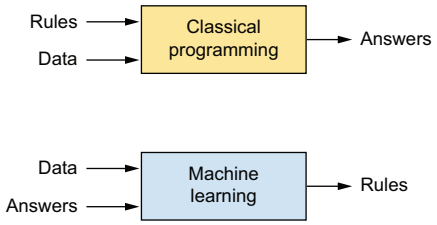
DEEP LEARNING
with Python

THIRD EDITION

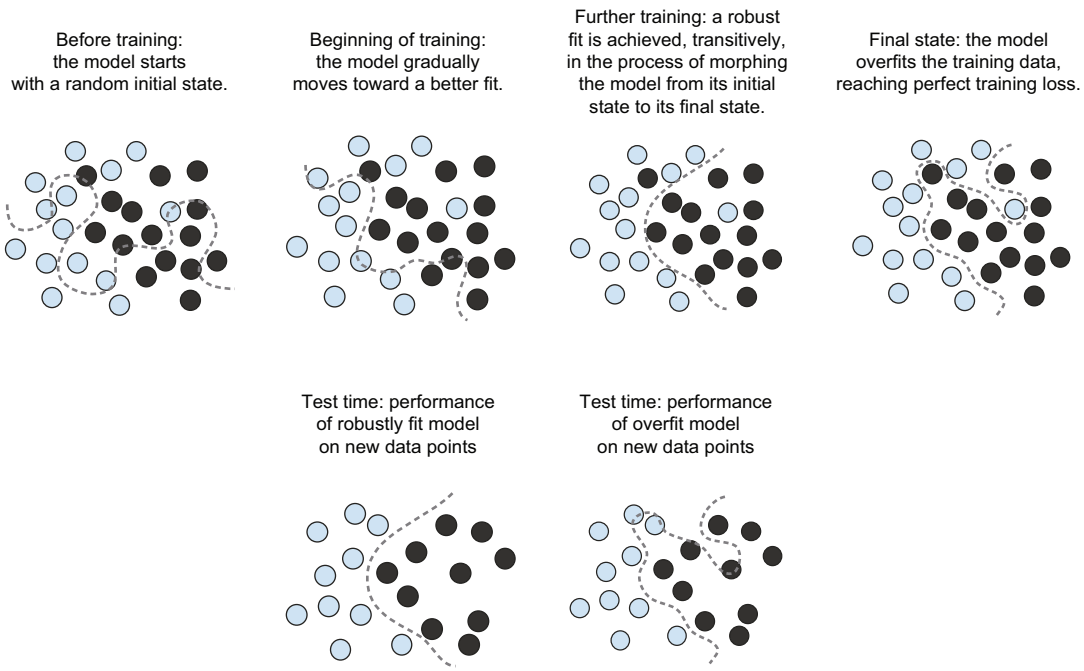
François Chollet
Matthew Watson

MANNING

Machine learning vs. classical programming (chapter 1)



Going from a random model to an overfit model (chapter 5)



Praise for the Second Edition

Chollet is a master of pedagogy and explains complex concepts with minimal fuss, cutting through the math with practical Python code. He is also an experienced ML researcher, and his insights on various model architectures or training tips are a joy to read.

—Martin Görner, Google

Immerse yourself in this exciting introduction to the topic with lots of real-world examples. A must-read for every deep learning practitioner.

—Sayak Paul, Carted

The modern classic just got better.

—Edmon Begoli, Oak Ridge National Laboratory

Truly the bible of deep learning.

—Yiannis Paraskevopoulos, University of West Attica

One of the best books on deep learning with Python.

—Raushan Jha, Microsoft

The book is full of insights, useful both for the novice and the more experienced machine learning professional.

—Viton Vitanis, Viseca Payment Services

Deep learning well explained, from A to Z.

—Todd Cook, Appen

This book really implements the democratization of AI: "AI to the people."

—Kjell Jansson, GubboIT

Deep Learning with Python, Third Edition

François Chollet
Matthew Watson

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com


Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Ian Hough
Review editor:	Dunja Nikitović
Production editor:	Kathy Rossland
Copy editor:	Alisa Larson
Proofreaders:	Katie Tennant and Melody Dolab
Technical proofreader:	Gabriel Rasskin
Typesetter:	Tamara Švelić Sabljić
Cover designer:	Marija Tudor

ISBN 9781633436589
Printed in the United States of America

brief contents

contents

preface

If you've picked up this book, you're probably aware of the extraordinary progress that deep learning has brought to the field of artificial intelligence. In just a handful of years, we went from near-unusable computer vision and natural language processing to highly performant systems deployed at scale in products you use every day. The consequences of this sudden progress extend to almost every industry. Deep learning is applied to a diverse range of important problems across domains as different as medical imaging, agriculture, autonomous driving, education, disaster prevention, and manufacturing. Digital assistants are becoming pervasive on nearly every consumer computing device.

Yet, we believe deep learning is still in its early days. AI is going to represent a much greater wave of disruption than anything that came before it. The technology will continue to make its way to every problem where it can provide some utility—a transformation that will take decades to play out. This is a transformation with profound societal implications, on a global scale, spanning industries and cultures. We strongly believe the only way to ensure such a transformation is beneficial for the people it will affect—all of us—is to radically democratize access to the underlying technology. We need to put understanding of deep learning—how it works, where it fails, and how to apply it—in the hands of as many people as possible, including people who aren't researchers in the field.

When I wrote the Keras deep learning framework in March 2015, the democratization of AI wasn't what I had in mind. I had been doing research in machine learning for several years and had built Keras to help with my own experiments. But since then, as newcomers have entered the field of deep learning, many have picked up Keras as their tool of choice. Accessibility quickly became an explicit goal in the development

of Keras, and over the last decade, the Keras developer community has made remarkable progress in this area. We've put deep learning into the hands of millions of people, who, in turn, are using it to solve problems that were, until recently, thought to be unsolvable.

The book you're holding is another step on the way toward broadening access to the field. We aim to make the concepts behind deep learning and their implementation as approachable as possible. Doing so doesn't require watering anything down; we believe that there are no difficult ideas in deep learning. This book will start with the very basics of the field and, layer by layer, build up to the cutting-edge generative AI models being deployed today.

This is the third edition of *Deep Learning with Python*. In an effort to make this content as broadly available as possible, we are making this edition available for free, online at https://deeplearningwithpython.io/. We hope you will consider purchasing a physical copy to support the project and read what we consider to be the best presentation of the content. This third edition amounts to a complete rewrite of the entire book, with a broader introduction to today's popular deep learning frameworks and greatly expanded content on large generative AI models.

We hope this book proves valuable and helps you solve the problems that matter to you.

acknowledgments

First, we'd like to thank the Keras community for making this book possible. Over the past decade, Keras has grown to have thousands of open source contributors and more than 2 million users. Their contributions and feedback have turned Keras into what it is today.

On a personal note, François would like to thank his wife for her endless support during the development of Keras and the writing of this book. Matthew would like to thank his partner Kate, his family, and all the friends who have supported him along the way.

We thank the people at Manning who made this book possible: publisher Marjan Bace and everyone on the editorial and production teams, including Michael Stephens, Aleksandar Dragosavljević, and many others who worked behind the scenes.

Many thanks go to the peer reviewers: Aakash Nain, Abheesht Sharma, Abhishek Shivanna, Aritra Roy Gosthipaty, Avinash Tiwari, Brandon Friar, Christopher Kardell, Srivathsan Srinivasagopalan, Edmon Begoli, Guillaume Alleon, Ian Stirk, Jacqueline Nolis, Kishore Reddy, Levi McClenny, Margaret Maynard-Reid, Nilson Chapagain, Prashanth Josyula, Preetish Kakkar, Sai Srinivas Somarouthu, Samuel Marks, Srivathsan Srinivasagopalan, Thiago Britto Borges, Todd Cook, and Varun Chawla—and all the other people who sent us feedback on the draft of the book.

On the coding side, special thanks go to Tomasz Kalinowski, who contributed to code examples in this book; Ian Hough, who served as the book's development editor; and Gabriel Rasskin, who served as the book's technical proofreader.

about this book

This book was written for anyone who wishes to explore deep learning from scratch or broaden their understanding of deep learning. Whether you're a practicing machine learning engineer, a software developer, or a college student, you'll find value in these pages.

You'll explore deep learning in an approachable way—starting simply and working up to state-of-the-art techniques. We hope you'll find that this book strikes a balance between intuition, theory, and hands-on practice. It avoids mathematical notation, preferring instead to explain the core ideas of deep learning via functioning code paired with explanations of the underlying principles. You'll train machine learning models from scratch in a number of different problem domains and learn practical recommendations for writing deep learning programs and deploying them in the real world.

After reading this book, you'll have a solid understanding of what deep learning is, when it's applicable, and what its limitations are. You'll be familiar with the standard workflow for approaching and solving machine learning problems, and you'll know how to address commonly encountered issues.

Who should read this book

This book is written for people with some Python programming experience who want to get started with machine learning and deep learning. But this book can also be valuable to many different types of readers:

- If you're a data scientist familiar with machine learning, this book will provide you with a solid, practical introduction to deep learning, the fastest-growing and most significant subfield of machine learning.

- If you're a deep learning researcher or practitioner looking to get started with the Keras framework, you'll find this book to be the ideal Keras crash course.
- If you're a graduate student studying deep learning in a formal setting, you'll find this book to be a practical complement to your education, helping you build intuition around the behavior of deep neural networks and familiarizing you with key best practices.

Even technically minded people who don't code regularly will find this book useful as an introduction to both basic and advanced deep learning concepts.

To understand the code examples, you'll need reasonable Python proficiency. You don't need previous experience with machine learning or deep learning: this book covers, from scratch, all the necessary basics. You don't need an advanced mathematics background either—high-school-level mathematics should suffice to follow along.

How this book is organized: A road map

This book contains 20 chapters, organized as follows:

- *Chapters 1–7: Foundations of deep learning*—These chapters give you a solid grounding in deep learning. We discuss the history of the field, provide an introduction to coding with the major deep learning frameworks currently available (JAX, PyTorch, and TensorFlow), and outline a practical guide to applying machine learning to real-world problems. We look closely at the Keras library for training machine learning models.
- *Chapters 8–12: Computer vision*—These chapters cover machine learning on image data. We train models to classify entire images and models to recognize and separate out individual objects in images. We look closely at ConvNets, a model architecture well suited to images and the first big success of the deep learning era.
- *Chapter 13: Timeseries forecasting*—This chapter covers working with timeseries data. We take data that is measured over time, like the weather or sales data, and use past patterns to predict future trends.
- *Chapters 14–15: Natural language processing*—These chapters cover applying deep learning to text. We build sentiment classifiers to predict whether text is positive or negative. We build an English-to-Spanish translation model. We pay special attention to the Transformer, a popular model architecture particularly well suited to text modeling.
- *Chapters 16–17: Generative modeling*—These chapters are all about using deep learning to create new content. We work with models that can produce new images and text as output. We discuss chatbots like ChatGPT and image generation systems. We both train generative models from scratch and use larger pretrained models trained on large amounts of data over weeks or months.
- *Chapters 18–20: Conclusions*—The final chapters of this book tie together what we have learned. We explore practical engineering concerns with deep learning systems. We dedicate a chapter to exploring potential futures for the field

and identifying new research directions, and then review the concepts presented across the book.

About the code

This book is an example of literate programming—it is simultaneously a prose explanation of deep learning concepts and a runnable Python program. Starting with chapter 2, each chapter will be interspersed with code listings that can be run as a Jupyter notebook, an interactive notebook for running Python code and visualizing data. We strongly suggest running, modifying, and playing with the examples in this book as you read to build your own intuitions on the concepts presented.

All code is written in the Python language with the deep learning library Keras. Keras can be run on top of TensorFlow, PyTorch, and JAX, the most popular low-level deep learning frameworks as of 2025. All code can be run on a local machine or directly in the browser using Google Colab, a hosted environment for Jupyter notebooks.

You can get executable snippets of code from the liveBook version of this book, included with your print or eBook purchase, at https://livebook.manning.com/book/deep-learning-with-python-third-edition. The complete code for the examples in the book is available for download from the Manning website at https://www.manning.com/books/deep-learning-with-python-third-edition and on GitHub, at https://github.com/fchollet/deep-learning-with-python-notebooks, along with instructions for running the code locally.

You can run the code directly through the web version of this book available at https://deeplearningwithpython.io. A Colab link appears at the top of each chapter allowing you to run the chapter's code in the browser on free, hosted hardware.

liveBook discussion forum

Purchase of *Deep Learning with Python, Third Edition,* includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the authors and other users. To access the forum, go to https://livebook.manning.com/book/deep-learning-with-python-third-edition/discussion.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors

FRANÇOIS CHOLLET has been working with deep learning since it started getting traction in academia in 2012. François is the author of Keras, one of the most popular libraries for deep learning. Keras is used in university classrooms; at companies like Google, Netflix, and Spotify; and in scientific organizations like CERN and NASA. Francois is the co-founder of the Ndea research lab for frontier AI systems and created the ARC-AGI challenge for measuring machine intelligence.

MATTHEW WATSON has been working on machine learning across Google since 2018, including the Gemini model and Google's open source deep learning ecosystem. He is a core maintainer of Keras, focusing on Keras's tools for natural language processing. He completed his master's in computer science at Stanford University, researching procedural modeling techniques at the Stanford Graphics Lab.

about the cover illustration

The figure on the cover of *Deep Learning with Python, Third Edition*, is captioned "Habit of a Persian Lady in 1568." The illustration is taken from Thomas Jefferys' *A Collection of the Dresses of Different Nations, Ancient and Modern*, published between 1757 and 1772.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

1 What is deep learning?

This chapter covers

- High-level definitions of fundamental concepts
- A soft introduction to the principles behind machine learning
- Deep learning's rising popularity and future potential

Over the past decade, artificial intelligence (AI) has been a subject of intense media hype. Machine learning, deep learning, and AI come up in countless articles, often outside of technology-minded publications. We're promised a future of intelligent chatbots, self-driving cars, and virtual assistants—a future sometimes painted in a grim light and other times as utopian, where human jobs will be scarce and most economic activity will be handled by robots or AI agents. For a practitioner of machine learning, it's important to be able to recognize the signal amid the noise, so that you can tell world-changing developments from overhyped press releases. Our future is at stake, and it's one in which you have an active role to play: after reading this book, you'll be one of those who can develop these AI systems. So let's

tackle these questions: What has deep learning achieved so far? How significant is it? Where are we headed next? Should you believe the hype?

1.1 Artificial intelligence, machine learning, and deep learning

First, we need to define clearly what we're talking about when we mention AI. What are artificial intelligence, machine learning, and deep learning (figure 1.1)? How do they relate to each other?

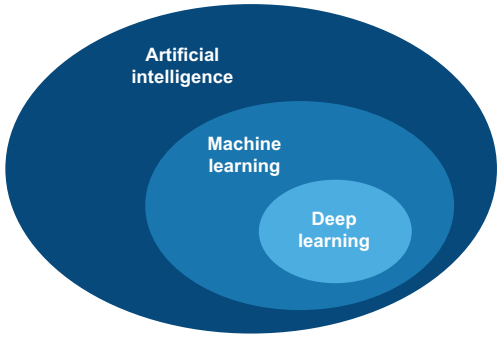


Figure 1.1 Artificial intelligence, machine learning, and deep learning

1.2 Artificial intelligence

Artificial intelligence was born in the 1950s, when a handful of pioneers from the nascent field of computer science started asking whether computers could be made to "think"—a question whose ramifications we're still exploring today.

While many of the underlying ideas had been brewing in the years and even decades prior, "artificial intelligence" finally crystallized as a field of research in 1956, when John McCarthy, then a young Assistant Professor of Mathematics at Dartmouth College, organized a summer workshop under the following proposal:

> *The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer.*

At the end of the summer, the workshop concluded without having fully solved the riddle it set out to investigate. Nevertheless, it was attended by many people who would move on to become pioneers in the field, and it set in motion an intellectual revolution that is still ongoing to this day.

Concisely, AI can be described as *the effort to automate intellectual tasks normally performed by humans*. As such, AI is a general field that encompasses machine learning and deep learning, but that also includes many more approaches that may not involve any learning. Consider that until the 1980s, most AI textbooks didn't mention "learning" at all! Early chess programs, for instance, only involved hardcoded rules crafted by programmers and didn't qualify as machine learning. In fact, for a fairly long time, most experts believed that human-level artificial intelligence could be achieved by having programmers handcraft a sufficiently large set of explicit rules for manipulating

knowledge stored in explicit databases. This approach is known as *symbolic AI*. It was the dominant paradigm in AI from the 1950s to the late 1980s, and it reached its peak popularity during the *expert systems* boom of the 1980s.

Although symbolic AI proved suitable to solve well-defined, logical problems, such as playing chess, it turned out to be intractable to figure out explicit rules for solving more complex, fuzzy problems, such as image classification, speech recognition, or natural language translation. A new approach arose to take symbolic AI's place: *machine learning*.

1.3 Machine learning

In Victorian England, Lady Ada Lovelace was a friend and collaborator of Charles Babbage, the inventor of the Analytical Engine: the first-known general-purpose mechanical computer. Although visionary and far ahead of its time, the Analytical Engine wasn't meant as a general-purpose computer when it was designed in the 1830s and 1840s, because the concept of general-purpose computation was yet to be invented. It was merely meant as a way to use mechanical operations to automate certain computations from the field of mathematical analysis—hence the name Analytical Engine. As such, it was the intellectual descendant of earlier attempts at encoding mathematical operations in gear form, such as the Pascaline, or Leibniz's step reckoner, a refined version of the Pascaline. Designed by Blaise Pascal in 1642 (at age 19!), the Pascaline was the world's first mechanical calculator—it could add, subtract, multiply, or even divide digits.

In 1843, Ada Lovelace remarked on the invention of the Analytical Engine:

> *The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform… Its province is to assist us in making available what we're already acquainted with.*

Even with 182 years of historical perspective, Lady Lovelace's observation remains arresting. Could a general-purpose computer "originate" anything, or would it always be bound to dully execute processes we humans fully understand? Could it ever be capable of any original thought? Could it learn from experience? Could it show creativity?

Her remark was later quoted by AI pioneer Alan Turing as "Lady Lovelace's objection" in his landmark 1950 paper "Computing Machinery and Intelligence,"[1] which introduced the *Turing test*[2] as well as key concepts that would come to shape AI. Turing was of the opinion—highly provocative at the time—that computers could, in principle, be made to emulate all aspects of human intelligence.

The usual way to make a computer do useful work is to have a human programmer write down rules—a computer program—to be followed to turn input data into appropriate answers, just like Lady Lovelace writing down step-by-step instructions for the Analytical

[1] A. M. Turing, "Computing Machinery and Intelligence," Mind 59, no. 236 (1950): 433-460.

[2] Although the Turing test has sometimes been interpreted as a literal test—a goal the field of AI should set out to reach—Turing merely meant it as a conceptual device in a philosophical discussion about the nature of cognition.

Engine to perform. Machine learning turns this around: the machine looks at the input data and the corresponding answers, and figures out what the rules should be (figure 1.2).

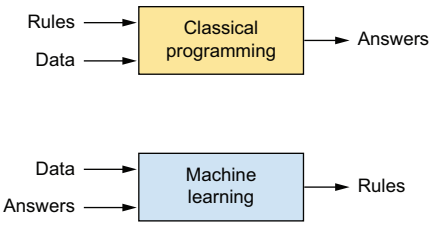


Figure 1.2 Machine learning: a new programming paradigm

A machine learning system is *trained* rather than explicitly programmed. It's presented with many examples relevant to a task, and it finds statistical structure in these examples that eventually allows the system to come up with rules for automating the task. For instance, if you wished to automate the task of tagging your vacation pictures, you could present a machine learning system with many examples of pictures already tagged by humans, and the system would learn statistical rules for associating specific pictures to specific tags like "landscape" or "food."

Although machine learning only started to flourish in the 1990s, it has quickly become the most popular and most successful subfield of AI, a trend driven by the availability of faster hardware and larger datasets. Machine learning is related to mathematical statistics, but it differs from statistics in several important ways—in the same sense that medicine is related to chemistry but cannot be reduced to chemistry, as medicine deals with its own distinct systems with their own distinct properties. Unlike statistics, machine learning tends to deal with large, complex datasets (such as a dataset of millions of images, each consisting of tens of thousands of pixels) for which classical statistical analysis such as Bayesian analysis would be impractical. As a result, machine learning, and especially deep learning, exhibits comparatively little mathematical theory—maybe too little—and is fundamentally an engineering discipline. Unlike theoretical physics or mathematics, machine learning is a very hands-on field driven by empirical findings and deeply reliant on advances in software and hardware.

1.4 Learning rules and representations from data

To define *deep learning* and understand the difference between deep learning and other machine learning approaches, first we need some idea of what machine learning algorithms do. We just stated that machine learning discovers rules to execute a data processing task, given examples of what's expected. So, to do machine learning, we need three things:

- *Input data points*—For instance, if the task is speech recognition, these data points could be sound files of people speaking. If the task is image tagging, they could be pictures.
- *Examples of the expected output*—In a speech-recognition task, these could be human-generated transcripts of sound files. In an image task, expected outputs could be tags such as "dog," "cat," and so on.
- *A way to measure whether the algorithm is doing a good job*—This is necessary to determine the distance between the algorithm's current output and its expected

output. The measurement is used as a feedback signal to adjust the way the algorithm works. This adjustment step is what we call *learning*.

A machine learning model transforms its input data into meaningful outputs, a process that is "learned" from exposure to known examples of inputs and outputs. Therefore, the central problem in machine learning and deep learning is to *meaningfully transform data*: in other words, to learn useful *representations* of the input data at hand—representations that get us closer to the expected output.

Before we go any further, what's a representation? At its core, it's a different way to look at data to represent or encode data. For instance, a color image can be encoded in the RGB format (red-green-blue) or in the HSV format (hue-saturation-value): these are two different representations of the same data. Some tasks that may be difficult with one representation can become easy with another. For example, the task "Select all red pixels in the image" is simpler in the RGB format, whereas "Make the image less saturated" is simpler in the HSV format. Machine learning models are all about finding appropriate representations for their input data—transformations of the data that make it more amenable to the task at hand.

Let's make this concrete. Consider an x-axis, a y-axis, and some points represented by their coordinates in the (x, y) system, as shown in figure 1.3.

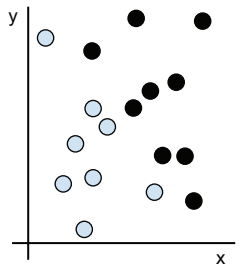


Figure 1.3 Some sample data

As you can see, we have a few white points and a few black points. Let's say we want to develop an algorithm that can take a point's (x, y) coordinates and output whether that point is likely black or white. In this case,

- The inputs are the coordinates of our points.
- The expected outputs are the colors of our points.
- A way to measure whether our algorithm is doing a good job could be, for instance, the percentage of points that are being correctly classified.

What we need here is a new representation of our data that cleanly separates the white points from the black points. One transformation we could use, among many other possibilities, would be a coordinate change, illustrated in figure 1.4.

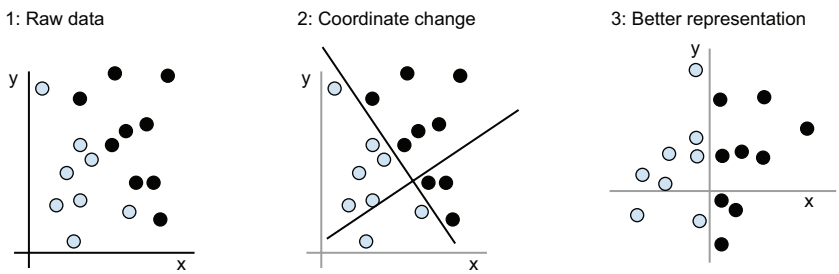


Figure 1.4 Coordinate change

In this new coordinate system, the coordinates of our points can be said to be a new representation of our data. And it's a good one! With this representation, the black/white classification problem can be expressed as a simple rule: "Black points are such that $x > 0$," or "White points are such that $x < 0$." This new representation, combined with this simple rule, neatly solves the classification problem.

In this case, we defined the coordinate change by hand: we used our human intelligence to come up with our own appropriate representation of the data. This is fine for such an extremely simple problem, but could you do the same if the task were to classify images of handwritten digits? Could you write explicit, computer-executable image transformations that would illuminate the difference between a 6 and an 8, between a 1 and a 7, across all kinds of different handwritings?

This is possible to an extent. Rules based on representations of digits such as "counting the number of closed loops" or vertical and horizontal pixel histograms can do a decent job at telling apart handwritten digits. But finding such useful representations by hand is hard work, and as you can imagine the resulting rule-based system would be brittle and a nightmare to maintain. Every time you would come across a new example of handwriting that would break your carefully thought-out rules, you would have to add new data transformations and new rules, while taking into account their interaction with every previous rule.

You're probably thinking, if this process is so painful, could we automate it? What if we tried systematically searching for different sets of automatically generated representations of the data and rules based on them, identifying good ones using the percentage of digits being correctly classified in some development dataset as feedback? We would then be doing machine learning. *Learning*, in the context of machine learning, describes an automatic search process for data transformations that produce useful representations of some data, guided by some feedback signal—representations that are amenable to simpler rules solving the task at hand.

These transformations can be coordinate changes (like in our 2D coordinates classification example) or a histogram of pixels and counting loops (like in our digits classification example), but they could also be linear projections, translations, and nonlinear operations (such as "Select all points such that $x > 0$"), and so on. Machine learning algorithms aren't usually creative in finding these transformations; they're merely searching through a predefined set of operations, called a *hypothesis space*. For instance, the space of all possible coordinate changes would be our hypothesis space in the 2D coordinates classification example.

So that's what machine learning is, concisely: searching for useful representations and rules over some input data, within a predefined space of possibilities, using guidance from a feedback signal. This simple idea allows us to solve a remarkably broad range of intellectual tasks, from autonomous driving to natural language question-answering.

Now that you understand what we mean by *learning*, let's take a look at what makes *deep learning* special.

1.5 The "deep" in "deep learning"

Deep learning is a specific subfield of machine learning; it's a new take on learning representations from data, which emphasizes learning successive layers of increasingly meaningful representations. The "deep" in "deep learning" isn't a reference to any kind of deeper understanding achieved by the approach; rather, it stands for this idea of successive layers of representations. How many layers contribute to a model of the data is called the *depth* of the model. Other appropriate names for the field could have been *layered representations learning* or *hierarchical representations learning*. Modern deep learning often involves tens or even hundreds of successive layers of representations, and they're all learned automatically from exposure to training data. Meanwhile, other approaches to machine learning tend to focus on learning only one or two layers of representations of the data (say, taking a pixel histogram and then applying a classification rule); hence they're sometimes called *shallow learning*.

In deep learning, these layered representations are learned via models called *neural networks*, structured in literal layers stacked on top of each other. The term *neural network* is a reference to neurobiology, but although some of the central concepts in deep learning were developed in part by drawing inspiration from our understanding of the brain (in particular, the visual cortex), deep learning models are not models of the brain. There's no evidence that the brain implements anything like the learning mechanisms used in modern deep learning models. You may come across pop science articles proclaiming that deep learning works like the brain or is modeled after the brain, but that isn't the case. It would be confusing and counterproductive for newcomers to the field to think of deep learning as being in any way related to neurobiology; you don't need that shroud of "just like our minds" mystique and mystery, and you may as well forget anything you may have read about hypothetical links between deep learning and biology. For our purposes, deep learning is a mathematical framework for learning representations from data.

What do the representations learned by a deep learning algorithm look like? Let's examine how a network several layers deep (see figure 1.5) transforms an image of a digit to recognize what digit it is.

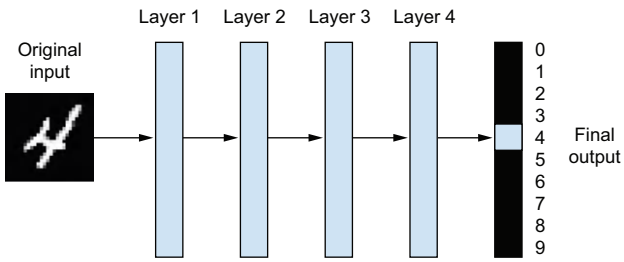


Figure 1.5 A deep neural network for digit classification

As you can see in figure 1.6, the network transforms the digit image into representations that are increasingly different from the original image and increasingly informative about the final result. You can think of a deep network as a multistage *information-distillation* process, where information goes through successive filters and comes out increasingly *purified* (that is, useful with regard to some task).

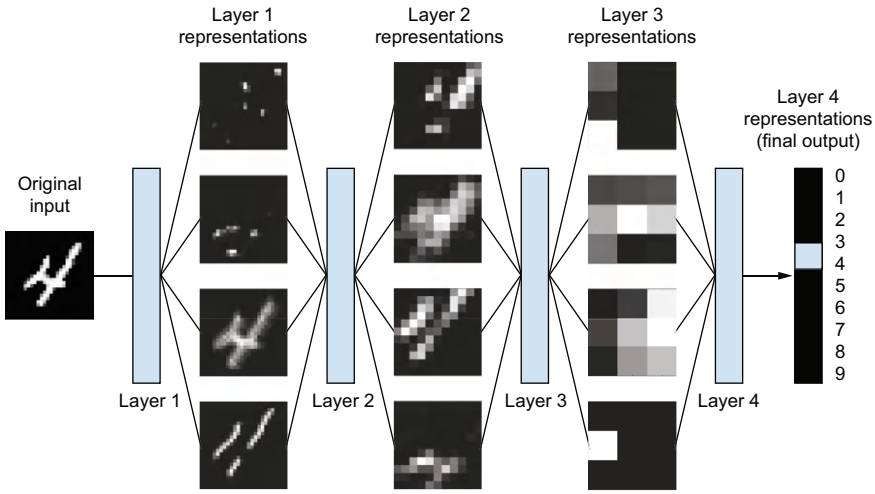


Figure 1.6 Deep representations learned by a digit-classification model

So that's what deep learning is, technically: a multistage way to learn data representations. It's a simple idea, but, as it turns out, very simple mechanisms, sufficiently scaled, can end up looking like magic.

1.6 Understanding how deep learning works, in three figures

At this point, you know that machine learning is about mapping inputs (such as images) to targets (such as the label "cat"), which is done by observing many examples of inputs and targets. You also know that deep neural networks do this input-to-target mapping via a deep sequence of simple data transformations (layers) and that these data transformations are learned by exposure to examples. Now let's look at how this learning happens, concretely.

The specification of what a layer does to its input data is stored in the layer's *weights*, which in essence are a bunch of numbers. In technical terms, we'd say that the transformation implemented by a layer is *parameterized* by its weights (see figure 1.7). (Weights are also sometimes called the parameters of a layer.) In this context, *learning* means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets. But here's the thing: a deep neural network can contain tens of millions of parameters. Finding the correct

value for all of them may seem like a daunting task, especially given that modifying the value of one parameter will affect the behavior of all the others!

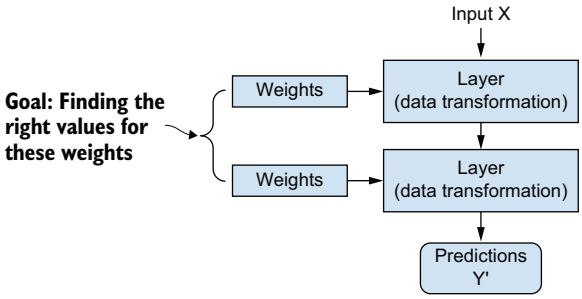


Figure 1.7 A neural network is parameterized by its weights.

To control something, first you need to be able to observe it. To control the output of a neural network, you need to be able to measure how far this output is from what you expected. This is the job of the *loss function* of the network, also sometimes called the *objective function* or *cost function*. The loss function takes the predictions of the network and the true target (what you wanted the network to output) and computes a distance score, capturing how well the network has done on this specific example (see figure 1.8).

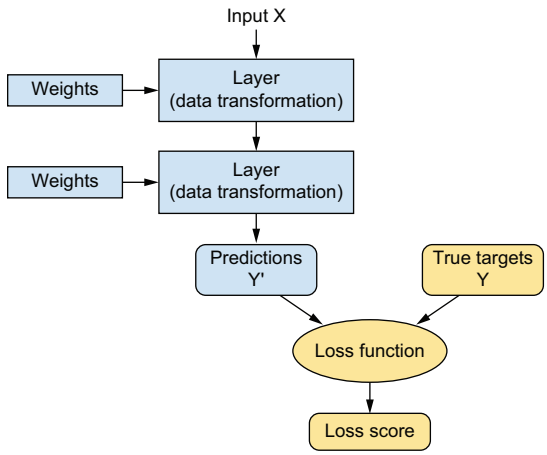


Figure 1.8 A loss function measures the quality of the network's output.

The fundamental trick in deep learning is to use this score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score for the current example (see figure 1.9). This adjustment is the job of the *optimizer*, which implements what's called the *Backpropagation* algorithm: the central algorithm in deep learning. The next chapter explains in more detail how backpropagation works.

Initially, the weights of the network are assigned random values, so the network merely implements a series of random transformations. Naturally, its output is far from what it should ideally be, and the loss score is accordingly very high. But with every example the network processes, the weights are adjusted a little in the correct direction, and the loss score decreases. This is the *training loop*, which, repeated a sufficient

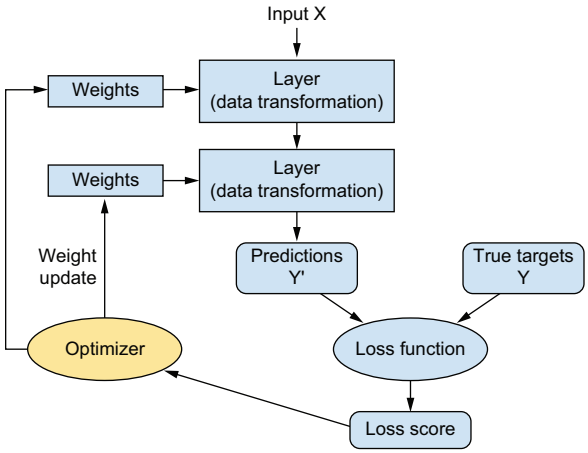


Figure 1.9 The loss score is used as a feedback signal to adjust the weights.

number of times (typically tens of passes over thousands of examples), yields weight values that minimize the loss function. A network with a minimal loss is one for which the outputs are as close as they can be to the targets: a trained network. Once again, it's a simple mechanism that, once scaled, ends up looking like magic.

1.7 What makes deep learning different

Is there anything special about deep neural networks that makes them the "right" approach for companies to invest in and for researchers to flock to? Will we still be using deep neural networks in 20 years?

Deep learning has several properties that justify its status as an AI revolution, and it's here to stay. We may not be using neural networks many decades from now, but whatever we use will directly inherit from modern deep learning and its core concepts. These important properties can be broadly sorted into three categories:

- *Simplicity*—Deep learning makes problem solving much easier, because it automates what used to be the most crucial step in a machine learning workflow: feature engineering. Previous machine learning techniques—shallow learning—only involved transforming the input data into one or two successive representation spaces, which wasn't expressive enough for most problems. As such, humans had to go to great lengths to make the initial input data more amenable to processing by these methods: they had to manually engineer good representations for their data. This is called *feature engineering*. Deep learning, on the other hand, completely automates this step: with deep learning, you learn all features in one pass rather than having to engineer them yourself. This has greatly simplified machine learning workflows, often replacing sophisticated multistage pipelines with a single, simple, end-to-end deep learning model.
- *Scalability*—Deep learning is highly amenable to parallelization on GPUs or more specialized machine learning hardware, so it can take full advantage of Moore's

law. In addition, deep learning models are trained by iterating over small batches of data, allowing them to be trained on datasets of arbitrary size. (The only bottleneck is the amount of parallel computational power available, which, thanks to Moore's law, is a fast-moving barrier.)

- *Versatility and reusability*—Unlike many prior machine learning approaches, deep learning models can be trained on additional data without restarting from scratch, making them viable for continuous online learning—an important property for very large production models. Furthermore, trained deep learning models are repurposable and thus reusable: this is the big idea behind "foundation models"—large models trained on humongous amounts of data, which can be used across many new tasks with little retraining, or even none at all.

1.8 The age of generative AI

Perhaps the most well-known example of deep learning today is the recent wave of generative AI applications—chatbot assistants like ChatGPT, Gemini, and Claude, as well as image generation services like Midjourney. These applications have captured the public imagination with their ability to produce informative or even creative content in response to simple prompts, blurring the lines between human and machine creativity.

Generative AI is powered by very large "foundation models" that learn to *reconstruct* the text and image content fed into them—reconstruct a sharp image from a noisy version, predict the next word in a sentence, and so on. This means that the *targets* from figure 1.8 are taken from the input itself. This is referred to as *self-supervised learning*, and it enables those models to use vast amounts of unlabeled data. Doing away with the manual data annotations that bottlenecked previous brands of machine learning has unlocked a level of scale never seen before—some of these foundation models have hundreds of billions of parameters and are trained on over 1 petabyte of data, at the cost of tens of millions of dollars.

These foundation models operate as a kind of fuzzy database of human knowledge, making them amenable to a very wide range of applications without needing special-purpose programming or retraining. Because they've already memorized so much, they can solve new problems merely via *prompting*—querying the knowledge representations they've learned and returning the output most likely to be associated with your prompt.

Generative AI only rose to mainstream awareness in 2022, but it has a long history—the earliest experiments with text generation date back to the 1990s. The first edition of this book, released in 2017, already had a hefty chapter titled "Generative AI" that explored the text generation and image generation techniques of the time, while promising the then-outlandish notion that, "soon," much of the cultural content we consume would be created with the help of AI.

1.9 What deep learning has achieved so far

Over the past decade, deep learning has achieved nothing short of a technological revolution, starting with remarkable results on perceptual tasks from 2013 to 2017,

then making fast progress on natural language processing tasks from 2017 to 2022, and culminating with a wave of transformative generative AI applications from 2022 to now.

Deep learning has enabled major breakthroughs, all in extremely challenging problems that had long eluded machines:

- Fluent and highly versatile chatbots such as ChatGPT and Gemini
- Programming assistants like GitHub Copilot
- Photorealistic image generation
- Human-level image classification
- Human-level speech transcription
- Human-level handwriting transcription and printed text transcription
- Dramatically improved machine translation
- Dramatically improved text-to-speech conversion
- Human-level autonomous driving, already deployed to the public in Phoenix, San Francisco, Los Angeles, and Austin as of 2025
- Improved recommender systems, as used by YouTube, Netflix, or Spotify
- Superhuman Go, Chess, and Poker playing

We're still exploring the full extent of what deep learning can do. We've started applying it with great success to a wide variety of problems that were thought to be impossible to solve just a few years ago—automatically transcribing the tens of thousands of ancient manuscripts held in the Vatican Secret Archive, detecting and classifying plant diseases in fields using a simple smartphone, assisting oncologists or radiologists with interpreting medical imaging data, predicting natural disasters such as floods, hurricanes, and even earthquakes. With every milestone, we're getting closer to an age where deep learning assists us in every activity and every field of human endeavor—science, medicine, manufacturing, energy, transportation, software development, agriculture, and even artistic creation.

1.10 Beware of the short-term hype

This seemingly unstoppable string of successes has led to a wave of intense hype, some of which is somewhat grounded, but most of which is just fancy fairy tales. In early 2023, soon after the release of GPT-4 by OpenAI, many pundits were claiming that "no one needed to work anymore" and that mass unemployment would be coming within a year, or that economic productivity would soon shoot up by 10× to 100×. Of course, two years later, none of this has come to pass—the unemployment rate in the US remains low, while productivity metrics are far from the promised explosion. Don't misunderstand: impact of AI—in particular, generative AI—is already considerable, and it is growing remarkably fast. As of mid-2025, generative AI was generating tens of billions of dollars in revenue per year, which is extremely impressive for an industry that did not exist three years prior! But it doesn't yet make much of a dent in the

overall economy and pales in comparison to the absolutely unbridled promises we were inundated with at its onset.

While discussions about unemployment and 100× productivity gains triggered by AI are already stirring anxieties, there's an even more sensational side to the AI hype. This side proclaims the imminent arrival of human-level general intelligence (AGI), or even "superintelligence" far surpassing human capabilities. These claims are fueling fears beyond economic disruption—the human species itself might be in danger of being replaced by our digital creations.

It might be tempting for those new to the field to assume that it is the practical successes of generative AI that caused the belief in near-term AGI, but that is actually backward. The claims of near-term AGI came first, and they significantly contributed to the rise of generative AI. As early as 2013, there were fears among tech elites that AGI might be coming within a few years. Back then, the idea was that DeepMind, a London AI research startup acquired by Google, was on track to achieve it. This belief was the impetus behind the founding of OpenAI in 2015, which initially aimed to be an open source counterweight to DeepMind. OpenAI played a critical role in kick-starting generative AI, so in a peculiar twist, it was the belief in near-term AGI that fueled the ascent of generative AI, not the other way around. In 2016, OpenAI's recruiting pitch was that it would achieve AGI by 2020! To be fair, though, only a minority of people in the tech industry believed in such an optimistic timeline back then. By early 2023, however, a significant fraction of engineers in the San Francisco Bay Area seemed convinced that AGI would be coming within the following couple of years.

It's crucial to approach such claims with a healthy dose of skepticism. Despite its name, today's "artificial intelligence" is more accurately described as "cognitive automation"—the encoding and operationalization of human skills and knowledge. AI excels at solving problems with narrowly defined requirements or those where ample precise examples are available. It's about enhancing the capabilities of computers, not about replicating human minds.

To be clear, cognitive automation is incredibly useful. But intelligence—cognitive autonomy—is a different creature altogether. Think of it this way: AI is like a cartoon character, while intelligence is like a living being. A cartoon, no matter how realistic, can only act out the scenes it was drawn for. A living being, on the other hand, can adapt to the unexpected.

"If the cartoon is drawn with sufficient realism and covers sufficiently many scenes, what's the difference?" you may ask. If a large language model can output a sufficiently human-sounding answer when asked a question, does it matter if it possesses true cognitive autonomy? The key difference is adaptability. Intelligence is the ability to face the unknown, adapt to it, and learn from it. Automation, even at its best, can only handle situations it's been trained on or programmed for. That's why creating robust automation is so challenging—it requires accounting for every possible scenario.

So don't worry about AI suddenly becoming self-aware and taking over humanity. Today's technology simply isn't headed in that direction. Even with significant

advancements, AI will remain a sophisticated tool, not a sentient being. It's like expecting a better clock to lead to time travel—they're just different things altogether.

1.11 Summer can turn to winter

The danger of inflated short-term expectations is that when technology inevitably falls short, research investment could dry up, slowing progress for a long time. This has happened before. Twice in the past, AI went through a cycle of intense optimism followed by disappointment and skepticism, with a dearth of funding as a result. It started with symbolic AI in the 1960s. In those early days, projections about AI were flying high. One of the best-known pioneers and proponents of the symbolic AI approach was Marvin Minsky, who claimed in 1967, "Within a generation . . . the problem of creating 'artificial intelligence' will substantially be solved." Three years later, in 1970, he made a more precisely quantified prediction: "In from three to eight years we will have a machine with the general intelligence of an average human being." In 2025, such an achievement still appears to be far in the future—in that we have no way to predict how long it will take—but in the 1960s and early 1970s, several experts believed it to be right around the corner (as do many people today). A few years later, as these high expectations failed to materialize, researchers and government funds turned away from the field, marking the start of the first *AI winter* (a reference to a nuclear winter, because this was shortly after the height of the Cold War).

It wouldn't be the last one. In the 1980s, a new take on symbolic AI, *expert systems*, started gathering steam among large companies. A few initial success stories triggered a wave of investment, with corporations around the world starting their own in-house AI departments to develop expert systems. Around 1985, companies were spending over $1 billion each year on the technology; but by the early 1990s, these systems had proven expensive to maintain, difficult to scale, and limited in scope, and interest died down. Thus began the second AI winter. We may be currently witnessing the third cycle of AI hype and disappointment—and we're still in the phase of intense optimism.

My current view is that we're unlikely to see a full-scale retreat away from AI research like we saw in the 1990s. If there is a winter, it should be very mild. AI has already demonstrated its world-changing value. However, it seems inevitable that some air will need to be let out of the 2023–2025 AI bubble. Currently, AI investment, primarily in data centers and GPUs, surpasses $200 billion annually, while revenue generation lags significantly, closer to $30 billion. AI is currently being judged by executives and investors not by what it has accomplished, but by what we are told it might soon become able to do—much of which will durably stay out of reach of existing technologies. Something will have to give. But what will happen precisely as the AI bubble deflates is still up in the air.

1.12 The promise of AI

Although we may have unrealistic short-term expectations for AI, the long-term picture is looking bright. We're only getting started in applying deep learning to many important problems for which it could prove transformative, from medical diagnoses to digital assistants.

In 2017, in this very book, I wrote:

> *Right now, it may seem hard to believe that AI could have a large impact on our world, because it isn't yet widely deployed—much as, back in 1995, it would have been difficult to believe in the future impact of the internet. Back then, most people didn't see how the internet was relevant to them and how it was going to change their lives. The same is true for deep learning and AI today. But make no mistake: AI is coming. In a not-so-distant future, AI will be your assistant, even your friend; it will answer your questions, help educate your kids, and watch over your health. It will deliver your groceries to your door and drive you from point A to point B. It will be your interface to an increasingly complex and information-intensive world. And, even more important, AI will help humanity as a whole move forward, by assisting human scientists in new breakthrough discoveries across all scientific fields, from genomics to mathematics.*

Fast-forwarding to 2025, most of these things have either come true or are on the verge of coming true—and this is just the beginning:

- Tens of millions of people are using AI chatbots like ChatGPT, Gemini, and Claude as assistants on a daily basis. In fact, question-answering and "educating your kids" (homework assistance) have turned out to be the top applications of these chatbots! For many people, AI is already the go-to interface to the world's information.
- Hundreds of thousands of people interact with AI "friends" in applications such as Character.ai.
- Fully autonomous driving is already deployed at scale in cities like Phoenix, San Francisco, Los Angeles, and Austin.
- AI is making major strides toward helping accelerate science. The AlphaFold model from DeepMind is helping biologists predict protein structures with unprecedented accuracy. Renowned mathematician Terence Tao believes that by around 2026, AI could become a reliable co-author in mathematical research and other fields when used appropriately.

The AI revolution, once a distant vision, is now rapidly unfolding before our eyes. On the way, we may face a few setbacks—in much the same way the internet industry was overhyped in 1998–1999 and suffered from a crash that dried up investment throughout the early 2000s. But we'll get there eventually. AI will end up being applied to nearly every process that makes up our society and our daily lives, much like the internet is today.

Don't believe the short-term hype, but do believe in the long-term vision. It may take a while for AI to be deployed to its true potential—a potential the full extent of which no one has yet dared to dream—but AI is coming, and it will transform our world in a fantastic way.

2 The mathematical building blocks of neural networks

This chapter covers

- A first example of a neural network
- Tensors and tensor operations
- How neural networks learn via backpropagation and gradient descent

Understanding deep learning requires familiarity with many simple mathematical concepts: *tensors*, *tensor operations*, *differentiation*, *gradient descent*, and so on. Our goal in this chapter will be to build up your intuition about these notions without getting overly technical. In particular, we'll steer away from mathematical notation, which can introduce unnecessary barriers for those without any mathematics background, and isn't necessary to explain things well. The most precise, unambiguous description of a mathematical operation is its executable code.

To provide sufficient context for introducing tensors and gradient descent, we'll begin the chapter with a practical example of a neural network. Then we'll go over every new concept that's been introduced, point by point. Keep in mind that these concepts will be essential for you to understand the practical examples that will come in the following chapters!

After reading this chapter, you'll have an intuitive understanding of the mathematical theory behind deep learning, and you'll be ready to start diving into modern deep learning frameworks, in chapter 3.

Running the code in this book

This book is full of runnable Python code. Each chapter is paired with a *Jupyter notebook* that contains all of the code from the chapter. A Jupyter notebook is a live Python scratch pad of sorts, where you can interactively run code, graph data, view images, and a lot more. You will gain a lot more practical knowledge from this book if you run and experiment with the code as you read.

By far the easiest way to set up a deep learning environment to run these notebooks is *Google Colaboratory* (or Colab for short), a hosted environment for Jupyter notebooks that has become the industry standard for ML practitioners. With Colab, you can run the code for this book interactively in the browser, connecting to cloud runtimes with configurable hardware. By default, the notebooks in this book will run on Colab's free GPU runtime.

If you would like, you can also run these notebooks locally on your own machine. A GPU is recommended, especially as you get to the larger and more compute-intensive models later in this book.

Instructions for running locally and on Colab, along with the code, can be found at https://github.com/fchollet/deep-learning-with-python-notebooks.

2.1 A first look at a neural network

Let's look at a concrete example of a neural network that uses the machine learning library *Keras* to learn to classify handwritten digits. We will use Keras extensively throughout this book. It's a simple, high-level library that will allow us to stay focused on the concepts we would like to cover.

Unless you already have experience with Keras or similar libraries, you won't understand everything about this first example right away. That's fine. In a few sections, we'll review each element in the example and explain it in detail. So don't worry if some steps seem arbitrary or look like magic to you! We've got to start somewhere.

The problem we're trying to solve here is to classify grayscale images of handwritten digits (28×28 pixels) into their 10 categories (0 through 9). We'll use the MNIST dataset, a classic in the machine learning community, which has been around almost as long as the field itself and has been intensively studied. It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s. You can think of "solving" MNIST as the "Hello World" of deep learning—it's what you do to verify that your algorithms are working as expected. As you become a machine learning practitioner, you'll see MNIST come up over and over again, in scientific papers, blog posts, and so on. You can see some MNIST samples in figure 2.1.

NOTE In machine learning, a *category* in a classification problem is called a *class*. Data points are called *samples*. The class associated with a specific sample is called a *label*.

Figure 2.1 MNIST sample digits

The MNIST dataset comes preloaded in Keras, in the form of a set of four NumPy arrays.

Listing 2.1 Loading the MNIST dataset in Keras

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

`train_images` and `train_labels` form the training set, the data that the model will learn from. The model will then be tested on the test set, `test_images` and `test_labels`. The images are encoded as NumPy arrays, and the labels are an array of digits, ranging from 0 to 9. The images and labels have a one-to-one correspondence.

NOTE NumPy is a highly popular Python library for numerical computation. You will see it pop up frequently in your machine learning journey. It is rarely used to implement modern machine learning algorithms, due to lacking GPU and *autodifferentiation* support, but NumPy arrays are often used as a numerical data exchange format—like here, for MNIST digits and their labels.

Let's look at the training data:

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

And here's the test data:

```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

The workflow will be as follows. First, we'll feed the neural network the training data, `train_images` and `train_labels`. The network will then learn to associate images and

labels. Finally, we'll ask the network to produce predictions for `test_images`, and we'll verify whether these predictions match the labels from `test_labels`.

Let's build the network—again, remember that you aren't expected to understand everything about this example yet.

Listing 2.2 The network architecture

```
import keras
from keras import layers

model = keras.Sequential(
    [
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)
```

The core building block of neural networks is the *layer*. You can think of a layer as a filter for data: some data goes in, and it comes out in a more useful form. Specifically, layers extract *representations* out of the data fed into them—hopefully, representations that are more meaningful for the problem at hand. Most of deep learning consists of chaining together simple layers that will implement a form of progressive *data distillation*. A deep learning model is like a sieve for data processing, made of a succession of increasingly refined data filters—the layers.

Here, our model consists of a sequence of two `Dense` layers, which are densely connected (also called *fully connected*) neural layers. The second (and last) layer is a 10-way `softmax` *classification* layer, which means it will return an array of 10 probability scores (summing to 1). Each score will be the probability that the current digit image belongs to one of our 10 digit classes.

To make the model ready for training, we need to pick three more things, as part of the *compilation* step:

- *A loss function*—How the model will be able to measure its performance on the training data and thus how it will be able to steer itself in the right direction.
- *An optimizer*—The mechanism through which the model will update itself based on the training data it sees, to improve its performance.
- *Metrics to monitor during training and testing*—Here, we'll only care about accuracy (the fraction of the images that were correctly classified).

The exact purpose of the loss function and the optimizer will be made clear throughout the next two chapters.

Listing 2.3 The compilation step

```
model.compile(
    optimizer="adam",
```

```
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

Before training, we'll *preprocess* the data by reshaping it into the shape the model expects and scaling it so that all values are in the `[0, 1]` interval. Previously, our training images were stored in an array of shape `(60000, 28, 28)` of type `uint8` with values in the `[0, 255]` interval. We transform it into a `float32` array of shape `(60000, 28 * 28)` with values between `0` and `1`.

Listing 2.4 Preparing the image data

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
```

We're now ready to train the model, which in Keras is done via a call to the model's `fit()` method—we *fit* the model to its training data.

Listing 2.5 "Fitting" the model

```
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

Two quantities are displayed during training: the loss of the model over the training data and the accuracy of the model over the training data. We quickly reach an accuracy of 0.989 (98.9%) on the training data.

Now that we have a trained model, we can use it to predict class probabilities for *new* digits—images that weren't part of the training data, like those from the test set.

Listing 2.6 Using the model to make predictions

```
>>> test_digits = test_images[0:10]
>>> predictions = model.predict(test_digits)
>>> predictions[0]
array([1.0726176e-10, 1.6918376e-10, 6.1314843e-08, 8.4106023e-06,
       2.9967067e-11, 3.0331331e-09, 8.3651971e-14, 9.9999106e-01,
       2.6657624e-08, 3.8127661e-07], dtype=float32)
```

Each number of index `i` in that array corresponds to the probability that digit image `test_digits[0]` belong to class `i`.

This first test digit has the highest probability score (0.99999106, almost 1) at index 7, so according to our model, it must be a 7:

```
>>> predictions[0].argmax()
7
>>> predictions[0][7]
0.99999106
```

We can check that the test label agrees:

```
>>> test_labels[0]
7
```

On average, how good is our model at classifying such never-before-seen digits? Let's check by computing average accuracy over the entire test set.

Listing 2.7 Evaluating the model on new data

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print(f"test_acc: {test_acc}")
test_acc: 0.9785
```

The test set accuracy turns out to be 97.8%—that's almost double the error rate of the training set (at 98.9% accuracy). This gap between training accuracy and test accuracy is an example of *overfitting*: the fact that machine learning models tend to perform worse on new data than on their training data. Overfitting is a central topic in chapter 5.

This concludes our first example. You just saw how you can build and train a neural network to classify handwritten digits in less than 15 lines of Python code. In this chapter and the next, we'll go into detail about every moving piece we just previewed and clarify what's going on behind the scenes. You'll learn about tensors, the data-storing objects going into the model; tensor operations, which layers are made of; and gradient descent, which allows your model to learn from its training examples.

2.2 Data representations for neural networks

In the previous example, we started from data stored in multidimensional NumPy arrays, also called *tensors*. In general, all current machine learning systems use tensors as their basic data structure. Tensors are fundamental to the field—so fundamental that the TensorFlow framework was named after them. So what's a tensor?

At its core, a tensor is a container for data—usually numerical data. So it's a container for numbers. You may already be familiar with matrices, which are rank-2 tensors: tensors are a generalization of matrices to an arbitrary number of dimensions (note that in the context of tensors, a dimension is often called an *axis*).

Going over the details of tensors might seem a bit abstract at first. But it's well worth it—manipulating tensors will be the bread and butter of any machine learning code you ever write.

2.2.1 Scalars (rank-0 tensors)

A tensor that contains only one number is called a *scalar* (or scalar tensor, rank-0 tensor, or 0D tensor). In NumPy, a `float32` or `float64` number is a scalar tensor (or scalar array). You can display the number of axes of a NumPy tensor via the `ndim` attribute; a scalar tensor has 0 axes (`ndim == 0`). The number of axes of a tensor is also called its *rank*. Here's a NumPy scalar:

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

2.2.2 Vectors (rank-1 tensors)

An array of numbers is called a vector (or rank-1 tensor or 1D tensor). A rank-1 tensor has exactly one axis. The following is a NumPy vector:

```
>>> x = np.array([12, 3, 6, 14, 7])
>>> x
array([12, 3, 6, 14, 7])
>>> x.ndim
1
```

This vector has five entries and so is called a *5-dimensional vector*. Don't confuse a 5D vector with a 5D tensor! A 5D vector has only one axis and has five dimensions along its axis, whereas a 5D tensor has five axes (and may have any number of dimensions along each axis). *Dimensionality* can denote either the number of entries along a specific axis (as in the case of our 5D vector) or the number of axes in a tensor (such as a 5D tensor), which can be confusing at times. In the latter case, it's technically more correct to talk about a *tensor of rank 5* (the rank of a tensor being the number of axes), but the ambiguous notation *5D tensor* is common regardless.

2.2.3 Matrices (rank-2 tensors)

An array of vectors is a *matrix* (or rank-2 tensor or 2D tensor). A matrix has two axes (often referred to as *rows* and *columns*). You can visually interpret a matrix as a rectangular grid of numbers. This is a NumPy matrix:

```
>>> x = np.array([[5, 78, 2, 34, 0],
...               [6, 79, 3, 35, 1],
...               [7, 80, 4, 36, 2]])
```

```
>>> x.ndim
2
```

The entries from the first axis are called the *rows*, and the entries from the second axis are called the *columns*. In the previous example, `[5, 78, 2, 34, 0]` is the first row of `x`, and `[5, 6, 7]` is the first column.

2.2.4 Rank-3 tensors and higher-rank tensors

If you pack such matrices in a new array, you obtain a rank-3 tensor (or 3D tensor), which you can visually interpret as a cube of numbers. The following is a NumPy rank-3 tensor:

```
>>> x = np.array([[[5, 78, 2, 34, 0],
...                [6, 79, 3, 35, 1],
...                [7, 80, 4, 36, 2]],
...               [[5, 78, 2, 34, 0],
...                [6, 79, 3, 35, 1],
...                [7, 80, 4, 36, 2]],
...               [[5, 78, 2, 34, 0],
...                [6, 79, 3, 35, 1],
...                [7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

By packing rank-3 tensors in an array, you can create a rank-4 tensor, and so on. In deep learning, you'll generally manipulate tensors with ranks 0 to 4, although you may go up to 5 if you process video data.

2.2.5 Key attributes

A tensor is defined by three key attributes:

- *Number of axes (rank)*—For instance, a rank-3 tensor has three axes, and a matrix has two axes. This is also called the tensor's `ndim` in Python libraries such as NumPy, JAX, TensorFlow, and PyTorch.
- *Shape*—This is a tuple of integers that describes how many dimensions the tensor has along each axis. For instance, the previous matrix example has shape `(3, 5)`, and the rank-3 tensor example has shape `(3, 3, 5)`. A vector has a shape with a single element, such as `(5,)`, whereas a scalar has an empty shape, `()`.
- *Data type (usually called `dtype` in Python libraries)*—This is the type of the data contained in the tensor; for instance, a tensor's type could be `float16`, `float32`, `float64`, `uint8`, `bool`, and so on. In TensorFlow, you are also likely to come across `string` tensors.

To make this more concrete, let's look back at the data we processed in the MNIST example. First, we load the MNIST dataset:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Next, we display the number of axes of the tensor `train_images`, the `ndim` attribute:

```
>>> train_images.ndim
3
```

Here's its shape:

```
>>> train_images.shape
(60000, 28, 28)
```

And this is its data type, the `dtype` attribute:

```
>>> train_images.dtype
uint8
```

So what we have here is a rank-3 tensor of 8-bit integers. More precisely, it's an array of 60,000 matrices of 28 × 28 integers. Each such matrix is a grayscale image, with coefficients between 0 and 255.

Let's display the fourth digit in this rank-3 tensor, using the library Matplotlib (part of the standard scientific Python suite); see figure 2.2.

Listing 2.8 Displaying the fourth digit

```
import matplotlib.pyplot as plt

digit = train_images[4]
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

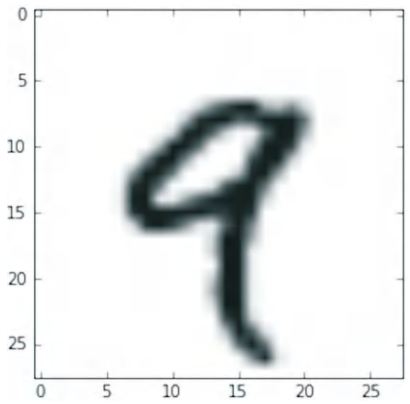


Figure 2.2 The fourth sample in our dataset

Naturally, the corresponding label is just the integer 9:

```
>>> train_labels[4]
9
```

2.2.6 Manipulating tensors in NumPy

In the previous example, we selected a specific digit alongside the first axis using the syntax `train_images[i]`. Selecting specific elements in a tensor is called *tensor slicing*. Let's look at the tensor-slicing operations you can do on NumPy arrays.

The following example selects digits #10 to #100 (#100 isn't included) and puts them in an array of shape `(90, 28, 28)`:

```
>>> my_slice = train_images[10:100]
>>> my_slice.shape
(90, 28, 28)
```

It's equivalent to this more detailed notation, which specifies a start index and stop index for the slice along each tensor axis. Note that `:` is equivalent to selecting the entire axis:

```
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

Equivalent to the previous example

Also equivalent to the previous example

In general, you may select slices between any two indices along each tensor axis. For instance, to select 14×14 pixels in the bottom-right corner of all images, you would do this:

```
my_slice = train_images[:, 14:, 14:]
```

It's also possible to use negative indices. Much like negative indices in Python lists, they indicate a position relative to the end of the current axis. To crop the images to patches of 14×14 pixels centered in the middle, do this:

```
my_slice = train_images[:, 7:-7, 7:-7]
```

2.2.7 The notion of data batches

In general, the first axis (axis 0, because indexing starts at 0) in all data tensors you'll come across in deep learning will be the *samples axis*. In the MNIST example, "samples" are images of digits.

In addition, deep learning models don't process an entire dataset at once; rather, they break the data into small "batches," or groups of samples with a fixed size. Concretely, here's one batch of our MNIST digits, with a batch size of 128:

```
batch = train_images[:128]
```

And here's the next batch:

```
batch = train_images[128:256]
```

And the nth batch:

```
n = 3
batch = train_images[128 * n : 128 * (n + 1)]
```

When considering such a batch tensor, the first axis (axis 0) is called the *batch axis* (or *batch dimension*). You'll frequently encounter this term when using Keras and other deep learning libraries.

2.2.8 Real-world examples of data tensors

Let's make data tensors more concrete with a few examples similar to what you'll encounter later. The data you'll manipulate will almost always fall into one of the following categories:

- *Vector data*—Rank-2 tensors of shape `(samples, features)`, where each sample is a vector of numerical attributes ("features")
- *Timeseries data or sequence data*—Rank-3 tensors of shape `(samples, timesteps, features)`, where each sample is a sequence (of length `timesteps`) of feature vectors
- *Images*—Rank-4 tensors of shape `(samples, height, width, channels)`, where each sample is a 2D grid of pixels, and each pixel is represented by a vector of values ("channels")
- *Video*—Rank-5 tensors of shape `(samples, frames, height, width, channels)`, where each sample is a sequence (of length `frames`) of images

VECTOR DATA

Vector data is one of the most common cases. In such a dataset, each single data point can be encoded as a vector, and thus a batch of data will be encoded as a rank-2 tensor (that is, an array of vectors), where the first axis is the *samples axis* and the second axis is the *features axis*.

Let's take a look at two examples:

- An actuarial dataset of people, where we consider each person's age, gender, and income. Each person can be characterized as a vector of three values, and thus an entire dataset of 100,000 people can be stored in a rank-2 tensor of shape `(100000, 3)`.

- A dataset of text documents, where we represent each document by the counts of how many times each word appears in it (out of a dictionary of 20,000 common words). Each document can be encoded as a vector of 20,000 values (one count per word in the dictionary), and thus an entire dataset of 500 documents can be stored in a tensor of shape `(500, 20000)`.

TIMESERIES DATA OR SEQUENCE DATA

Whenever time matters in your data (or the notion of sequence order), it makes sense to store it in a rank-3 tensor with an explicit time axis. Each sample can be encoded as a sequence of vectors (a rank-2 tensor), and thus a batch of data will be encoded as a rank-3 tensor (see figure 2.3).

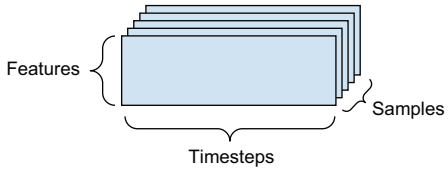


Figure 2.3 A rank-3 timeseries data tensor

The time axis is always the second axis (axis of index 1), by convention. Let's look at a few examples:

- *A dataset of stock prices*—Every minute, we store the current price of the stock, the highest price in the past minute, and the lowest price in the past minute. Thus every minute is encoded as a 3D vector, an entire day of trading is encoded as a matrix of shape `(390, 3)` (there are 390 minutes in a trading day), and 250 days' worth of data can be stored in a rank-3 tensor of shape `(250, 390, 3)`. Here, each sample would be one day's worth of data.
- *A dataset of tweets, where we encode each tweet as a sequence of 280 characters out of an alphabet of 128 unique characters*—In this setting, each character can be encoded as a binary vector of size 128 (an all-zeros vector except for a 1 entry at the index corresponding to the character). Then each tweet can be encoded as a rank-2 tensor of shape `(280, 128)`, and a dataset of 1 million tweets can be stored in a tensor of shape `(1000000, 280, 128)`.

IMAGE DATA

Images typically have three dimensions: height, width, and color depth. Although grayscale images (like our MNIST digits) have only a single color channel and could thus be stored in rank-2 tensors, by convention image tensors are always rank-3, with a one-dimensional color channel for grayscale images. A batch of 128 grayscale images of size 256 × 256 could thus be stored in a tensor of shape `(128, 256, 256, 1)`, and a batch of 128 color images could be stored in a tensor of shape `(128, 256, 256, 3)` (see figure 2.4).

There are two conventions for the shapes of image tensors: the *channels-last* convention (which is standard in JAX and TensorFlow, as well as most other deep learning tools out there) and the *channels-first* convention (which is standard in PyTorch).

The channels-last convention places the color-depth axis at the end: `(samples, height, width, color_depth)`. Meanwhile, the channels-first convention places the color depth axis right after the batch axis: `(samples, color_depth, height, width)`.

With the channels-first convention, the previous examples would become `(128, 1, 256, 256)` and `(128, 3, 256, 256)`. The Keras API provides support for both formats.

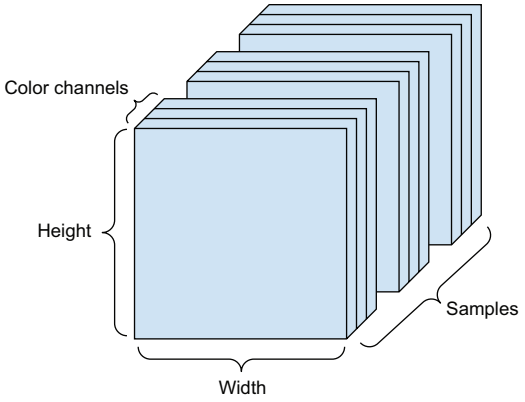


Figure 2.4 A rank-4 image data tensor

VIDEO DATA

Video data is one of the few types of real-world data for which you'll need rank-5 tensors. A video can be understood as a sequence of frames, each frame being a color image. Because each frame can be stored in a rank-3 tensor `(height, width, color_depth)`, a sequence of frames can be stored in a rank-4 tensor `(frames, height, width, color_depth)`, and thus a batch of different videos can be stored in a rank-5 tensor of shape `(samples, frames, height, width, color_depth)`.

For instance, a 60-second, 144 × 256 YouTube video clip sampled at 4 frames per second would have 240 frames. A batch of four such video clips would be stored in a tensor of shape `(4, 240, 144, 256, 3)`. That's a total of 106,168,320 values! If the `dtype` of the tensor was `float32`, then each value would be stored in 32 bits, so the tensor would represent 425 MB. Heavy! Videos you encounter in real life are much lighter because they aren't stored in `float32`, and they're typically compressed by a large factor (such as the MPEG format).

2.3 The gears of neural networks: Tensor operations

Just like any computer program can be ultimately reduced to a small set of binary operations on binary inputs (`AND`, `OR`, `NOR`, and so on), all transformations learned by deep neural networks can be reduced to a handful of *tensor operations* (or *tensor functions*) applied to tensors of numeric data. For instance, it's possible to add tensors, multiply tensors, and so on.

In our initial example, we were building our model by stacking `Dense` layers on top of each other. A Keras layer instance looks like this:

```
keras.layers.Dense(512, activation="relu")
```

This layer can be interpreted as a function, which takes as input a matrix and returns another matrix—a new representation for the input tensor. Specifically, the function is as follows (where `W` is a matrix and `b` is a vector, both attributes of the layer):

```
output = relu(matmul(input, W) + b)
```

Let's unpack this. We have three tensor operations here:

- A tensor product (`matmul`) between the input tensor and a tensor named `W`.
- An addition (`+`) between the resulting matrix and a vector `b`.
- A `relu` operation: `relu(x)` is `max(x, 0)`. `"relu"` stands for "REctified Linear Unit."

NOTE Although this section deals entirely with linear algebra expressions, you won't find any mathematical notation in this book. I've found that mathematical concepts can be more readily mastered by programmers with no mathematical background if they're expressed as short Python snippets instead of mathematical equations. So we'll use NumPy code throughout.

2.3.1 Element-wise operations

The `relu` operation and addition are element-wise operations: operations that are applied independently to each entry in the tensors being considered. This means these operations are highly amenable to massively parallel implementations (*vectorized* implementations, a term that comes from the *vector processor* supercomputer architecture from the 1970–1990 period). If you want to write a naive Python implementation of an element-wise operation, you use a `for` loop, as in this naive implementation of an element-wise `relu` operation:

```
def naive_relu(x):
    assert len(x.shape) == 2
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

x is a rank-2 NumPy tensor.

Avoids overwriting the input tensor

You could do the same for addition:

```
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

x and y are rank-2 NumPy tensors.

Avoids overwriting the input tensor

On the same principle, you can do element-wise multiplication, subtraction, and so on.

In practice, when dealing with NumPy arrays, these operations are available as well-optimized built-in NumPy functions, which themselves delegate the heavy lifting to a Basic Linear Algebra Subprograms (BLAS) implementation. BLAS are low-level,

highly parallel, efficient tensor-manipulation routines that are typically implemented in Fortran or C.

So, in NumPy, you can do the following element-wise operation, and it will be blazing fast:

```
import numpy as np

z = x + y                          <- Element-wise addition
z = np.maximum(z, 0.0)             <- Element-wise relu
```

Let's actually time the difference:

```
import time

x = np.random.random((20, 100))
y = np.random.random((20, 100))

t0 = time.time()
for _ in range(1000):
    z = x + y
    z = np.maximum(z, 0.0)
print("Took: {0:.2f} s".format(time.time() - t0))
```

This takes 0.02 seconds. Meanwhile, the naive version takes a stunning 2.45 seconds:

```
t0 = time.time()
for _ in range(1000):
    z = naive_add(x, y)
    z = naive_relu(z)
print("Took: {0:.2f} s".format(time.time() - t0))
```

Likewise, when running JAX/TensorFlow/PyTorch code on a GPU, element-wise operations are executed via fully vectorized CUDA implementations that can best utilize the highly parallel GPU chip architecture.

2.3.2 Broadcasting

Our earlier naive implementation of `naive_add` only supports the addition of rank-2 tensors with identical shapes. But in the `Dense` layer introduced earlier, we added a rank-2 tensor with a vector. What happens with addition when the shapes of the two tensors being added differ?

When possible, and if there's no ambiguity, the smaller tensor will be *broadcast* to match the shape of the larger tensor. Broadcasting consists of two steps:

- Axes (called *broadcast axes*) are added to the smaller tensor to match the `ndim` of the larger tensor.

- The smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor.

Let's look at a concrete example. Consider `X` with shape `(32, 10)` and `y` with shape `(10,)`:

```
import numpy as np

X = np.random.random((32, 10))
y = np.random.random((10,))
```

X is a random matrix with shape (32, 10).

y is a random vector with shape (10,).

First, we add an empty first axis to `y`, whose shape becomes `(1, 10)`:

```
y = np.expand_dims(y, axis=0)
```

The shape of y is now (1, 10).

Then, we repeat `y` 32 times alongside this new axis, so that we end up with a tensor `Y` with shape `(32, 10)`, where `Y[i, :] == y` for `i` in `range(0, 32)`:

```
Y = np.tile(y, (32, 1))
```

Repeat y 32 times along axis 0 to obtain Y with shape (32, 10).

At this point, we can add `X` and `Y` because they have the same shape.

In terms of implementation, no new rank-2 tensor is created because that would be terribly inefficient. The repetition operation is entirely virtual: it happens at the algorithmic level rather than at the memory level. But thinking of the vector being repeated 32 times alongside a new axis is a helpful mental model. Here's what a naive implementation would look like:

```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

x is a rank-2 NumPy tensor.

y is a NumPy vector.

Avoids overwriting the input tensor

With broadcasting, you can generally apply two-tensor element-wise operations if one tensor has shape `(a, b, … n, n + 1, … m)` and the other has shape `(n, n + 1, … m)`. The broadcasting will then automatically happen for axes `a` through `n - 1`.

The following example applies the element-wise `maximum` operation to two tensors of different shapes via broadcasting:

```
import numpy as np

x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))
z = np.maximum(x, y)
```

x is a random tensor with shape (64, 3, 32, 10).

y is a random tensor with shape (32, 10).

The output z has shape (64, 3, 32, 10) like x.

2.3.3 Tensor product

The *tensor product*, also called *dot product* or *matmul* (short for "matrix multiplication") is one of the most common, most useful tensor operations.

In NumPy, a tensor product is done using the `np.matmul` function, and in Keras, with the `keras.ops.matmul` function. Its shorthand is the `@` operator in Python:

```
x = np.random.random((32,))
y = np.random.random((32,))

z = np.matmul(x, y)
z = x @ y
```

Takes the product between x and y

This is equivalent.

In mathematical notation, you'd note the operation with a dot (•) (hence the name "dot product"):

```
z = x • y
```

Mathematically, what does the `matmul` operation do? Let's start with the product of two vectors `x` and `y`. It's computed as follows:

```
def naive_vector_product(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 0.0
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

x and y are NumPy vectors.

You'll have noticed that the product between two vectors is a scalar and that only vectors with the same number of elements are compatible for this operation.

You can also take the product between a matrix `x` and a vector `y`, which returns a vector where the coefficients are the products between `y` and the rows of `x`. You implement it as follows:

```
def naive_matrix_vector_product(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z
```

x is a NumPy matrix.

y is a NumPy vector.

The 1st dimension of x must equal the 0th dimension of y!

This operation returns a vector of 0s with as many rows as x.

You could also reuse the code we wrote previously, which highlights the relationship between a matrix-vector product and a vector product:

```
def naive_matrix_vector_product(x, y):
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        z[i] = naive_vector_product(x[i, :], y)
    return z
```

Note that as soon as one of the two tensors has an `ndim` greater than 1, `matmul` is no longer *symmetric*, which is to say that `matmul(x, y)` isn't the same as `matmul(y, x)`.

Of course, a tensor product generalizes to tensors with an arbitrary number of axes. The most common applications may be the product between two matrices. You can take the product of two matrices `x` and `y` (`matmul(x, y)`) if and only if `x.shape[1] == y.shape[0]`. The result is a matrix with shape `(x.shape[0], y.shape[1])`, where the coefficients are the vector products between the rows of `x` and the columns of `y`. Here's the naive implementation:

```
def naive_matrix_product(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]
    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
```

x and y are NumPy matrices.

The 1st dimension of x must equal the 0th dimension of y!

This operation returns a matrix of 0s with a specific shape.

Iterates over the rows of x …

… and over the columns of y.

```
            z[i, j] = naive_vector_product(row_x, column_y)
    return z
```

To understand vector product shape compatibility, it helps to visualize the input and output tensors by aligning them as shown in figure 2.5.

`x`, `y`, and `z` are pictured as rectangles (literal boxes of coefficients). Because the rows of `x` and the columns of `y` must have the same size, it follows that the width of `x` must match the height of `y`. If you go on to develop new machine learning algorithms, you'll likely be drawing such diagrams often.

More generally, you can take the product between higher-dimensional tensors, following the same rules for shape compatibility as outlined earlier for the 2D case:

```
(a, b, c, d) • (d,) -> (a, b, c)
(a, b, c, d) • (d, e) -> (a, b, c, e)
```

And so on.

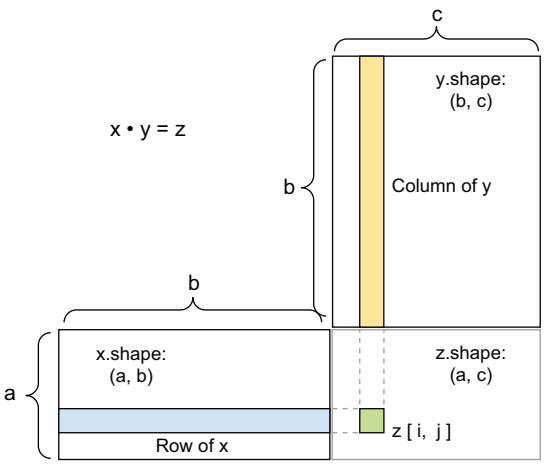


Figure 2.5 Matrix product box diagram

2.3.4 Tensor reshaping

A third type of tensor operation that's essential to understand is *tensor reshaping*. Although it wasn't used in the `Dense` layers in our first neural network example, we used it when we preprocessed the digits data before feeding it into our model:

```
train_images = train_images.reshape((60000, 28 * 28))
```

Reshaping a tensor means rearranging its rows and columns to match a target shape. Naturally, the reshaped tensor has the same total number of coefficients as the initial tensor. Reshaping is best understood via simple examples:

```
>>> x = np.array([[0., 1.],
...               [2., 3.],
...               [4., 5.]])
>>> x.shape
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
>>> x = x.reshape((2, 3))
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
```

A special case of reshaping that's commonly encountered is *transposition*. *Transposing* a matrix means exchanging its rows and its columns, so that `x[i, :]` becomes `x[:, i]`:

```
>>> x = np.zeros((300, 20))
>>> x = np.transpose(x)
>>> x.shape
(20, 300)
```

Creates an all-zeros matrix of shape (300, 20)

2.3.5 Geometric interpretation of tensor operations

Because the contents of the tensors manipulated by tensor operations can be interpreted as coordinates of points in some geometric space, all tensor operations have a geometric interpretation. For instance, let's consider addition. We'll start with the following vector:

```
A = [0.5, 1]
```

It's a point in a 2D space (see figure 2.6). It's common to picture a vector as an arrow linking the origin to the point, as shown in figure 2.7.

Let's consider a new point, `B = [1, 0.25]`, which we'll add to the previous one. This is done geometrically by chaining together the vector arrows, with the resulting location being the vector representing the sum of the previous two vectors (see figure 2.8). As you can see, adding a vector `B` to a vector `A` represents the action of copying point A in

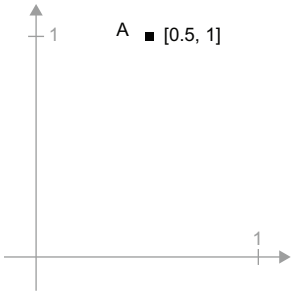


Figure 2.6 A point in a 2D space

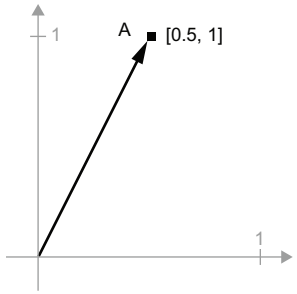


Figure 2.7 A point in a 2D space pictured as an arrow

new location, whose distance and direction from the original point A is determined by the vector `B`. If you apply the same vector addition to a group of points in the plane (an "object"), you would be creating a copy of the entire object in a new location (see figure 2.9). Tensor addition thus represents the action of *translating an object* (moving the object without distorting it) by a certain amount in a certain direction.

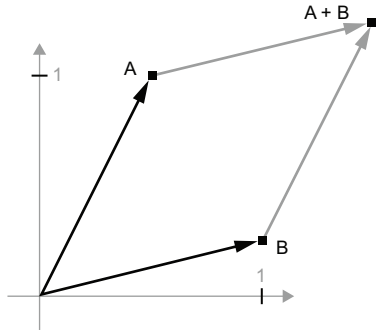


Figure 2.8 Geometric interpretation of the sum of two vectors

In general, elementary geometric operations, such as translation, rotation, scaling, skewing, and so on, can be expressed as tensor operations. Here are a few examples:

- *Translation*—As you just saw, adding a vector to a point will move this point by a fixed amount in a fixed direction. Applied to a set of points (such as a 2D object), this is called a "translation" (see figure 2.9).

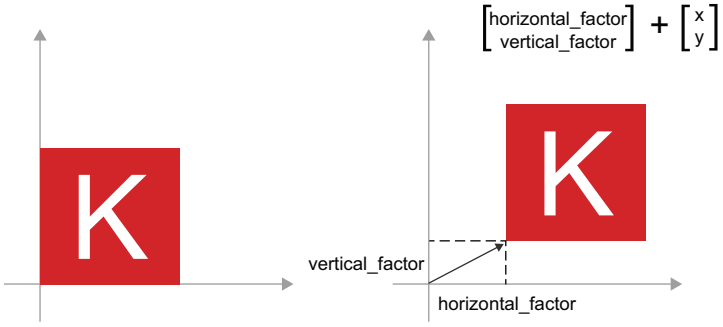


Figure 2.9 2D translation as a vector addition

- *Rotation*—A counterclockwise rotation of a 2D vector by an angle theta (see figure 2.10) can be achieved via a product with a 2 × 2 matrix `R = [[cos(theta), -sin(theta)], [sin(theta), cos(theta)]]`.

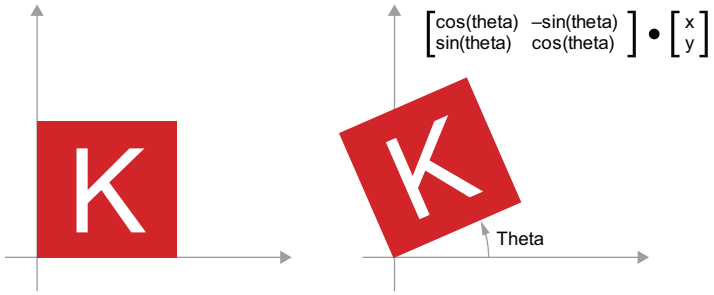


Figure 2.10 2D rotation (counterclockwise) as a matrix product

- *Scaling*—A vertical and horizontal scaling of the image (see figure 2.11) can be achieved via a product with a 2 × 2 matrix `S =` (note that such a matrix is called a "diagonal matrix" because it only has non-zero coefficients in its "diagonal," going from the top left to the bottom right).

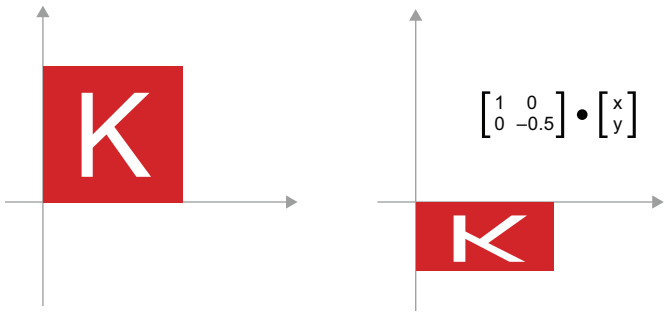


Figure 2.11 2D scaling as a matrix product

- *Linear transform*—A product with an arbitrary matrix implements a linear transform. Note that *scaling* and *rotation*, seen previously, are, by definition, linear transforms.
- *Affine transform*—An affine transform (see figure 2.12) is the combination of a linear transform (achieved via a matrix product) and a translation (achieved via a vector addition). As you have probably recognized, that's exactly the `y = W @ x + b` computation implemented by the `Dense` layer! A `Dense` layer without an activation function is an affine layer.

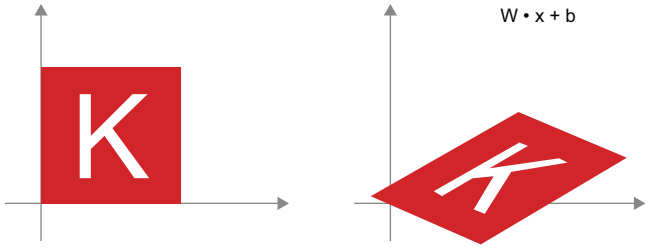


Figure 2.12 Affine transform in the plane

- *`Dense` layer with `relu` activation*—An important observation about affine transforms is that if you apply many of them repeatedly, you still end up with an affine transform (so you could just have applied that one affine transform in the first place). Let's try it with two: `affine2(affine1(x)) = W2 @ (W1 @ x + b1) + b2 = (W2 @ W1) @ x + (W2 @ b1 + b2)`. That's an affine transform where the linear part is the matrix `W2 @ W1` and the translation part is the vector `W2 @ b1 + b2`. As a consequence, a multilayer neural network made entirely of `Dense` layers without activations would be equivalent to a single `Dense` layer. This "deep" neural network would just be a linear model in disguise! This is why we need activation functions, like `relu` (seen in action in figure 2.13). Thanks to activation functions, a chain of `Dense` layers can be made to implement very complex, nonlinear geometric transformation, resulting in very rich hypothesis spaces for your deep neural networks. We cover this idea in more detail in the next chapter.

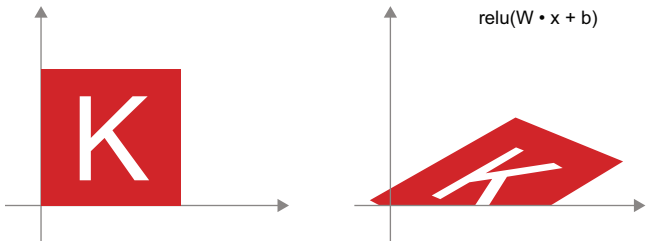


Figure 2.13 Affine transform followed by `relu` activation

2.3.6 A geometric interpretation of deep learning

You just learned that neural networks consist entirely of chains of tensor operations and that all of these tensor operations are just simple geometric transformations of the input data. It follows that you can interpret a neural network as a very complex geometric transformation in a high-dimensional space, implemented via a series of simple steps.

In 3D, the following mental image may prove useful. Imagine two sheets of colored paper: one red and one blue. Put one on top of the other. Now crumple them together into a small ball. That crumpled paper ball is your input data, and each sheet of paper is a class of data in a classification problem. What a neural network is meant to do is figure

out a transformation of the paper ball that would uncrumple it to make the two classes cleanly separable again (see figure 2.14). With deep learning, this would be implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time.

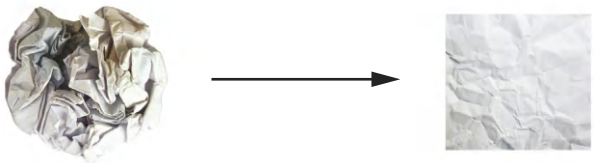


Figure 2.14 Uncrumpling a complicated manifold of data

Uncrumpling paper balls is what machine learning is about: finding neat representations for complex, highly folded data *manifolds* in high-dimensional spaces (a manifold is a continuous surface, like our crumpled sheet of paper). At this point, you should have a pretty good intuition as to why deep learning excels at this: it takes the approach of incrementally decomposing a complicated geometric transformation into a long chain of elementary ones, which is pretty much the strategy a human would follow to uncrumple a paper ball. Each layer in a deep network applies a transformation that disentangles the data a little—and a deep stack of layers makes tractable an extremely complicated disentanglement process.

2.4 The engine of neural networks: Gradient-based optimization

As you saw in the previous section, each neural layer from our first model example transforms its input data as follows:

```
output = relu(matmul(input, W) + b)
```

In this expression, `W` and `b` are tensors that are attributes of the layer. They're called the *weights* or *trainable parameters* of the layer (the `kernel` and `bias` attributes, respectively). These weights contain the information learned by the model from exposure to training data.

Initially, these weight matrices are filled with small random values (a step called *random initialization*). Of course, there's no reason to expect that `relu(matmul(input, W) + b)`, when `W` and `b` are random, will yield any useful representations. The resulting representations are meaningless—but they're a starting point. What comes next is to gradually adjust these weights, based on a feedback signal. This gradual adjustment, also called *training*, is basically the learning that machine learning is all about.

This happens within what's called a *training loop*, which works as follows. Repeat these steps in a loop, until the loss seems sufficiently low:

1. Draw a batch of training samples `x` and corresponding targets `y_true`.
2. Run the model on `x` (a step called the *forward pass*) to obtain predictions `y_pred`.
3. Compute the loss of the model on the batch, a measure of the mismatch between `y_pred` and `y_true`.
4. Update all weights of the model in a way that slightly reduces the loss on this batch.

You'll eventually end up with a model that has a very low loss on its training data: a low mismatch between predictions `y_pred` and expected targets `y_true`. The model has "learned" to map its inputs to correct targets. From afar, it may look like magic, but when you reduce it to elementary steps, it turns out to be simple.

Step 1 sounds easy enough—it's just I/O code. Steps 2 and 3 are merely the application of a handful of tensor operations, so you could implement these steps purely from what you learned in the previous section. The difficult part is step 4: updating the model's weights. Given an individual weight coefficient in the model, how can you compute whether the coefficient should be increased or decreased, and by how much?

One naive solution would be to freeze all weights in the model except the one scalar coefficient being considered and try different values for this coefficient. Let's say the initial value of the coefficient is 0.3. After the forward pass on a batch of data, the loss of the model on the batch is 0.5. If you change the coefficient's value to 0.35 and rerun the forward pass, the loss increases to 0.6. But if you lower the coefficient to 0.25, the loss falls to 0.4. In this case, it seems that updating the coefficient by –0.05 would contribute to minimizing the loss. This would have to be repeated for all coefficients in the model.

But such an approach would be horribly inefficient because you'd need to compute two forward passes (which are expensive) for every individual coefficient (of which there are many, usually at least a few thousands and potentially up to billions). Thankfully, there's a much better approach: *gradient descent*.

Gradient descent is the optimization technique that powers modern neural networks. Here's the gist of it. All of the functions used in our models (such as `matmul` or `+`) transform their input in a smooth and continuous way: if you look at `z = x + y`, for instance, a small change in `y` only results in a small change in `z`, and if you know the direction of the change in `y`, you can infer the direction of the change in `z`. Mathematically, you'd say these functions are *differentiable*. If you chain together such functions, the bigger function you obtain is still differentiable. In particular, this applies to the function that maps the model's coefficients to the loss of the model on a batch of data: a small change of the model's coefficients results in a small, predictable change of the loss value. This enables you to use a mathematical operator called the *gradient* to describe how the loss varies as you move the model's coefficients in different directions. If you compute this gradient, you can use it to move the coefficients (all at once in a single update, rather than one at a time) in a direction that decreases the loss.

If you already know what *differentiable* means and what a *gradient* is, you can skip the next two sections. Otherwise, the following will help you understand these concepts.

2.4.1 What's a derivative?

Consider a continuous, smooth function `f(x) = y`, mapping a number `x` to a new number `y`. We can use the function in figure 2.15 as an example.

Because the function is *continuous*, a small change in `x` can only result in a small change in `y`—that's the intuition behind *continuity*. Let's say you increase `x` by a small factor `epsilon_x`: this results in a small `epsilon_y` change to `y`, as shown in figure 2.16.

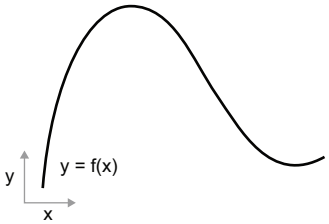


Figure 2.15 A continuous, smooth function

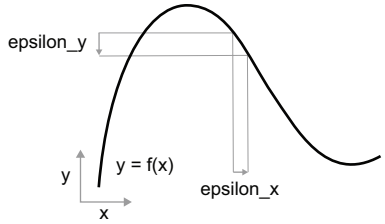


Figure 2.16 With a continuous function, a small change in `x` results in a small change in `y`.

In addition, because the function is *smooth* (its curve doesn't have any abrupt angles), when `epsilon_x` is small enough, around a certain point `p`, it's possible to approximate `f` as a linear function of slope `a`, so that `epsilon_y` becomes `a * epsilon_x`:

```
f(x + epsilon_x) = y + a * epsilon_x
```

Obviously, this linear approximation is valid only when `x` is close enough to `p`.

The slope `a` is called the *derivative* of `f` in `p`. If `a` is negative, it means a small increase in `x` around `p` will result in a decrease of `f(x)`, as shown in figure 2.17, and if `a` is positive, a small increase in `x` will result in an increase of `f(x)`. Further, the absolute value of `a` (the *magnitude* of the derivative) tells you how quickly this increase or decrease will happen.

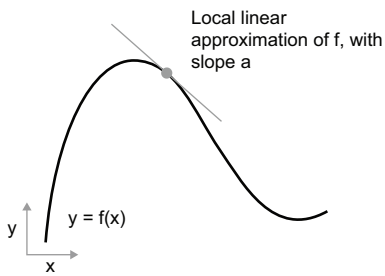


Figure 2.17 Derivative of `f` in `p`

For every differentiable function `f(x)` (*differentiable* means "can be derived": for example, smooth, continuous functions can be derived), there exists a derivative function `f'(x)` that maps values of `x` to the slope of the local linear approximation of `f` in those points. For instance, the derivative of `cos(x) is -sin(x)`, the derivative of `f(x) = a * x` is `f'(x) = a`, and so on.

Being able to derive functions is a very powerful tool when it comes to *optimization*, the task of finding values of `x` that minimize the value of `f(x)`. If you're trying to update `x` by a factor `epsilon_x` to minimize `f(x)` and you know the derivative of `f`, then your job is done: the derivative completely describes how `f(x)` evolves as you change `x`. If you want to reduce the value of `f(x)`, you just need to move `x` a little in the opposite direction from the derivative.

2.4.2 Derivative of a tensor operation: The gradient

The function we were just looking at turned a scalar value `x` into another scalar value `y`: you could plot it as a curve in a 2D plane. Now, imagine a function that turns a tuple of scalars `(x, y)` into a scalar value `z`: that would be a vector operation. You could plot it as a 2D *surface* in a 3D space (indexed by coordinates `x, y, z`). Likewise, you can imagine functions that take as input matrices, functions that take as input rank-3 tensors, etc.

The concept of derivation can be applied to any such function, as long as the surfaces they describe are continuous and smooth. The derivative of a tensor operation (or tensor function) is called a *gradient*. Gradients are just the generalization of the concept of derivatives to functions that take tensors as inputs. Remember how, for a scalar function, the derivative represents the *local slope* of the curve of the function? In just the same way, the gradient of a tensor function represents the *curvature* of the multidimensional surface described by the function. It characterizes how the output of the function varies when its input parameters vary.

Let's look at an example grounded in machine learning. Consider

- An input vector `x` (a sample in a dataset)
- A matrix `W` (the weights of a model)
- A target `y_true` (what the model should learn to associated to `x`)
- A loss function `loss` (meant to measure the gap between the model's current predictions and `y_true`).

You can use `W` to compute a target candidate `y_pred` and then compute the loss, or mismatch, between the target candidate `y_pred` and the target `y_true`:

```
y_pred = matmul(x, W)
loss_value = loss(y_pred, y_true)
```

We use the model weights W to make a prediction for x.

We estimate how far off the prediction was.

Now, we'd like to use gradients to figure out how to update `W` to make `loss_value` smaller. How do we do that?

Given fixed inputs `x` and `y_true`, the previous operations can be interpreted as a function mapping values of `W` (the model's weights) to loss values:

```
loss_value = f(W)
```
← f describes the curve (or high-dimensional surface) formed by loss values when W varies.

Let's say the current value of `W` is `W0`. Then the derivative of `f` in the point `W0` is a tensor `grad(loss_value, W0)`, with the same shape as `W`, where each coefficient `grad(loss_value, W0)[i, j]` indicates the direction and magnitude of the change in `loss_value` you observe when modifying `W0[i, j]`. That tensor `grad(loss_value, W0)` is the gradient of the function `f(W) = loss_value` in `W0`, also called "gradient of `loss_value` with respect to `W` around `W0`."

> **NOTE** The tensor operation `grad(f(W), W)` (which takes as input a matrix `W`) can be expressed as a combination of scalar functions `grad_ij(f(W), w_ij)`, each of which would return the derivative of `loss_value = f(W)` with respect to the coefficient `W[i, j]` of `W`, assuming all other coefficients are constant. `grad_ij` is called the *partial derivative* of `f` with respect to `W[i, j]`.

Concretely, what does `grad(loss_value, W0)` represent? You saw earlier that the derivative of a function `f(x)` of a single coefficient can be interpreted as the slope of the curve of `f`. Likewise, `grad(loss_value, W0)` can be interpreted as the tensor describing the *curvature* of `loss_value = f(W)` around `W0`. Each partial derivative describes the curvature of `f` in a specific direction.

We just saw how for a function `f(x)`, you can reduce the value of `f(x)` by moving `x` a little in the opposite direction from the derivative. In much the same way, with a function `f(W)` of a tensor, you can reduce `loss_value = f(W)` by moving `W` in the opposite direction from the gradient, such as an update of `W1 = W0 - step * grad(f(W0), W0)` where `step` is a small scaling factor. That means going against the curvature, which intuitively should put you lower on the curve. Note that the scaling factor `step` is needed because `grad(loss_value, W0)` only approximates the curvature when you're close to `W0`, so you don't want to get too far from `W0`.

2.4.3 Stochastic gradient descent

Given a differentiable function, it's theoretically possible to find its minimum analytically: it's known that a function's minimum is a point where the derivative is 0, so all you have to do is find all the points where the derivative goes to 0 and check for which of these points the function has the lowest value.

Applied to a neural network, that means finding analytically the combination of weight values that yields the smallest possible loss function. This can be done by solving the equation `grad(f(W), W) = 0` for `W`. This is a polynomial equation of `N` variables, where `N` is the number of coefficients in the model. Although it would be possible to solve such an equation for `N = 2` or `N = 3`, doing so is intractable for real neural networks, where the number of parameters is never less than a few thousand and can sometimes be in the billions.

Instead, you can use the four-step algorithm outlined at the beginning of this section: modify the parameters little by little based on the current loss value on a random batch of data. Because you're dealing with a differentiable function, you can compute its gradient, which gives you an efficient way to implement step 4. If you update the weights in the opposite direction from the gradient, the loss will be a little less every time:

1. Draw a batch of training samples `x` and corresponding targets `y_true`.
2. Run the model on `x` to obtain predictions `y_pred` (this is called the *forward pass*).
3. Compute the loss of the model on the batch, a measure of the mismatch between `y_pred` and `y_true`.
4. Compute the gradient of the loss with regard to the model's parameters (this is called the *backward pass*).
5. Move the parameters a little in the opposite direction from the gradient—for example, `W -= learning_rate * gradient`—thus reducing the loss on the batch a bit. The *learning rate* (`learning_rate` here) would be a scalar factor modulating the "speed" of the gradient descent process.

Easy enough! What we just described is called *mini-batch stochastic gradient descent* (mini-batch SGD). The term *stochastic* refers to the fact that each batch of data is drawn at random (*stochastic* is a scientific synonym of *random*). Figure 2.18 illustrates what happens in 1D, when the model has only one parameter and you have only one training sample.

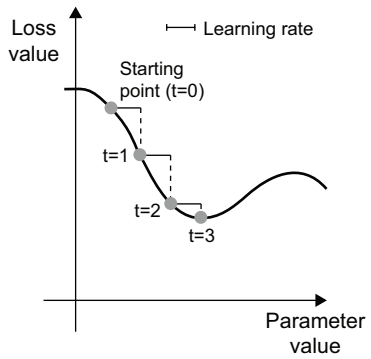


Figure 2.18 SGD down a 1D loss curve (one learnable parameter)

We can see intuitively that it's important to pick a reasonable value for the `learning_rate` factor. If it's too small, the descent down the curve will take many iterations, and it could get stuck in a local minimum. If `learning_rate` is too large, your updates may end up taking you to completely random locations on the curve.

Note that a variant of the mini-batch SGD algorithm would be to draw a single sample and target at each iteration, rather than drawing a batch of data. This would be *true* SGD (as opposed to *mini-batch* SGD). Alternatively, going to the opposite extreme, you could run every step on *all* data available, which is called *batch gradient descent*. Each update would then be more accurate, but far more expensive. The efficient compromise between these two extremes is to use mini-batches of reasonable size.

Although figure 2.18 illustrates gradient descent in a 1D parameter space, in practice, you'll use gradient descent in highly dimensional spaces: every weight coefficient in a neural network is a free dimension in the space, and there may be tens of thousands or even millions of them. To help you build intuition about loss surfaces, you can also visualize gradient descent along a 2D loss surface, as shown in figure 2.19. But

you can't possibly visualize what the actual process of training a neural network looks like—you can't represent a 1,000,000-dimensional space in a way that makes sense to humans. As such, it's good to keep in mind that the intuitions you develop through these low-dimensional representations may not always be accurate in practice. This has historically been a source of issues in the world of deep learning research.

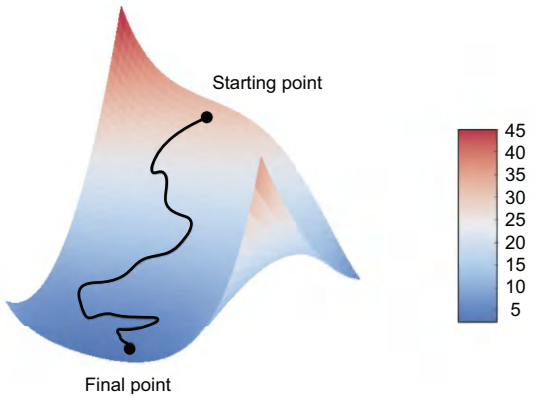


Figure 2.19 Gradient descent down a 2D loss surface (two learnable parameters)

Additionally, there exist multiple variants of SGD that differ by taking into account previous weight updates when computing the next weight update, rather than just looking at the current value of the gradients. There is, for instance, SGD with momentum, as well as Adagrad, RMSprop, and several others. Such variants are known as *optimization methods* or *optimizers*. In particular, the concept of *momentum*, which is used in many of these variants, deserves your attention. Momentum addresses two issues with SGD: convergence speed and local minima. Consider figure 2.20, which shows the curve of a loss as a function of a model parameter.

As you can see, around a certain parameter value, there is a *local minimum*: around that point, moving left would result in the loss increasing, but so would moving right. If the parameter under consideration were being optimized via SGD with a small learning rate, then the optimization process would get stuck at the local minimum instead of making its way to the global minimum.

You can avoid such issues by using momentum, which draws inspiration from physics. A useful mental image here is to think of the optimization process as a small ball rolling down the loss curve. If it has enough

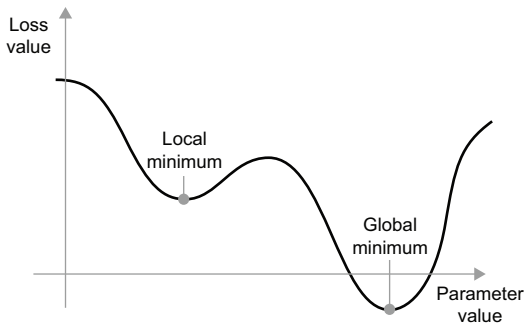


Figure 2.20 A local minimum and a global minimum

momentum, the ball won't get stuck in a ravine and will end up at the global minimum. Momentum is implemented by moving the ball at each step based not only on the current slope value (current acceleration) but also on the current velocity (resulting from past acceleration). In practice, this means updating the parameter `w` based not only on the current gradient value but also on the previous parameter update, such as in this naive implementation:

```
past_velocity = 0.0
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum - learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

Constant momentum factor

Optimization loop

2.4.4 Chaining derivatives: The Backpropagation algorithm

In the previously discussed algorithm, we casually assumed that because a function is differentiable, we can easily compute its gradient. But is that true? How can we compute the gradient of complex expressions in practice? In our two-layer network example, how can we get the gradient of the loss with regard to the weights? That's where the *Backpropagation algorithm* comes in.

THE CHAIN RULE

Backpropagation is a way to use the derivative of simple operations (such as addition, `relu`, or tensor product) to easily compute the gradient of arbitrarily complex combinations of these atomic operations. Crucially, a neural network consists of many tensor operations chained together, each of which has a simple, known derivative. For instance, the model from our first example can be expressed as a function parameterized by the variables `W1`, `b1`, `W2`, and `b2` (belonging to the first and second `Dense` layers, respectively), involving the atomic operations `matmul`, `relu`, `softmax`, and `+`, as well as our loss function, `loss`, which are all easily differentiable:

```
loss_value = loss(
    y_true,
    softmax(matmul(relu(matmul(inputs, W1) + b1), W2) + b2),
)
```

Calculus tells us that such a chain of functions can be derived using the following identity, called the *chain rule*. Consider two functions `f` and `g`, as well as the composed function `fg` such that `y = fg(x) == f(g(x))`:

```
def fg(x):
    x1 = g(x)
    y = f(x1)
    return y
```

Then the chain rule states that `grad(y, x) == grad(y, x1) * grad(x1, x)`. This enables you to compute the derivative of `fg` as long as you know the derivatives of `f` and `g`. The chain rule is named like this because when you add more intermediate functions, it starts looking like a chain:

```
def fghj(x):
    x1 = j(x)
    x2 = h(x1)
    x3 = g(x2)
    y = f(x3)
    return y

grad(y, x) == grad(y, x3) * grad(x3, x2) * grad(x2, x1) * grad(x1, x)
```

Applying the chain rule to the computation of the gradient values of a neural network gives rise to an algorithm called *backpropagation*. Let's see how that works, concretely.

Automatic differentiation with computation graphs

A useful way to think about backpropagation is in terms of *computation graphs*. A computation graph is the data structure at the heart of the deep learning revolution. It's a directed acyclic graph of operations—in our case, tensor operations. For instance, figure 2.21 is the graph representation of our first model.

Computation graphs have been an extremely successful abstraction in computer science because they enable us to *treat computation as data*: a computable expression is encoded as a machine-readable data structure that can be used as the input or output of another program. For instance, you could imagine a program that receives a computation graph and returns a new computation graph that implements a large-scale distributed version of the same computation—this would mean that you could distribute any computation without having to write the distribution logic yourself. Or imagine … a program that receives a computation graph and can automatically generate the derivative of the expression it represents. It's much easier to do these things if your computation is expressed as an explicit graph data structure rather than, say, lines of ASCII characters in a `.py` file.

To explain backpropagation clearly, let's look at a really basic example of a computation graph. We'll consider a simplified version of the graph in figure 2.21, where we only have one linear layer and where all variables are scalar, shown in figure 2.22. We'll take two scalar variables `w`, `b`, a scalar input `x`, and apply some operations to them to combine into an output `y`. Finally, we'll apply an absolute value error loss function: `loss_val = abs(y_true - y)`. Since we want to update `w` and `b` in a way that would minimize `loss_val`, we are interested in computing `grad(loss_val, b)` and `grad(loss_val, w)`.

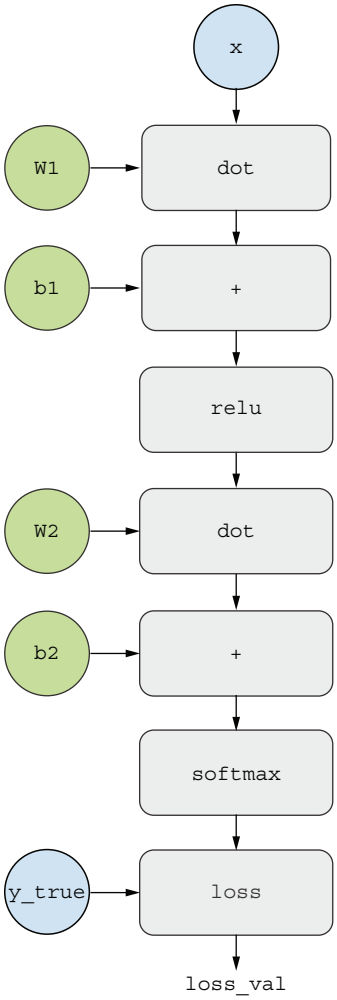


Figure 2.21 The computation graph representation of our two-layer model

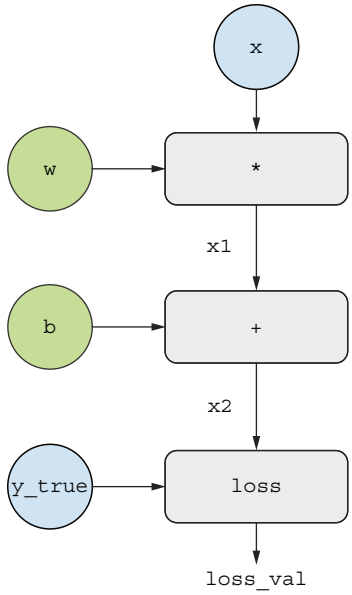


Figure 2.22 A basic example of a computation graph

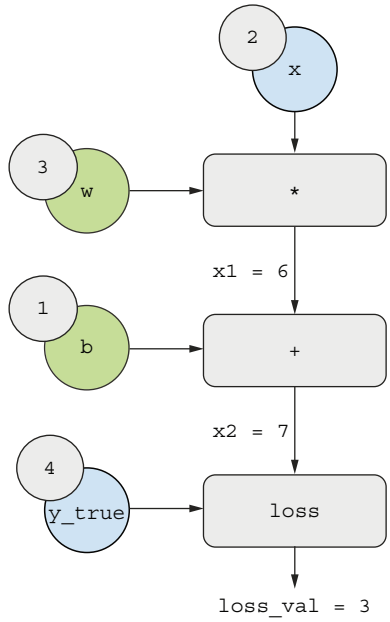


Figure 2.23 Running a forward pass

Let's set concrete values for the "input nodes" in the graph—that is, the input `x`, the target `y_true`, `w` and `b` (figure 2.23). We propagate these values to all nodes in the graph, from top to bottom, until we reach `loss_val`. This is the *forward pass*.

Now let's "reverse" the graph: for each edge in the graph going from `A` to `B`, we will create an opposite edge from `B` to `A`, and ask, "How much does `B` vary when `A` varies?" That is, what is

`grad(B, A)`? We'll annotate each inverted edge with this value (figure 2.24). This backward graph represents the *backward pass*.

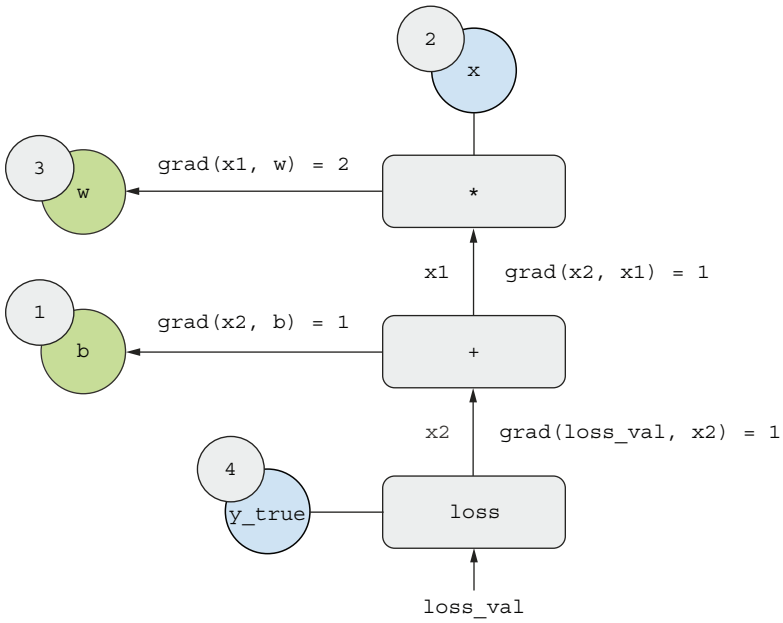


Figure 2.24 Running a backward pass

We have

- `grad(loss_val, x2) = 1` because as `x2` varies by an amount epsilon, `loss_val = abs(4 - x2)` varies by the same amount.
- `grad(x2, x1) = 1` because as `x1` varies by an amount epsilon, `x2 = x1 + b = x1 + 1` varies by the same amount.
- `grad(x2, b) = 1` because as `b` varies by an amount epsilon, `x2 = x1 + b = 6 + b` varies by the same amount.
- `grad(x1, w) = 2` because as `w` varies by an amount epsilon, `x1 = x * w = 2 * w` varies by `2 * epsilon`.

What the chain rule says about this backward graph is that you can obtain the derivative of a node with respect to another node by *multiplying the derivatives for each edge along the path linking the two nodes*. For instance, `grad(loss_val, w) = grad(loss_val, x2) * grad(x2, x1) * grad(x1, w)` (see figure 2.25).

By applying the chain rule to our graph, we obtain what we were looking for:

- `grad(loss_val, w) = 1 * 1 * 2 = 2`
- `grad(loss_val, b) = 1 * 1 = 1`

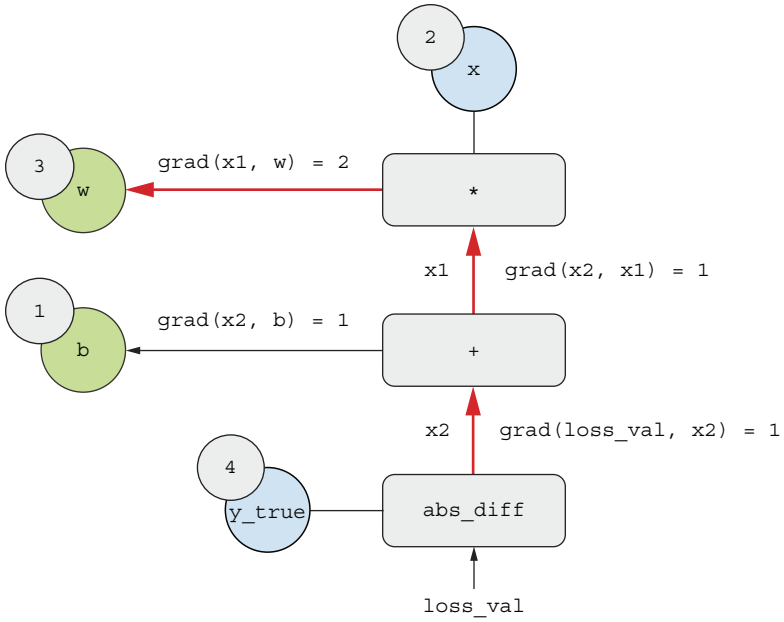


Figure 2.25 Path from `loss_val` to `w` in the backward graph

> **NOTE** If there are multiple paths linking the two nodes of interest `a`, `b` in the backward graph, we would obtain `grad(b, a)` by summing the contributions of all the paths.

And with that, you just saw backpropagation in action! Backpropagation is simply the application of the chain rule to a computation graph. There's nothing more to it. Backpropagation starts with the final loss value and works backward from the top layers to the bottom layers, computing the contribution that each parameter had in the loss value. That's where the name "backpropagation" comes from: we "back propagate" the loss contributions of different nodes in a computation graph.

Nowadays, people implement neural networks in modern frameworks that are capable of *automatic differentiation*, such as JAX, TensorFlow, and PyTorch. Automatic differentiation is implemented with the kind of computation graph previously presented. Automatic differentiation makes it possible to retrieve the gradients of arbitrary compositions of differentiable tensor operations without doing any extra work besides writing down the forward pass. When I wrote my first neural networks in C in the 2000s, I had to write my gradients by hand. Now, thanks to modern automatic differentiation tools, you'll never have to implement backpropagation yourself. Consider yourself lucky!

2.5 Looking back at our first example

You're nearing the end of this chapter, and you should now have a general understanding of what's going on behind the scenes in a neural network. What was a magical black box at the start of the chapter has turned into a clearer picture, as illustrated in figure 2.26: the model, composed of layers that are chained together, maps the input data to predictions. The loss function then compares these predictions to the targets, producing a loss value: a measure of how well the model's predictions match what was expected. The optimizer uses this loss value to update the model's weights.

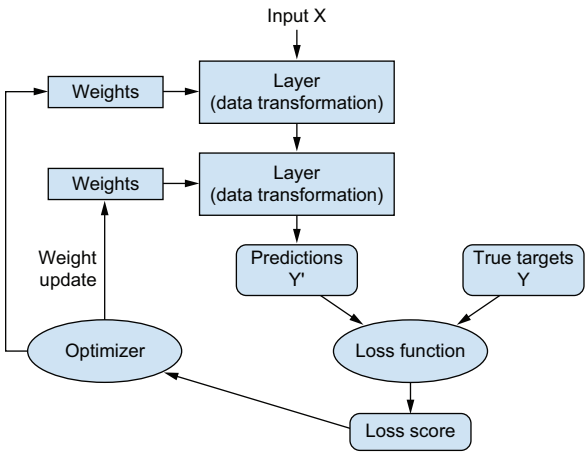


Figure 2.26 Relationship between the network, layers, loss function, and optimizer

Let's go back to the first example and review each piece of it in the light of what you've learned in the previous sections.

This was the input data:

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
```

Now you understand that the input images are stored in NumPy tensors, which are here formatted as `float32` tensors of shape `(60000, 784)` (training data) and `(10000, 784)` (test data), respectively.

This was our model:

```
model = keras.Sequential(
    [
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)
```

Now you understand that this model consists of a chain of two `Dense` layers, that each layer applies a few simple tensor operations to the input data, and that these operations involve weight tensors. Weight tensors, which are attributes of the layers, are where the *knowledge* of the model persists.

This was the model-compilation step:

```
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

Now you understand that `"sparse_categorical_crossentropy"` is the loss function that's used as a feedback signal for learning the weight tensors, which the training phase will attempt to minimize. You also know that this reduction of the loss happens via mini-batch stochastic gradient descent. The exact rules governing a specific use of gradient descent are defined by the `"adam"` optimizer passed as the first argument.

Finally, this was the training loop:

```
model.fit(
    train_images,
    train_labels,
    epochs=5,
    batch_size=128,
)
```

Now you understand what happens when you call `fit`: the model will start to iterate on the training data in mini-batches of 128 samples, 5 times over (each iteration over all the training data is called an *epoch*). For each batch, the model will compute the gradient of the loss with regard to the weights (using the Backpropagation algorithm, which derives from the chain rule in calculus) and move the weights in the direction that will reduce the value of the loss for this batch.

After these 5 epochs, the model will have performed 2,345 gradient updates (469 per epoch), and the loss of the model will be sufficiently low that the model will be capable of classifying handwritten digits with high accuracy.

At this point, you already know most of what there is to know about neural networks. Let's prove it by reimplementing a simplified version of that first example step by step, using only low-level operations.

2.5.1 Reimplementing our first example from scratch

What's better to demonstrate full, unambiguous understanding than to implement everything from scratch? Of course, what "from scratch" means here is relative: we won't reimplement basic tensor operations, and we won't implement backpropagation. But we'll go to such a low level that each computation step will be made explicit.

Don't worry if you don't understand every little detail in this example just yet. The next chapter will dive in more detail into the Keras API. For now, just try to follow the gist of what's going on—the intent of this example is to help crystallize your understanding of the mathematics of deep learning using a concrete implementation. Let's go!

A SIMPLE DENSE CLASS

You've learned earlier that the `Dense` layer implements the following input transformation, where `W` and `b` are model parameters, and `activation` is an element-wise function (usually `relu`):

```
output = activation(matmul(input, W) + b)
```

Let's implement a simple Python class `NaiveDense` that creates two Keras variables `W` and `b`, and exposes a `__call__()` method that applies the previous transformation:

```
import keras
from keras import ops

class NaiveDense:
    def __init__(self, input_size, output_size, activation=None):
        self.activation = activation
        self.W = keras.Variable(
            shape=(input_size, output_size), initializer="uniform"
        )
        self.b = keras.Variable(shape=(output_size,), initializer="zeros")

    def __call__(self, inputs):
        x = ops.matmul(inputs, self.W)
        x = x + self.b
        if self.activation is not None:
            x = self.activation(x)
        return x
```

keras.ops is where you will find all the tensor operations you need.

Creates a matrix W of shape (input_size, output_size), initialized with random values drawn from a uniform distribution

Applies the forward pass

Creates a vector b of shape (output_size,), initialized with zeros

```
    @property
    def weights(self):
        return [self.W, self.b]
```

The convenience method for retrieving the layer's weights

A SIMPLE SEQUENTIAL CLASS

Now, let's create a `NaiveSequential` class to chain these layers. It wraps a list of layers and exposes a `__call__()` method that simply calls the underlying layers on the inputs, in order. It also features a `weights` property to easily keep track of the layers' parameters:

```
class NaiveSequential:
    def __init__(self, layers):
        self.layers = layers

    def __call__(self, inputs):
        x = inputs
        for layer in self.layers:
            x = layer(x)
        return x

    @property
    def weights(self):
        weights = []
        for layer in self.layers:
            weights += layer.weights
        return weights
```

Using this `NaiveDense` class and this `NaiveSequential` class, we can create a mock Keras model:

```
model = NaiveSequential(
    [
        NaiveDense(input_size=28 * 28, output_size=512, activation=ops.relu),
        NaiveDense(input_size=512, output_size=10, activation=ops.softmax),
    ]
)
assert len(model.weights) == 4
```

A BATCH GENERATOR

Next, we need a way to iterate over the MNIST data in mini-batches. This is easy:

```
import math

class BatchGenerator:
    def __init__(self, images, labels, batch_size=128):
```

```
        assert len(images) == len(labels)
        self.index = 0
        self.images = images
        self.labels = labels
        self.batch_size = batch_size
        self.num_batches = math.ceil(len(images) / batch_size)

    def next(self):
        images = self.images[self.index : self.index + self.batch_size]
        labels = self.labels[self.index : self.index + self.batch_size]
        self.index += self.batch_size
        return images, labels
```

2.5.2 Running one training step

The most difficult part of the process is the "training step": updating the weights of the model after running it on one batch of data. We need to

- Compute the predictions of the model for the images in the batch
- Compute the loss value for these predictions given the actual labels
- Compute the gradient of the loss with regard to the model's weights
- Move the weights by a small amount in the direction opposite to the gradient

Listing 2.9 A single step of training

```
def one_training_step(model, images_batch, labels_batch):
    predictions = model(images_batch)
    loss = ops.sparse_categorical_crossentropy(labels_batch, predictions)
    average_loss = ops.mean(loss)
    gradients = get_gradients_of_loss_wrt_weights(loss, model.weights)
    update_weights(gradients, model.weights)
    return loss
```

Runs the "forward pass"

Computes the gradient of the loss with regard to the weights. The output, gradients, is a list where each entry corresponds to a weight from the model. weights list. We haven't defined this function yet!

Updates the weights using the gradients. We haven't defined this function yet!

THE WEIGHT UPDATE STEP

As you already know, the purpose of the "weight update" step (represented by the `update_weights()` function) is to move the weights by "a bit" in a direction that will reduce the loss on this batch. The magnitude of the move is determined by the

"learning rate," typically a small quantity. The simplest way to implement this `update_weights()` function is to subtract `gradient * learning_rate` from each weight:

```
learning_rate = 1e-3

def update_weights(gradients, weights):
    for g, w in zip(gradients, weights):
        w.assign(w - g * learning_rate)
```

Assigns a new value to the variable, in place

In practice, you will almost never implement a weight update step like this by hand. Instead, you would use an `Optimizer` instance from Keras—like this:

```
from keras import optimizers

optimizer = optimizers.SGD(learning_rate=1e-3)

def update_weights(gradients, weights):
    optimizer.apply_gradients(zip(gradients, weights))
```

GRADIENT COMPUTATION

Now, there's just one thing we're still missing: gradient computation (represented by the `get_gradients_of_loss_wrt_weights()` function in listing 2.9). In the previous section, we outlined how we could use the chain rule to obtain the gradients of a chain of functions given their individual derivatives, a process known as backpropagation. We could reimplement backpropagation from scratch here, but that would be rather cumbersome, especially since we're using a `softmax` operation and a crossentropy loss, which have fairly verbose derivatives.

Instead, we can rely on the automatic differentiation mechanism that's built into one of the low-level frameworks supported by Keras, such as TensorFlow, JAX, or PyTorch. For the sake of the example, let's go with TensorFlow here. You'll learn more about TensorFlow, JAX, and PyTorch in the next chapter.

The API through which you can use TensorFlow's automatic differentiation capabilities is the `tf.GradientTape` object. It's a Python scope that will "record" the tensor operations that run inside it, in the form of a computation graph (sometimes called a *tape*). This graph can then be used to retrieve the gradient of any scalar value with respect to any set of input values:

```
import tensorflow as tf

x = tf.zeros(shape=())
```

Instantiates a scalar tensor with value 0

```
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

Opens a GradientTape scope

Inside the scope, applies some tensor operations to our variable

Uses the tape to retrieve the gradient of the output y with respect to our variable x

Let's rewrite our function `one_training_step()` using the TensorFlow `GradientTape` (skipping the need for a separate `get_gradients_of_loss_wrt_weights()` function):

```
def one_training_step(model, images_batch, labels_batch):
    with tf.GradientTape() as tape:
        predictions = model(images_batch)
        loss = ops.sparse_categorical_crossentropy(labels_batch, predictions)
        average_loss = ops.mean(loss)
    gradients = tape.gradient(average_loss, model.weights)
    update_weights(gradients, model.weights)
    return average_loss
```

Now that our per-batch training step is ready, we can move on to implementing an entire epoch of training.

2.5.3 The full training loop

An epoch of training simply consists of the repetition of the training step for each batch in the training data, and the full training loop is simply the repetition of one epoch:

```
def fit(model, images, labels, epochs, batch_size=128):
    for epoch_counter in range(epochs):
        print(f"Epoch {epoch_counter}")
        batch_generator = BatchGenerator(images, labels)
        for batch_counter in range(batch_generator.num_batches):
            images_batch, labels_batch = batch_generator.next()
            loss = one_training_step(model, images_batch, labels_batch)
            if batch_counter % 100 == 0:
                print(f"loss at batch {batch_counter}: {loss:.2f}")
```

Let's test-drive it:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28 * 28))
```

```
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255

fit(model, train_images, train_labels, epochs=10, batch_size=128)
```

2.5.4 Evaluating the model

We can evaluate the model by taking the `argmax` of its predictions over the test images, and comparing it to the expected labels:

```
>>> predictions = model(test_images)
>>> predicted_labels = ops.argmax(predictions, axis=1)
>>> matches = predicted_labels == test_labels
>>> f"accuracy: {ops.mean(matches):.2f}"
accuracy: 0.83
```

All done! As you can see, it's quite a bit of work to do "by hand" what you can do in a few lines of Keras code. But because you've gone through these steps, you should now have a crystal-clear understanding of what goes on inside a neural network when you call `fit()`. Having this low-level mental model of what your code is doing behind the scenes will make you better able to take advantage of the high-level features of the Keras API.

2.6 Summary

- *Tensors* form the foundation of modern machine learning systems. They come in various flavors of `dtype`, `rank`, and `shape`.
- You can manipulate numerical tensors via *tensor operations* (such as addition, tensor product, or element-wise multiplication), which can be interpreted as encoding geometric transformations. In general, everything in deep learning is amenable to a geometric interpretation.
- Deep learning models consist of chains of simple tensor operations, parameterized by *weights*, which are themselves tensors. The weights of a model are where its "knowledge" is stored.
- *Learning* means finding a set of values for the model's weights that minimizes a *loss function* for a given set of training data samples and their corresponding targets.
- Learning happens by drawing random batches of data samples and their targets and computing the gradient of the model parameters with respect to the loss on the batch. The model parameters are then moved a bit (the magnitude of the

move is defined by the learning rate) in the opposite direction from the gradient. This is called *mini-batch gradient descent*.

- The entire learning process is made possible by the fact that all tensor operations in neural networks are differentiable, and thus it's possible to apply the chain rule of derivation to find the gradient function mapping the current parameters and current batch of data to a gradient value. This is called *backpropagation*.
- Two key concepts you'll see frequently in future chapters are *loss* and *optimizers*. These are the two things you need to define before you begin feeding data into a model:
  - The *loss* is the quantity you'll attempt to minimize during training, so it should represent a measure of success for the task you're trying to solve.
  - The *optimizer* specifies the exact way in which the gradient of the loss will be used to update parameters: for instance, it could be the RMSProp optimizer, SGD with momentum, and so on.

3 Introduction to TensorFlow, PyTorch, JAX, and Keras

This chapter covers

- A closer look at all major deep learning frameworks and their relationships
- An overview of how core deep learning concepts translate to code across all frameworks

This chapter is meant to give you everything you need to start doing deep learning in practice. First, you'll get familiar with three popular deep learning frameworks that can be used with Keras:

- TensorFlow (https://tensorflow.org)
- PyTorch (https://pytorch.org/)
- JAX (https://jax.readthedocs.io/)

Then, building on top of the first contact you've had with Keras in chapter 2, we'll review the core components of neural networks and how they translate to Keras APIs.

By the end of this chapter, you'll be ready to move on to practical, real-world applications—which will start with chapter 4.

3.1 A brief history of deep learning frameworks

In the real world, you're not going to be writing low-level code from scratch like we did at the end of chapter 2. Instead, you're going to use a framework. Besides Keras, the main deep learning frameworks today are JAX, TensorFlow, and PyTorch. This book will teach you about all four.

If you're just getting started with deep learning, it may seem like all these frameworks have been here forever. In reality, they're all quite recent, with Keras being the oldest among the four (launched in March 2015). The ideas behind these frameworks, however, have a long history—the first paper about automatic differentiation was published in 1964.[1]

All these frameworks combine three key features:

- A way to compute gradients for arbitrary differentiable functions (automatic differentiation)
- A way to run tensor computations on CPUs and GPUs (and possibly even on other specialized deep learning hardware)
- A way to distribute computation across multiple devices or multiple computers, such as multiple GPUs on one computer, or even multiple GPUs across multiple separate computers

Together, these three simple features unlock all modern deep learning.

It took a long time for the field to develop robust solutions for all three problems and package those solutions in a reusable form. Since its inception in the 1960s and until the 2000s, autodifferentiation had no practical applications in machine learning—folks who worked with neural networks simply wrote their own gradient logic by hand, usually in a language like C++. Meanwhile, GPU programming was all but impossible.

Things started to slowly change in the late 2000s. First, Python and its ecosystem were slowly rising in popularity in the scientific community, gaining traction over MATLAB and C++. Second, NVIDIA released CUDA in 2006, unlocking the possibility of building neural networks that could run on consumer GPUs. The initial focus on CUDA was on physics simulation rather than machine learning, but that didn't stop machine learning researchers from starting to implement CUDA-based neural networks from 2009 onward. They were typically one-off implementations that ran on a single GPU without any autodifferentiation.

The first framework to enable autodifferentiation and GPU computation to train deep learning models was Theano, circa 2009. Theano is the conceptual ancestor of all modern deep learning tools. It started getting good traction in the machine learning research community in 2013–2014, after the results of the ImageNet 2012 competition ignited the world's interest in deep learning. Around the same time, a few other GPU-enabled deep learning libraries started gaining popularity in the computer vision

[1] R. E. Wengert, "A Simple Automatic Derivative Evaluation Program," Communications of the ACM, 7 no. 8 (1964).

world—in particular, Torch 7 (Lua-based) and Caffe (C++-based). Keras launched in early 2015 as a higher-level, easier-to-use deep learning library powered by Theano, and it quickly gained traction with the few thousands of people who were into deep learning at the time.

Then in late 2015, Google launched TensorFlow, which took many of the key ideas from Theano and added support for large-scale distributed computation. The release of TensorFlow was a watershed moment that precipitated deep learning in the mainstream developer zeitgeist. Keras immediately added support for TensorFlow. By mid-2016, over half of all TensorFlow users were using it through Keras.

In response to TensorFlow, Meta (named Facebook at the time) launched PyTorch about one year later, taking ideas from Chainer (a niche but innovative framework launched in mid-2015, now long dead) and NumPy-Autograd, a CPU-only autodifferentiation library for NumPy released by Maclaurin et al. in 2014. Meanwhile, Google released TPUs as an alternative to GPUs, alongside XLA, a high-performance compiler developed to enable TensorFlow to run on TPUs.

A few years later, at Google, Matthew Johnson—one of the developers who worked on NumPy-Autograd—released JAX as an alternative way to use autodifferentiation with XLA. JAX quickly gained traction with researchers thanks to its minimalistic API and high scalability. Today, Keras, TensorFlow, PyTorch, and JAX are the top frameworks in the deep learning world.

Looking back on this chaotic history, we can ask, What's next? Will a new framework arise tomorrow? Will we switch to a new programming language or a new hardware platform?

If you ask me, three things today are certain:

- Python has won. Its machine learning and data science ecosystem simply has too much momentum at this point. There won't be a brand new language to replace it—at least not in the next 15 years.
- We're in a multiframework world—all four frameworks are well established and are unlikely to go anywhere in the next few years. It's a good idea for you to learn a little bit about each one. However, it's highly possible that *new* frameworks will gain popularity in the future, in addition to them; Apple's recently released MLX could be one such example. In this context, using Keras is a considerable advantage: you should be able to run your existing Keras models on any new up-and-coming framework via a new Keras backend. Keras will keep providing future-proof stability to machine learning developers in the future, like it has since 2015—back when neither TensorFlow nor PyTorch nor JAX existed.
- New chips may certainly arise in the future, alongside NVIDIA's GPUs and Google's TPUs. For instance, AMD's GPU line likely has bright days ahead. But any new such chip will have to work with the existing frameworks to gain traction. New hardware is unlikely to disrupt your workflows.

3.2 How these frameworks relate to each other

Keras, TensorFlow, PyTorch, and JAX don't all have the same feature set and aren't interchangeable. They have some overlap, but to a large extent, they serve different roles for different use cases. The biggest difference is between Keras and the three others. Keras is a high-level framework, while the others are lower level. Imagine building a house. Keras is like a prefabricated building kit: it provides a streamlined interface for setting up and training neural networks. In contrast, TensorFlow, PyTorch, and JAX are like the raw materials used in construction.

As you saw in the previous chapters, training a neural network revolves around the following concepts:

- *First, low-level tensor manipulation*—The infrastructure that underlies all modern machine learning. This translates to low-level APIs found in TensorFlow, PyTorch,[2] and JAX:
  - *Tensors*, including special tensors that store the network's state (*variables*)
  - *Tensor operations* such as addition, `relu`, or `matmul`
  - *Backpropagation*, a way to compute the gradient of mathematical expressions
- *Second, high-level deep learning concepts*—This translates to Keras APIs:
  - *Layers*, which are combined into a *model*
  - A *loss function*, which defines the feedback signal used for learning
  - An *optimizer*, which determines how learning proceeds
  - *Metrics* to evaluate model performance, such as accuracy
  - A *training loop* that performs mini-batch stochastic gradient descent

Further, Keras is unique in that it isn't a fully standalone framework. It needs a *backend engine* to run, much like a prefabricated house-building kit needs to source building materials from somewhere. TensorFlow, PyTorch, and JAX can all be used as Keras backends. In addition, Keras can run on NumPy, but since NumPy does not provide an API for gradients, Keras workflows on NumPy are restricted to making predictions from a model—training is impossible.

Now that you have a clearer understanding of how all these frameworks came to be and how they relate to each other, let's dive into what it's like to work with them. We'll cover them in chronological order: TensorFlow first, then PyTorch, and finally JAX.

3.3 Introduction to TensorFlow

TensorFlow is a Python-based open source machine learning framework developed primarily by Google. Its initial release was in November 2015, followed by a v1 release in February 2017, and a v2 release in October 2019. TensorFlow is heavily used in production-grade machine learning applications across the industry.

[2] Note that PyTorch is a bit of an intermediate case: while it is mainly a lower-level framework, it also includes its own layers and its own optimizers. However, if you use PyTorch in conjunction with Keras, then you will only interact with low-level PyTorch APIs such as tensor operations.

It's important to keep in mind that TensorFlow is more than a single library. It's really a platform, home to a vast ecosystem of components, some developed by Google, some developed by third parties. For instance, there's TFX for industry-strength machine learning workflow management, TF-Serving for production deployment, the TF Optimization Toolkit for model quantization and pruning, and TFLite and MediaPipe for mobile application deployment.

Together, these components cover a very wide range of use cases, from cutting-edge research to large-scale production applications.

3.3.1 First steps with TensorFlow

Over the next paragraphs, you'll get familiar with all the basics of TensorFlow. We'll cover the following key concepts:

- Tensors and variables
- Numerical operations in TensorFlow
- Computing gradients with a `GradientTape`
- Making TensorFlow functions fast by using just-in-time compilation

We'll then conclude the introduction with an end-to-end example: a pure-TensorFlow implementation of linear regression.

Let's get those tensors flowing.

TENSORS AND VARIABLES IN TENSORFLOW

To do anything in TensorFlow, we're going to need some tensors. There are a few different ways you can create them.

CONSTANT TENSORS

Tensors need to be created with some initial value, so common ways to create tensors are via `tf.ones` (equivalent to `np.ones`) and `tf.zeros` (equivalent to `np.zeros`). You can also create a tensor from Python or NumPy values using `tf.constant`.

Listing 3.1 All-ones or all-zeros tensors

```
>>> import tensorflow as tf
>>> tf.ones(shape=(2, 1))
tf.Tensor([[1.], [1.]], shape=(2, 1), dtype=float32)
>>> tf.zeros(shape=(2, 1))
tf.Tensor([[0.], [0.]], shape=(2, 1), dtype=float32)
>>> tf.constant([1, 2, 3], dtype="float32")
tf.Tensor([1., 2., 3.], shape=(3,), dtype=float32)
```

Equivalent to np.ones(shape=(2, 1))

Equivalent to np.zeros(shape=(2, 1))

Equivalent to np.array([1, 2, 3], dtype="float32")

RANDOM TENSORS

You can also create tensors filled with random values via one of the methods of the `tf.random` submodule (equivalent to the `np.random` submodule).

Listing 3.2 Random tensors

```
>>> x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)
>>> print(x)
tf.Tensor(
[[-0.14208166]
 [-0.95319825]
 [ 1.1096532 ]], shape=(3, 1), dtype=float32)
>>> x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
>>> print(x)
tf.Tensor(
[[0.33779848]
 [0.06692922]
 [0.7749394 ]], shape=(3, 1), dtype=float32)
```

Tensor of random values drawn from a normal distribution with mean 0 and standard deviation 1. Equivalent to np.random.normal(size=(3, 1), loc=0., scale=1.).

Tensor of random values drawn from a uniform distribution between 0 and 1. Equivalent to np.random.uniform(size=(3, 1), low=0., high=1.).

Tensor assignment and the Variable class

A significant difference between NumPy arrays and TensorFlow tensors is that TensorFlow tensors aren't assignable: they're constant. For instance, in NumPy, you can do the following.

Listing 3.3 NumPy arrays are assignable

```
import numpy as np

x = np.ones(shape=(2, 2))
x[0, 0] = 0.0
```

Try to do the same thing in TensorFlow: you will get an error, `EagerTensor object does not support item assignment`.

Listing 3.4 TensorFlow tensors are not assignable

```
x = tf.ones(shape=(2, 2))
x[0, 0] = 0.0
```

This will fail, as a tensor isn't assignable.

To train a model, we'll need to update its state, which is a set of tensors. If tensors aren't assignable, how do we do it, then? That's where variables come in. `tf.Variable` is the class meant to manage modifiable state in TensorFlow.

To create a variable, you need to provide some initial value, such as a random tensor.

Listing 3.5 Creating a `tf.Variable`

```
>>> v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1)))
>>> print(v)
```

```
array([[-0.75133973],
       [-0.4872893 ],
       [ 1.6626885 ]], dtype=float32)>
```

The state of a variable can be modified via its `assign` method.

Listing 3.6 Assigning a value to a `Variable`

```
>>> v.assign(tf.ones((3, 1)))
array([[1.],
       [1.],
       [1.]], dtype=float32)>
```

Assignment also works for a subset of the coefficients.

Listing 3.7 Assigning a value to a subset of a `Variable`

```
>>> v[0, 0].assign(3.)
array([[3.],
       [1.],
       [1.]], dtype=float32)>
```

Similarly, `assign_add` and `assign_sub` are efficient equivalents of `+=` and `-=`.

Listing 3.8 Using `assign_add`

```
>>> v.assign_add(tf.ones((3, 1)))
array([[2.],
       [2.],
       [2.]], dtype=float32)>
```

TENSOR OPERATIONS: DOING MATH IN TENSORFLOW

Just like NumPy, TensorFlow offers a large collection of tensor operations to express mathematical formulas. Here are a few examples.

Listing 3.9 A few basic math operations in TensorFlow

```
a = tf.ones((2, 2))
b = tf.square(a)
c = tf.sqrt(a)
d = b + c
e = tf.matmul(a, b)
f = tf.concat((a, b), axis=0)
```

Takes the square, same as np.square

Takes the square root, same as np.sqrt

Adds two tensors (element-wise)

Takes the product of two tensors (see chapter 2), same as np.matmul

Concatenates a and b along axis 0, same as np.concatenate

Here's an equivalent of the `Dense` layer we saw in chapter 2:

```
def dense(inputs, W, b):
    return tf.nn.relu(tf.matmul(inputs, W) + b)
```

Gradients in TensorFlow: A second look at the GradientTape API

So far, TensorFlow seems to look a lot like NumPy. But here's something NumPy can't do: retrieve the gradient of any differentiable expression with respect to any of its inputs. Just open a `GradientTape` scope, apply some computation to one or several input tensors, and retrieve the gradient of the result with respect to the inputs.

Listing 3.10 Using the `GradientTape`

```
input_var = tf.Variable(initial_value=3.0)
with tf.GradientTape() as tape:
    result = tf.square(input_var)
gradient = tape.gradient(result, input_var)
```

This is most commonly used to retrieve the gradients of the loss of a model with respect to its weights: `gradients = tape.gradient(loss, weights)`.

In chapter 2, you saw how the `GradientTape` works on either a single input or a list of inputs and how inputs could be either scalars or high-dimensional tensors.

So far, you've only seen the case where the input tensors in `tape.gradient()` were TensorFlow variables. It's actually possible for these inputs to be any arbitrary tensor. However, only *trainable variables* are being tracked by default. With a constant tensor, you'd have to manually mark it as being tracked, by calling `tape.watch()` on it.

Listing 3.11 Using the `GradientTape` with constant tensor inputs

```
input_const = tf.constant(3.0)
with tf.GradientTape() as tape:
    tape.watch(input_const)
    result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

Why? Because it would be too expensive to preemptively store the information required to compute the gradient of anything with respect to anything. To avoid wasting resources, the tape needs to know what to watch. Trainable variables are watched by default because computing the gradient of a loss with regard to a list of trainable variables is the most common use case of the gradient tape.

The gradient tape is a powerful utility, even capable of computing *second-order gradients*—that is, the gradient of a gradient. For instance, the gradient of the position of

an object with regard to time is the speed of that object, and the second-order gradient is its acceleration.

If you measure the position of a falling apple along a vertical axis over time, and find that it verifies `position(time) = 4.9 * time ** 2`, what is its acceleration? Let's use two nested gradient tapes to find out.

Listing 3.12 Using nested gradient tapes to compute second-order gradients

```
time = tf.Variable(0.0)
with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as inner_tape:
        position = 4.9 * time**2
    speed = inner_tape.gradient(position, time)
acceleration = outer_tape.gradient(speed, time)
```

We use the outer tape to compute the gradient of the gradient from the inner tape. Naturally, the answer is 4.9 * 2 = 9.8.

MAKING TENSORFLOW FUNCTIONS FAST USING COMPILATION

All the TensorFlow code you've written so far has been executing "eagerly." This means operations are executed one after the other in the Python runtime, much like any Python code or NumPy code. Eager execution is great for debugging, but it is typically quite slow. It can often be beneficial to parallelize some computation, or "fuse" operations—replacing two consecutive operations, like `matmul` followed by `relu`, with a single, more efficient operation that does the same thing without materializing the intermediate output.

This can be achieved via *compilation*. The general idea of compilation is to take certain functions you've written in Python, lift them out of Python, automatically rewrite them into a faster and more efficient "compiled program," and then call that program from the Python runtime.

The main benefit of compilation is improved performance. There's a drawback too: the code you write is no longer the code that gets executed, which can make the debugging experience painful. Only turn on compilation after you've already debugged your code in the Python runtime.

You can apply compilation to any TensorFlow function by wrapping it in a `tf.function` decorator, like this:

```
@tf.function
def dense(inputs, W, b):
    return tf.nn.relu(tf.matmul(inputs, W) + b)
```

When you do this, any call to `dense()` is replaced with a call to a compiled program that implements a more optimized version of the function. The first call to the function will take a bit longer, because TensorFlow will be compiling your code. This only happens once—all subsequent calls to the same function will be fast.

TensorFlow has two compilation modes:

- First, the default one, which we refer to as "graph mode." Any function decorated with `@tf.function` runs in graph mode.
- Second, compilation with XLA, a high-performance compiler for ML (it's short for Accelerated Linear Algebra). You can turn it on by specifying `jit_compile=True`, like this:

```
@tf.function(jit_compile=True)
def dense(inputs, W, b):
    return tf.nn.relu(tf.matmul(inputs, W) + b)
```

It is often the case that compiling a function with XLA will make it run faster than graph mode—though it takes more time to execute the function the first time, since the compiler has more work to do.

3.3.2 An end-to-end example: A linear classifier in pure TensorFlow

You know about tensors, variables, and tensor operations, and you know how to compute gradients. That's enough to build any TensorFlow-based machine learning model based on gradient descent. Let's walk through an end-to-end example to make sure everything is crystal clear.

In a machine learning job interview, you may be asked to implement a linear classifier from scratch: a very simple task that serves as a filter between candidates who have some minimal machine learning background, and those who don't. Let's get you past that filter, and use your newfound knowledge of TensorFlow to implement such a linear classifier.

First, let's come up with some nicely linearly separable synthetic data to work with: two classes of points in a 2D plane.

Listing 3.13 Generating two classes of random points in a 2D plane

```
import numpy as np

num_samples_per_class = 1000
negative_samples = np.random.multivariate_normal(
    mean=[0, 3], cov=[[1, 0.5], [0.5, 1]], size=num_samples_per_class
)
positive_samples = np.random.multivariate_normal(
    mean=[3, 0], cov=[[1, 0.5], [0.5, 1]], size=num_samples_per_class
)
```

Generates the first class of points: 1,000 random 2D points with specified "mean" and "covariance matrix." Intuitively, the "covariance matrix" describes the shape of the point cloud, and the "mean" describes its position in the plane. cov=[[1, 0.5],[0.5, 1]] corresponds to "an oval-like point cloud oriented from bottom left to top right."

Generates the other class of points with a different mean and the same covariance matrix (point cloud with a different position and the same shape)

`negative_samples` and `positive_samples` are both arrays with shape `(1000, 2)`. Let's stack them into a single array with shape `(2000, 2)`.

Listing 3.14 Stacking the two classes into an array with shape (2000, 2)

```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
```

Let's generate the corresponding target labels, an array of 0s and 1s of shape `(2000, 1)`, where `targets[i, 0]` is 0 if `inputs[i]` belongs to class 0 (and inversely).

Listing 3.15 Generating the corresponding targets (0 and 1)

```
targets = np.vstack(
    (
        np.zeros((num_samples_per_class, 1), dtype="float32"),
        np.ones((num_samples_per_class, 1), dtype="float32"),
    )
)
```

Let's plot our data with Matplotlib, a well-known Python data visualization library (it comes preinstalled in Colab, so no need for you to install it yourself), as shown in figure 3.1.

Listing 3.16 Plotting the two point classes

```
import matplotlib.pyplot as plt

plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])
plt.show()
```

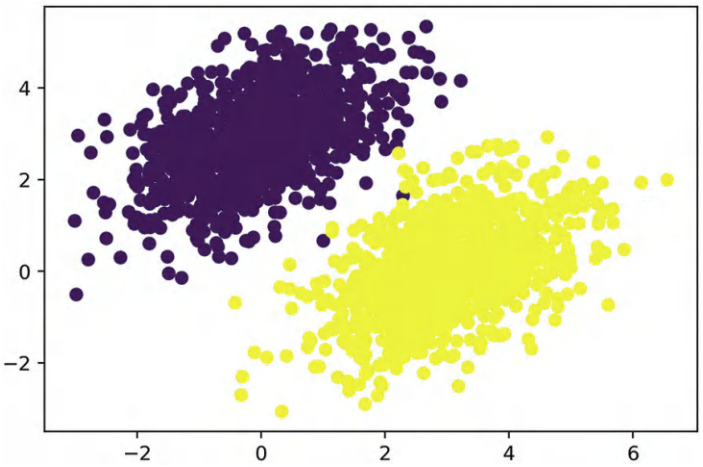


Figure 3.1 Our synthetic data: two classes of random points in the 2D plane

Now, let's create a linear classifier that can learn to separate these two blobs. A linear classifier is an affine transformation (`prediction = matmul(input, W) + b`) trained to minimize the square of the difference between predictions and the targets.

As you'll see, it's actually a much simpler example than the end-to-end example of a toy two-layer neural network from the end of chapter 2. However, this time, you should be able to understand everything about the code, line by line.

Let's create our variables `W` and `b`, initialized with random values and with zeros, respectively.

Listing 3.17 Creating the linear classifier variables

```
input_dim = 2
output_dim = 1
W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
```

The inputs will be 2D points.

The output predictions will be a single score per sample (close to 0 if the sample is predicted to be in class 0, and close to 1 if the sample is predicted to be in class 1).

Here's our forward pass function.

Listing 3.18 The forward pass function

```
def model(inputs, W, b):
    return tf.matmul(inputs, W) + b
```

Because our linear classifier operates on 2D inputs, `W` is really just two scalar coefficients: `W = [[w1], [w2]]`. Meanwhile, `b` is a single scalar coefficient. As such, for given input point `[x, y]`, its prediction value is `prediction = [[w1], [w2]] • [x, y] + b = w1 * x + w2 * y + b`.

Here's our loss function.

Listing 3.19 The mean squared error loss function

per_sample_losses will be a tensor with the same shape as targets and predictions, containing per-sample loss scores.

```
def mean_squared_error(targets, predictions):
    per_sample_losses = tf.square(targets - predictions)
    return tf.reduce_mean(per_sample_losses)
```

We need to average these per-sample loss scores into a single scalar loss value: reduce_mean does this.

Now, we move to the training step, which receives some training data and updates the weights `W` and `b` to minimize the loss on the data.

Listing 3.20 The training-step function

```
learning_rate = 0.1

@tf.function(jit_compile=True)
def training_step(inputs, targets, W, b):
    with tf.GradientTape() as tape:
        predictions = model(inputs, W, b)
        loss = mean_squared_error(predictions, targets)
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
    W.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b * learning_rate)
    return loss
```

Wraps the function in a tf.function decorator to speed it up

Forward pass, inside of a gradient tape scope

Retrieves the gradient of the loss with regard to weights

Updates the weights

For simplicity, we'll do *batch training* instead of *mini-batch training*: we'll run each training step (gradient computation and weight update) on the entire data, rather than iterate over the data in small batches. On one hand, this means that each training step will take much longer to run, since we compute the forward pass and the gradients for 2,000 samples at once. On the other hand, each gradient update will be much more effective at reducing the loss on the training data, since it will encompass information from all training samples instead of, say, only 128 random samples. As a result, we will need many fewer steps of training, and we should use a larger learning rate than what we would typically use for mini-batch training (we'll use `learning_rate = 0.1`, as previously defined).

Listing 3.21 The batch training loop

```
for step in range(40):
    loss = training_step(inputs, targets, W, b)
    print(f"Loss at step {step}: {loss:.4f}")
```

After 40 steps, the training loss seems to have stabilized around 0.025. Let's plot how our linear model classifies the training data points, as shown in figure 3.2. Because our targets are 0s and 1s a given input point will be classified as "0" if its prediction value is below 0.5, and as "1" if it is above 0.5:

```
predictions = model(inputs, W, b)
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
plt.show()
```

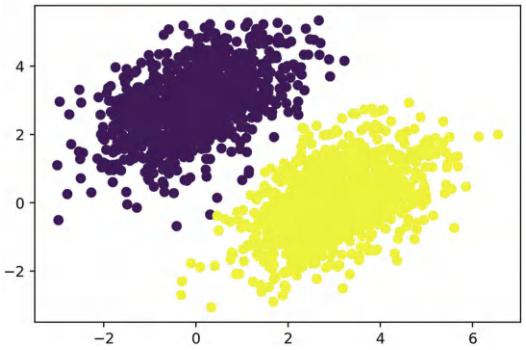


Figure 3.2 Our model's predictions on the training inputs: pretty similar to the training targets

Recall that the prediction value for a given point `[x, y]` is simply `prediction == [[w1], [w2]] • [x, y] + b == w1 * x + w2 * y + b`. Thus, class "0" is defined as `w1 * x + w2 * y + b < 0.5` and class "1" is defined as `w1 * x + w2 * y + b > 0.5`. You'll notice that what you're looking at is really the equation of a line in the 2D plane: `w1 * x + w2 * y + b = 0.5`. Class 1 is above the line; class 0 is below the line. You may be used to seeing line equations in the format `y = a * x + b`; in the same format, our line becomes `y = - w1 / w2 * x + (0.5 - b) / w2`.

Let's plot this line, as shown in figure 3.3:

Generates 100 regularly spaced numbers between -1 and 4, which we will use to plot our line

This is our line's equation.

Plots our line ("-r" means "plot it as a red line")

```
x = np.linspace(-1, 4, 100)
y = -W[0] / W[1] * x + (0.5 - b) / W[1]
plt.plot(x, y, "-r")
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
```

Plots our model's predictions on the same plot

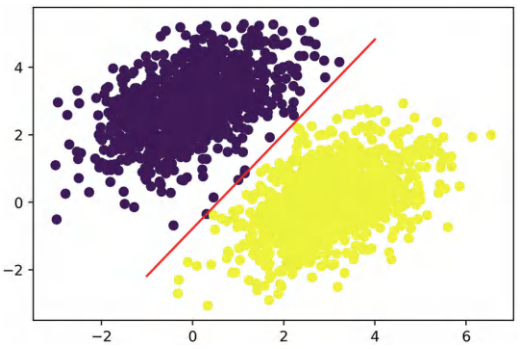


Figure 3.3 Our model, visualized as a line

This is really what a linear classifier is all about: finding the parameters of a line (or, in higher-dimensional spaces, a hyperplane) neatly separating two classes of data.

3.3.3 What makes the TensorFlow approach unique

You're now familiar with all the basic APIs that underlie TensorFlow-based workflows, and you're about to dive into more frameworks—in particular, PyTorch and JAX. What makes working with TensorFlow different from working with any other framework? When should you use TensorFlow, and when could you use something else?

If you ask us, here are the main benefits of TensorFlow:

- Thanks to graph mode and XLA compilation, it's fast. It's usually significantly faster than PyTorch and NumPy, though JAX is often even faster.
- It is extremely feature complete. Unique among all frameworks, it has support for string tensors as well as "ragged tensors" (tensors where different entries may have different dimensions—very useful for handling sequences without requiring to pad them to a shared length). It also has outstanding support for data preprocessing, via the highly performant `tf.data` API. `tf.data` is so good that even JAX recommends it for data preprocessing. Whatever you need to do, TensorFlow has a solution for it.
- Its ecosystem for production deployment is the most mature among all frameworks, especially when it comes to deploying on mobile or in the browser.

However, TensorFlow also has some noticeable flaws:

- It has a sprawling API—the flipside of being very feature complete. TensorFlow includes thousands of different operations.
- Its numerical API is occasionally inconsistent with the NumPy API, making it a bit harder to approach if you're already familiar with NumPy.
- The popular pretrained model-sharing platform Hugging Face has less support for TensorFlow, which means that the latest generative AI models may not always be available in TensorFlow.

Now, let's move on to PyTorch.

3.4 Introduction to PyTorch

PyTorch is a Python-based open source machine learning framework developed primarily by Meta (formerly Facebook). It was originally released in September 2016 (as a response to the release of TensorFlow), with its 1.0 version launched in 2018, and its 2.0 version launched in 2023. PyTorch inherits its programming style from the now-defunct Chainer framework, which was itself inspired by NumPy-Autograd. PyTorch is used extensively in the machine learning research community.

Like TensorFlow, PyTorch is at the center of a large ecosystem of related packages, such as `torchvision`, `torchaudio`, or the popular model-sharing platform Hugging Face.

The PyTorch API is higher level than that of TensorFlow and JAX: it includes layers and optimizers, like Keras. These layers and optimizers are compatible with Keras workflows when you use Keras with the PyTorch backend.

3.4.1 First steps with PyTorch

Over the next paragraphs, you'll get familiar with all the basics of PyTorch. We'll cover the following key concepts:

- Tensors and parameters
- Numerical operations in PyTorch
- Computing gradients with the `backward()` method
- Packaging computation with the `Module` class
- Speeding up PyTorch by using compilation

We'll conclude the introduction by reimplementing our linear regression end-to-end example in pure PyTorch.

Tensors and parameters in PyTorch

A first gotcha about PyTorch is that the package isn't named `pytorch`. It's actually named `torch`. You'd install it via `pip install torch` and you'd import it via `import torch`.

Like in NumPy and TensorFlow, the object at the heart of the framework is the tensor. First, let's get our hands on some PyTorch tensors.

Constant tensors

Here are some constant tensors.

Listing 3.22 All-ones or all-zeros tensors

```
>>> import torch
>>> torch.ones(size=(2, 1))
tensor([[1.], [1.]])
>>> torch.zeros(size=(2, 1))
tensor([[0.], [0.]])
>>> torch.tensor([1, 2, 3], dtype=torch.float32)
tensor([1., 2., 3.])
```

Unlike in other frameworks, the shape argument is named "size" rather than "shape."

Unlike in other frameworks, you cannot pass dtype="float32" as a string. The dtype argument must be a torch dtype instance.

Random tensors

Random tensor creation is similar to NumPy and TensorFlow, but with divergent syntax. Consider the function `normal`: it doesn't take a shape argument. Instead, the mean and standard deviation should be provided as PyTorch tensors with the expected output shape.

Listing 3.23 Random tensors

```
>>> torch.normal(
... mean=torch.zeros(size=(3, 1)),
... std=torch.ones(size=(3, 1)))
tensor([[-0.9613],
        [-2.0169],
        [ 0.2088]])
```

Equivalent to tf.random.normal(shape=(3, 1), mean=0., stddev=1.)

As for creating a random uniform tensor, you'd do that via `torch.rand`. Unlike `np.random.uniform` or `tf.random.uniform`, the output shape should be provided as independent arguments for each dimension, like this:

```
>>> torch.rand(3, 1)
```

Equivalent to tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)

TENSOR ASSIGNMENT AND THE PARAMETER CLASS

Like NumPy arrays, but unlike TensorFlow tensors, PyTorch tensors are assignable. You can do operations like this:

```
>>> x = torch.zeros(size=(2, 1))
>>> x[0, 0] = 1.
>>> x
tensor([[1.],
        [0.]])
```

While you can just use a regular `torch.Tensor` to store the trainable state of a model, PyTorch does provide a specialized tensor subclass for that purpose, the `torch.nn.parameter.Parameter` class. Compared to a regular tensor, it provides semantic clarity—if you see a `Parameter`, you'll know it's a piece of trainable state, whereas a `Tensor` could be anything. As a result, it enables PyTorch to automatically track and retrieve the `Parameters` you assign to PyTorch models—similar to what Keras does with Keras `Variable` instances.

Here's a `Parameter`.

Listing 3.24 Creating a PyTorch parameter

```
>>> x = torch.zeros(size=(2, 1))
>>> p = torch.nn.parameter.Parameter(data=x)
```

A Parameter can only be created using a torch.Tensor value—no NumPy arrays allowed.

TENSOR OPERATIONS: DOING MATH IN PYTORCH

Math in PyTorch works just the same as math in NumPy or TensorFlow, although much like TensorFlow, the PyTorch API often diverges in subtle ways from the NumPy API.

Listing 3.25 A few basic math operations in PyTorch

```
a = torch.ones((2, 2))
b = torch.square(a)
c = torch.sqrt(a)
d = b + c
e = torch.matmul(a, b)
f = torch.cat((a, b), dim=0)
```

Takes the square, same as np.square

Takes the square root, same as np.sqrt

Adds two tensors (element-wise)

Takes the product of two tensors (see chapter 2), same as np.matmul

Concatenates a and b along axis 0, same as np.concatenate

Here's a dense layer:

```
def dense(inputs, W, b):
    return torch.nn.relu(torch.matmul(inputs, W) + b)
```

COMPUTING GRADIENTS WITH PYTORCH

There's no explicit "gradient tape" in PyTorch. A similar mechanism does exist: when you run any computation in PyTorch, the framework creates a one-time computation graph (a "tape") that records what just happened. However, that tape is hidden from the user. The public API for using it is at the level of tensors themselves: you can call `tensor.backward()` to run backpropagation through all operations previously executed that led to that tensor. Doing this will populate the `.grad` attribute of all tensors that are tracking gradients.

Listing 3.26 Computing a gradient with `.backward()`

To compute gradients with respect to a tensor, it must be created with requires_grad=True.

```
>>> input_var = torch.tensor(3.0, requires_grad=True)
>>> result = torch.square(input_var)
>>> result.backward()
>>> gradient = input_var.grad
>>> gradient
tensor(6.)
```

Calling backward() populates the "grad" attribute on all tensors create with requires_grad=True.

If you call `backward()` multiple times in a row, the `.grad` attribute will "accumulate" gradients: each new call will sum the new gradient with the preexisting one. For instance,

in the following code, `input_var.grad` is not the gradient of `square(input_var)` with respect to `input_var`; rather, it is the sum of that gradient and the previously computed gradient—its value has doubled since our last code snippet:

```
>>> result = torch.square(input_var)
>>> result.backward()
>>> input_var.grad
tensor(12.)
```

.grad will sum all gradient values from each time backward() is called.

To reset gradients, you can just set `.grad` to `None`:

```
>>> input_var.grad = None
```

Now let's put this into practice!

3.4.2 An end-to-end example: A linear classifier in pure PyTorch

You now know enough to rewrite our linear classifier in PyTorch. It will stay very similar to the TensorFlow one—the only major difference is how we compute the gradients.

Let's start by creating our model variables. Don't forget to pass `requires_grad=True` so we can compute gradients with respect to them:

```
input_dim = 2
output_dim = 1

W = torch.rand(input_dim, output_dim, requires_grad=True)
b = torch.zeros(output_dim, requires_grad=True)
```

This is our model—no difference so far. We just went from `tf.matmul` to `torch.matmul`:

```
def model(inputs, W, b):
    return torch.matmul(inputs, W) + b
```

This is our loss function. We just switch from `tf.square` to `torch.square` and from `tf.reduce_mean` to `torch.mean`:

```
def mean_squared_error(targets, predictions):
    per_sample_losses = torch.square(targets - predictions)
    return torch.mean(per_sample_losses)
```

Now for the training step. Here's how it works:

1. `loss.backward()` runs backpropagation starting from the `loss` output node and populates the `tensor.grad` attribute on all tensors that were involved in the computation of `loss`. `tensor.grad` represents the gradient of the loss with regard to that tensor.
2. We use the `.grad` attribute to recover the gradients of the loss with regard to `W` and `b`.
3. We update `W` and `b` using those gradients. Because these updates are not intended to be part of the backward pass, we do them inside a `torch.no_grad()` scope, which skips gradient computation for everything inside it.
4. We reset the contents of the `.grad` property of our `W` and `b` parameters, by setting it to `None`. If we didn't do this, gradient values would accumulate across multiple calls to `training_step()`, resulting in invalid values:

```
learning_rate = 0.1

def training_step(inputs, targets, W, b):
    predictions = model(inputs)
    loss = mean_squared_error(targets, predictions)
    loss.backward()
    grad_loss_wrt_W, grad_loss_wrt_b = W.grad, b.grad
    with torch.no_grad():
        W -= grad_loss_wrt_W * learning_rate
        b -= grad_loss_wrt_b * learning_rate
    W.grad = None
    b.grad = None
    return loss
```

Forward pass

Computes gradients

Retrieves gradients

Updates weights inside a no_grad scope

Resets gradients

This could be made even simpler—let's see how.

Packaging state and computation with the Module class

PyTorch also has a higher-level, object-oriented API for performing backpropagation, which requires relying on two new classes: the `torch.nn.Module` class and an optimizer class from the `torch.optim` module, such as `torch.optim.SGD` (the equivalent of `keras.optimizers.SGD`).

The general idea is to define a subclass of `torch.nn.Module`, which will

- Hold some `Parameters`, to store state variables. Those are defined in the `__init__()` method.
- Implement the forward pass computation in the `forward()` method.

It should look just like the following.

Listing 3.27 Defining a `torch.nn.Module`

```
class LinearModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
```

```
            self.W = torch.nn.Parameter(torch.rand(input_dim, output_dim))
            self.b = torch.nn.Parameter(torch.zeros(output_dim))

        def forward(self, inputs):
            return torch.matmul(inputs, self.W) + self.b
```

We can now instantiate our `LinearModel`:

```
model = LinearModel()
```

When using an instance of `torch.nn.Module`, rather than calling the `forward()` method directly, you'd use `__call__()` (i.e., directly call the model class on inputs), which redirects to `forward()` but adds a few framework hooks to it:

```
torch_inputs = torch.tensor(inputs)
output = model(torch_inputs)
```

Now, let's get our hands on a PyTorch optimizer. To instantiate it, you will need to provide the list of parameters that the optimizer is intended to update. You can retrieve it from our `Module` instance via `.parameters()`:

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Using our `Module` instance and the PyTorch `SGD` optimizer, we can run a simplified training step:

```
def training_step(inputs, targets):
    predictions = model(inputs)
    loss = mean_squared_error(targets, predictions)
    loss.backward()
    optimizer.step()
    model.zero_grad()
    return loss
```

Previously, updating the model parameters looked like this:

```
with torch.no_grad():
    W -= grad_loss_wrt_W * learning_rate
    b -= grad_loss_wrt_b * learning_rate
```

Now we can just do `optimizer.step()`.

Similarly, previously we needed to reset parameter gradients by hand by doing `tensor.grad = None` on each one. Now we can just do `model.zero_grad()`.

Overall, this may feel a bit confusing—somehow the loss tensor, the optimizer, and the `Module` instance all seem to be aware of each other through some hidden background mechanism. They're all interacting with one another via spooky action at a distance. Don't worry though—you can just treat this sequence of steps (`loss.backward()` - `optimizer.step()` - `model.zero_grad()`) as a magic incantation to be recited any time you need to write a training step function. Just make sure not to forget `model.zero_grad()`. That would be a major bug (and it is unfortunately quite common)!

Making PyTorch modules fast using compilation

One last thing. Similarly to how TensorFlow lets you compile functions for better performance, PyTorch lets you compile functions or even `Module` instances via the `torch.compile()` utility. This API uses PyTorch's very own compiler, named Dynamo.

Let's try it on our linear regression `Module`:

```
compiled_model = torch.compile(model)
```

The resulting object is intended to work identically to the original—except the forward and backward pass should run faster.

You can also use `torch.compile()` as a function decorator:

```
@torch.compile
def dense(inputs, W, b):
    return torch.nn.relu(torch.matmul(inputs, W) + b)
```

In practice, most PyTorch code out there does not use compilation and simply runs eagerly, as the compiler may not always work with all models and may not always result in a speedup when it does work. Unlike in TensorFlow and Jax where compilation was built in from the inception of the library, PyTorch's compiler is a relatively recent addition.

3.4.3 What makes the PyTorch approach unique

Compared to TensorFlow and JAX, which we will cover next, what makes PyTorch stand out? Why should you use it or not use it?

Here are PyTorch's two key strengths:

- PyTorch code executes eagerly by default, making it easy to debug. Note that this is also the case for TensorFlow code and JAX code, but a big difference is that PyTorch is generally intended to be run eagerly at all times, whereas any serious TensorFlow or JAX project will inevitably need compilation at some point, which can significantly hurt the debugging experience.

- The popular pretrained model-sharing platform Hugging Face has first-class support for PyTorch, which means that any model you'd like to use is likely available in PyTorch. This is the primary drive behind PyTorch adoption today.

Meanwhile, there are also some downsides to using PyTorch:

- Like with TensorFlow, the PyTorch API is inconsistent with NumPy. Further, it's also internally inconsistent. For instance, the commonly used keyword `axis` is occasionally named `dim` instead, depending on the function. Some pseudo-random number generation operations take a `seed` argument; others don't. And so on. This can make PyTorch frustrating to learn, especially when coming from NumPy.
- Due to its focus on eager execution, PyTorch is quite slow—it's the slowest of all the major frameworks by a large margin. For most models, you may see a 20% or 30% speedup with JAX. For some models—especially large ones—you may even see a 3× or a 5× speedup with JAX, even after using `torch.compile()`.
- While it is possible to make PyTorch code faster via `torch.compile()`, the PyTorch Dynamo compiler remains at this time (in 2025) quite ineffective and full of trapdoors. As a result, only a very small percentage of the PyTorch user base uses compilation. Perhaps this will be improved in future versions!

3.5 Introduction to JAX

JAX is an open source library for differentiable computation, primarily developed by Google. After its release in 2018, JAX quickly gained traction in the research community, particularly for its ability to use Google's TPUs at scale. Today, JAX is in use by most of the top players in the generative AI space—companies like DeepMind, Apple, Midjourney, Anthropic, Cohere, and so on.

JAX embraces a *stateless* approach to computation, meaning that functions in JAX do not maintain any persistent state. This contrasts with traditional imperative programming, where variables can hold values between function calls.

The stateless nature of JAX functions has several advantages. In particular, it enables effective automatic parallelization and distributed computation, as functions can be executed independently without the need for synchronization. The extreme scalability of JAX is essential for handling the very large-scale machine learning problems faced by companies like Google and DeepMind.

3.5.1 First steps with JAX

We'll go over the following key concepts:

- The `array` class
- Random operations in JAX
- Numerical operations in JAX

- Computing gradients via `jax.grad` and `jax.value_and_grad`
- Making JAX functions fast by leveraging just-in-time compilation

Let's get started.

3.5.2 Tensors in JAX

One of the best features of JAX is that it doesn't try to implement its own independent, similar-to-NumPy-but-slightly-divergent numerical API. Instead, it just implements the NumPy API, as is. It is available as the `jax.numpy` namespace, and you will often see it imported as `jnp` for short.

Here are some JAX arrays.

Listing 3.28 All-ones or all-zeros tensors

```
>>> from jax import numpy as jnp
>>> jnp.ones(shape=(2, 1))
Array([[1.],
       [1.]], dtype=float32)
>>> jnp.zeros(shape=(2, 1))
Array([[0.],
       [0.]], dtype=float32)
>>> jnp.array([1, 2, 3], dtype="float32")
Array([1., 2., 3.], dtype=float32)
```

There are, however, two minor differences between `jax.numpy` and the actual NumPy API: random number generation and array assignment. Let's take a look.

3.5.3 Random number generation in JAX

The first difference between JAX and NumPy has to do with the way JAX handles random operations—what is known as "PRNG" (Pseudo-Random Number Generation) operations. We said earlier that JAX is *stateless*, which implies that JAX code can't rely on any hidden global state. Consider the following NumPy code.

Listing 3.29 Random tensors

```
>>> np.random.normal(size=(3,))
array([-1.68856166,  0.16489586,  0.67707523])
>>> np.random.normal(size=(3,))
array([-0.73671259,  0.3053194 ,  0.84124895])
```

How did the second call to `np.random.normal()` know to return a different value from the first call? That's right—it's a hidden piece of global state. You can actually retrieve that global state via `np.random.get_state()` and set it via `np.random.seed(seed)`.

In a stateless framework, we can't have any such global state. The same API call must always return the same value. As a result, in a stateless version of NumPy, you would have to rely on passing different seed arguments to your `np.random` calls to get different values.

Now, it's often the case that your PRNG calls are going to be in functions that get called multiple times and that are intended to use different random values each time. If you don't want to rely on any global state, this requires you to manage your seed state outside of the target function, like this:

```
def apply_noise(x, seed):
    np.random.seed(seed)
    x = x * np.random.normal((3,))
    return x

seed = 1337
y = apply_noise(x, seed)
seed += 1
z = apply_noise(x, seed)
```

It's basically the same in JAX. However, JAX doesn't use integer seeds. It uses special array structures called *keys*. You can create one from an integer value, like this:

```
import jax

seed_key = jax.random.key(1337)
```

To force you to always provide a seed "key" to PRNG calls, all JAX PRNG-using operations take `key` (the random seed) as their first positional argument. Here's how to use `random.normal()`:

```
>>> seed_key = jax.random.key(0)
>>> jax.random.normal(seed_key, shape=(3,))
Array([ 1.8160863 , -0.48262316,  0.33988908], dtype=float32)
```

Two calls to `random.normal()` that receive the same seed key will always return the same value.

Listing 3.30 Using a random seed in Jax

```
>>> seed_key = jax.random.key(123)
>>> jax.random.normal(seed_key, shape=(3,))
Array([-0.1470326,  0.5524756,  1.648498 ], dtype=float32)
>>> jax.random.normal(seed_key, shape=(3,))
Array([-0.1470326,  0.5524756,  1.648498 ], dtype=float32)
```

If you need a new seed key, you can simply create a new one from an existing one using the `jax.random.split()` function. It is deterministic, so the same sequence of splits will always result in the same final seed key:

```
>>> seed_key = jax.random.key(123)
>>> jax.random.normal(seed_key, shape=(3,))
Array([-0.1470326,  0.5524756,  1.648498 ], dtype=float32)
>>> new_seed_key = jax.random.split(seed_key, num=1)[0]
>>> jax.random.normal(new_seed_key, shape=(3,))
Array([ 0.5362355, -1.1920372,  2.450225 ], dtype=float32)
```

You could even split your key into multiple new keys at once!

This is definitely more work than `np.random`! But the benefits of statelessness far outweigh the costs: it makes your code *vectorizable* (i.e., the JAX compiler can automatically turn it into highly parallel code) while maintaining determinism (i.e., you can run the same code twice with the same results). That is impossible to achieve with a global PRNG state.

TENSOR ASSIGNMENT

The second difference between JAX and NumPy is tensor assignment. Like in TensorFlow, JAX arrays are not assignable in place. That's because any sort of in-place modification would go against JAX's stateless design. Instead, if you need to update a tensor, you must create a new tensor with the desired value. JAX makes this easy by providing the `at()/set()` API. These methods allow you to create a new tensor with an updated element at a specific index. Here's an example of how you would update the first element of a JAX array to a new value.

Listing 3.31 Modifying values in a JAX array

```
>>> x = jnp.array([1, 2, 3], dtype="float32")
>>> new_x = x.at[0].set(10)
```

Simple enough!

TENSOR OPERATIONS: DOING MATH IN JAX

Doing math in JAX looks exactly the same as it does in NumPy. No need to learn anything new this time!

Listing 3.32 A few basic math operations in JAX

```
a = jnp.ones((2, 2))
b = jnp.square(a)
c = jnp.sqrt(a)
d = b + c
e = jnp.matmul(a, b)
e *= d
```

Takes the square

Takes the square root

Adds two tensors (element-wise)

Takes the product of two tensors (see chapter 2)

Multiplies two tensors (element-wise)

Here's a dense layer:

```
def dense(inputs, W, b):
    return jax.nn.relu(jnp.matmul(inputs, W) + b)
```

COMPUTING GRADIENTS WITH JAX

Unlike TensorFlow and PyTorch, JAX takes a *metaprogramming* approach to gradient computation. Metaprogramming refers to the idea of having *functions that return functions*—you could call them "meta-functions." In practice, JAX lets you *turn a loss-computation function into a gradient-computation function*. So computing gradients in JAX is a three-step process:

1. Define a loss function, `compute_loss()`.
2. Call `grad_fn = jax.grad(compute_loss)` to retrieve a gradient-computation function.
3. Call `grad_fn` to retrieve the gradient values.

The loss function should verify the following properties:

- It should return a scalar loss value.
- Its first argument (which, in the following example, is also the only argument) should contain the state arrays we need gradients for. This argument is usually named `state`. For instance, this first argument could be a single array, a list of arrays, or a dict of arrays.

Let's take a look at a simple example. Here's a loss-computation function that takes a single scalar, `input_var` and returns a scalar loss value—just the square of the input:

```
def compute_loss(input_var):
    return jnp.square(input_var)
```

We can now call the JAX utility `jax.grad()` on this loss function. It returns a gradient-computation function—a function that takes the same arguments as the original loss function and returns the gradient of the loss with respect to `input_var`:

```
grad_fn = jax.grad(compute_loss)
```

Once you've obtained `grad_fn()`, you can call it with the same arguments as `compute_loss()`, and it will return gradients arrays corresponding to the first argument of `compute_loss()`. In our case, our first argument was a single array, so `grad_fn()` directly returns the gradient of the loss with respect to that one array:

```
input_var = jnp.array(3.0)
grad_of_loss_wrt_input_var = grad_fn(input_var)
```

JAX GRADIENT-COMPUTATION BEST PRACTICES

So far so good! Metaprogramming is a big word, but it turns out to be quite simple. Now, in real-world use cases, there are a few more things you'll need to take into account. Let's take a look.

RETURNING THE LOSS VALUE

It's usually the case that you don't just need the gradient array; you also need the loss value. It would be quite inefficient to recompute it independently outside of `grad_fn()`, so instead, you can just configure your `grad_fn()` to also return the loss value. This is done by using the JAX utility `jax.value_and_grad()` instead of `jax.grad()`. It works identically, but it returns a tuple of values, where the first entry is the loss value, and the second entry is the gradient(s):

```
grad_fn = jax.value_and_grad(compute_loss)
output, grad_of_loss_wrt_input_var = grad_fn(input_var)
```

GETTING GRADIENTS FOR A COMPLEX FUNCTION

Now, what if you need gradients for more than a single variable? And what if your `compute_loss()` function has more than one input?

Let's say your state contains three variables, `a`, `b`, and `c`, and your loss function has two inputs, `x` and `y`. You would simply structure it like this:

```
def compute_loss(state, x, y):
    ...
    return loss

grad_fn = jax.value_and_grad(compute_loss)
state = (a, b, c)
loss, grads_of_loss_wrt_state = grad_fn(state, x, y)
```

state contains a, b, and c. It must be the first argument.

grads_of_loss_wrt_state has the same structure as state.

Note that `state` doesn't have to be a tuple—it could be a dict, a list, or any nested structure of tuples, dicts, and lists. In JAX parlance, such a nested structure is called a *tree*.

RETURNING AUXILIARY OUTPUTS

Finally, what if your `compute_loss()` function needs to return more than just the loss? Let's say you want to return an additional value `output` that's computed as a by-product of the loss computation. How to get it out?

You would use the `has_aux` argument:

1. Edit the loss function to return a tuple where the first entry is the loss, and the second entry is your extra output.
2. Pass the argument `has_aux=True` to `value_and_grad()`. This tells `value_and_grad()` to return not just the gradient but also the "auxiliary" output(s) of `compute_loss()`, like this:

```
def compute_loss(state, x, y):
    ...
    return loss, output                    <-- Returns a tuple

grad_fn = jax.value_and_grad(compute_loss, has_aux=True)   <-- Passes has_aux=True here
loss, (grads_of_loss_wrt_state, output) = grad_fn(state, x, y)   <-- Gets back a nested tuple
```

Admittedly, things are starting to be pretty convoluted at this point. Don't worry, though; this is about as hard as JAX gets! Almost everything else is simpler by comparison.

MAKING JAX FUNCTIONS FAST WITH @JAX.JIT

One more thing. As a JAX user, you will frequently use the `@jax.jit` decorator, which behaves identically to the `@tf.function(jit_compile=True)` decorator. It turns any stateless JAX function into an XLA-compiled piece of code, typically delivering a considerable execution speedup:

```
@jax.jit
def dense(inputs, W, b):
    return jax.nn.relu(jnp.matmul(inputs, W) + b)
```

Be mindful that you can only decorate a stateless function—any tensors that get updated by the function should be part of its return values.

3.5.4 An end-to-end example: A linear classifier in pure JAX

Now you know enough JAX to write the JAX version of our linear classifier example. There are two major differences from the TensorFlow and PyTorch versions you've already seen:

- All functions we will create will be *stateless*. That means the state (the arrays `W` and `b`) will be provided as function arguments, and if they get modified by the function, their new value will be returned by the function.
- Gradients are computed using the JAX `value_and_grad()` utility.

Let's get started. The model function and the mean squared error function should look familiar:

```
def model(inputs, W, b):
    return jnp.matmul(inputs, W) + b

def mean_squared_error(targets, predictions):
    per_sample_losses = jnp.square(targets - predictions)
    return jnp.mean(per_sample_losses)
```

To compute gradients, we need to package loss computation in a single `compute_loss()` function. It returns the total loss as a scalar, and it takes `state` as its first argument—a tuple of all the tensors we need gradients for:

```
def compute_loss(state, inputs, targets):
    W, b = state
    predictions = model(inputs, W, b)
    loss = mean_squared_error(targets, predictions)
    return loss
```

Calling `jax.value_and_grad()` on this function gives us a new function, with the same argument as `compute_loss`, which returns both the loss and the gradients of the loss with regard to the elements of `state`:

```
grad_fn = jax.value_and_grad(compute_loss)
```

Next, we can set up our training step function. It looks straightforward. Be mindful that, unlike its TensorFlow and PyTorch equivalents, it needs to be stateless, and so it must return the updated values of the `W` and `b` tensors:

```
learning_rate = 0.1

@jax.jit
def training_step(inputs, targets, W, b):
    loss, grads = grad_fn((W, b), inputs, targets)
    grad_wrt_W, grad_wrt_b = grads
    W = W - grad_wrt_W * learning_rate
    b = b - grad_wrt_b * learning_rate
    return loss, W, b
```

We use the jax.jit decorator to take advantage of XLA compilation.

Computes the forward pass and backward pass in one go

Updates W and b

Make sure to return the new values of W and b in addition to the loss!

Because we won't change the `learning_rate` during our example, we can consider it part of the function itself and not our model's state. If we wanted to modify our learning rate during training, we'd need to pass it through as well.

Finally, we're ready to run the full training loop. We initialize `W` and `b`, and we repeatedly update them via stateless calls to `training_step()`:

```
input_dim = 2
output_dim = 1

W = jax.numpy.array(np.random.uniform(size=(input_dim, output_dim)))
b = jax.numpy.array(np.zeros(shape=(output_dim,)))
state = (W, b)
for step in range(40):
    loss, W, b = training_step(inputs, targets, W, b)
    print(f"Loss at step {step}: {loss:.4f}")
```

That's it! You're now able to write a custom training loop in JAX.

3.5.5 What makes the JAX approach unique

The main thing that makes JAX unique among modern machine learning frameworks is its functional, stateless philosophy. While it may seem to cause friction at first, it is what unlocks the power of JAX—its ability to compile to extremely fast code and to scale to arbitrarily large models and arbitrarily many devices.

There's a lot to like about JAX:

- It's fast. For most models, it is the fastest of all frameworks you've seen so far.
- Its numerical API is fully consistent with NumPy, making it pleasant to learn.
- It's the best fit for training models on TPUs, as it was developed from the ground up for XLA and TPUs.

Using JAX can also come with some amount of developer friction:

- Its use of metaprogramming and compilation can make it significantly harder to debug compared to pure eager execution.
- Low-level training loops tend to be more verbose and more difficult to write than in TensorFlow or PyTorch.

At this point, you know the basics of TensorFlow, PyTorch, and JAX, and you can use these frameworks to implement a basic linear classifier from scratch. That's a solid foundation to build upon. It's now time to move on to a more productive path to deep learning: the Keras API.

3.6 Introduction to Keras

Keras is a deep learning API for Python that provides a convenient way to define and train any kind of deep learning model. It was released in March 2015, with its v2 in 2017 and its v3 in 2023.

Keras users range from academic researchers, engineers, and data scientists at both startups and large companies to graduate students and hobbyists. Keras is used at Google, Netflix, Uber, YouTube, CERN, NASA, Yelp, Instacart, Square, Waymo, YouTube, and thousands of smaller organizations working on a wide range of problems across every industry. Your YouTube recommendations originate from Keras models. The

Waymo self-driving cars rely on Keras models for processing sensor data. Keras is also a popular framework on Kaggle, the machine learning competition website.

Because Keras has a diverse user base, it doesn't force you to follow a single "true" way of building and training models. Rather, it enables a wide range of different workflows, from the very high-level to the very low-level, corresponding to different user profiles. For instance, you have an array of ways to build models and an array of ways to train them, each representing a certain tradeoff between usability and flexibility. In chapter 7, we'll review in detail a good fraction of this spectrum of workflows.

3.6.1 First steps with Keras

Before we get to writing Keras code, there are a few things to consider when setting up the library before it's imported.

Picking a backend framework

Keras can be used together with JAX, TensorFlow, or PyTorch. They're the "backend frameworks" of Keras. Through these backend frameworks, Keras can run on top of different types of hardware (see figure 3.4)—GPU, TPU, or plain CPU—can be seamlessly scaled to thousands of machines, and can be deployed to a variety of platforms.

Backend frameworks are pluggable: you can switch to a different backend framework *after* you've written some Keras code. You aren't locked into a single framework and a single ecosystem—you can move your models from JAX to TensorFlow to PyTorch depending on your current needs. For instance, when you develop a Keras model, you could debug it with PyTorch, train it on TPU with JAX for maximum efficiency, and finally run inference with the excellent tooling from the TensorFlow ecosystem.

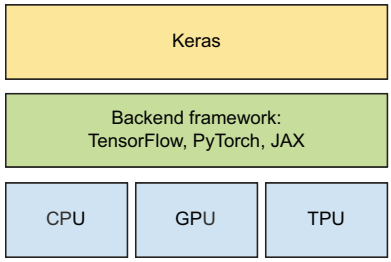


Figure 3.4 Keras and its backends. A backend is a low-level tensor-computing platform; Keras is a high-level deep learning API.

The default backend for Keras right now is TensorFlow, so if you run `import keras` in a fresh environment, without having configured anything, you will be running on top of TensorFlow. There are two ways to pick a different backend:

- Set the environment variable `KERAS_BACKEND`. Before you start your `python` repl, you can run the following shell command to use JAX as your Keras backend: `export KERAS_BACKEND=jax`. Alternatively, you can add the following code snippet

at the top of your Python file or notebook (note that it must imperatively go before the first `import keras`):

```
import os

os.environ["KERAS_BACKEND"] = "jax"

import keras
```

Sets the environment variable from within the Python runtime

Only then should you import Keras.

- Edit your local Keras configuration file at `~/.keras/keras.json`. If you have already imported Keras once, this file has already been created with default settings. You can use any text editor to open and modify it—it's a human-readable JSON file. It should look like this:

```
{
    "floatx": "float32",
    "epsilon": 1e-07,
    "backend": "tensorflow",
    "image_data_format": "channels_last",
}
```

Default floating-point precision. It should typically not be changed.

Default numerical fuzzing factor. It should typically not be changed.

Change "tensorflow" to "jax" or "torch."

This is the default image layout. We'll talk about this in chapter 8.

NOTE When configuring the Keras backend, you should use the string `"torch"` to refer to the PyTorch backend, rather than the string `"pytorch"`, which would be invalid. This is because the PyTorch package name is `torch` (as in `import torch` or `pip install torch`).

Now, you may ask, which backend should I be picking? It's really your own choice: all Keras code examples in the rest of the book will be compatible with all three backends. If the need for backend-specific code arises (as in chapter 7, for instance), we will show you all three versions—TensorFlow, PyTorch, JAX. If you have no particular backend preference, our personal recommendation is JAX. It's usually the most performant backend.

Once your backend is configured, you can start actually building and training Keras models. Let's take a look.

3.6.2 Layers: The building blocks of deep learning

The fundamental data structure in neural networks is the *layer*, to which you were introduced in chapter 2. A layer is a data processing module that takes as input one or more tensors and that outputs one or more tensors. Some layers are stateless, but more frequently layers have a state: the layer's *weights*, one or several tensors learned with stochastic gradient descent, which together contain the network's *knowledge*.

Different types of layers are appropriate for different tensor formats and different types of data processing. For instance, simple vector data, stored in 2D tensors of shape `(samples, features)`, is often processed by *densely connected* layers, also called *fully connected* or *dense* layers (the `Dense` class in Keras). Sequence data, stored in 3D tensors of shape `(samples, timesteps, features)`, is typically processed by *recurrent* layers, such as an LSTM layer, or 1D convolution layers (`Conv1D`). Image data, stored in rank-4 tensors, is usually processed by 2D convolution layers (`Conv2D`).

You can think of layers as the LEGO bricks of deep learning, a metaphor that is made explicit by Keras. Building deep learning models in Keras is done by clipping together compatible layers to form useful data transformation pipelines.

The base Layer class in Keras

A simple API should have a single abstraction around which everything is centered. In Keras, that's the `Layer` class. Everything in Keras is either a `Layer` or something that closely interacts with a `Layer`.

A `Layer` is an object that encapsulates some state (weights) and some computation (a forward pass). The weights are typically defined in a `build()` (although they could also be created in the constructor `__init__()`), and the computation is defined in the `call()` method.

In the previous chapter, we implemented a `NaiveDense` class that contained two weights `W` and `b` and applied the computation `output = activation(matmul(input, W) + b)`. The following is what the same layer would look like in Keras.

Listing 3.33 A simple dense layer from scratch in Keras

```
import keras

class SimpleDense(keras.Layer):
    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units
        self.activation = activation

    def build(self, input_shape):
        batch_dim, input_dim = input_shape
        self.W = self.add_weight(
            shape=(input_dim, self.units), initializer="random_normal"
        )
        self.b = self.add_weight(shape=(self.units,), initializer="zeros")

    def call(self, inputs):
        y = keras.ops.matmul(inputs, self.W) + self.b
        if self.activation is not None:
            y = self.activation(y)
        return y
```

All Keras layers inherit from the base Layer class.

Weight creation takes place in the build() method.

We define the forward pass computation in the call() method.

add_weight is a shortcut method for creating weights. It's also possible to create standalone variables and assign them as layer attributes, like self.W = keras.Variable(shape=…, initializer=…).

In the next section, we'll cover in detail the purpose of these `build()` and `call()` methods. Don't worry if you don't understand everything just yet!

Once instantiated, a layer like this can be used just like a function, taking as input a tensor:

```
>>> my_dense = SimpleDense(units=32, activation=keras.ops.relu)
>>> input_tensor = keras.ops.ones(shape=(2, 784))
>>> output_tensor = my_dense(input_tensor)
>>> print(output_tensor.shape)
(2, 32)
```

Instantiates our layer, defined previously

Creates some test inputs

Calls the layer on the inputs, just like a function

Now, you're probably wondering, why did we have to implement `call()` and `build()`, since we ended up using our layer by plainly calling it, that is to say, by using its `__call__()` method? It's because we want to be able to create the state just in time. Let's see how that works.

AUTOMATIC SHAPE INFERENCE: BUILDING LAYERS ON THE FLY

Just like with LEGO bricks, you can only "clip" together layers that are *compatible*. The notion of *layer compatibility* here refers specifically to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape. Consider the following example:

```
from keras import layers

layer = layers.Dense(32, activation="relu")
```

A dense layer with 32 output units

This layer will return a tensor whose non-batch dimension is 32. It can only be connected to a downstream layer that expects 32-dimensional vectors as its input.

When using Keras, you don't have to worry about size compatibility most of the time because the layers you add to your models are dynamically built to match the shape of the incoming inputs. For instance, suppose you write the following:

```
from keras import models
from keras import layers

model = models.Sequential(
    [
        layers.Dense(32, activation="relu"),
        layers.Dense(32),
    ]
)
```

The layers didn't receive any information about the shape of their inputs. Instead, they automatically inferred their input shape as being the shape of the first inputs they see.

In the toy version of a `Dense` layer that we've implemented in chapter 2, we had to pass the layer's input size explicitly to the constructor in order to be able to create its weights. That's not ideal, because it would lead to models that look like this, where each new layer needs to be made aware of the shape of the layer before it:

```
model = NaiveSequential(
    [
        NaiveDense(input_size=784, output_size=32, activation="relu"),
        NaiveDense(input_size=32, output_size=64, activation="relu"),
        NaiveDense(input_size=64, output_size=32, activation="relu"),
        NaiveDense(input_size=32, output_size=10, activation="softmax"),
    ]
)
```

It would be even worse when the rules used by a layer to produce its output shape are complex. For instance, what if our layer returned outputs of shape `(batch, input_size * 2 if input_size % 2 == 0 else input_size * 3)`?

If we were to reimplement our `NaiveDense` layer as a Keras layer capable of automatic shape inference, it would look like the `SimpleDense` layer, with its `build()` and `call()` methods.

In the Keras `SimpleDense`, we no longer create weights in the constructor like in the previous example. Instead, we create them in a dedicated state-creation method `build()`, which receives as argument the first input shape seen by the layer. The `build()` method is called automatically the first time the layer is called (via its `__call__()` method). In fact, that's why we defined the computation in a separate `call()` method rather than in the `__call__()` method directly! The `__call__()` method of the base layer schematically looks like this:

```
def __call__(self, inputs):
    if not self.built:
        self.build(inputs.shape)
        self.built = True
    return self.call(inputs)
```

With automatic shape inference, our previous example becomes simple and neat:

```
model = keras.Sequential(
    [
        SimpleDense(32, activation="relu"),
        SimpleDense(64, activation="relu"),
        SimpleDense(32, activation="relu"),
```

```
            SimpleDense(10, activation="softmax"),
        ]
    )
```

Note that automatic shape inference is not the only thing that the `Layer` class's `__call__()` method handles. It takes care of many more things, in particular routing between *eager* and *graph* execution, and input masking (which we cover in chapter 14). For now, just remember: when implementing your own layers, put the forward pass in the `call()` method.

3.6.3 From layers to models

A deep learning model is a graph of layers. In Keras, that's the `Model` class. For now, you've only seen `Sequential` models (a subclass of `Model`), which are simple stacks of layers, mapping a single input to a single output. But as you move forward, you'll be exposed to a much broader variety of network topologies. Some common ones are

- Two-branch networks
- Multihead networks
- Residual connections

Network topology can get quite involved. For instance, figure 3.5 shows topology of the graph of layers of a Transformer, a common architecture designed to process text data.

There are generally two ways of building such models in Keras: you can directly subclass the `Model` class, or you can use the Functional API, which lets you do more with less code. We'll cover both approaches in chapter 7.

The topology of a model defines a *hypothesis space*. You may remember that in chapter 1, we described machine learning as "searching for useful representations of some input data, within a predefined *space of possibilities*, using guidance from a feedback signal." By choosing a network topology, you constrain your space of possibilities (hypothesis space) to a specific series of tensor operations, mapping input data to output data. What you'll then be searching for is a good set of values for the weight tensors involved in these tensor operations.

To learn from data, you have to make assumptions about it. These assumptions define what can be learned. As such, the structure of your hypothesis space—the architecture of your model—is extremely important. It encodes the assumptions you make about your problem, the prior knowledge that the model starts with. For instance, if you're working on a two-class classification problem with a model made of a single `Dense` layer with no activation (a pure affine transformation), you are assuming that your two classes are linearly separable.

Picking the right network architecture is more an art than a science, and although there are some best practices and principles you can rely on, only practice can help you become a proper neural network architect. The next few chapters will both teach you explicit principles for building neural networks and help you develop intuition as to

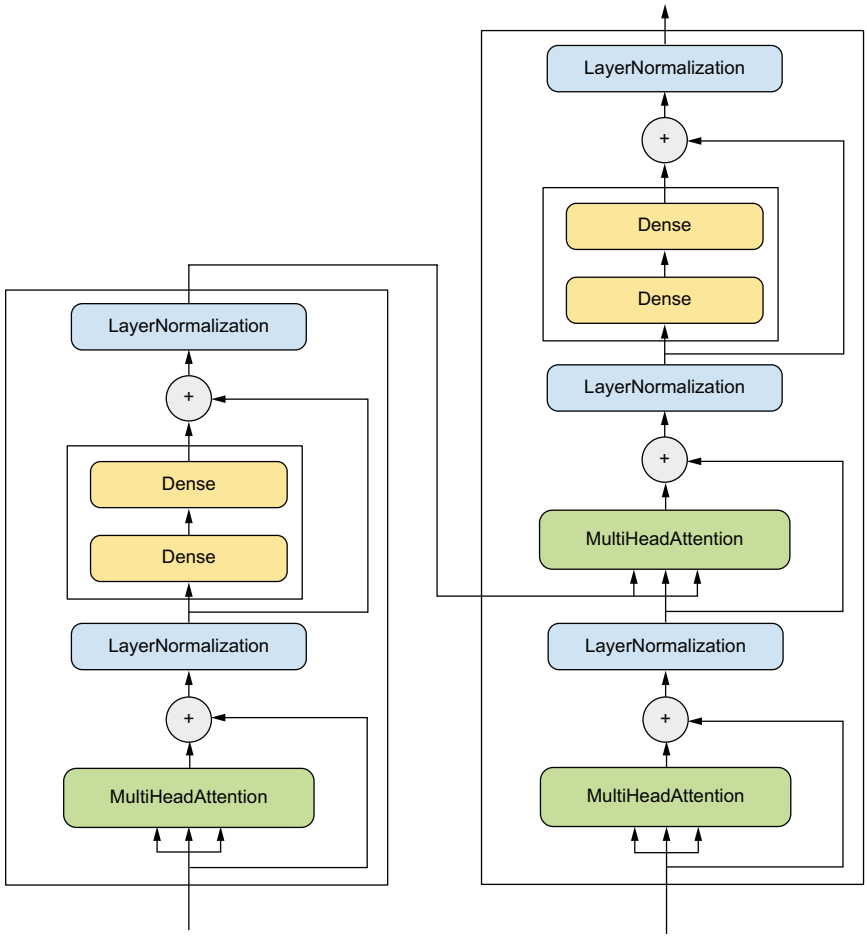


Figure 3.5 The Transformer architecture. There's a lot going on here. Throughout the next few chapters, you'll climb your way up to understanding it (in chapter 15).

what works or doesn't work for specific problems. You'll build a solid intuition about what type of model architectures work for different kinds of problems, how to build these networks in practice, how to pick the right learning configuration, and how to tweak a model until it yields the results you want to see.

3.6.4 The "compile" step: Configuring the learning process

Once the model architecture is defined, you still have to choose three more things:

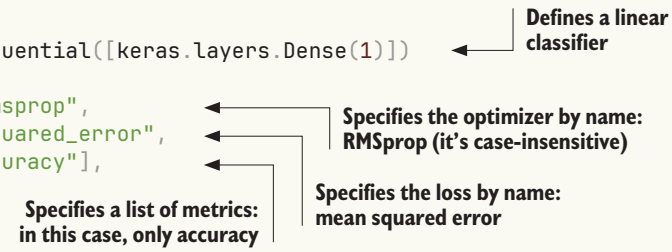
- *Loss function (objective function)*—The quantity that will be minimized during training. It represents a measure of success for the task at hand.
- *Optimizer*—Determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD).

- *Metrics*—The measures of success you want to monitor during training and validation, such as classification accuracy. Unlike the loss, training will not optimize directly for these metrics. As such, metrics don't need to be differentiable.

Once you've picked your loss, optimizer, and metrics, you can use the built-in `compile()` and `fit()` methods to start training your model. Alternatively, you can write your own custom training loops—we cover how to do this in chapter 7. It's a lot more work! For now, let's take a look at `compile()` and `fit()`.

The `compile()` method configures the training process—you've already been introduced to it in your very first neural network example in chapter 2. It takes the arguments `optimizer`, `loss`, and `metrics` (a list):

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(
    optimizer="rmsprop",
    loss="mean_squared_error",
    metrics=["accuracy"],
)
```



In the previous call to `compile()`, we passed the optimizer, loss, and metrics as strings (such as `"rmsprop"`). These strings are actually shortcuts that get converted to Python objects. For instance, `"rmsprop"` becomes `keras.optimizers.RMSprop()`. Importantly, it's also possible to specify these arguments as object instances, like this:

```
model.compile(
    optimizer=keras.optimizers.RMSprop(),
    loss=keras.losses.MeanSquaredError(),
    metrics=[keras.metrics.BinaryAccuracy()],
)
```

This is useful if you want to pass your own custom losses or metrics or if you want to further configure the objects you're using—for instance, by passing a `learning_rate` argument to the optimizer:

```
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),
    loss=my_custom_loss,
    metrics=[my_custom_metric_1, my_custom_metric_2],
)
```

In chapter 7, we cover how to create custom losses and metrics. In general, you won't have to create your own losses, metrics, or optimizers from scratch because Keras offers a wide range of built-in options that is likely to include what you need:

- *Optimizers*
  - `SGD()` (with or without momentum)
  - `RMSprop()`
  - `Adam()`
  - Etc.
- *Losses*
  - `CategoricalCrossentropy()`
  - `SparseCategoricalCrossentropy()`
  - `BinaryCrossentropy()`
  - `MeanSquaredError()`
  - `KLDivergence()`
  - `CosineSimilarity()`
  - `Etc.`
- *Metrics*
  - `CategoricalAccuracy()`
  - `SparseCategoricalAccuracy()`
  - `BinaryAccuracy()`
  - `AUC()`
  - `Precision()`
  - `Recall()`
  - Etc.

Throughout this book, you'll see concrete applications of many of these options.

3.6.5 Picking a loss function

Choosing the right loss function for the right problem is extremely important: your network will take any shortcut it can to minimize the loss. So if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted. Imagine a stupid, omnipotent AI trained via SGD, with this poorly chosen objective function: "Maximize the average well-being of all humans alive." To make its job easier, this AI might choose to kill all humans except a few and focus on the well-being of the remaining ones because average well-being isn't affected by how many humans are left. That might not be what you intended! Just remember that all neural networks you build will be just as ruthless in lowering their loss function, so choose the objective wisely, or you'll have to face unintended side effects.

Fortunately, when it comes to common problems such as classification, regression, and sequence prediction, there are simple guidelines you can follow to choose the

correct loss. For instance, you'll use binary crossentropy for a two-class classification problem, categorical crossentropy for a many-class classification problem, and so on. Only when you're working on truly new research problems will you have to develop your own loss functions. In the next few chapters, we'll detail explicitly which loss functions to choose for a wide range of common tasks.

3.6.6 Understanding the fit method

After `compile()` comes `fit()`. The `fit` method implements the training loop itself. Its key arguments are

- The *data* (inputs and targets) to train on. It will typically be passed either in the form of NumPy arrays or a TensorFlow `Dataset` object. You'll learn more about the `Dataset` API in the next chapters.
- The number of *epochs* to train for: how many times the training loop should iterate over the data passed.
- The batch size to use within each epoch of mini-batch gradient descent: the number of training examples considered to compute the gradients for one weight update step.

Listing 3.34 Calling `fit` with NumPy data

```
history = model.fit(
    inputs,
    targets,
    epochs=5,
    batch_size=128,
)
```

The input examples, as a NumPy array

The corresponding training targets, as a NumPy array

The training loop will iterate over the data 5 times.

The training loop will iterate over the data in batches of 128 examples.

The call to `fit` returns a `History` object. This object contains a `history` field, which is a dict mapping key, such as `"loss"` or specific metric names to the list of their per-epoch values:

```
>>> history.history
{"binary_accuracy": [0.855, 0.9565, 0.9555, 0.95, 0.951],
 "loss": [0.6573270302042366,
  0.07434618508815766,
  0.07687718723714351,
  0.07412414988875389,
  0.07617757616937161]}
```

3.6.7 Monitoring loss and metrics on validation data

The goal of machine learning is not to obtain models that perform well on the training data, which is easy—all you have to do is follow the gradient. The goal is to obtain models that perform well in general, particularly on data points that the model has never encountered before. Just because a model performs well on its training data doesn't mean it will perform well on data it has never seen! For instance, it's possible that your model could end up merely *memorizing* a mapping between your training samples and their targets, which would be useless for the task of predicting targets for data the model has never seen before. We'll go over this point in much more detail in the chapter 5.

To keep an eye on how the model does on new data, it's standard practice to reserve a subset of the training data as "validation data": you won't be training the model on this data, but you will use it to compute a loss value and metrics value. You do this by using the `validation_data` argument in `fit()`. Like the training data, the validation data could be passed as NumPy arrays or as a TensorFlow `Dataset` object.

Listing 3.35 Using the validation data argument

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
    loss=keras.losses.MeanSquaredError(),
    metrics=[keras.metrics.BinaryAccuracy()],
)

indices_permutation = np.random.permutation(len(inputs))
shuffled_inputs = inputs[indices_permutation]
shuffled_targets = targets[indices_permutation]

num_validation_samples = int(0.3 * len(inputs))
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]
model.fit(
    training_inputs,
    training_targets,
    epochs=5,
    batch_size=16,
    validation_data=(val_inputs, val_targets),
)
```

To avoid having samples from only one class in the validation data, shuffles the inputs and targets using a random indices permutation

Reserves 30% of the training inputs and targets for "validation." (We'll exclude these samples from training and reserve them to compute the "validation loss" and metrics.)

Training data, used to update the weights of the model

Validation data, used only to monitor the "validation loss" and metrics

The value of the loss on the validation data is called the *validation loss*, to distinguish it from the *training loss*. Note that it's essential to keep the training data and validation

data strictly separate: the purpose of validation is to monitor whether what the model is learning is actually useful on new data. If any of the validation data has been seen by the model during training, your validation loss and metrics will be flawed.

If you want to compute the validation loss and metrics after training is complete, you can call the `evaluate` method:

```
loss_and_metrics = model.evaluate(val_inputs, val_targets, batch_size=128)
```

`evaluate()` will iterate in batches (of size `batch_size`) over the data passed and return a list of scalars, where the first entry is the validation loss and the following entries are the validation metrics. If the model has no metrics, only the validation loss is returned (rather than a list).

3.6.8 Inference: Using a model after training

Once you've trained your model, you're going to want to use it to make predictions on new data. This is called *inference*. To do this, a naive approach would simply be to `__call__()` the model:

```
predictions = model(new_inputs)
```

Takes a NumPy array or a tensor for your current backend and returns a tensor for your current backend

However, this will process all inputs in `new_inputs` at once, which may not be feasible if you're looking at a lot of data (in particular, it may require more memory than your GPU has).

A better way to do inference is to use the `predict()` method. It will iterate over the data in small batches and return a NumPy array of predictions. And unlike `__call__()`, it can also process TensorFlow `Dataset` objects:

```
predictions = model.predict(new_inputs, batch_size=128)
```

Takes a NumPy array or a Dataset and returns a NumPy array

For instance, if we use `predict()` on some of our validation data with the linear model we trained earlier, we get scalar scores that correspond to the model's prediction for each input sample:

```
>>> predictions = model.predict(val_inputs, batch_size=128)
>>> print(predictions[:10])
```

```
[[0.3590725 ]
 [0.82706255]
 [0.74428225]
 [0.682058  ]
 [0.7312616 ]
 [0.6059811 ]
 [0.78046083]
 [0.025846  ]
 [0.16594526]
 [0.72068727]]
```

For now, this is all you need to know about Keras models. At this point, you are ready to move on to solving real-world machine problems with Keras, in the next chapter.

Summary

- TensorFlow, PyTorch, and JAX are three popular low-level frameworks for numerical computation and autodifferentiation. They all have their own way of doing things and their own strengths and weaknesses.
- Keras is a high-level API for building and training neural networks. It can be used with either TensorFlow, PyTorch, or JAX—just pick the backend you like best.
- The central class of Keras is the `Layer`. A layer encapsulates some weights and some computation. Layers are assembled into models.
- Before you start training a model, you need to pick an optimizer, a loss, and some metrics, which you specify via the `model.compile()` method.
- To train a model, you can use the `fit()` method, which runs mini-batch gradient descent for you. You can also use it to monitor your loss and metrics on validation data, a set of inputs that the model doesn't see during training.
- Once your model is trained, you can use the `model.predict()` method to generate predictions on new inputs.

4 Classification and regression

This chapter covers

- Your first examples of real-world machine learning workflows
- Handling binary and categorical classification problems
- Handling continuous regression problems

This chapter is designed to get you started with using neural networks to solve real problems. You'll consolidate the knowledge you gained from chapters 2 and 3, and you'll apply what you've learned to three new tasks covering the three most common use cases of neural networks—binary classification, categorical classification, and scalar regression:

- Classifying movie reviews as positive or negative (binary classification)
- Classifying news wires by topic (categorical classification)
- Estimating the price of a house, given real estate data (scalar regression)

These examples will be your first contact with end-to-end machine learning workflows: you'll get introduced to data preprocessing, basic model architecture principles, and model evaluation.

By the end of this chapter, you'll be able to use neural networks to handle simple classification and regression tasks over vector data. You'll then be ready to start building a more principled, theory-driven understanding of machine learning in chapter 5.

Classification and regression glossary

Classification and regression involve many specialized terms. You've come across some of them in earlier examples, and you'll see more of them in future chapters. They have precise, machine learning–specific definitions, and you should be familiar with them:

- *Sample or input*—One data point that goes into your model.
- *Prediction or output*—What comes out of your model.
- *Target*—The truth. What your model should ideally have predicted, according *to an external source of data.*
- *Prediction error or loss value*—A measure of the distance between your model's prediction and the target.
- *Classes*—A set of possible labels to choose from in a classification problem. For example, when classifying cat and dog pictures, "dog" and "cat" are the two classes.
- *Label*—A specific instance of a class annotation in a classification problem. For instance, if picture #1234 is annotated as containing the class "dog," then "dog" is a label of picture #1234.
- *Ground-truth or annotations*—All targets for a dataset, typically collected by humans.
- *Binary classification*—A classification task where each input sample should be categorized into two exclusive categories.
- *Categorical classification or multiclass classification*—A classification task where each input sample should be categorized into more than two categories: for instance, classifying handwritten digits.
- *Multilabel classification*—A classification task where each input sample can be assigned multiple labels. For instance, a given image may contain both a cat and a dog and should be annotated with both the "cat" label and the "dog" label. The number of labels per image is usually variable.
- *Scalar regression*—A task where the target is a continuous scalar value. Predicting house prices is a good example: the different target prices form a continuous space.
- *Vector regression*—A task where the target is a set of continuous values: for example, a continuous vector. If you're doing regression against multiple values (such as the coordinates of a bounding box in an image), then you're doing vector regression.
- *Mini-batch or just batch*—A small set of samples (typically between 8 and 128) that are processed simultaneously by the model. The number of samples is often a power of 2, to facilitate memory allocation on GPU. When training, a mini-batch is used to compute a single gradient-descent update applied to the weights of the model.

4.1 Classifying movie reviews: A binary classification example

Two-class classification, or binary classification, is one of the most common kinds of machine learning problem. In this example, you'll learn to classify movie reviews as positive or negative, based on the text content of the reviews.

4.1.1 The IMDb dataset

You'll work with the IMDb dataset: a set of 50,000 highly polarized reviews from the Internet Movie Database. They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.

Just like the MNIST dataset, the IMDb dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary. This enables us to focus on model building, training, and evaluation. In chapter 14, you'll learn how to process raw text input from scratch.

The following code will load the dataset (when you run it the first time, about 80 MB of data will be downloaded to your machine).

Listing 4.1 Loading the IMDb dataset

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000
)
```

The argument `num_words=10000` means you'll only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows you to work with vector data of manageable size. If we didn't set this limit, we'd be working with 88,585 unique words in the training data, which is unnecessarily large. Many of these words only occur in a single sample, and thus can't be meaningfully used for classification.

The variables `train_data` and `test_data` are NumPy arrays of reviews; each review is a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are NumPy arrays of 0s and 1s, where 0 stands for *negative* and 1 stands for *positive*:

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]
>>> train_labels[0]
1
```

Because you're restricting yourself to the top 10,000 most frequent words, no word index will exceed 10,000:

```
>>> max([max(sequence) for sequence in train_data])
9999
```

For kicks, let's quickly decode one of these reviews back to English words.

Listing 4.2 Decoding reviews back to text

```
word_index = imdb.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_review = " ".join(
    [reverse_word_index.get(i - 3, "?") for i in train_data[0]]
)
```

word_index is a dictionary mapping words to an integer index.

Reverses it, mapping integer indices to words

Decodes the review. Note that the indices are offset by 3 because 0, 1, and 2 are reserved indices for "padding," "start of sequence," and "unknown."

Let's take a look at what we got:

```
>>> decoded_review[:100]
? this film was just brilliant casting location scenery story direction everyone
```

Note that the leading `?` corresponds to a start token that has been prefixed to each review.

4.1.2 Preparing the data

You can't directly feed lists of integers into a neural network. They have all different lengths, while a neural network expects to process contiguous batches of data. You have to turn your lists into tensors. There are two ways to do that:

- Pad your lists so that they all have the same length, then turn them into an integer tensor of shape `(samples, max_length)`, and start your model with a layer capable of handling such integer tensors (the `Embedding` layer, which we'll cover in detail later in the book).
- *Multi-hot encode* your lists to turn them into vectors of 0s and 1s reflecting the presence or absence of all possible words. This would mean, for instance, turning the sequence `[8, 5]` into a 10,000-dimensional vector that would be all 0s except for indices 5 and 8, which would be 1s.

Let's go with the latter solution to vectorize the data. When done manually, the process looks like the following.

Listing 4.3 Encoding the integer sequences via multi-hot encoding

```
import numpy as np

def multi_hot_encode(sequences, num_classes):
    results = np.zeros((len(sequences), num_classes))
    for i, sequence in enumerate(sequences):
        results[i][sequence] = 1.0
    return results

x_train = multi_hot_encode(train_data, num_classes=10000)
x_test = multi_hot_encode(test_data, num_classes=10000)
```

Creates an all-zero matrix of shape (len(sequences), num_classes)

Sets specific indices of results[i] to 1s

Vectorized training data

Vectorized test data

Here's what the samples look like now:

```
>>> x_train[0]
array([ 0.,  1.,  1., ...,  0.,  0.,  0.])
```

In addition to vectorizing the input sequences, you should also vectorize their labels, which is straightforward. Our labels are already NumPy arrays, so just convert the type from ints to floats:

```
y_train = train_labels.astype("float32")
y_test = test_labels.astype("float32")
```

Now the data is ready to be fed into a neural network.

4.1.3 Building your model

The input data is vectors, and the labels are scalars (1s and 0s): this is one of the simplest problem setups you'll ever encounter. A type of model that performs well on such a problem is a plain stack of densely connected (`Dense`) layers with `relu` activations.

There are two key architecture decisions to be made about such a stack of `Dense` layers:

- How many layers to use
- How many units to choose for each layer

In chapter 5, you'll learn formal principles to guide you in making these choices. For the time being, you'll have to trust us with the following architecture choice:

- Two intermediate layers with 16 units each
- A third layer that will output the scalar prediction regarding the sentiment of the current review

Figure 4.1 shows what the model looks like. Here's the Keras implementation, similar to the MNIST example you saw previously.

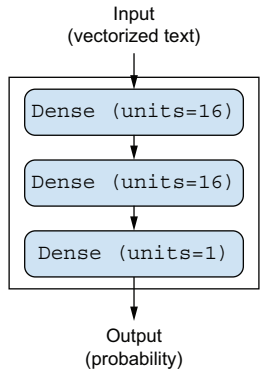


Figure 4.1 The three-layer model

Listing 4.4 Model definition

```
import keras
from keras import layers

model = keras.Sequential(
    [
        layers.Dense(16, activation="relu"),
        layers.Dense(16, activation="relu"),
        layers.Dense(1, activation="sigmoid"),
    ]
)
```

The first argument being passed to each `Dense` layer is the number of *units* in the layer: the dimensionality of representation space of the layer. You remember from chapters 2 and 3 that each such `Dense` layer with a `relu` activation implements the following chain of tensor operations:

```
output = relu(dot(input, W) + b)
```

Having 16 units means the weight matrix `W` will have shape `(input_dimension, 16)`: the dot product with `W` will project the input data onto a 16-dimensional representation space (and then you'll add the bias vector `b` and apply the `relu` operation). You can intuitively understand the dimensionality of your representation space as "how much freedom you're allowing the model to have when learning internal representations." Having more units (a higher-dimensional representation space) allows your model to learn more complex representations, but it makes the model more computationally expensive and may lead to learning unwanted patterns (patterns that will improve performance on the training data but not on the test data).

The intermediate layers use `relu` as their activation function, and the final layer uses a sigmoid activation to output a probability (a score between 0 and 1, indicating how likely the review is to be positive). A `relu` (rectified linear unit) is a function meant to zero-out negative values (see figure 4.2), whereas a sigmoid "squashes" arbitrary values into the `[0, 1]` interval (see figure 4.3), outputting something that can be interpreted as a probability.

What are activation functions, and why are they necessary?

Without an activation function like `relu` (also called a *non-linearity*), the `Dense` layer would consist of two linear operations—a dot product and an addition:

(continued)

```
output = dot(input, W) + b
```

So the layer could only learn *linear transformations* (affine transformations) of the input data: the *hypothesis space* of the layer would be the set of all possible linear transformations of the input data into a 16-dimensional space. Such a hypothesis space is too restricted and wouldn't benefit from multiple layers of representations because a deep stack of linear layers would still implement a linear operation: adding more layers wouldn't extend the hypothesis space (as you saw in chapter 2).

To get access to a much richer hypothesis space that would benefit from deep representations, you need a non-linearity or activation function. `relu` is the most popular activation function in deep learning, but there are many other candidates, which all come with similarly strange names: `prelu`, `elu`, and so on.

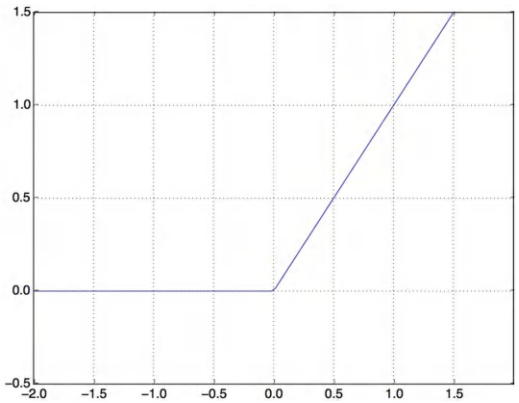


Figure 4.2 The rectified linear unit function

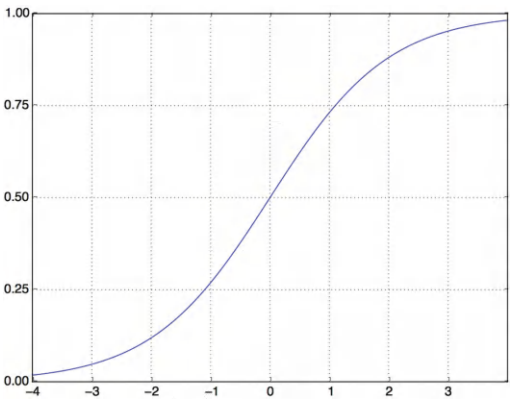


Figure 4.3 The sigmoid function

Finally, you need to choose a loss function and an optimizer. Because you're facing a binary classification problem and the output of your model is a probability (you end your model with a single-unit layer with a sigmoid activation), it's best to use the `binary_crossentropy` loss. It isn't the only viable choice: you could use, for instance, `mean_squared_error`. But crossentropy is usually the best choice when you're dealing with models that output probabilities. *Crossentropy* is a quantity from the field of information theory that measures the distance between probability distributions or, in this case, between the ground-truth distribution and your predictions.

As for the choice of the optimizer, we'll go with `adam`, which is usually a good default choice for virtually any problem.

Here's the step where you configure the model with the `adam` optimizer and the `binary_crossentropy` loss function. Note that you'll also monitor accuracy during training.

Listing 4.5 Compiling the model

```
model.compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
```

4.1.4 Validating your approach

As you learned in chapter 3, a deep learning model should never be evaluated on its training data—it's standard practice to use a "validation set" to monitor the accuracy of the model during training. Here, you'll create a validation set by setting apart 10,000 samples from the original training data.

You might ask, why not simply use the *test* data to evaluate the model? That seems like it would be easier. The reason is that you're going to want to use the results you get on the validation set to inform your next choices to improve training—for instance, your choice of what model size to use or how many epochs to train for. When you start doing this, your validation scores stop being an accurate reflection of the performance of the model on brand-new data, since the model has been deliberately modified to perform better on the validation data. It's good to keep around a set of never-before-seen samples that you can use to perform the final evaluation round in a completely unbiased way, and that's exactly what the test set is. We'll talk more about this in the next chapter.

Listing 4.6 Setting aside a validation set

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

You'll now train the model for 20 epochs (20 iterations over all samples in the training data), in mini-batches of 512 samples. At the same time, you'll monitor loss and accuracy on the 10,000 samples that you set apart. You do so by passing the validation data as the `validation_data` argument to `model.fit()`.

Listing 4.7 Training your model

```
history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=20,
    batch_size=512,
    validation_data=(x_val, y_val),
)
```

The validation_split argument

Instead of manually splitting out validation data from your training data and passing it as the `validation_data` argument, you can also use the `validation_split` argument in `fit()`. It specifies a fraction of the training data to use as validation data, like this:

```
history = model.fit(
    x_train,
    y_train,
    epochs=20,
    batch_size=512,
    validation_split=0.2,
)
```

In this example, 20% of the samples in the `x_train` and `y_train` arrays are being held out from training and used as validation data.

On CPU, this will take less than 2 seconds per epoch—training is over in 20 seconds. At the end of every epoch, there is a slight pause as the model computes its loss and accuracy on the 10,000 samples of the validation data.

Note that the call to `model.fit()` returns a `History` object, as you've seen in chapter 3. This object has a member `history`, which is a dictionary containing data about everything that happened during training. Let's look at it:

```
>>> history_dict = history.history
>>> history_dict.keys()
dict_keys(["accuracy", "loss", "val_accuracy", "val_loss"])
```

The dictionary contains four entries: one per metric that was being monitored during training and during validation. In the following two listings, let's use Matplotlib to plot the training and validation loss side by side (see figure 4.4), as well as the training and validation accuracy (see figure 4.5). Note that your own results may vary slightly due to a different random initialization of your model.

Listing 4.8 Plotting the training and validation loss

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict["loss"]
val_loss_values = history_dict["val_loss"]
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, "r--", label="Training loss")
plt.plot(epochs, val_loss_values, "b", label="Validation loss")
plt.title("[IMDB] Training and validation loss")
plt.xlabel("Epochs")
```

"b" is for "solid blue line."

"r--" is for "dashed red line."

```
plt.xticks(epochs)
plt.ylabel("Loss")
plt.legend()
plt.show()
```

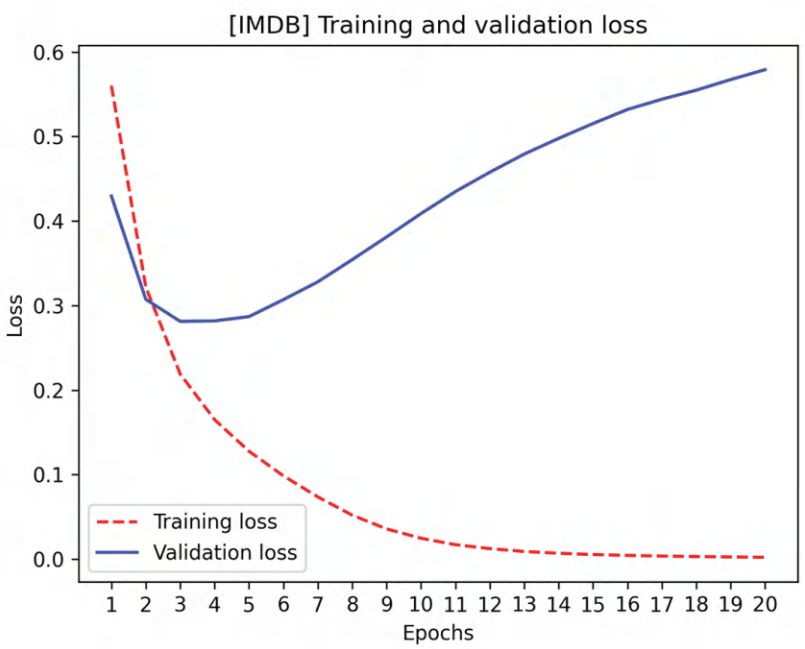


Figure 4.4 Training and validation loss

Listing 4.9 Plotting the training and validation accuracy

```
plt.clf()                                   # Clears the figure
acc = history_dict["accuracy"]
val_acc = history_dict["val_accuracy"]
plt.plot(epochs, acc, "r--", label="Training acc")
plt.plot(epochs, val_acc, "b", label="Validation acc")
plt.title("[IMDB] Training and validation accuracy")
plt.xlabel("Epochs")
plt.xticks(epochs)
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

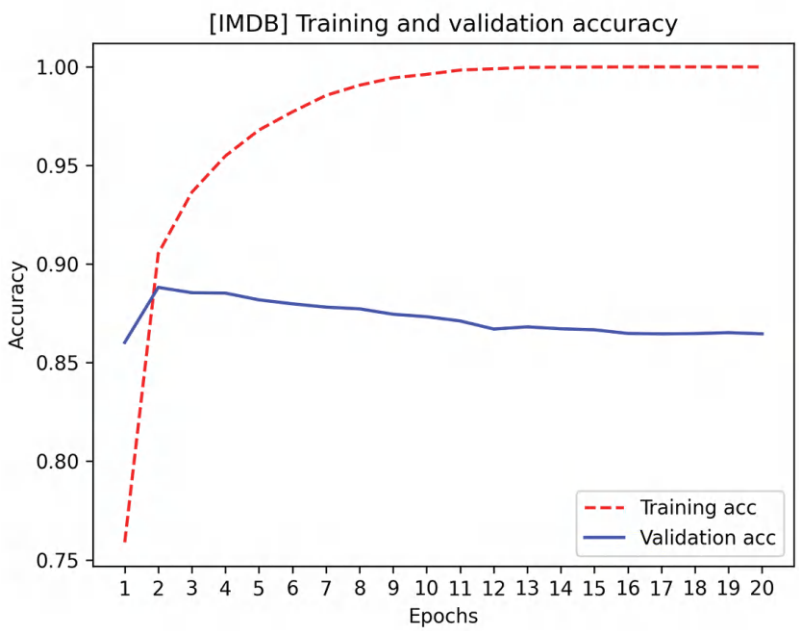


Figure 4.5 Training and validation accuracy

As you can see, the training loss decreases with every epoch, and the training accuracy increases with every epoch. That's what you would expect when running gradient-descent optimization—the quantity you're trying to minimize should be less with every iteration. But that isn't the case for the validation loss and accuracy: they seem to peak at the fourth epoch. This is an example of what we warned against earlier: a model that performs better on the training data isn't necessarily a model that will do better on data it has never seen before. In precise terms, what you're seeing is *overfitting*: after the fourth epoch, you're overoptimizing on the training data, and you end up learning representations that are specific to the training data and don't generalize to data outside of the training set.

In this case, to prevent overfitting, you could stop training after four epochs. In general, you can use a range of techniques to mitigate overfitting, which we'll cover in chapter 5.

Let's train a new model from scratch for four epochs and then evaluate it on the test data.

Listing 4.10 Training the model for four epochs

```
model = keras.Sequential(
    [
        layers.Dense(16, activation="relu"),
```

```
        layers.Dense(16, activation="relu"),
        layers.Dense(1, activation="sigmoid"),
    ]
)
model.compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

The final results are as follows:

```
>>> results
[0.2929924130630493, 0.88327999999999995]
```

The first number, 0.29, is the test loss, and the second number, 0.88, is the test accuracy.

This fairly naive approach achieves an accuracy of 88%. With state-of-the-art approaches, you should be able to get close to 95%.

4.1.5 Using a trained model to generate predictions on new data

After having trained a model, you'll want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the `predict` method, as you've learned in chapter 3:

```
>>> model.predict(x_test)
array([[ 0.98006207]
       [ 0.99758697]
       [ 0.99975556]
       ...,
       [ 0.82167041]
       [ 0.02885115]
       [ 0.65371346]], dtype=float32)
```

As you can see, the model is confident for some samples (0.99 or more, or 0.01 or less) but less confident for others (0.6, 0.4).

4.1.6 Further experiments

The following experiments will help convince you that the architecture choices you've made are all fairly reasonable, although there's still room for improvement:

- You used two representation layers before the final classification layer. Try using one or three representation layers and see how doing so affects validation and test accuracy.

- Try using layers with more units or fewer units: 32 units, 64 units, and so on.
- Try using the `mean_squared_error` loss function instead of `binary_crossentropy`.
- Try using the `tanh` activation (an activation that was popular in the early days of neural networks) instead of `relu`.

4.1.7 Wrapping up

Here's what you should take away from this example:

- You usually need to do quite a bit of preprocessing on your raw data to be able to feed it—as tensors—into a neural network. Sequences of words can be encoded as binary vectors, but there are other encoding options, too.
- Stacks of `Dense` layers with `relu` activations can solve a wide range of problems (including sentiment classification), and you'll use them frequently.
- In a binary classification problem (two output classes), your model should end with a `Dense` layer with one unit and a `sigmoid` activation: the output of your model should be a scalar between 0 and 1, encoding a probability.
- With such a scalar sigmoid output on a binary classification problem, the loss function you should use is `binary_crossentropy`.
- The `adam` optimizer is generally a good enough choice, whatever your problem. That's one less thing for you to worry about.
- As they get better on their training data, neural networks eventually start overfitting and end up obtaining increasingly worse results on data they've never seen before. Be sure to always monitor performance on data that is outside of the training set!

4.2 Classifying newswires: A multiclass classification example

In the previous section, you saw how to classify vector inputs into two mutually exclusive classes using a densely connected neural network. But what happens when you have more than two classes?

In this section, you'll build a model to classify Reuters newswires into 46 mutually exclusive topics. Because you have many classes, this problem is an instance of *multiclass classification*, and because each data point should be classified into only one category, the problem is more specifically an instance of *single-label, multiclass classification*. If each data point could belong to multiple categories (in this case, topics), you'd be facing a *multilabel, multiclass classification* problem.

4.2.1 The Reuters dataset

You'll work with the Reuters dataset, a set of short newswires and their topics, published by Reuters in 1986. It's a simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set.

Like IMDb and MNIST, the Reuters dataset comes packaged as part of Keras. Let's take a look.

Listing 4.11 Loading the Reuters dataset

```
from keras.datasets import reuters

(train_data, train_labels), (test_data, test_labels) = reuters.load_data(
    num_words=10000
)
```

As with the IMDb dataset, the argument `num_words=10000` restricts the data to the 10,000 most frequently occurring words found in the data.

You have 8,982 training examples and 2,246 test examples:

```
>>> len(train_data)
8982
>>> len(test_data)
2246
```

As with the IMDb reviews, each example is a list of integers (word indices):

```
>>> train_data[10]
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979,
3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]
```

Here's how you can decode it back to words, in case you're curious.

Listing 4.12 Decoding newswires back to text

```
word_index = reuters.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_newswire = " ".join(
    [reverse_word_index.get(i - 3, "?") for i in train_data[10]]
)
```

The indices are offset by 3 because 0, 1, and 2 are reserved indices for "padding," "start of sequence," and "unknown."

The label associated with an example is an integer between 0 and 45—a topic index:

```
>>> train_labels[10]
3
```

4.2.2 Preparing the data

You can vectorize the data with the exact same code as in the previous example.

Listing 4.13 Encoding the input data

```
x_train = multi_hot_encode(train_data, num_classes=10000)
x_test = multi_hot_encode(test_data, num_classes=10000)
```

Vectorized training data

Vectorized test data

To vectorize the labels, there are two possibilities: you can leave the labels untouched as integers, or you can use *one-hot encoding*. One-hot encoding is a widely used format for categorical data, also called *categorical encoding*. In this case, one-hot encoding of the labels consists of embedding each label as an all-zero vector with a 1 in the place of the label index. Here's an example.

Listing 4.14 Encoding the labels

```
def one_hot_encode(labels, num_classes=46):
    results = np.zeros((len(labels), num_classes))
    for i, label in enumerate(labels):
        results[i, label] = 1.0
    return results

y_train = one_hot_encode(train_labels)
y_test = one_hot_encode(test_labels)
```

Vectorized training labels

Vectorized test labels

Note that there is a built-in way to do this in Keras:

```
from keras.utils import to_categorical

y_train = to_categorical(train_labels)
y_test = to_categorical(test_labels)
```

4.2.3 Building your model

This topic classification problem looks similar to the previous movie review classification problem: in both cases, you're trying to classify short snippets of text. But there is a new constraint here: the number of output classes has gone from 2 to 46. The dimensionality of the output space is much larger.

In a stack of `Dense` layers like those you've been using, each layer can only access information present in the output of the previous layer. If one layer drops some information relevant to the classification problem, this information can never be

recovered by later layers: each layer can potentially become an information bottleneck. In the previous example, you used 16-dimensional intermediate layers, but a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information.

For this reason, you'll use larger intermediate layers. Let's go with 64 units.

Listing 4.15 Model definition

```
model = keras.Sequential(
    [
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(46, activation="softmax"),
    ]
)
```

There are two other things you should note about this architecture:

- You end the model with a `Dense` layer of size 46. This means for each input sample, the network will output a 46-dimensional vector. Each entry in this vector (each dimension) will encode a different output class.
- The last layer uses a `softmax` activation. You saw this pattern in the MNIST example. It means the model will output a *probability distribution* over the 46 different output classes—for every input sample, the model will produce a 46-dimensional output vector, where `output[i]` is the probability that the sample belongs to class `i`. The 46 scores will sum to 1.

The best loss function to use in this case is `categorical_crossentropy`. It measures the distance between two probability distributions—here, between the probability distribution outputted by the model and the true distribution of the labels. By minimizing the distance between these two distributions, you train the model to output something as close as possible to the true labels.

Like last time, we'll also monitor accuracy. However, accuracy is a bit of a crude metric in this case: if the model has the correct class as its second choice for a given sample, with an incorrect first choice, the model will still have an accuracy of zero on that sample—even though such a model would be much better than a random guess. A more nuanced metric in this case is top-k accuracy, such as top-3 or top-5 accuracy. It measures whether the correct class was among the top-k predictions of the model. Let's add top-3 accuracy to our model.

Listing 4.16 Compiling the model

```
top_3_accuracy = keras.metrics.TopKCategoricalAccuracy(
    k=3, name="top_3_accuracy"
```

```
)
model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy", top_3_accuracy],
)
```

4.2.4 *Validating your approach*

Let's set apart 1,000 samples in the training data to use as a validation set.

Listing 4.17 Setting aside a validation set

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]
y_val = y_train[:1000]
partial_y_train = y_train[1000:]
```

Now, let's train the model for 20 epochs.

Listing 4.18 Training the model

```
history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=20,
    batch_size=512,
    validation_data=(x_val, y_val),
)
```

And finally, let's display its loss and accuracy curves (see figures 4.6 and 4.7).

Listing 4.19 Plotting the training and validation loss

```
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, "r--", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.xticks(epochs)
plt.ylabel("Loss")
plt.legend()
plt.show()
```

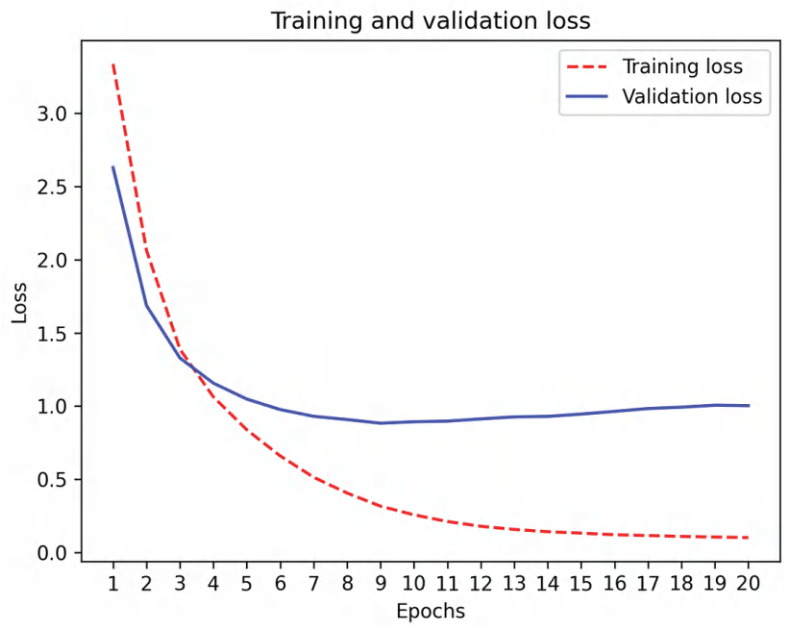


Figure 4.6 Training and validation loss

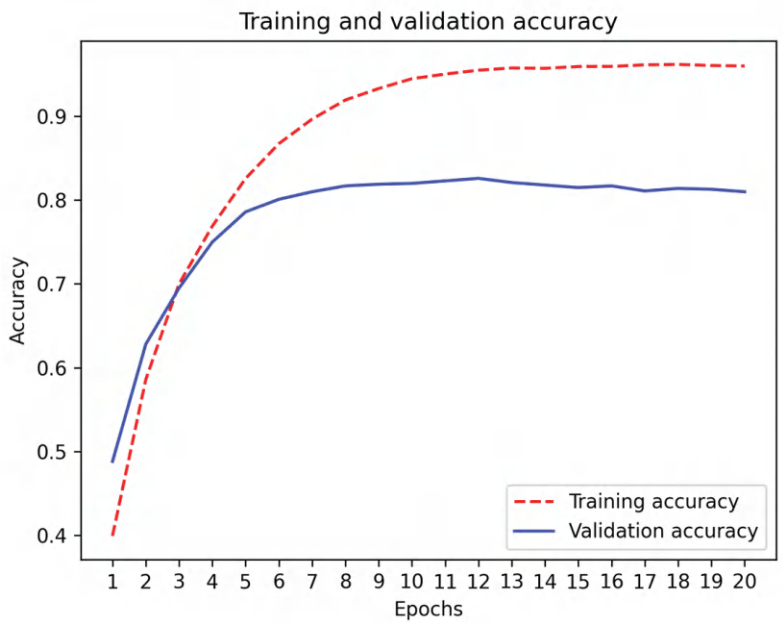


Figure 4.7 Training and validation accuracy

Listing 4.20 Plotting the training and validation accuracy

```
plt.clf()
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
plt.plot(epochs, acc, "r--", label="Training accuracy")
plt.plot(epochs, val_acc, "b", label="Validation accuracy")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.xticks(epochs)
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

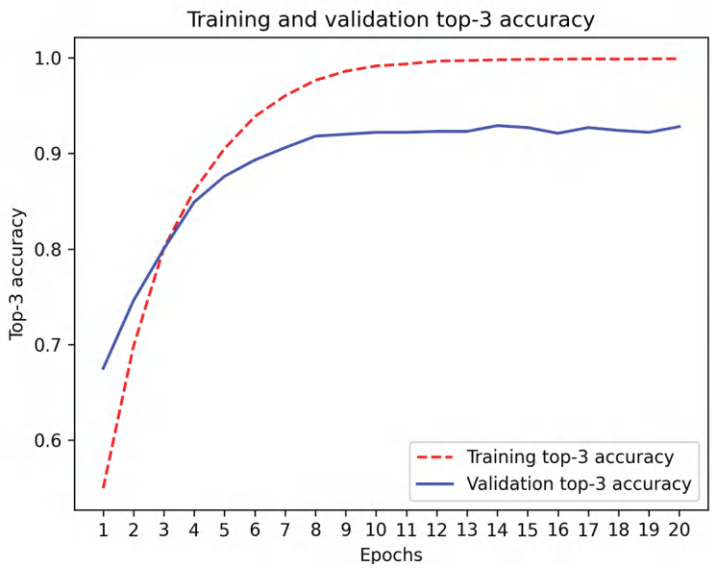


Figure 4.8 Training and validation top-3 accuracy

Listing 4.21 Plotting the training and validation top-3 accuracy

```
plt.clf()
acc = history.history["top_3_accuracy"]
val_acc = history.history["val_top_3_accuracy"]
plt.plot(epochs, acc, "r--", label="Training top-3 accuracy")
plt.plot(epochs, val_acc, "b", label="Validation top-3 accuracy")
plt.title("Training and validation top-3 accuracy")
plt.xlabel("Epochs")
plt.xticks(epochs)
plt.ylabel("Top-3 accuracy")
plt.legend()
plt.show()
```

The model begins to overfit after nine epochs. Let's train a new model from scratch for nine epochs and then evaluate it on the test set.

Listing 4.22 Retraining a model from scratch

```
model = ke ras.Sequential(
    [
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(46, activation="softmax"),
    ]
)
model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy"],
)
model.fit(
    x_train,
    y_train,
    epochs=9,
    batch_size=512,
)
results = model.evaluate(x_test, y_test)
```

Here are the final results:

```
>>> results
[0.9565213431445807, 0.79697239536954589]
```

This approach reaches an accuracy of approximately 80%. With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%. But in this case, we have 46 classes, and they may not be equally represented. What would be the accuracy of a random baseline? We could try quickly implementing one to check this empirically:

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> hits_array = np.array(test_labels == test_labels_copy)
>>> hits_array.mean()
0.18655387355298308
```

As you can see, a random classifier would score around 19% classification accuracy, so the results of our model seem pretty good in that light.

4.2.5 Generating predictions on new data

Calling the model's `predict` method on new samples returns a class probability distribution over all 46 topics for each sample. Let's generate topic predictions for all of the test data:

```
predictions = model.predict(x_test)
```

Each entry in "predictions" is a vector of length 46:

```
>>> predictions[0].shape
(46,)
```

The coefficients in this vector sum to 1, as they form a probability distribution:

```
>>> np.sum(predictions[0])
1.0
```

The largest entry is the predicted class—the class with the highest probability:

```
>>> np.argmax(predictions[0])
4
```

4.2.6 A different way to handle the labels and the loss

We mentioned earlier that another way to encode the labels would be to leave them untouched as integer tensors, like this:

```
y_train = train_labels
y_test = test_labels
```

The only thing this approach would change is the choice of the loss function. The loss function used in listing 4.22, `categorical_crossentropy`, expects the labels to follow a categorical encoding. With integer labels, you should use `sparse_categorical_crossentropy`:

```
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

This new loss function is still mathematically the same as `categorical_crossentropy`; it just has a different interface.

4.2.7 The importance of having sufficiently large intermediate layers

We mentioned earlier that because the final outputs are 46-dimensional, you should avoid intermediate layers with much fewer than 46 units. Now let's see what happens when you introduce an information bottleneck by having intermediate layers that are significantly less than 46-dimensional: for example, 4-dimensional.

Listing 4.23 A model with an information bottleneck

```
model = keras.Sequential(
    [
        layers.Dense(64, activation="relu"),
        layers.Dense(4, activation="relu"),
        layers.Dense(46, activation="softmax"),
    ]
)
model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy"],
)
model.fit(
    partial_x_train,
    partial_y_train,
    epochs=20,
    batch_size=128,
    validation_data=(x_val, y_val),
)
```

The model now peaks at approximately 71% validation accuracy, an 8% absolute drop. This drop is mostly due to the fact that you're trying to compress a lot of information (enough information to recover the separation hyperplanes of 46 classes) into an intermediate space that is too low-dimensional. The model is able to cram *most* of the necessary information into these 4-dimensional representations, but not all of it.

4.2.8 Further experiments

Like in the previous example, we encourage you to try out the following experiments to train your intuition about the kind of configuration decisions you have to make with such models:

- Try using larger or smaller layers: 32 units, 128 units, and so on.
- You used two intermediate layers before the final softmax classification layer. Now try using a single intermediate layer, or three intermediate layers.

4.2.9 Wrapping up

Here's what you should take away from this example:

- If you're trying to classify data points among *N* classes, your model should end with a `Dense` layer of size *N*.
- In a single-label, multiclass classification problem, your model should end with a `softmax` activation so that it will output a probability distribution over the *N* output classes.
- Categorical crossentropy is almost always the loss function you should use for such problems. It minimizes the distance between the probability distributions output by the model and the true distribution of the targets.
- There are two ways to handle labels in multiclass classification:
  - Encoding the labels via categorical encoding (also known as one-hot encoding) and using `categorical_crossentropy` as a loss function
  - Encoding the labels as integers and using the `sparse_categorical_crossentropy` loss function
- If you need to classify data into a large number of categories, you should avoid creating information bottlenecks in your model due to intermediate layers that are too small.

4.3 Predicting house prices: A regression example

The two previous examples were considered classification problems, where the goal was to predict a single discrete label of an input data point. Another common type of machine learning problem is *regression*, which consists of predicting a continuous value instead of a discrete label: for instance, predicting the temperature tomorrow given meteorological data, or predicting the time that a software project will take to complete given its specifications.

> **NOTE** Confusingly, logistic regression isn't a regression algorithm—it's a classification algorithm.

4.3.1 The California Housing Price dataset

You'll attempt to predict the median price of homes in different areas of California, based on data from the 1990 census.

Each data point in the dataset represents information about a "block group," a group of homes located in the same area. You can think of it as a district. This dataset has two versions, the "small" version with just 600 districts, and the "large" version with 20,640 districts. Let's use the small version, because real-world datasets can often be tiny, and you need to know how to handle such cases.

For each district, we know

- The longitude and latitude of the approximate geographic center of the area.
- The median age of houses in the district.
- The population of the district. The districts are pretty small: the average population is 1,425.5.
- The total number of households.
- The median income of those households.
- The total number of rooms in the district, across all homes located there. This is typically in the low thousands.
- The total number of bedrooms in the district.

That's eight variables in total (longitude and latitude count as two variables). The goal is to use these variables to predict the median value of the houses in the district. Let's get started by loading the data.

Listing 4.24 Loading the California Housing Prices dataset

```
from keras.datasets import california_housing

(train_data, train_targets), (test_data, test_targets) = (
    california_housing.load_data(version="small")
)
```

Make sure to pass version="small" to get the right dataset.

Let's look at the data:

```
>>> train_data.shape
(480, 8)
>>> test_data.shape
(120, 8)
```

As you can see, we have 480 training samples and 120 test samples, each with 8 numerical features. The targets are the median values of homes in the district considered, in dollars:

```
>>> train_targets
array([252300., 146900., 290900., ..., 140500., 217100.],
      dtype=float32)
```

The prices are between $60,000 and $500,000. If that sounds cheap, remember that this was in 1990, and these prices aren't adjusted for inflation.

4.3.2 Preparing the data

It would be problematic to feed into a neural network values that all take wildly different ranges. The model might be able to automatically adapt to such heterogeneous data, but it would definitely make learning more difficult. A widespread best practice to deal with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), you subtract the mean of the feature and divide by the standard deviation, so that the feature is centered around 0 and has a unit standard deviation. This is easily done in NumPy.

Listing 4.25 Normalizing the data

```
mean = train_data.mean(axis=0)
std = train_data.std(axis=0)
x_train = (train_data - mean) / std
x_test = (test_data - mean) / std
```

Note that the quantities used for normalizing the test data are computed using the training data. You should never use in your workflow any quantity computed on the test data, even for something as simple as data normalization.

In addition, we should also scale the targets. Our normalized inputs have their value in a small range close to 0, and our model's weights are initialized with small random values. This means that our model's prediction will also be small values when we start training. If the targets are in the range 60,000–500,000, the model is going to need very large weight values to output those. With a small learning rate, it would take a very long time to get there. The simplest fix is to divide all target values by 100,000, so that the smallest target becomes 0.6, and the largest becomes 5. We can then convert the model's predictions back to dollar values by multiplying them by 100,000 accordingly.

Listing 4.26 Scaling the targets

```
y_train = train_targets / 100000
y_test = test_targets / 100000
```

4.3.3 Building your model

Because so few samples are available, you'll use a very small model with two intermediate layers, each with 64 units. In general, the less training data you have, the worse overfitting will be, and using a small model is one way to mitigate overfitting.

Listing 4.27 Model definition

```
def get_model():
    model = keras.Sequential(
        [
```

Because you need to instantiate the same model multiple times, you use a function to construct it.

```
            layers.Dense(64, activation="relu"),
            layers.Dense(64, activation="relu"),
            layers.Dense(1),
        ]
    )
    model.compile(
        optimizer="adam",
        loss="mean_squared_error",
        metrics=["mean_absolute_error"],
    )
    return model
```

The model ends with a single unit and no activation: it will be a linear layer. This is a typical setup for scalar regression—a regression where you're trying to predict a single continuous value. Applying an activation function would constrain the range the output can take; for instance, if you applied a `sigmoid` activation function to the last layer, the model could only learn to predict values between 0 and 1. Here, because the last layer is purely linear, the model is free to learn to predict values in any range.

Note that you compile the model with the `mean_squared_error` loss function—*mean squared error*, the square of the difference between the predictions and the targets. This is a widely used loss function for regression problems.

You're also monitoring a new metric during training: *mean absolute error* (MAE). It's the absolute value of the difference between the predictions and the targets. For instance, an MAE of 0.5 on this problem would mean your predictions are off by $50,000 on average (remember the target scaling of factor 100,000).

4.3.4 Validating your approach using K-fold validation

To evaluate your model while you keep adjusting its parameters (such as the number of epochs used for training), you could split the data into a training set and a validation set, as you did in the previous examples. But because you have so few data points, the validation set would end up being very small (for instance, about 100 examples). As a consequence, the validation scores might change a lot depending on which data points you chose to use for validation and which you chose for training: the validation scores might have a high *variance* with regard to the validation split. This would prevent you from reliably evaluating your model.

The best practice in such situations is to use *K-fold* cross-validation (see figure 4.9). It consists of splitting the available data into K partitions (typically $K = 4$ or 5), instantiating K identical models, and training each one on $K - 1$ partitions while evaluating on the remaining partition. The validation score for the model used is then the average of the K validation scores obtained. In terms of code, this is straightforward.

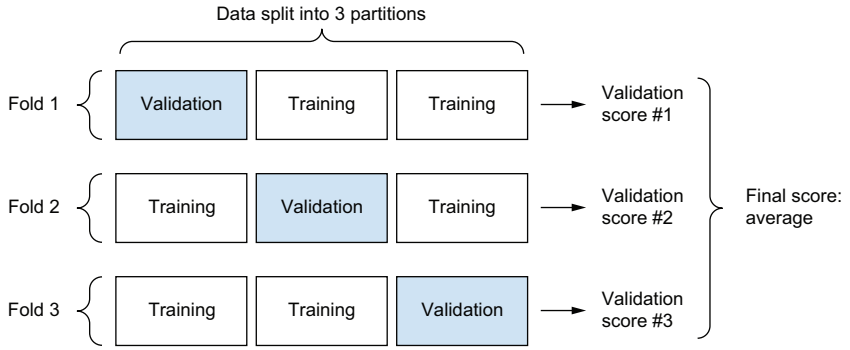


Figure 4.9 Three-fold cross-validation

Listing 4.28 K-fold validation

```
k = 4
num_val_samples = len(x_train) // k
num_epochs = 50
all_scores = []
for i in range(k):
    print(f"Processing fold #{i + 1}")
    fold_x_val = x_train[i * num_val_samples : (i + 1) * num_val_samples]
    fold_y_val = y_train[i * num_val_samples : (i + 1) * num_val_samples]
    fold_x_train = np.concatenate(
        [x_train[: i * num_val_samples], x_train[(i + 1) * num_val_samples :]],
        axis=0,
    )
    fold_y_train = np.concatenate(
        [y_train[: i * num_val_samples], y_train[(i + 1) * num_val_samples :]],
        axis=0,
    )
    model = get_model()
    model.fit(
        fold_x_train,
        fold_y_train,
        epochs=num_epochs,
        batch_size=16,
        verbose=0,
    )
    scores = model.evaluate(fold_x_val, fold_y_val, verbose=0)
    val_loss, val_mae = scores
    all_scores.append(val_mae)
```

Prepares the training data: data from all other partitions

Prepares the validation data: data from partition #k

Builds the Keras model (already compiled)

Trains the model

Evaluates the model on the validation data

Running this with `num_epochs = 50` yields the following results:

```
>>> [round(value, 3) for value in all_scores]
[0.298, 0.349, 0.232, 0.305]
>>> round(np.mean(all_scores), 3)
0.296
```

The different runs do indeed show meaningfully different validation scores, from 0.232 to 0.349. The average (0.296) is a much more reliable metric than any single score—that's the entire point of K-fold cross-validation. In this case, you're off by $29,600 on average, which is significant considering that the prices range from $60,000 to $500,000.

Let's try training the model a bit longer: 200 epochs. To keep a record of how well the model does at each epoch, you'll modify the training loop to save the per-epoch validation score log.

Listing 4.29 Saving the validation logs at each fold

```
k = 4
num_val_samples = len(x_train) // k
num_epochs = 200
all_mae_histories = []
for i in range(k):
    print(f"Processing fold #{i + 1}")
    fold_x_val = x_train[i * num_val_samples : (i + 1) * num_val_samples]
    fold_y_val = y_train[i * num_val_samples : (i + 1) * num_val_samples]
    fold_x_train = np.concatenate(
        [x_train[: i * num_val_samples], x_train[(i + 1) * num_val_samples :]],
        axis=0,
    )
    fold_y_train = np.concatenate(
        [y_train[: i * num_val_samples], y_train[(i + 1) * num_val_samples :]],
        axis=0,
    )
    model = get_model()
    history = model.fit(
        fold_x_train,
        fold_y_train,
        validation_data=(fold_x_val, fold_y_val),
        epochs=num_epochs,
        batch_size=16,
        verbose=0,
    )
    mae_history = history.history["val_mean_absolute_error"]
    all_mae_histories.append(mae_history)
```

Prepares the training data: data from all other partitions

Prepares the validation data: data from partition #k

Builds the Keras model (already compiled)

Trains the model

You can then compute the average of the per-epoch mean absolute error (MAE) scores for all folds.

Listing 4.30 Building the history of successive mean K-fold validation scores

```
average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)
]
```

Let's plot this; see figure 4.10.

Listing 4.31 Plotting validation scores

```
epochs = range(1, len(average_mae_history) + 1)
plt.plot(epochs, average_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()
```

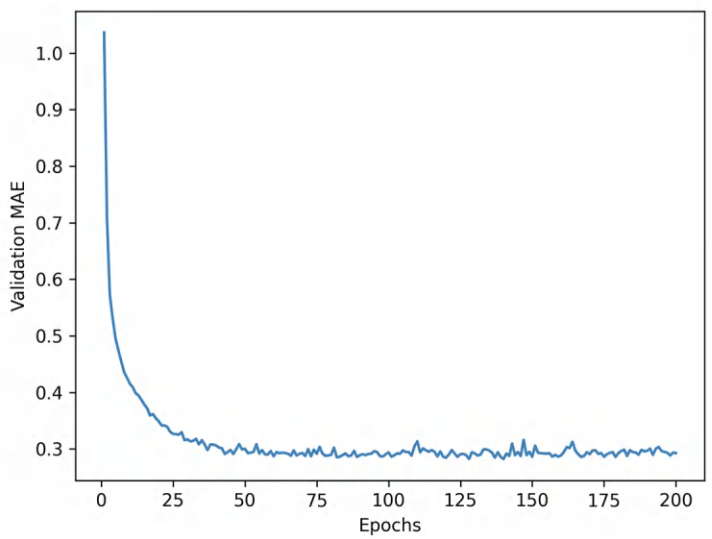


Figure 4.10 Validation MAE by epoch

It may be a little difficult to read the plot due to a scaling issue: the validation MAE for the first few epochs is dramatically higher than the values that follow. Let's omit the first 10 data points, which are on a different scale than the rest of the curve.

Listing 4.32 Plotting validation scores, excluding the first 10 data points

```
truncated_mae_history = average_mae_history[10:]
epochs = range(10, len(truncated_mae_history) + 10)
```

```
plt.plot(epochs, truncated_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()
```

According to this plot (see figure 4.11), validation MAE stops improving significantly after 120–140 epochs (this number includes the 10 epochs we omitted). Past that point, you start overfitting.

Once you're finished tuning other parameters of the model (in addition to the number of epochs, you could also adjust the size of the intermediate layers), you can train a final production model on all of the training data, with the best parameters, and then look at its performance on the test data.

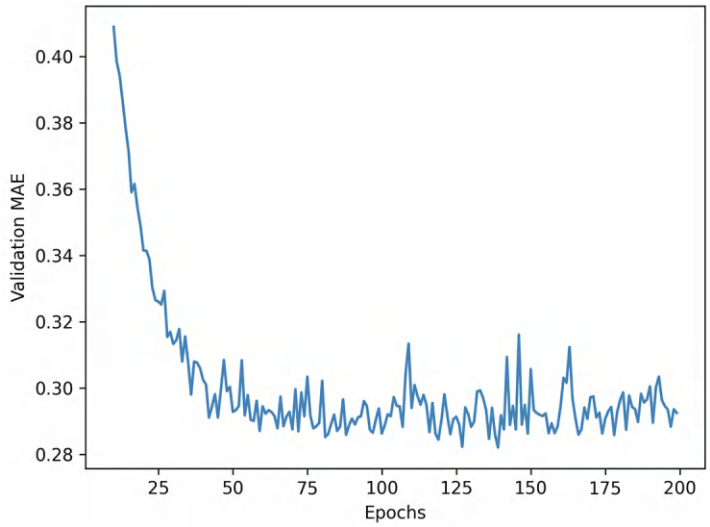


Figure 4.11 Validation MAE by epoch, excluding the first 10 data points

Listing 4.33 Training the final model

Gets a fresh, compiled model

Trains it on the entirety of the data

```
model = get_model()
model.fit(x_train, y_train, epochs=130, batch_size=16, verbose=0)
test_mean_squared_error, test_mean_absolute_error = model.evaluate(
    x_test, y_test
)
```

Here's the final result:

```
>>> round(test_mean_absolute_error, 3)
0.31
```

We're still off by about $31,000 on average.

4.3.5 Generating predictions on new data

When calling `predict()` on our binary classification model, we retrieved a scalar score between 0 and 1 for each input sample. With our multiclass classification model, we retrieved a probability distribution over all classes for each sample. Now, with this scalar regression model, `predict()` returns the model's guess for the sample's price in hundreds of thousands of dollars:

```
>>> predictions = model.predict(x_test)
>>> predictions[0]
array([2.834494], dtype=float32)
```

The first district in the test set is predicted to have an average home price of about $283,000.

4.3.6 Wrapping up

Here's what you should take away from this scalar regression example:

- Regression is done using a different loss function than what we used for classification. Mean squared error (MSE) is a loss function commonly used for regression.
- Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally, the concept of accuracy doesn't apply for regression. A common regression metric is MAE.
- When features in the input data have values in different ranges, each feature should be scaled independently as a preprocessing step.
- When there is little data available, using K-fold validation is a great way to reliably evaluate a model.
- When little training data is available, it's preferable to use a small model with few intermediate layers (typically only one or two), in order to avoid severe overfitting.

Summary

- The three most common kinds of machine learning tasks on vector data are binary classification, multiclass classification, and scalar regression. Each task uses different loss functions:
  - `binary_crossentropy` for binary classification
  - `categorical_crossentropy` for multiclass classification
  - `mean_squared_error` for scalar regression
- You'll usually need to preprocess raw data before feeding it into a neural network.
- When your data has features with different ranges, scale each feature independently as part of preprocessing.
- As training progresses, neural networks eventually begin to overfit and obtain worse results on never-before-seen data.
- If you don't have much training data, use a small model with only one or two intermediate layers, to avoid severe overfitting.
- If your data is divided into many categories, you may cause information bottlenecks if you make the intermediate layers too small.
- When you're working with little data, K-fold validation can help reliably evaluate your model.

5 Fundamentals of machine learning

This chapter covers

- Understanding the tension between generalization and optimization, the fundamental issue in machine learning
- Evaluation methods for machine learning models
- Best practices to improve model fitting
- Best practices to achieve better generalization

After the three practical examples in chapter 4, you should be starting to feel familiar with how to approach classification and regression problems using neural networks, and you've witnessed the central problem of machine learning: overfitting. This chapter will formalize some of your new intuition about machine learning into a solid conceptual framework, highlighting the importance of accurate model evaluation and the balance between training and generalization.

5.1 Generalization: The goal of machine learning

In the three examples presented in chapter 4—predicting movie reviews, topic classification, and house-price regression—we split the data into a training set, a

validation set, and a test set. The reason not to evaluate the models on the same data they were trained on quickly became evident: after just a few epochs, performance on never-before-seen data started diverging from performance on the training data, which always improves as training progresses. The models started to *overfit*. Overfitting happens in every machine-learning problem.

The fundamental issue in machine learning is the tension between optimization and generalization. *Optimization* refers to the process of adjusting a model to get the best performance possible on the training data (the *learning* in *machine learning*), whereas *generalization* refers to how well the trained model performs on data it has never seen before. The goal of the game is to get good generalization, of course, but you don't control generalization; you can only fit the model to its training data. If you do that *too well*, overfitting kicks in and generalization suffers.

But what causes overfitting? How can we achieve good generalization?

5.1.1 Underfitting and overfitting

For all models you've seen in the previous chapter, performance on the held-out validation data initially improved as training went on and then inevitably peaked after a while. This pattern (illustrated in figure 5.1) is universal. You'll see it with any model type and any dataset.

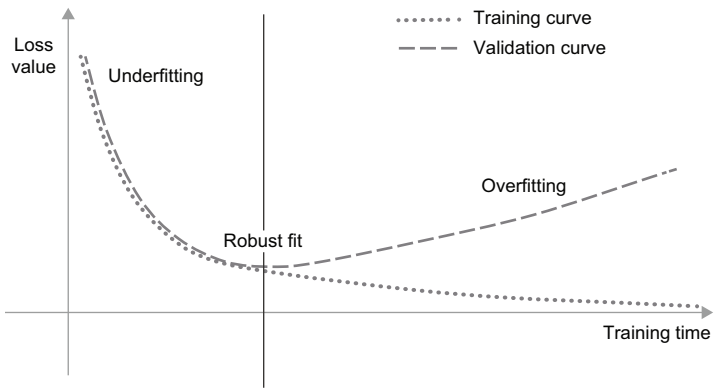


Figure 5.1 Canonical overfitting behavior

At the beginning of training, optimization and generalization are correlated: the lower the loss on training data, the lower the loss on test data. While this is happening, your model is said to be *underfit*: there is still progress to be made; the network hasn't yet modeled all relevant patterns in the training data. But after a certain number of iterations on the training data, generalization stops improving, and validation metrics stall and then begin to degrade: the model is starting to overfit. That is, it's beginning to

learn patterns that are specific to the training data but that are misleading or irrelevant when it comes to new data.

Overfitting is particularly likely to occur when your data is noisy, if it involves uncertainty, or if it includes rare features. Let's look at concrete examples.

NOISY TRAINING DATA

In real-world datasets, it's fairly common for some inputs to be invalid. Perhaps a MNIST digit could be an all-black image, for instance—or something like figure 5.2.

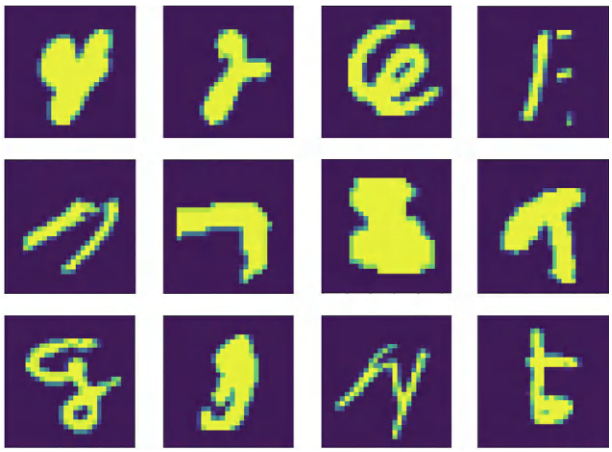


Figure 5.2 Some pretty weird MNIST training samples

What are these? We don't know either. But they're all part of the MNIST training set. What's even worse, however, is having perfectly valid inputs that end up mislabeled, like those shown in figure 5.3.

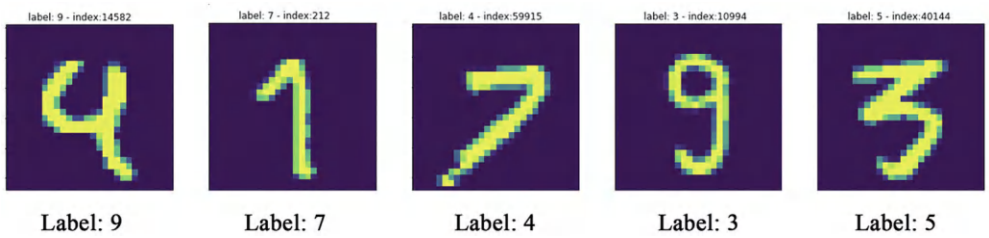


Figure 5.3 Mislabeled MNIST training samples

If a model goes out of its way to incorporate such outliers, its generalization performance will degrade, as shown in figure 5.4. For instance, a 4 that looks very close to the mislabeled 4 in figure 5.3 may end up getting classified as a 9.

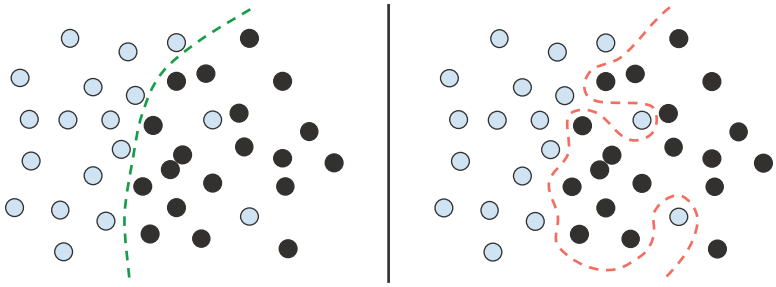


Figure 5.4 Dealing with outliers: robust fit vs. overfitting

AMBIGUOUS FEATURES

Not all data noise comes from inaccuracies—even perfectly clean and neatly labeled data can be noisy when the problem involves uncertainty and ambiguity (see figure 5.5). In classification tasks, it is often the case that some regions of the input feature space are associated with multiple classes at the same time. Let's say you're developing a model that takes an image of a banana and predicts whether the banana is unripened, ripe, or rotten. These categories have no objective boundaries, so the same picture might be classified as either unripened or ripe by different human labelers. Similarly, many problems involve randomness. You could use atmospheric pressure data to predict whether it will rain tomorrow, but the exact same measurements may be followed sometimes by rain, sometimes by a clear sky—with some probability.

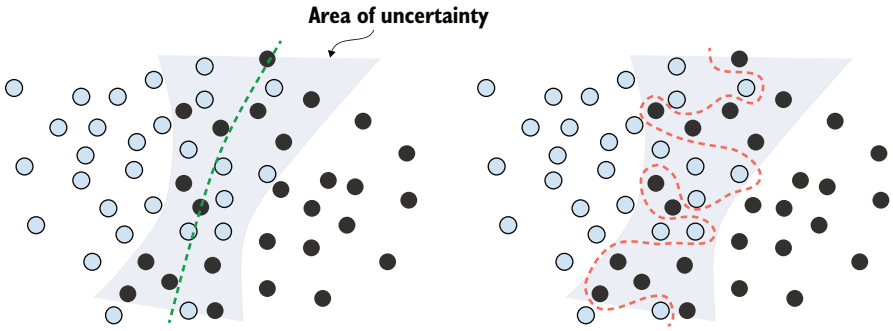


Figure 5.5 Robust fit vs. overfitting giving an ambiguous area of the feature space

A model could overfit to such probabilistic data by being too confident about ambiguous regions of the feature space, like in figure 5.6. A more robust fit would ignore individual data points and look at the bigger picture.

RARE FEATURES AND SPURIOUS CORRELATIONS

If you've only ever seen two orange tabby cats in your life, and they both happened to be terribly antisocial, you might infer that orange tabby cats are generally likely to be antisocial. That's overfitting: if you had been exposed to a wider variety of cats, including more orange ones, you'd have learned that cat color is not well correlated with character.

Likewise, machine learning models trained on datasets that include rare feature values are highly susceptible to overfitting. In a sentiment classification task, if the word "cherimoya" (a fruit native to the Andes) only appears in one text in the training data, and this text happens to be negative in sentiment, a poorly regularized model might put a very high weight on this word and always classify new texts that mention cherimoyas as negative, whereas, objectively, there's nothing negative about the cherimoya.[1]

Importantly, a feature value doesn't need to occur only a couple of times to lead to spurious correlations. Consider a word that occurs in 100 samples in your training data, and that's associated with a positive sentiment 54% of the time and with a negative sentiment 46% of the time. That difference may well be a complete statistical fluke, yet your model is likely to learn to use that feature for its classification task. This is one of the most common sources of overfitting.

Here's a striking example. Take MNIST. Create a new training set by concatenating 784 white noise dimensions to the existing 784 dimensions of the data—so half of the data is now noise. For comparison, also create an equivalent dataset by concatenating 784 all-zeros dimensions. Our concatenation of meaningless features does not at all affect the information content of the data: we're only adding irrelevant data points. Human classification accuracy wouldn't be affected by these transformations at all.

Listing 5.1 Adding white noise channels or all-zeros channels to MNIST

```
from keras.datasets import mnist
import numpy as np

(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

train_images_with_noise_channels = np.concatenate(
    [train_images, np.random.random((len(train_images), 784))], axis=1
)

train_images_with_zeros_channels = np.concatenate(
    [train_images, np.zeros((len(train_images), 784))], axis=1
)
```

Now, let's train the model from chapter 2 on both of these training sets.

[1] Mark Twain even called it "the most delicious fruit known to men."

Listing 5.2 Training the same model on MNIST data with noise channels or all-zero channels

```
import keras
from keras import layers

def get_model():
    model = keras.Sequential(
        [
            layers.Dense(512, activation="relu"),
            layers.Dense(10, activation="softmax"),
        ]
    )
    model.compile(
        optimizer="adam",
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model

model = get_model()
history_noise = model.fit(
    train_images_with_noise_channels,
    train_labels,
    epochs=10,
    batch_size=128,
    validation_split=0.2,
)

model = get_model()
history_zeros = model.fit(
    train_images_with_zeros_channels,
    train_labels,
    epochs=10,
    batch_size=128,
    validation_split=0.2,
)
```

Listing 5.3 Plotting a validation accuracy comparison

```
import matplotlib.pyplot as plt

val_acc_noise = history_noise.history["val_accuracy"]
val_acc_zeros = history_zeros.history["val_accuracy"]
epochs = range(1, 11)
plt.plot(
    epochs,
    val_acc_noise,
    "b-",
    label="Validation accuracy with noise channels",
)
plt.plot(
    epochs,
```

```
    val_acc_zeros,
    "r--",
    label="Validation accuracy with zeros channels",
)
plt.title("Effect of noise channels on validation accuracy")
plt.xlabel("Epochs")
plt.xticks(epochs)
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

Despite the data holding the same information in both cases, the validation accuracy of the model trained with noise channels ends up about one percentage point lower—purely through the influence of spurious correlations (figure 5.6). The more noise channels you might add, the further accuracy would degrade.

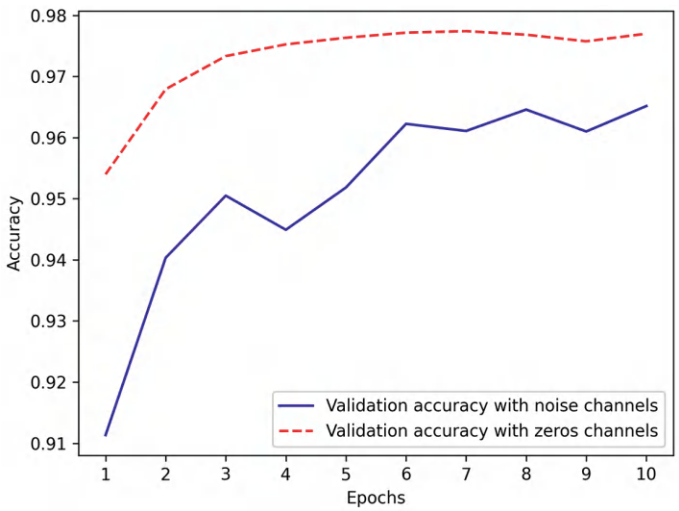


Figure 5.6 Effect of noise channels on validation accuracy

Noisy features inevitably lead to overfitting. As such, in cases where you aren't sure whether the features you have are informative or distracting, it's common to do *feature selection* before training. Restricting the IMDB data to the top 10,000 most common words was a crude form of feature selection, for instance. The typical way to do feature selection is to compute some usefulness score for each feature available—a measure of how informative the feature is with respect to the task, such as the mutual information between the feature and the labels—and only keep features that are above some threshold. Doing this would filter out the white noise channels in the preceding example.

5.1.2 The nature of generalization in deep learning

A remarkable fact about deep learning models is that they can be trained to fit anything, as long as they have enough representational power.

Don't believe me? Try shuffling the order of the MNIST labels and train a model on that. Even though there is no relationship whatsoever between the inputs and the shuffled labels, the training loss goes down just fine, even with a relatively small model. Naturally, the validation loss does not improve at all over time, since there is no possibility of generalization in this setting.

Listing 5.4 Fitting an MNIST model with randomly shuffled labels

```
(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

random_train_labels = train_labels[:]
np.random.shuffle(random_train_labels)

model = keras.Sequential(
    [
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)
model.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
model.fit(
    train_images,
    random_train_labels,
    epochs=100,
    batch_size=128,
    validation_split=0.2,
)
```

Copies train_labels

In fact, you don't even need to do this with MNIST data—you could just generate white noise inputs and random labels. You could fit a model on that, too, as long as it has enough parameters. It would just end up memorizing specific inputs, much like a Python dictionary.

If this is the case, then why do deep learning models generalize at all? Shouldn't they just learn an ad hoc mapping between training inputs and targets, like a fancy `dict`? What expectation can we have that this mapping will work for new inputs?

As it turns out, the nature of generalization in deep learning has rather little to do with deep learning models themselves and much to do with the structure of information in the real world. Let's take a look at what's really going on here.

THE MANIFOLD HYPOTHESIS

The input to an MNIST classifier (before preprocessing) is a 28 × 28 array of integers between 0 and 255. The total number of possible input values is thus 256 to the power of 784—much greater than the number of atoms in the universe. However, very few of these inputs would look like valid MNIST samples: actual handwritten digits only occupy a tiny *subspace* of the parent space of all possible 28 × 28 `uint8` arrays. What's more, this subspace isn't just a set of points sprinkled at random in the parent space: it is highly structured.

First, the subspace of valid handwritten digits is *continuous*: if you take a sample and modify it a little, it will still be recognizable as the same handwritten digit. Further, all samples in the valid subspace are *connected* by smooth paths that run through the subspace. This means that if you take two random MNIST digits A and B, there exists a sequence of "intermediate" images that morph A into B, such that two consecutive digits are very close to each other (see figure 5.7). Perhaps there will be a few ambiguous shapes close to the boundary between two classes, but even these shapes would still look very digit-like.

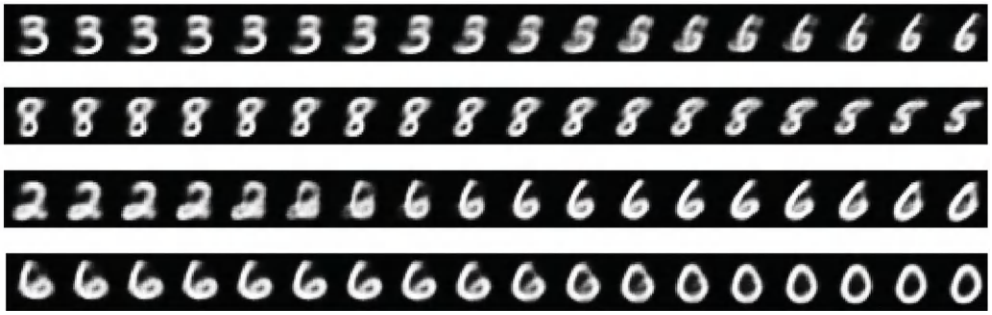


Figure 5.7 Different MNIST digits gradually morphing into one another, showing that the space of handwritten digits forms a "manifold." This image was generated using code from chapter 17.

In technical terms, you would say that handwritten digits form a *manifold* within the space of possible 28 × 28 `uint8` arrays. That's a big word, but the concept is pretty intuitive. A manifold is a lower-dimensional subspace of some parent space that is locally similar to a linear (Euclidean) space. For instance, a smooth curve in the plane is a 1D manifold within a 2D space because for every point of the curve, you can draw a tangent (the curve can be approximated by a line in every point). A smooth surface within a 3D space is a 2D manifold. And so on.

More generally, the *manifold hypothesis* posits that all natural data lies on a low-dimensional manifold within the high-dimensional space where it is encoded. That's a pretty strong statement about the structure of information in the universe. As far as we know, it's accurate, and it's the reason why deep learning works. It's true for MNIST

digits, as well as for human faces, tree morphology, the sounds of the human voice, and even natural language.

The manifold hypothesis implies

- Machine learning models only have to fit relatively simple, low-dimensional, highly structured subspaces within their potential input space (latent manifolds).
- Within one of these manifolds, it's always possible to *interpolate* between two inputs—that is, morph one into another via a continuous path along which all points fall on the manifold.

The ability to interpolate between samples is the key to understanding generalization in deep learning.

Interpolation as a source of generalization

If you work with data points that can be interpolated, you can start making sense of points you've never seen before by relating them to other points that lie close on the manifold. In other words, you can make sense of the *totality* of the space using only a *sample* of the space. You can use interpolation to fill in the blanks.

Note that interpolation on the latent manifold is different from linear interpolation in the parent space, as illustrated in figure 5.8. For instance, the average of pixels between two MNIST digits is usually not a valid digit.

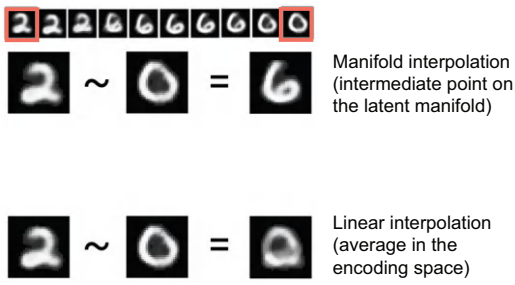


Figure 5.8 Difference between linear interpolation and interpolation on the latent manifold. Every point on the latent manifold of digits is a valid digit, but the average of two digits usually isn't.

Crucially, while deep learning achieves generalization via interpolation on a learned approximation of the data manifold, it would be a mistake to assume that interpolation is *all* there is to generalization. It's the tip of the iceberg. Interpolation can only help you make sense of things that are very close to what you've seen before: it enables *local generalization*. But remarkably, humans deal with extreme novelty all the time, and they do just fine. You don't need to be trained in advance on countless examples of every situation you'll ever have to encounter. Every single one of your days is different from any day you've experienced before, and different from any day experienced by anyone since the dawn of humanity. You can switch between spending a week in NYC, a week in Shanghai, and a week in Bangalore without requiring thousands of lifetimes of learning and rehearsal for each city.

Humans are capable of *extreme generalization*, which is enabled by cognitive mechanisms other than interpolation—abstraction, symbolic models of the world, reasoning, logic, common sense, innate priors about the world—what we generally call *reason*, as

opposed to intuition and pattern recognition. The latter are largely interpolative in nature, but the former isn't. Both are essential to intelligence. We'll talk more about this in chapter 19.

WHY DEEP LEARNING WORKS

Remember the crumpled paper ball metaphor from chapter 2? A sheet of paper represents a 2D manifold within 3D space (figure 5.9). A deep learning model is a tool for uncrumpling paper balls—that is, for disentangling latent manifolds.

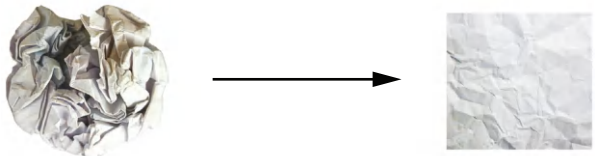


Figure 5.9 Uncrumpling a complicated manifold of data

A deep learning model is basically a very high-dimensional curve. The curve is smooth and continuous (with additional constraints on its structure, originating from model architecture priors) because it needs to be differentiable. And that curve is fitted to data points via gradient descent—smoothly and incrementally. *By construction*, deep learning is about taking a big, complex curve—a manifold—and incrementally adjusting its parameters until it fits some training data points.

The curve involves enough parameters that it could fit anything. Indeed, if you let your model train for long enough, it will effectively end up purely memorizing its training data and won't generalize at all. However, the data you're fitting to isn't made of isolated points sparsely distributed across the underlying space. Your data forms a highly structured, low-dimensional manifold within the input space—that's the manifold hypothesis. And because fitting your model curve to this data happens gradually and smoothly over time, as gradient descent progresses, there will be an intermediate point during training at which the model roughly approximates the natural manifold of the data, as you can see in figure 5.10.

Moving along the curve learned by the model at that point will come close to moving along the actual latent manifold of the data. As such, the model will be capable of making sense of never-before-seen inputs via interpolation between training inputs.

Besides the trivial fact that they have sufficient representational power, there are a few properties of deep learning models that make them particularly well suited to learning latent manifolds:

- Deep learning models implement a smooth, continuous mapping from their inputs to their outputs. It has to be smooth and continuous because it must be differentiable, by necessity (you couldn't do gradient descent otherwise). This smoothness helps approximate latent manifolds, which follow the same properties.

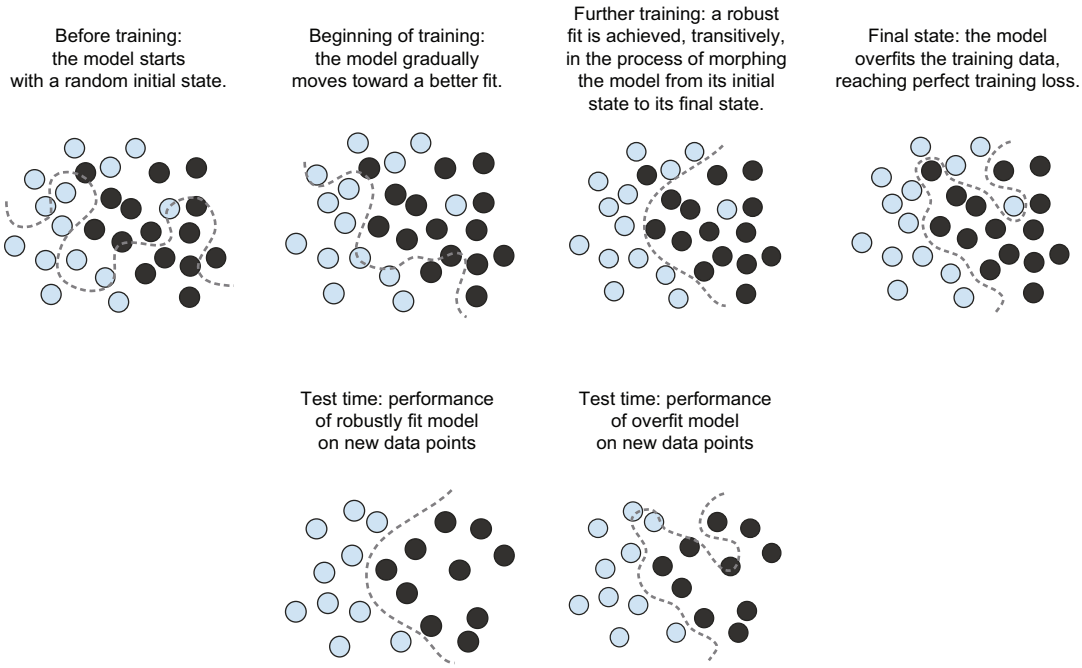


Figure 5.10 Going from a random model to an overfit model and achieving a robust fit as an intermediate state

- Deep learning models tend to be structured in a way that mirrors the "shape" of the information in their training data (via architecture priors). This is the case in particular for image-processing models (see chapters 8–12) and sequence-processing models (see chapter 13). More generally, deep neural networks structure their learned representations in a hierarchical and modular way, which echoes the way natural data is organized.

Training data is paramount

While deep learning is indeed well suited to manifold learning, the power to generalize is more a consequence of the natural structure of your data than a consequence of any property of your model. You'll only be able to generalize if your data forms a manifold where points can be interpolated. The more informative and the less noisy your features are, the better you will be able to generalize, since your input space will be simpler and better structured. Data curation and feature engineering are essential to generalization.

Further, because deep learning is curve fitting, for a model to perform well, *it needs to be trained on a dense sampling of its input space*. A "dense sampling" in this context means that the training data should densely cover the entirety of the input data manifold (see figure 5.11). This is especially true near decision boundaries. With a sufficiently dense sampling, it becomes possible to make sense of new inputs by interpolating between

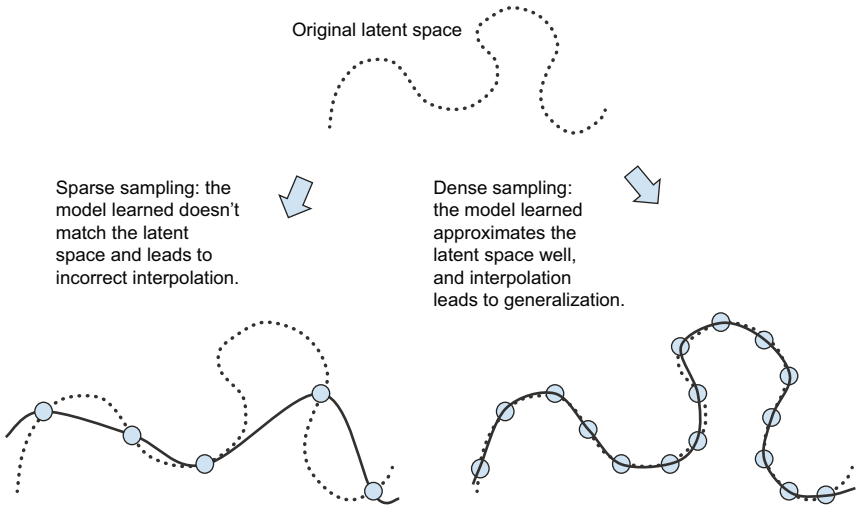


Figure 5.11 A dense sampling of the input space is necessary to learn a model capable of accurate generalization.

past training inputs, without having to use common-sense, abstract reasoning, or external knowledge about the world—all things that machine learning models have no access to.

As such, you should always keep in mind that the best way to improve a deep learning model is to train it on more data or better data (of course, adding overly noisy or inaccurate data will harm generalization). A denser coverage of the input data manifold will yield a model that generalizes better. You should never expect a deep learning model to perform anything more than crude interpolation between its training samples, and thus, you should do everything you can to make interpolation as easy as possible. The only thing you will find in a deep learning model is what you put into it: the priors encoded in its architecture and the data it was trained on.

When getting more data isn't possible, the next-best solution is to modulate the quantity of information that your model is allowed to store, or to add constraints on the smoothness of the model curve. If a network can only afford to memorize a small number of patterns, or very regular patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well. The process of fighting overfitting this way is called *regularization*. We'll review regularization techniques in depth in section 5.4.4.

Before you can start tweaking your model to help it generalize better, you need a way to assess how your model is currently doing. In the following section, you'll learn about how you can monitor generalization during model development: model evaluation.

5.2 Evaluating machine-learning models

You can only control what you can observe. Since your goal is to develop models that can successfully generalize to new data, it's essential to be able to reliably measure the generalization power of your model. In this section, we'll formally introduce the different ways you can evaluate machine learning models. You've already seen most of them in action in the previous chapter.

5.2.1 Training, validation, and test sets

Evaluating a model always boils down to splitting the available data into three sets: training, validation, and test. You train on the training data and evaluate your model on the validation data. Once your model is ready for prime time, you test it one final time on the test data, which is meant to be as similar as possible to production data. Then you can deploy the model in production.

You may ask, why not have two sets: a training set and a test set? You'd train on the training data and evaluate on the test data. Much simpler!

The reason is that developing a model always involves tuning its configuration: for example, choosing the number of layers or the size of the layers (called the *hyperparameters* of the model, to distinguish them from the *parameters*, which are the network's weights). You do this tuning by using as a feedback signal the performance of the model on the validation data. In essence, this tuning is a form of *learning*: a search for a good configuration in some parameter space. As a result, tuning the configuration of the model based on its performance on the validation set can quickly result in *overfitting to the validation set*, even though your model is never directly trained on it.

Central to this phenomenon is the notion of *information leaks*. Every time you tune a hyperparameter of your model based on the model's performance on the validation set, some information about the validation data leaks into the model. If you do this only once, for one parameter, then very few bits of information will leak, and your validation set will remain reliable to evaluate the model. But if you repeat this many times—running one experiment, evaluating on the validation set, and modifying your model as a result—then you'll leak an increasingly significant amount of information about the validation set into the model.

At the end of the day, you'll end up with a model that performs artificially well on the validation data because that's what you optimized it for. You care about performance on completely new data, not the validation data, so you need to use a completely different, never-before-seen dataset to evaluate the model: the test dataset. Your model shouldn't have had access to *any* information about the test set, even indirectly. If anything about the model has been tuned based on test set performance, then your measure of generalization will be flawed.

Splitting your data into training, validation, and test sets may seem straightforward, but there are a few advanced ways to do it that can come in handy when little data is available. Let's review three classic evaluation recipes: simple hold-out validation, K-fold validation, and iterated K-fold validation with shuffling. We'll also

talk about the use of common-sense baselines to check that your training is going somewhere.

SIMPLE HOLD-OUT VALIDATION

Set apart some fraction of your data as your test set. Train on the remaining data, and evaluate on the test set. As you saw in the previous sections, to prevent information leaks, you shouldn't tune your model based on the test set, and therefore you should *also* reserve a validation set.

Schematically, hold-out validation looks like figure 5.12. The following listing shows a simple implementation.

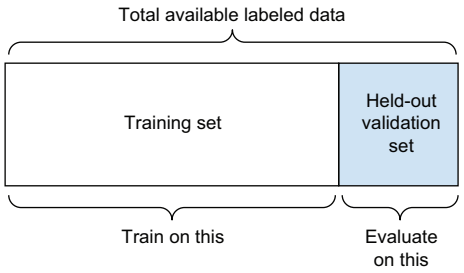


Figure 5.12 Simple hold-out validation split

Listing 5.5 Hold-out validation (note that labels are omitted for simplicity)

```
num_validation_samples = 10000
np.random.shuffle(data)
validation_data = data[:num_validation_samples]
training_data = data[num_validation_samples:]
model = get_model()
model.fit(training_data, ...)
validation_score = model.evaluate(validation_data, ...)

...

model = get_model()
model.fit(
    np.concatenate([training_data, validation_data]),
    ...,
)
test_score = model.evaluate(test_data, ...)
```

Defines the training set

Shuffling the data is usually appropriate.

Defines the validation set

Trains a model on the training data and evaluates it on the validation data

At this point, you can tune your model, retrain it, evaluate it, tune it again, and so on.

Once you've tuned your hyperparameters, it's common to train your final model from scratch on all non-test data available.

This is the simplest evaluation protocol, and it suffers from one flaw: if little data is available, then your validation and test sets may contain too few samples to be statistically representative of the data at hand. This is easy to recognize: if different random shuffling rounds of the data before splitting end up yielding very different measures of model performance, then you're having this issue. K-fold validation and iterated K-fold validation are two ways to address this, as discussed next.

K-FOLD VALIDATION

With this approach, you split your data into `K` partitions of equal size. For each partition `i`, train a model on the remaining `K - 1` partitions and evaluate it on partition `i`.

Your final score is then the averages of the K scores obtained. This method is helpful when the performance of your model shows significant variance based on your train/test split. Like hold-out validation, this method doesn't exempt you from using a distinct validation set for model calibration.

Schematically, K-fold cross-validation looks like figure 5.13. Listing 5.6 shows a simple implementation.

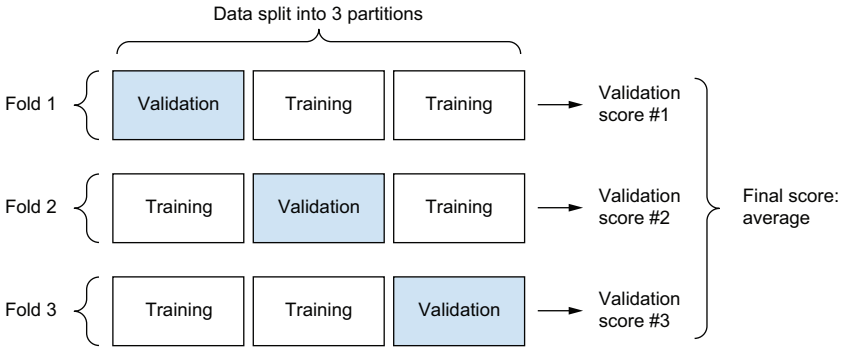


Figure 5.13 Three-fold validation

Listing 5.6 K-fold cross-validation (note that labels are omitted for simplicity)

```
k = 3
num_validation_samples = len(data) // k
np.random.shuffle(data)
validation_scores = []
for fold in range(k):
    validation_data = data[
        num_validation_samples * fold : num_validation_samples * (fold + 1)
    ]
    training_data = np.concatenate(
        data[: num_validation_samples * fold],
        data[num_validation_samples * (fold + 1) :],
    )
    model = get_model()
    model.fit(training_data, ...)
    validation_score = model.evaluate(validation_data, ...)
    validation_scores.append(validation_score)
validation_score = np.average(validation_scores)
model = get_model()
model.fit(data, ...)
test_score = model.evaluate(test_data, ...)
```

Selects the validation-data partition

Uses the remainder of the data as training data

Creates a brand-new instance of the model (untrained)

Trains the final model on all non-test data available

Validation score: average of the validation scores of the k folds

ITERATED K-FOLD VALIDATION WITH SHUFFLING

This one is for situations in which you have relatively little data available and you need to evaluate your model as precisely as possible. I've found it to be extremely helpful in Kaggle competitions. It consists of applying K-fold validation multiple times, shuffling the data every time before splitting it `K` ways. The final score is the average of the scores obtained at each run of K-fold validation. Note that you end up training and evaluating `P * K` models (where `P` is the number of iterations you use), which can be very expensive.

5.2.2 Beating a common-sense baseline

Besides the different evaluation protocols you have available, one last thing you should know about is the use of common-sense baselines.

Training a deep learning model is a bit like pressing a button that launches a rocket in a parallel world. You can't hear it or see it. You can't observe the manifold learning process—it's happening in a space with thousands of dimensions, and even if you projected it to 3D, you couldn't interpret it. The only feedback you have is your validation metrics—like an altitude meter on your invisible rocket.

A particularly important point is to be able to tell whether you're getting off the ground at all. What was the altitude you started at? Your model seems to have an accuracy of 15%, is that any good? Before you start working with a dataset, you should always pick a trivial baseline that you'll try to beat. If you cross that threshold, you'll know you're doing something right: your model is actually using the information in the input data to make predictions that generalize—you can keep going. This baseline could be performance of a random classifier, or the performance of the simplest non-machine learning technique you can imagine.

For instance, in the MNIST digit-classification example, a simple baseline would be a validation accuracy greater than 0.1 (random classifier); in the IMDB example, it would be a validation accuracy greater than 0.5. In the Reuters example, it would be around 0.18–0.19, due to class imbalance. If you have a binary classification problem where 90% of samples belong to class A and 10% belong to class B, then a classifier that always predicts A already achieves 0.9 in validation accuracy, and you'll need to do better than that.

Having a common sense baseline you can refer to is essential when you're getting started on a problem no one has solved before. If you can't beat a trivial solution, your model is worthless—perhaps you're using the wrong model or perhaps the problem you're tackling can't even be approached with machine learning in the first place. Time to go back to the drawing board.

5.2.3 Things to keep in mind about model evaluation

Keep an eye out for the following when you're choosing an evaluation protocol:

- *Data representativeness*—You want both your training set and test set to be representative of the data at hand. For instance, if you're trying to classify images of digits, and you're starting from an array of samples where the samples are

ordered by their class, taking the first 80% of the array as your training set and the remaining 20% as your test set will result in your training set containing only classes 0–7, whereas your test set contains only classes 8–9. This seems like a ridiculous mistake, but it's surprisingly common. For this reason, you usually should *randomly shuffle* your data before splitting it into training and test sets.

- *The arrow of time*—If you're trying to predict the future given the past (for example, tomorrow's weather, stock movements, and so on), you should not randomly shuffle your data before splitting it because doing so will create a *temporal leak*: your model will effectively be trained on data from the future. In such situations, you should always make sure all data in your test set is *posterior* to the data in the training set.
- *Redundancy in your data*—If some data points in your data appear twice (fairly common with real-world data), then shuffling the data and splitting it into a training set and a validation set will result in redundancy between the training and validation sets. In effect, you'll be testing on part of your training data, which is the worst thing you can do! Make sure your training set and validation set are disjoint.

Having a reliable way to evaluate the performance of your model is how you'll be able to monitor the tension at the heart of machine learning—between optimization and generalization, underfitting and overfitting.

5.3 Improving model fit

To achieve the perfect fit, you must first overfit. Since you don't know in advance where the boundary lies, you must cross it to find it. Thus, your initial goal as you start working on a problem is to achieve a model that shows some generalization power, and that is able to overfit. Once you have such a model, you'll focus on refining generalization by fighting overfitting.

There are three common problems you'll encounter at this stage:

- Training doesn't get started: your training loss doesn't go down over time.
- Training gets started just fine, but your model doesn't meaningfully generalize: you can't beat the common-sense baseline you set.
- Training and validation loss both go down over time, and you can beat your baseline, but you don't seem to be able to overfit, which indicates you're still underfitting.

Let's see how you can address these issues to achieve the first big milestone of a machine learning project: getting a model that has some generalization power (it can beat a trivial baseline) and is able to overfit.

5.3.1 Tuning key gradient descent parameters

Sometimes, training doesn't get started or stalls too early. Your loss is stuck. This is *always* something you can overcome: remember that you can fit a model to random

data. Even if nothing about your problem makes sense, you should *still* be able to train something—if only by memorizing the training data.

When this happens, it's always a problem with the configuration of the gradient descent process: your choice of optimizer, the distribution of initial values in the weights of your model, your learning rate, or your batch size. All these parameters are interdependent, and as such, it is usually sufficient to tune the learning rate and the batch size while maintaining the rest of the parameters constant.

Let's look at a concrete example: let's train the MNIST model from chapter 2 with an inappropriately large learning rate, of value 1.

Listing 5.7 Training an MNIST model with an incorrectly high learning rate

```
(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

model = keras.Sequential(
    [
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=1.0),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
model.fit(
    train_images, train_labels, epochs=10, batch_size=128, validation_split=0.2
)
```

The model quickly reaches a training and validation accuracy in the 20% to 40% range, but cannot get past that. Let's try to lower the learning rate to a more reasonable value of `1e-2`.

Listing 5.8 The same model with a more appropriate learning rate

```
model = keras.Sequential(
    [
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=1e-2),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

```
model.fit(
    train_images, train_labels, epochs=10, batch_size=128, validation_split=0.2
)
```

The model is now able to train.

If you find yourself in a similar situation, try

- Lowering or increasing the learning rate. A learning rate that is too high may lead to updates that vastly overshoot a proper fit, like in the previous example, and a learning rate that is too low may make training so slow that it appears to stall.
- Increasing the batch size. A batch with more samples will lead to gradients that are more informative and less noisy (lower variance).

You will, eventually, find a configuration that gets training started.

5.3.2 Using better architecture priors

You have a model that fits, but for some reason your validation metrics aren't improving at all. They remain no better than what a random classifier would achieve: your model trains, but doesn't generalize. What's going on?

This is perhaps the worst machine learning situation you can find yourself in. It indicates that *something is fundamentally wrong with your approach*, and it may not be easy to tell what. Here are some tips.

First, it may be that the input data you're using simply doesn't contain sufficient information to predict your targets: the problem as formulated is not solvable. This is what happened earlier when we tried to fit an MNIST model where the labels were shuffled: the model would train just fine, but validation accuracy would stay stuck at 10%, because it was plainly impossible to generalize with such a dataset.

It may also be that the kind of model you're using is not suited for the problem at hand. For instance, in chapter 13, you'll see an example of a timeseries prediction problem where a densely connected architecture isn't able to beat a trivial baseline, whereas a more appropriate recurrent architecture does manage to generalize well. Using a model that makes the right assumptions about the problem is essential to achieve generalization: you should use the right architecture priors.

In the following chapters, you'll learn about the best architectures to use for a variety of data modalities—images, text, timeseries, and so on. In general, you should always make sure to read up on architecture best practices for the kind of task you're attacking—chances are you're not the first person to attempt it.

5.3.3 Increasing model capacity

If you manage to get to a model that fits, where validation metrics are going down, and that seems to achieve at least some level of generalization power, congratulations: you're almost there. Next, you need to get your model to start overfitting.

Consider the following small model—a simple logistic regression—trained on MNIST pixels.

Listing 5.9 A simple logistic regression on MNIST

```
model = keras.Sequential([layers.Dense(10, activation="softmax")])
model.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
history_small_model = model.fit(
    train_images, train_labels, epochs=20, batch_size=128, validation_split=0.2
)
```

You get loss curves that look like this (see figure 5.14):

```
import matplotlib.pyplot as plt

val_loss = history_small_model.history["val_loss"]
epochs = range(1, 21)
plt.plot(epochs, val_loss, "b-", label="Validation loss")
plt.title("Validation loss for a model with insufficient capacity")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

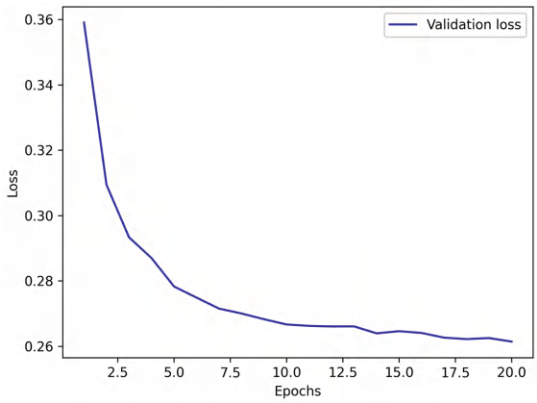


Figure 5.14 Effect of insufficient model capacity on loss curves

Validation metrics seem to stall or to improve very slowly, instead of peaking and reversing course. The validation loss goes to 0.26 and just stays there. You can fit, but

you can't clearly overfit, even after many iterations over the training data. You're likely to encounter similar curves often in your career.

Remember that it should always be possible to overfit. Much like the problem "the training loss doesn't go down," this is an issue that can always be solved. If you can't seem to be able to overfit, it's likely a problem with the *representational power* of your model: you're going to need a bigger model, one with more *capacity*—that is, able to store more information. You can increase representational power by adding more layers, using bigger layers (layers with more parameters), or using kinds of layers that are more appropriate for the problem at hand (better architecture priors).

Let's try training a bigger model, one with two intermediate layers with 128 units each:

```
model = keras.Sequential(
    [
        layers.Dense(128, activation="relu"),
        layers.Dense(128, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)
model.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
history_large_model = model.fit(
    train_images,
    train_labels,
    epochs=20,
    batch_size=128,
    validation_split=0.2,
)
```

The training curves now look exactly like they should: the model fits fast and starts overfitting after eight epochs (see figure 5.15).

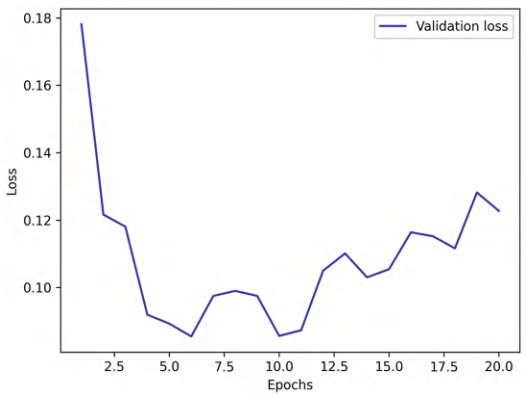


Figure 5.15 Validation loss for a model with appropriate capacity

Note that while it is standard to work with models that are way overparameterized for the problem at hand, there can definitely be such a thing as *too much* memorization capacity. You'll know your model is too large if it starts overfitting right away. Here's what happens for an MNIST model with three intermediate layers with 2,048 units each (see figure 5.16):

```
model = keras.Sequential(
    [
        layers.Dense(2048, activation="relu"),
        layers.Dense(2048, activation="relu"),
        layers.Dense(2048, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)
model.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
history_very_large_model = model.fit(
    train_images,
    train_labels,
    epochs=20,
    batch_size=32,
    validation_split=0.2,
)
```

When training larger models, you can reduce the batch size to limit memory consumption.

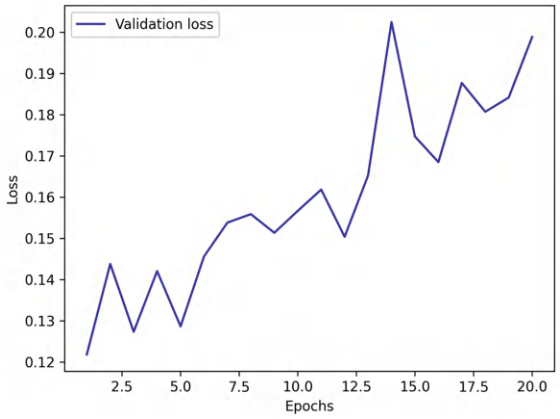


Figure 5.16 Effect of excessive model capacity on validation loss

5.4 Improving generalization

Once your model has shown to have some generalization power and to be able to overfit, it's time to switch your focus toward maximizing generalization.

5.4.1 Dataset curation

You've already learned that generalization in deep learning originates from the latent structure of your data. If your data makes it possible to smoothly interpolate between samples, then you will be able to train a deep learning model that generalizes. If your problem is overly noisy or fundamentally discrete, like, say, list sorting, deep learning will not help you. Deep learning is curve fitting, not magic.

As such, it is essential that you make sure that you're working with an appropriate dataset. Spending more effort and money on data collection almost always yields a much greater return on investment than spending the same on developing a better model:

- Make sure you have enough data. Remember that you need a *dense sampling* of the input-cross-output space. More data will yield a better model. Sometimes, problems that seem impossible at first become solvable with a larger dataset.
- Minimize labeling errors—visualize your inputs to check for anomalies, and proofread your labels.
- Clean your data and deal with missing values (we cover this in the next chapter).
- If you have many features and you aren't sure which ones are actually useful, do feature selection.

A particularly important way you can improve the generalization potential of your data is *feature engineering*. For most machine learning problems, *feature engineering* is a key ingredient for success. Let's take a look.

5.4.2 Feature engineering

Feature engineering is the process of using your own knowledge about the data and about the machine learning algorithm at hand (in this case, a neural network) to make the algorithm work better by applying hardcoded (non-learned) transformations to the data before it goes into the model. In many cases, it isn't reasonable to expect a machine learning model to be able to learn from completely arbitrary data. The data needs to be presented to the model in a way that will make the model's job easier.

Let's look at an intuitive example. Suppose you're trying to develop a model that can take as input an image of a clock and can output the time of day (see figure 5.17). If you choose to use the raw pixels of the image as input data, then you have a difficult machine learning problem on your hands. You'll need a convolutional neural network to solve it, and you'll have to expend quite a bit of computational resources to train the network.

But if you already understand the problem at a high level (you understand how humans read time on a clock face), then you can come up with much better input features for a machine learning algorithm: for instance, it's easy to write a five-line Python script to follow the black pixels of the clock hands and output the `(x, y)` coordinates of the tip of each hand. Then a simple machine learning algorithm can learn to associate these coordinates with the appropriate time of day.

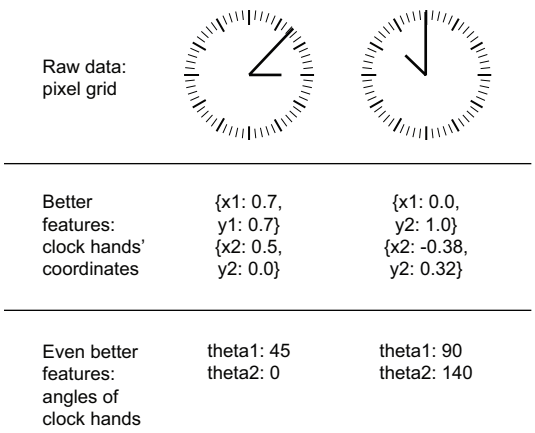


Figure 5.17 Feature engineering for reading the time on a clock

You can go even further: do a coordinate change and express the `(x, y)` coordinates as polar coordinates with regard to the center of the image. Your input will become the angle `theta` of each clock hand. At this point, your features are making the problem so easy that no machine learning is required; a simple rounding operation and dictionary lookup are enough to recover the approximate time of day.

That's the essence of feature engineering: making a problem easier by expressing it in a simpler way. Make the latent manifold smoother, simpler, and better organized. It usually requires understanding the problem in depth.

Before deep learning, feature engineering used to be the most important part of the machine learning workflow because classical shallow algorithms didn't have hypothesis spaces rich enough to learn useful features by themselves. The way you presented the data to the algorithm was absolutely critical to its success. For instance, before convolutional neural networks became successful on the MNIST digit-classification problem, solutions were typically based on hardcoded features such as the number of loops in a digit image, the height of each digit in an image, a histogram of pixel values, and so on.

Fortunately, modern deep learning removes the need for most feature engineering because neural networks are capable of automatically extracting useful features from raw data. Does this mean you don't have to worry about feature engineering as long as you're using deep neural networks? No, for two reasons:

- Good features still allow you to solve problems more elegantly while using fewer resources. For instance, it would be ridiculous to solve the problem of reading a clock face using a convolutional neural network.
- Good features let you solve a problem with far less data. The ability of deep-learning models to learn features on their own relies on having lots of training data available; if you have only a few samples, then the information value in their features becomes critical.

5.4.3 Using early stopping

In deep learning, we always use models that are vastly overparameterized: they have way more degrees of freedom than the minimum necessary to fit to the latent manifold of the data. This overparameterization is not an issue because *you never fully fit a deep learning model*. Such a fit wouldn't generalize at all. You will always interrupt training long before you've reached the minimum possible training loss.

Finding the exact point during training where you've reached the most generalizable fit—the exact boundary between an underfit curve and an overfit curve—is one of the most effective things you can do to improve generalization.

In the examples from the previous chapter, we would start by training our models for longer than needed to figure out the number of epochs that yielded the best validation metrics, then we would retrain a new model for exactly that number of epochs. This is pretty standard. However, it requires you to do redundant work, which can sometimes be expensive. Naturally, you could just save your model at the end of each epoch, then once you've found the best epoch, reuse the closest saved model you have. In Keras, it's typical to do this with an `EarlyStopping` callback, which will interrupt training as soon as validation metrics have stopped improving, while remembering the best known model state. You'll learn to use callbacks in chapter 7.

5.4.4 Regularizing your model

Regularization techniques are a set of best practices that actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation. This is called "regularizing" the model because it tends to make the model simpler, more "regular," its curve smoother, and more "generic"—thus less specific to the training set and better able to generalize by more closely approximating the latent manifold of the data. Keep in mind that "regularizing" a model is a process that should always be guided by an accurate evaluation procedure. You will only achieve generalization if you can measure it.

Let's review some of the most common regularization techniques and apply them in practice to improve the movie classification model from chapter 4.

Reducing the network's size

You've already learned that a model that is too small will not overfit. The simplest way to mitigate overfitting is to reduce the size of the model (the number of learnable parameters in the model, determined by the number of layers and the number of units per layer). If the model has limited memorization resources, it won't be able to simply memorize its training data. To minimize its loss, it will have to resort to learning compressed representations that have predictive power regarding the targets—precisely the type of representations we're interested in. At the same time, keep in mind that you should use models that have enough parameters that they don't underfit: your model shouldn't be starved for memorization resources. There is a compromise to be found between *too much capacity* and *not enough capacity*.

Unfortunately, there is no magical formula to determine the right number of layers or the right size for each layer. You must evaluate an array of different architectures (on your validation set, not on your test set, of course) to find the correct model size for your data. The general workflow to find an appropriate model size is to start with relatively few layers and parameters and increase the size of the layers or add new layers until you see diminishing returns with regard to validation loss.

Let's try this on the movie-review classification model. Here's a condensed version of the model from chapter 4.

Listing 5.10 Original model

```
from keras.datasets import imdb

(train_data, train_labels), _ = imdb.load_data(num_words=10000)

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.0
    return results

train_data = vectorize_sequences(train_data)

model = keras.Sequential(
    [
        layers.Dense(16, activation="relu"),
        layers.Dense(16, activation="relu"),
        layers.Dense(1, activation="sigmoid"),
    ]
)
model.compile(
    optimizer="rmsprop",
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
history_original = model.fit(
    train_data,
    train_labels,
    epochs=20,
    batch_size=512,
    validation_split=0.4,
)
```

Now let's try to replace it with this smaller model.

Listing 5.11 Version of the model with lower capacity

```
model = keras.Sequential(
    [
        layers.Dense(4, activation="relu"),
```

```
        layers.Dense(4, activation="relu"),
        layers.Dense(1, activation="sigmoid"),
    ]
)
model.compile(
    optimizer="rmsprop",
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
history_smaller_model = model.fit(
    train_data,
    train_labels,
    epochs=20,
    batch_size=512,
    validation_split=0.4,
)
```

Figure 5.18 shows a comparison of the validation losses of the original model and the smaller model.

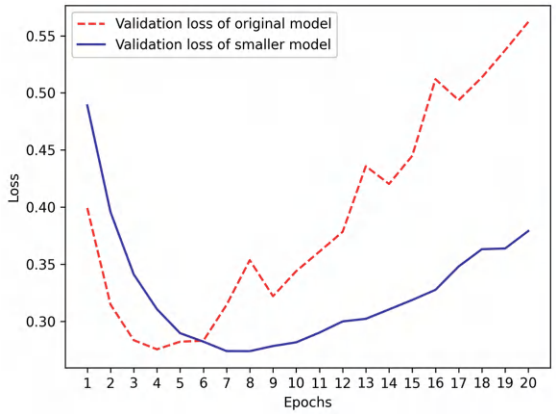


Figure 5.18 Original model vs. smaller model on IMDb review classification

As you can see, the smaller model starts overfitting later than the reference model (after six epochs rather than four), and its performance degrades more slowly once it starts overfitting.

Now, let's add to our benchmark a model that has much more capacity—far more than the problem warrants.

Listing 5.12 Version of the model with higher capacity

```
model = keras.Sequential(
    [
        layers.Dense(512, activation="relu"),
```

```
        layers.Dense(512, activation="relu"),
        layers.Dense(1, activation="sigmoid"),
    ]
)
model.compile(
    optimizer="rmsprop",
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
history_larger_model = model.fit(
    train_data,
    train_labels,
    epochs=20,
    batch_size=512,
    validation_split=0.4,
)
```

Figure 5.19 shows how the bigger model fares compared to the reference model. The bigger model starts overfitting almost immediately, after just one epoch, and it overfits much more severely. Its validation loss is also noisier. It gets training loss near zero very quickly. The more capacity the model has, the more quickly it can model the training data (resulting in a low training loss), but the more susceptible it is to overfitting (resulting in a large difference between the training and validation loss).

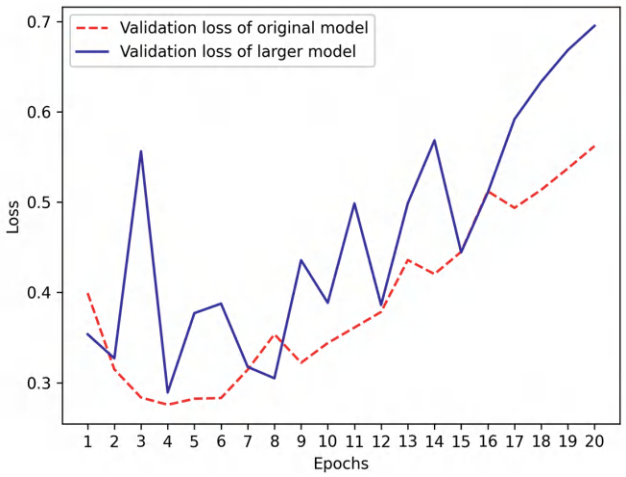


Figure 5.19 Original model vs. much larger model on IMDB review classification

ADDING WEIGHT REGULARIZATION

You may be familiar with the principle of *Occam's razor*: given two explanations for something, the explanation most likely to be correct is the simplest one—the one that makes fewer assumptions. This idea also applies to the models learned by neural

networks: given some training data and a network architecture, multiple sets of weight values (multiple *models*) could explain the data. Simpler models are less likely to overfit than complex ones.

A *simple model* in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters, as you saw in the previous section). Thus a common way to mitigate overfitting is to put constraints on the complexity of a model by forcing its weights to take only small values, which makes the distribution of weight values more *regular*. This is called *weight regularization*, and it's done by adding to the loss function of the model a cost associated with having large weights. This cost comes in two flavors:

- *L1 regularization*—The cost added is proportional to the *absolute value of the weight coefficients* (the *L1 norm* of the weights).
- *L2 regularization*—The cost added is proportional to the *square of the value of the weight coefficients* (the *L2 norm* of the weights). L2 regularization is also called *weight decay* in the context of neural networks. Don't let the different name confuse you: weight decay is mathematically the same as L2 regularization.

In Keras, weight regularization is added by passing *weight regularizer instances* to layers as keyword arguments. Let's add L2 weight regularization to the movie review classification model.

Listing 5.13 Adding L2 weight regularization to the model

```
from keras.regularizers import l2

model = keras.Sequential(
    [
        layers.Dense(16, kernel_regularizer=l2(0.002),
activation="relu"),
        layers.Dense(16, kernel_regularizer=l2(0.002),
activation="relu"),
        layers.Dense(1, activation="sigmoid"),
    ]
)
model.compile(
    optimizer="rmsprop",
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
history_l2_reg = model.fit(
    train_data,
    train_labels,
    epochs=20,
    batch_size=512,
    validation_split=0.4,
)
```

`l2(0.002)` means every coefficient in the weight matrix of the layer will add `0.002 * weight_coefficient_value ** 2` to the total loss of the model. Note that because this penalty is *only added at training time*, the loss for this model will be much higher at training than at test time.

Figure 5.20 shows the effect of the L2 regularization penalty. As you can see, the model with L2 regularization has become much more resistant to overfitting than the reference model, even though both models have the same number of parameters.

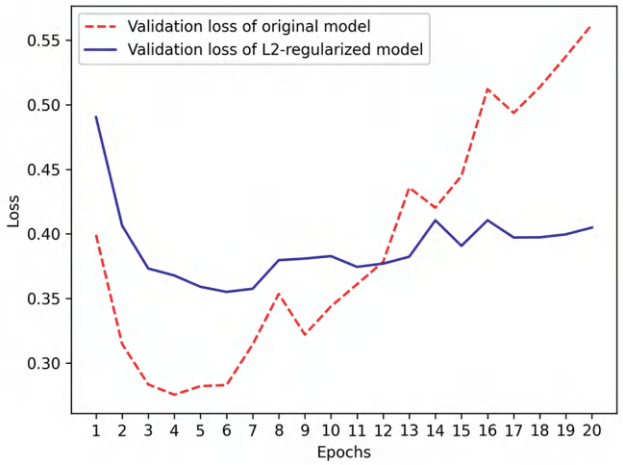


Figure 5.20 Effect of L2 weight regularization on validation loss

As an alternative to L2 regularization, you can use one of the following Keras weight regularizers.

Listing 5.14 Different weight regularizers available in Keras

```
from keras import regularizers

regularizers.l1(0.001)                    <-- L1 regularization
regularizers.l1_l2(l1=0.001, l2 01)       <-- Simultaneous L1 and L2 regularization
```

Note that weight regularization is more typically used for smaller deep learning models. Large deep learning models tend to be so overparameterized that imposing constraints on weight values does not have much effect on model capacity and generalization. In these cases, a different regularization technique is preferred: *dropout*.

ADDING DROPOUT

Dropout, developed by Geoff Hinton and his students at the University of Toronto, is one of the most effective and most commonly used regularization techniques for

neural networks. Dropout, applied to a layer, consists of randomly *dropping out* (setting to zero) a number of output features of the layer during training. Let's say a given layer would normally return a vector `[0.2, 0.5, 1.3, 0.8, 1.1]` for a given input sample during training. After applying dropout, this vector will have a few zero entries distributed at random: for example, `[0, 0.5, 1.3, 0, 1.1]`. The *dropout rate* is the fraction of the features that are zeroed out; it's usually set between 0.2 and 0.5. At test time, no units are dropped out; instead, the layer's output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time.

Consider a NumPy matrix containing the output of a layer, `layer_output`, of shape `(batch_size, features)`. At training time, we zero-out at random a fraction of the values in the matrix:

```
layer_output *= np.random.randint(low=0, high=2, size=layer_output.shape)
```

At training time, drops out 50% of the units in the output

At test time, we scale down the output by the dropout rate. Here, we scale by 0.5 (because we previously dropped half the units):

```
layer_output *= 0.5
```

At test time

Note that this process can be implemented by doing both operations at training time and leaving the output unchanged at test time, which is often the way it's implemented in practice (see figure 5.21):

At training time

```
layer_output *= np.random.randint(low=0, high=2, size=layer_output.shape)
layer_output /= 0.5
```

Note that we're scaling up rather scaling down in this case.

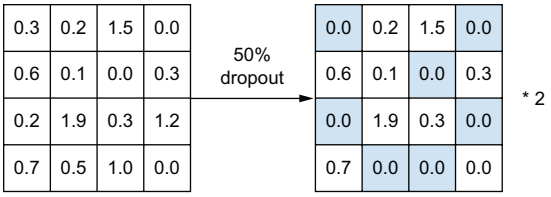


Figure 5.21 Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time, the activation matrix is unchanged.

This technique may seem strange and arbitrary. Why would this help reduce overfitting? Hinton says he was inspired by, among other things, a fraud-prevention mechanism used by banks:

> *I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting.*

The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that aren't significant (what Hinton refers to as *conspiracies*), which the model will start memorizing if no noise is present.

In Keras, you can introduce dropout in a model via the `Dropout` layer, which is applied to the output of the layer right before it. Let's add two `Dropout` layers in the IMDB model to see how well they do at reducing overfitting.

Listing 5.15 Adding dropout to the IMDB model

```
model = keras.Sequential(
    [
        layers.Dense(16, activation="relu"),
        layers.Dropout(0.5),
        layers.Dense(16, activation="relu"),
        layers.Dropout(0.5),
        layers.Dense(1, activation="sigmoid"),
    ]
)
model.compile(
    optimizer="rmsprop",
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
history_dropout = model.fit(
    train_data,
    train_labels,
    epochs=20,
    batch_size=512,
    validation_split=0.4,
)
```

Figure 5.22 shows a plot of the results. This is a clear improvement over the reference model. It also seems to be working much better than L2 regularization since the lowest validation loss reached has improved.

To recap, these are the most common ways to maximize generalization and prevent overfitting in neural networks:

- Getting more training data, or better training data
- Developing better features

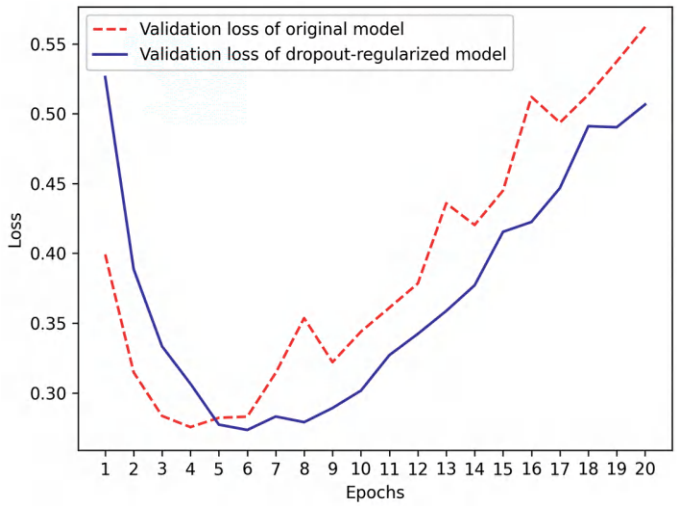


Figure 5.22 Effect of dropout on validation loss

- Reducing the capacity of the model
- Adding weight regularization (for smaller models)
- Adding dropout

Summary

- The purpose of a machine learning model is to *generalize*: to perform accurately on never-before-seen inputs. It's harder than it seems.
- A deep neural network achieves generalization by learning a parametric model that can successfully *interpolate* between training samples. Such a model can be said to have learned the *latent manifold* of the training data. This is why deep learning models can only make sense of inputs that are very close to what they've seen during training.
- The fundamental problem in machine learning is *the tension between optimization and generalization*: to attain generalization, you must first achieve a good fit to the training data, but improving your model's fit to the training data will inevitably start hurting generalization after a while. Every single deep learning best practice deals with managing this tension.
- The ability of deep learning models to generalize comes from the fact that they manage to learn to approximate the *latent manifold* of their data and can thus make sense of new inputs via interpolation.
- It's essential to be able to accurately evaluate the generalization power of your model while you're developing it. You have at your disposal an array of evaluation methods, from simple hold-out validation to K-fold cross-validation and iterated K-fold cross-validation with shuffling. Remember to always keep a completely

separate test set for final model evaluation, since information leaks from your validation data to your model may have occurred.

- When you start working on a model, your goal is first to achieve a model that has some generalization power and that can overfit. Best practices to do this include tuning your learning rate and batch size, using better architecture priors, increasing model capacity, or simply training longer.
- As your model starts overfitting, your goal switches to improving generalization through *model regularization*. You can reduce your model's capacity, add dropout or weight regularization, and use early stopping. And naturally, a larger or better dataset is always the number one way to help a model generalize.

6 The universal workflow of machine learning

This chapter covers

- Framing a machine learning problem
- Developing a working model
- Deploying your model in production and maintaining it

Our previous examples have assumed that we already had a labeled dataset to start from, and that we could immediately start training a model. In the real world, this is often not the case. You don't start from a dataset; you start from a problem.

Imagine that you're launching your own machine learning consulting shop. You incorporate, you put up a fancy website, you notify your network. The projects start rolling in:

- A personalized photo search engine for a picture-sharing social network—type in "wedding" and retrieve all the pictures you took at weddings, without any manual tagging needed.
- Flagging spam and offensive text content among the posts of a budding chat app.

- Building a music recommendation system for users of an online radio.
- Detecting credit card fraud for an e-commerce website.
- Predicting display ad click-through rate to decide which ad to serve to a given user at a given time.
- Flagging anomalous cookies on the conveyor belt of a cookie-manufacturing line.
- Using satellite images to predict the location of as-yet unknown archaeological sites.

It would be very convenient if you could import the correct dataset from `keras.datasets` and start fitting some deep learning models. Unfortunately, in the real world, you'll have to start from scratch.

In this chapter, you'll learn about the universal step-by-step blueprint that you can use to approach and solve any machine learning problem, like those previously listed. This template will bring together and consolidate everything you've learned in chapters 4 and 5 and give you the wider context that should anchor what you will learn in the next chapters.

The universal workflow of machine learning is broadly structured in three parts:

- *Define the task*—Understand the problem domain and the business logic underlying what the customer asked. Collect a dataset, understand what the data represents, and choose how you will measure success on the task.
- *Develop a model*—Prepare your data so that it can be processed by a machine learning model, select a model evaluation protocol and a simple baseline to beat, train a first model that has generalization power that can overfit, and then regularize and tune your model until you achieve the best possible generalization performance.
- *Deploy the model*—Present your work to stakeholders, ship the model to a web server, a mobile app, a web page, or an embedded device, monitor the model's performance in the wild, and start collecting the data you'll need to build the next model generation.

Let's dive in.

6.1 Defining the task

You can't do good work without a deep understanding of the context of what you're doing. Why is your customer trying to solve this particular problem? What value will they derive from the solution? How will your model be used? How will it fit into your customer's business processes? What kind of data is available or could be collected? What kind of machine learning task can be mapped to the business problem?

6.1.1 Framing the problem

Framing a machine learning problem usually involves many detailed discussions with stakeholders. Here are the questions that should be on top of your mind:

- What will your input data be? What are you trying to predict? You can only learn to predict something if you have available training data: for example, you can only learn to classify the sentiment of movie reviews if you have both movie reviews and sentiment annotations available. As such, data availability is usually the limiting factor at this stage. In many cases, you will have to resort to collecting and annotating new datasets yourself (which we cover in the next section).
- What type of machine learning task are you facing? Is it binary classification? Multiclass classification? Scalar regression? Vector regression? Multiclass, multilabel classification? Image segmentation? Ranking? Something else, like clustering, generation, or reinforcement learning? In some cases, it may be that machine learning isn't even the best way to make sense of your data, and you should use something else, such as plain old-school statistical analysis:
  - The photo search engine project is a multiclass, multilabel classification task.
  - The spam detection project is a binary classification task. If you set "offensive content" as a separate class, it's a three-way classification task.
  - The music recommendation engine turns out to be better handled not via deep learning, but via matrix factorization (collaborative filtering).
  - The credit card fraud detection project is a binary classification task.
  - The click-through rate prediction project is a scalar regression task.
  - Anomalous cookie detection is a binary classification task, but it will also require an object detection model as a first stage to correctly crop out the cookies in raw images. Note that the set of machine learning techniques known as "anomaly detection" would not be a good fit in this setting!
  - The project about finding new archaeological sites from satellite images is an image similarity ranking task: you need to retrieve new images that look the most like known archaeological sites.
- What do existing solutions look like? Perhaps your customer already has a hand-crafted algorithm that handles spam filtering or credit card fraud detection—with lots of nested `if` statements. Perhaps a human is currently in charge of manually handling the process considered—monitoring the conveyor belt at the cookie plant and manually removing the bad cookies, or crafting playlists of song recommendations to be sent out to users who liked a specific artist. You should make sure to understand what systems are already in place, and how they work.
- Are there particular constraints you will need to deal with? For example, you could find out that the app for which you're building a spam detection system is strictly end-to-end encrypted, so that the spam detection model will have to live on the end-user's phone, and must be trained on an external dataset. Perhaps the cookie-filtering model has such latency constraints that it will need to run on an embedded device at the factory rather than on a remote server. You should understand the full context in which your work will fit.

Once you've done your research, you should know what your inputs will be, what your targets will be, and what broad type of machine learning task the problem maps to. Be aware of the hypotheses you're making at this stage:

- You hypothesize that your targets can be predicted given your inputs.
- You hypothesize that the data that's available (or that you will soon collect) is sufficiently informative to learn the relationship between inputs and targets.

Until you have a working model, these are merely hypotheses, waiting to be validated or invalidated. Not all problems can be solved with machine learning; just because you've assembled examples of inputs X and targets Y doesn't mean X contains enough information to predict Y. For instance, if you're trying to predict the movements of a stock on the stock market given its recent price history, you're unlikely to succeed, because price history doesn't contain much predictive information.

Note on ethics

You may sometimes be offered ethically dubious projects, such as "building an AI that rates the trustworthiness of someone from a picture of their face." First of all, the validity of the project is in doubt: it isn't clear why trustworthiness would be reflected on someone's face. Second, such a task opens the door to all kinds of ethical problems. Collecting a dataset for this task would amount to recording the biases and prejudices of the people who label the pictures. The models you would train on such data would be merely encoding these same biases—into a black box algorithm, which would give them a thin veneer of legitimacy. In a largely tech-illiterate society like ours, "The AI algorithm said this person cannot be trusted" strangely appears to carry more weight and objectivity than "John Smith said this person cannot be trusted"—despite the former being a learned approximation of the latter. Your model would be laundering and operationalizing at scale the worst aspects of human judgement, with negative effects on the lives of real people.

Technology is never neutral. If your work has any impact on the world, then this impact has a moral direction: technical choices are also ethical choices. Always be deliberate about the values you want your work to support.

6.1.2 Collecting a dataset

Once you understand the nature of the task and you know what your inputs and targets are going to be, it's time for data collection—the most arduous, time-consuming, and costly part of most machine learning projects:

- The photo search engine project requires you to first select the set of labels you want to classify—you settle on 10,000 common image categories. Then, you need to manually tag hundreds of thousands of your past user-uploaded images with labels from this set.
- For the chat app spam detection project, because user chats are end-to-end encrypted, you cannot use their contents for training a model. You need to gain

access to a separate dataset of tens of thousands of unfiltered social media posts, and manually tag them as spam, offensive, or acceptable.

- For the music recommendation engine, you can just use the "likes" of your users. No new data needs to be collected. Likewise, for the click-through rate prediction project, you have an extensive record of click-through rate for your past ads, going back years.
- For the cookie-flagging model, you will need to install cameras above the conveyor belts to collect tens of thousands of images, and then someone will need to manually label these images. The people who know how to do this currently work at the cookie factory—but it doesn't seem too difficult, you should be able to train people to do it.
- The satellite imagery project will require a team of archaeologists to collect a database of existing sites of interest, and for each site, you will need to find existing satellite images taken in different weather conditions. To get a good model, you're going to need thousands of different sites.

You've learned in chapter 5 that a model's ability to generalize comes almost entirely from the properties of the data it is trained on—the number of data points you have, the reliability of your labels, the quality of your features. A good dataset is an asset worthy of care and investment. If you get an extra 50 hours to spend on a project, chances are that the most effective way to allocate them is to collect more data, rather than search for incremental modeling improvements.

The point that data matters more than algorithms was most famously made in a 2009 paper by Google researchers titled "The Unreasonable Effectiveness of Data" (the title is a riff on the well-known 1960 book *The Unreasonable Effectiveness of Mathematics in the Natural Sciences* by Eugene Wigner). This was before deep learning was popular, but remarkably, the rise of deep learning has only increased the importance of data.

If you're doing supervised learning, then once you've collected inputs (such as images) you're going to need *annotations* for them (such as tags for those images): the targets you will train your model to predict.

Sometimes, annotations can be retrieved automatically—for instance, in the case of our music recommendation task or our click-through rate prediction task. But often, you have to annotate your data by hand. This is a labor-heavy process.

Investing in data annotation infrastructure

Your data annotation process will determine the quality of your targets, which, in turn, determines the quality of your model. Carefully consider the options you have available:

- Should you annotate the data yourself?
- Should you use a crowdsourcing platform like Mechanical Turk to collect labels?
- Should you use the services of a specialized data-labeling company?

Outsourcing can potentially save you time and money but takes away control. Using something like Mechanical Turk is likely to be inexpensive and to scale well, but your annotations may end up being quite noisy.

To pick the best option, consider the constraints you're working with:

- Do the data labelers need to be subject matter experts, or could anyone annotate the data? The labels for a cat-versus-dog image classification problem can be selected by anyone, but those for a dog breed classification task require specialized knowledge. Meanwhile, annotating CT scans of bone fractures pretty much requires a medical degree.
- If annotating the data requires specialized knowledge, can you train people to do it? If not, how can you get access to relevant experts?
- Do you, yourself, understand the way experts come up with the annotations? If you don't, you will have to treat your dataset as a black box, and you won't be able to perform manual feature engineering—this isn't critical, but it can be limiting.

If you decide to label your data in-house, ask yourself what software you will use to record annotations. You may well need to develop that software yourself. Productive data annotation software will save you a lot of time, so it's something worth investing in early in a project.

BEWARE OF NONREPRESENTATIVE DATA

Machine learning models can only make sense of inputs that are similar to what they've seen before. As such, it's critical that the data used for training should be *representative* of the production data. This concern should be the foundation of all of your data collection work.

Suppose you're developing an app where users can take pictures of a dish to find out its name. You train a model using pictures from an image-sharing social network that's popular with foodies. Come deployment time, and feedback from angry users starts rolling in: your app gets the answer wrong 8 times out of 10. What's going on? Your accuracy on the test set was well over 90%! A quick look at user-uploaded data reveals that mobile picture uploads of random dishes from random restaurants taken with random smartphones look nothing like the professional-quality, well-lit, appetizing pictures you trained the model on: *your training data wasn't representative of the production data.* That's a cardinal sin—welcome to machine learning hell.

If possible, collect data directly from the environment where your model will be used. A movie review sentiment classification model should be used on new IMDB reviews, not on Yelp restaurant reviews, nor on Twitter status updates. If you want to rate the sentiment of a tweet, start by collecting and annotating actual tweets—from a similar set of users as those you're expecting in production. If it's not possible to train on production data, then make sure you fully understand how your training and production data differ, and that you are actively correcting these differences.

A related phenomenon you should be aware of is *concept drift.* You'll encounter concept drift in almost all real-world problems, especially those that deal with

user-generated data. Concept drift occurs when the properties of the production data change over time, causing model accuracy to gradually decay. A music recommendation engine trained in the year 2013 may not be very effective today. Likewise, the IMDB dataset you worked with was collected in 2011, and a model trained on it would likely not perform as well on reviews from 2020 compared to reviews from 2012, as vocabulary, expressions, and movie genres evolve over time. Concept drift is particularly acute in adversarial contexts like credit card fraud detection, where fraud patterns change practically every day. Dealing with fast concept drift requires constant data collection, annotation, and model retraining.

Keep in mind that machine learning can only be used to memorize patterns that are present in your training data. You can only recognize what you've seen before. Using machine learning trained on past data to predict the future is making the assumption that the future will behave like the past. That often isn't the case.

The problem of sampling bias

A particularly insidious and common case of nonrepresentative data is *sampling bias*. Sampling bias occurs when your data collection process interacts with what you are trying to predict, resulting in biased measurements. A famous historical example occurred in the 1948 US presidential election. On election night, the Chicago Tribune printed the headline "DEWEY DEFEATS TRUMAN." The next morning, Truman emerged as the winner. The editor of the Tribune had trusted the results of a phone survey—but phone users in 1948 were not a random, representative sample of the voting population. They were more likely to be richer, conservative, and to vote for Dewey, the Republican candidate. Nowadays, every phone survey takes sampling bias into account. That doesn't mean that sampling bias is a thing of the past in political polling—far from it. But unlike in 1948, pollsters are aware of it and take steps to correct it.

"DEWEY DEFEATS TRUMAN": a famous example of sampling bias

6.1.3 Understanding your data

It's bad practice to treat a dataset as a black box. Before you start training models, you should explore and visualize your data to gain insights about what makes it predictive—which will inform feature engineering—and screen for potential issues:

- If your data includes images or natural language text, take a look at a few samples (and their labels) directly.
- If your data contains numerical features, it's a good idea to plot the histogram of feature values to get a feel for the range of values taken and the frequency of different values.
- If your data includes location information, plot it on a map. Do any clear patterns emerge?
- Are some samples missing values for some features? If so, you'll need to deal with this when you prepare the data (we cover how to do this in the next section).
- If your task is a classification problem, print the number of instances of each class in your data. Are the classes roughly equally represented? If not, you will need to account for this imbalance.
- Check for *target leaking*—the presence of features in your data that provide information about the targets that may not be available in production. If you're training a model on medical records to predict whether someone will be treated for cancer in the future, and the records include the feature "This person has been diagnosed with cancer," then your targets are being artificially leaked into your data. Always ask yourself, is every feature in your data something that will be available in the same form in production?

6.1.4 Choosing a measure of success

To control something, you need to be able to observe it. To achieve success on a project, you must first define what you mean by success. Accuracy? Precision and recall? Customer retention rate? Your metric for success will guide all of the technical choices you will make throughout the project. It should directly align with your higher-level goals, such as the business success of your customer.

For balanced classification problems, where every class is equally likely, accuracy and *area under curve* (AUC) of the *receiver operating characteristic* (ROC) are common metrics. For class-imbalanced problems, ranking problems, or multilabel classification, you can use precision and recall or a metric that counts false positives, true positives, false negatives, and true negatives. And it isn't uncommon to have to define your own custom metric by which to measure success. To get a sense of the diversity of machine learning success metrics and how they relate to different problem domains, it's helpful to browse the data science competitions on Kaggle (https://kaggle.com); it showcases a wide range of problems and evaluation metrics.

6.2 Developing a model

Once you know how you will measure your progress, you can get started with model development. Most tutorials and research projects assume that this is the only step—skipping problem definition and dataset collection, which are assumed to be already done, and skipping model deployment and maintenance, which is assumed to be handled by someone else. In fact, model development is only *one step* in the machine learning workflow, and if you ask us, it's not the most difficult one. The hardest things in machine learning are framing problems and collecting, annotating, and cleaning data. So cheer up, what comes next will be easy in comparison!

6.2.1 Preparing the data

As you've learned before, deep learning models typically don't ingest raw data. Data preprocessing aims at making the raw data at hand more amenable to neural networks. This includes vectorization, normalization, or handling missing values. Many preprocessing techniques are domain specific (for example, specific to text data or image data); we'll cover those in the following chapters as we encounter them in practical examples. For now, we'll review the basics that are common to all data domains.

VECTORIZATION

All inputs and targets in a neural network must be typically tensors of floating-point data (or, in specific cases, tensors of integers or strings). Whatever data you need to process—sound, images, text—you must first turn into tensors, a step called *data vectorization*. For instance, in the two previous text classification examples from chapter 4, we started from text represented as lists of integers (standing for sequences of words), and we used multi-hot encoding to turn them into a tensor of `float32` data. In the examples of classifying digits and predicting house prices, the data already came in vectorized form, so you were able to skip this step.

VALUE NORMALIZATION

In the MNIST digit-classification example from chapter 2, you started from image data encoded as integers in the 0–255 range, encoding grayscale values. Before you fed this data into your network, you had to cast it to `float32` and divide by 255 so you'd end up with floating-point values in the 0–1 range. Similarly, when predicting house prices, you started from features that took a variety of ranges—some features had small floating-point values, others had fairly large integer values. Before you fed this data into your network, you had to normalize each feature independently so that it had a standard deviation of 1 and a mean of 0.

In general, it isn't safe to feed into a neural network data that takes relatively large values (for example, multi-digit integers, which are much larger than the initial values taken by the weights of a network) or data that is heterogeneous (for example, data where one feature is in the range 0–1, and another is in the range 100–200). Doing so can trigger large gradient updates that will prevent the network from

converging. To make learning easier for your network, your data should have the following characteristics:

- *Take small values*—Typically, most values should be in the 0–1 range.
- *Be homogeneous*—That is, all features should take values in roughly the same range.

Additionally, the following stricter normalization practice is common and can help, although it isn't always necessary (for example, you didn't do this in the digit-classification example):

- Normalize each feature independently to have a mean of 0.
- Normalize each feature independently to have a standard deviation of 1.

This is easy to do with NumPy arrays:

```
x -= x.mean(axis=0)
x /= x.std(axis=0)
```

Assuming x is a 2D data matrix of shape (samples, features)

HANDLING MISSING VALUES

You may sometimes have missing values in your data. For instance, in the house price example, the second feature was the median age of houses in the district. What if this feature wasn't available for all samples? You'd then have missing values in the training or test data.

You could just discard the feature entirely, but you don't necessarily have to:

- If the feature is categorical, it's safe to create a new category that means "the value is missing." The model will automatically learn what this implies with respect to the targets.
- If the feature is numerical, avoid inputting an arbitrary value like 0 because it may create a discontinuity in the latent space formed by your features, making it harder for a model trained on it to generalize. Instead, consider replacing the missing value with the average or median value for the feature in the dataset. You could also train a model to predict the feature value given the values of other features.

Note that if you're expecting missing categorical features in the test data, but the network was trained on data without any missing values, the network won't have learned to ignore missing values! In this situation, you should artificially generate training samples with missing entries: copy some training samples several times, and drop some of the categorical features that you expect are likely to be missing in the test data.

6.2.2 Choosing an evaluation protocol

As you've learned in the previous chapter, the purpose of a model is to achieve generalization, and every modeling decision you will make throughout the model development process will be guided by *validation metrics* that seek to measure generalization

performance. The goal of your validation protocol is to accurately estimate what your success metric of choice (such as accuracy) will be on actual production data. The reliability of that process is critical to building a useful model.

In chapter 5, we reviewed three common evaluation protocols:

- *Maintaining a hold-out validation set*—The way to go when you have plenty of data
- *Doing K-fold cross-validation*—The right choice when you have too few samples for hold-out validation to be reliable
- *Doing iterated K-fold validation*—For performing highly accurate model evaluation when little data is available

Just pick one of these. In most cases, the first will work well enough. As you've learned before, always be mindful of the *representativity* of your validation set(s) and be careful not to have redundant samples between your training set and your validation set(s).

6.2.3 Beating a baseline

As you start working on the model itself, your initial goal is to achieve *statistical power*, as you saw in chapter 5—that is, to develop a small model that is capable of beating a simple baseline.

At this stage, these are the three most important things you should focus on:

- *Feature engineering*—Filter out uninformative features (feature selection) and use your knowledge of the problem to develop new features that are likely to be useful.
- *Selecting the correct architecture priors*—What type of model architecture will you use? A densely connected network, a ConvNet, a recurrent neural network, a Transformer? Is deep learning even a good approach for the task, or should you use something else?
- *Selecting a good enough training configuration*—What loss function should you use? What batch size and learning rate?

Picking the right loss function

It's often not possible to directly optimize for the metric that measures success on a problem. Sometimes there is no easy way to turn a metric into a loss function; loss functions, after all, need to be computable given only a mini-batch of data (ideally, a loss function should be computable for as little as a single data point) and must be differentiable (otherwise, you can't use backpropagation to train your network). For instance, the widely used classification metric ROC AUC can't be directly optimized. Hence, in classification tasks, it's common to optimize for a proxy metric of ROC AUC, such as crossentropy. In general, you can hope that the lower the crossentropy gets, the higher the ROC AUC will be.

Table 6.1 can help you choose a last-layer activation, a loss function, and metrics for a few common problem types.

Table 6.1 Which loss, last-layer activation, and metrics to use for different tasks

Task	Last-layer activation	Loss function	Metrics
Binary classification	Sigmoid	Binary crossentropy	Binary accuracy, ROC AUC
Multiclass, single-label classification	Softmax	Categorical crossentropy	Categorical accuracy, top-k categorical accuracy, ROC AUC
Multiclass, multi-label classification	Sigmoid	Binary crossentropy	Binary accuracy, ROC AUC
Regression	None	Mean squared error	Mean absolute error

For most problems, there are existing templates you can start from. You're not the first person to try to build a spam detector, a music recommendation engine, or an image classifier. Make sure to research prior art to identify the feature engineering techniques and model architectures that are most likely to perform well on your task.

Note that it's not always possible to achieve statistical power. If you can't beat a simple baseline after trying multiple reasonable architectures, it may be that the answer to the question you're asking isn't present in the input data. Remember that you're making two hypotheses:

- You hypothesize that your outputs can be predicted given your inputs.
- You hypothesize that the available data is sufficiently informative to learn the relationship between inputs and outputs.

It may well be that these hypotheses are false, in which case you must go back to the drawing board.

6.2.4 Scaling up: Developing a model that overfits

Once you've obtained a model that has statistical power, the question becomes, is your model sufficiently powerful? Does it have enough layers and parameters to properly model the problem at hand? For instance, a logistic regression model has statistical power on MNIST but wouldn't be sufficient to solve the problem well. Remember that the universal tension in machine learning is between optimization and generalization; the ideal model is one that stands right at the border between underfitting and overfitting, between undercapacity and overcapacity. To figure out where this border lies, first you must cross it.

To figure out how big a model you'll need, you must develop a model that overfits. This is fairly easy, as you learned in chapter 5:

- Add layers.
- Make the layers bigger.
- Train for more epochs.

Always monitor the training loss and validation loss, as well as the training and validation values for any metrics you care about. When you see that the model's performance on the validation data begins to degrade, you've achieved overfitting.

6.2.5 Regularizing and tuning your model

Once you've achieved statistical power and you're able to overfit, you know you're on the right path. At this point, your goal becomes to maximize generalization performance.

This phase will take the most time: you'll repeatedly modify your model, train it, evaluate on your validation data (not the test data, at this point), modify it again, and repeat, until the model is as good as it can get. Here are some things you should try:

- Try different architectures; add or remove layers.
- Add dropout.
- If your model is small, add L1 or L2 regularization.
- Try different hyperparameters (such as the number of units per layer or the learning rate of the optimizer) to find the optimal configuration.
- Optionally, iterate on data curation or feature engineering: collect and annotate more data, develop better features, or remove features that don't seem to be informative.

It's possible to automate a large chunk of this work by using *automated hyperparameter-tuning software*, such as KerasTuner. We'll cover this in chapter 18.

Be mindful of the following: every time you use feedback from your validation process to tune your model, you leak information about the validation process into the model. Repeated just a few times, this is innocuous; however, done systematically over many iterations, it will eventually cause your model to overfit to the validation process (even though no model is directly trained on any of the validation data). This makes the evaluation process less reliable.

Once you've developed a satisfactory model configuration, you can train your final production model on all the available data (training and validation) and evaluate it one last time on the test set. If it turns out that performance on the test set is significantly worse than the performance measured on the validation data, this may mean either that your validation procedure wasn't reliable after all, or that you began overfitting to the validation data while tuning the parameters of the model. In this case, you may want to switch to a more reliable evaluation protocol (such as iterated K-fold validation).

6.3 Deploying your model

After your model has successfully cleared its final evaluation on the test set, it's ready to be deployed and to begin its productive life.

6.3.1 Explaining your work to stakeholders and setting expectations

Success and customer trust are about consistently meeting or exceeding people's expectations; the actual system you deliver is only half of that picture. The other half is setting appropriate expectations before launch.

The expectations of nonspecialists toward AI systems are often unrealistic. For example, they might expect that the system "understands" its task and is capable of exercising human-like common sense in the context of the task. To address this, you should consider showing some examples of the *failure modes* of your model (for instance, show what incorrectly classified samples look like, especially those for which the misclassification seems surprising).

They might also expect human-level performance, especially for processes that were previously handled by people. Most machine learning models, because they are (imperfectly) trained to approximate human-generated labels, do not nearly get there. You should clearly convey model performance expectations. Avoid using abstract statements like "The model has 98% accuracy" (which most people mentally round up to 100%), and prefer talking, for instance, about false-negative rates and false-positive rates. You could say, "With these settings, the fraud detection model would have a 5% false-negative rate and a 2.5% false-positive rate. Every day, an average of 200 valid transactions would be flagged as fraudulent and sent for manual review, and an average of 14 fraudulent transactions would be missed. An average of 266 fraudulent transactions would be correctly caught." Clearly relate the model's performance metrics to business goals.

You should also make sure to discuss with stakeholders the choice of key launch parameters—for instance, the probability threshold at which a transaction should be flagged (different thresholds will produce different false-negative and false-positive rates). Such decisions involve tradeoffs that can only be handled with a deep understanding of the business context.

6.3.2 Shipping an inference model

A machine learning project doesn't end when you arrive at a Colab notebook that can save a trained model. You rarely put into production the exact same Python model object that you manipulated during training.

First, you may want to export your model to something other than Python:

- Your production environment may not support Python at all—for instance, if it's a mobile app or an embedded system.
- If the rest of the app isn't in Python (it could be in JavaScript, C++, etc.), the use of Python to serve a model may induce significant overhead.

Second, since your production model will only be used to output predictions (a phase called *inference*), rather than for training, you have room to perform various optimizations that can make the model faster and reduce its memory footprint.

Let's take a quick look at the different model deployment options you have available.

Deploying a model as a REST API

Perhaps the easiest way to turn a model into a product is to serve it online via a REST API. There are a number of libraries out there for making this happen. Keras supports two of the most popular approaches out of the box—*TensorFlow Serving* and *ONNX* (short for Open Neural Network Exchange). Both libraries operate by lifting all model weights and a computation graph outside of the Python program, so you can serve it from a number of different environments (for example, a C++ server). If this sounds a lot like the compilation mechanism discussed in chapter 3 you are spot-on. TensorFlow Serving is essentially a library for serving `tf.function` computation graphs with a specific set of saved weights.

Keras allows access to both TensorFlow Serving and ONNX via an easy-to-use `export()` method available on all Keras models. Here's a code snippet showing how this works for TensorFlow Serving:

```
model.export("path/to/location", format="tf_saved_model")

reloaded_artifact = tf.saved_model.load("path/to/location")
predictions = reloaded_artifact.serve(input_data)
```

Exports the model as a TensorFlow SavedModel artifact

ads the artifact in a different process, environment, or programming language

A similar flow exists for ONNX:

```
model.export("path/to/location", format="onnx")

ort_session = onnxruntime.InferenceSession("path/to/location")
predictions = ort_session.run(None, input_data)
```

You should use this deployment setup when

- The application that will consume the model's prediction will have reliable access to the internet (obviously). For instance, if your application is a mobile app, serving predictions from a remote API means that the application won't be usable in airplane mode or in a low-connectivity environment.
- The application does not have strict latency requirements: the request, inference, and answer round trip will typically take around 500 ms.
- The input data sent for inference is not highly sensitive: the data will need to be available on the server in a decrypted form, since it will need to be seen by the model (but note that you should use SSL encryption for the HTTP request and answer).

For instance, the image search engine project, the music recommender system, the credit card fraud detection project, and the satellite imagery project are all a good fit for serving via a REST API.

An important question when deploying a model as a REST API is whether you want to host the code on your own or whether you want to use a fully managed third-party cloud service. For instance, Cloud AI Platform, a Google product, lets you simply upload your TensorFlow model to Google Cloud Storage (GCS) and gives you an API endpoint to query it. It takes care of many practical details such as batching predictions, load balancing, and scaling.

DEPLOYING A MODEL ON A DEVICE

Sometimes, you may need your model to live on the same device that runs the application that uses it—maybe a smartphone, an embedded ARM CPU on a robot, or a microcontroller on a tiny device. For instance, perhaps you've already seen a camera capable of automatically detecting people and faces in the scenes you pointed it at: that was probably a small deep learning model running directly on the camera.

You should use this setup when

- Your model has strict latency constraints or needs to run in a low-connectivity environment. If you're building an immersive augmented-reality application, querying a remote server is not a viable option.
- Your model can be made sufficiently small that it can run under the memory and power constraints of the target device.
- Getting the highest possible accuracy isn't mission critical for your task: there is always a tradeoff between runtime efficiency and accuracy, so memory and power constraints often require you to ship a model that isn't quite as good as the best model you could run on a large GPU.
- The input data is strictly sensitive and thus shouldn't be decryptable on a remote server.

For instance, our spam detection model will need to run on the end user's smartphone as part of the chat app, because messages are end-to-end encrypted and thus cannot be read by a remotely hosted model at all. Likewise, the bad-cookie-detection model has strict latency constraints and will need to run at the factory. Thankfully, in this case, we don't have any power or space constraints, so we can actually run the model on a GPU.

To deploy a Keras model on a smartphone or embedded device, you can again use the `export()` method to create a TensorFlow or ONNX save of your model including the computation graph. TensorFlow Lite (https://www.tensorflow.org/lite) is a framework for efficient on-device deep learning inference that runs on Android and iOS smartphones, as well as ARM CPUs, Raspberry Pi, or certain microcontrollers. It uses the same TensorFlow save model format as TensorFlow Serving. The ONNX runtime can also run on mobile devices.

Deploying a model in the browser

Deep learning is often used in browser-based or desktop-based JavaScript applications. While it is usually possible to have the application query a remote model via a REST API, there can be key advantages in instead having the model run directly in the browser, on the user's computer (utilizing GPU resources if available).

Use this setup when

- You want to offload compute to the end user, which can dramatically reduce server costs.
- The input data needs to stay on the end user's computer or phone. For instance, in our spam detection project, the web version and the desktop version of the chat app (implemented as a cross-platform app written in JavaScript) should use a locally run model.
- Your application has strict latency constraints: while a model running on the end user's laptop or smartphone is likely to be slower than one running on a large GPU on your own server, you don't have the extra 100 ms of network round trip.
- You need your app to keep working without connectivity, after the model has been downloaded and cached.

Of course, you should only go with this option if your model is small enough that it won't hog the CPU, GPU, or RAM of your user's laptop or smartphone. In addition, since the entire model will be downloaded to the user's device, you should make sure that nothing about the model needs to stay confidential. Be mindful of the fact that, given a trained deep learning model, it is usually possible to recover some information about the training data: better not to make your trained model public if it was trained on sensitive data.

To deploy a model in JavaScript, the TensorFlow ecosystem includes TensorFlow.js (https://www.tensorflow.org/js), and ONNX supports a native JavaScript runtime. TensorFlow.js even implements almost all of the Keras API (it was originally developed under the working name WebKeras) as well as many lower-level TensorFlow APIs. You can easily import a saved Keras model into TensorFlow.js to query it as part of your browser-based JavaScript app or your desktop Electron app.

Inference model optimization

Optimizing your model for inference is especially important when deploying in an environment with strict constraints on available power and memory (smartphones and embedded devices) or for applications with low-latency requirements. You should always seek to optimize your model before importing it into TensorFlow.js or exporting it to TensorFlow Lite.

There are two popular optimization techniques you can apply:

- *Weight pruning*—Not every coefficient in a weight tensor contributes equally to the predictions. It's possible to considerably lower the number of parameters in the layers of your model by only keeping the most significant ones. This reduces

the memory and compute footprint of your model at a small cost in performance metrics. By tuning how much pruning you want to apply, you are in control of the tradeoff between size and accuracy.

- *Weight quantization*—Deep learning models are trained with single-precision floating-point (`float32`) weights. However, it's possible to *quantize* weights to 8-bit signed integers (`int8`) to get an inference-only model that's four times smaller but remains near the accuracy of the original model. Keras models come with a built-in `quantize()` API that can help with this. Simply call `model.quantize("int8")` to compress each weight in your model to a single byte.

6.3.3 Monitoring your model in the wild

You've exported an inference model, you've integrated it into your application, and you've done a dry run on production data—the model behaved exactly as you expected. You've written unit tests as well as logging and status-monitoring code—perfect. Now it's time to press the big red button and deploy to production.

Even this is not the end. Once you've deployed a model, you need to keep monitoring its behavior, its performance on new data, its interaction with the rest of the application, and its eventual impact on business metrics:

- Is user engagement in your online radio up or down after deploying the new music recommender system? Has average ad click-through rate increased after switching to the new click-through rate prediction model? Consider using *randomized A/B testing* to isolate the impact of the model itself from other changes: a subset of cases should go through the new model, while another control subset should stick to the old process. Once sufficiently many cases have been processed, the difference in outcomes between the two is likely attributable to the model.
- If possible, do a regular manual audit of the model's predictions on production data. It's generally possible to reuse the same infrastructure as for data annotation: send some fraction of the production data to be manually annotated and compare the model's predictions to the new annotations. For instance, you should definitely do this for the image search engine and the bad-cookie-flagging system.
- When manual audits are impossible, consider alternative evaluation avenues such as user surveys (for example, in the case of the spam and offensive content–flagging system).

6.3.4 Maintaining your model

Lastly, no model lasts forever. You've already learned about *concept drift*: over time, the characteristics of your production data will change, gradually degrading the performance and relevance of your model. The lifespan of your music recommender system will be counted in weeks. For the credit card fraud detection system, it would be days; a couple of years in the best case for the image search engine.

As soon as your model has launched, you should be getting ready to train the next generation that will replace it:

- Watch out for changes in the production data. Are new features becoming available? Should you expand or otherwise edit the label set?
- Keep collecting and annotating data, and keep improving your annotation pipeline over time. In particular, you should pay special attention to collecting samples that seem to be difficult to classify for your current model—such samples are the most likely to help improve performance.

This concludes the universal workflow of machine learning—that's a lot of things to keep in mind. It takes time and experience to become an expert, but don't worry, you're already a lot wiser than you were a few chapters ago. You are now familiar with the big picture—the entire spectrum of what machine learning projects entail. While most of this book will focus on the model development part, you're now aware that it's only one part of the entire workflow. Always keep in mind the big picture!

Summary

- When you take on a new machine learning project, first, define the problem at hand:
  - Understand the broader context of what you're setting out to do—what's the end goal and what are the constraints?
  - Collect and annotate a dataset; make sure you understand your data in depth.
  - Choose how you'll measure success on your problem. What metrics will you monitor on your validation data?
- Once you understand the problem and you have an appropriate dataset, develop a model:
  - Prepare your data.
  - Pick your evaluation protocol. Hold-out validation? K-fold validation? Which portion of the data should you use for validation?
  - Achieve statistical power: beat a simple baseline.
  - Scale up: develop a model that can overfit.
  - Regularize your model and tune its hyperparameters, based on performance on the validation data. A lot of machine learning research tends to focus only on this step—but keep the big picture in mind.
- When your model is ready and yields good performance on the test data, it's time for deployment:
  - First, make sure to set appropriate expectations with stakeholders.
  - Optimize a final model for inference, and ship the model to the deployment environment of choice—web server, mobile, browser, embedded device, etc.
  - Monitor your model's performance in production and keep collecting data so you can develop the next generation of the model.

7 A deep dive on Keras

This chapter covers

- The different ways to create Keras models: the `Sequential` class, the Functional API, and model subclassing
- How to use the built-in Keras training and evaluation loops, including how to use custom metrics and custom losses
- Using Keras callbacks to further customize how training proceeds
- Using TensorBoard for monitoring your training and evaluation metrics over time
- How to write your own training and evaluation loops from scratch

You're starting to have some amount of experience with Keras. You're familiar with the Sequential model, `Dense` layers, and built-in APIs for training, evaluation, and inference—`compile()`, `fit()`, `evaluate()`, and `predict()`. You've even learned in chapter 3 how to inherit from the `Layer` class to create custom layers, and how to

use the gradient APIs in TensorFlow, JAX, and PyTorch to implement a step-by-step training loop.

In the coming chapters, we'll dig into computer vision, timeseries forecasting, natural language processing, and generative deep learning. These complex applications will require much more than a `Sequential` architecture and the default `fit()` loop. So let's first turn you into a Keras expert! In this chapter, you'll get a complete overview of the key ways to work with Keras APIs: everything you're going to need to handle the advanced deep learning use cases you'll encounter next.

7.1 A spectrum of workflows

The design of the Keras API is guided by the principle of *progressive disclosure of complexity*: make it easy to get started, yet make it possible to handle high-complexity use cases, only requiring incremental learning at each step. Simple use cases should be easy and approachable, and arbitrarily advanced workflows should be *possible*: no matter how niche and complex the thing you want to do, there should be a clear path to it: a path that builds upon the various things you've learned from simpler workflows. This means that you can grow from beginner to expert and still use the same tools—only in different ways.

As such, there's not a single "true" way of using Keras. Rather, Keras offers a *spectrum of workflows*, from the very simple to the very flexible. There are different ways to build Keras models, and different ways to train them, answering different needs.

For instance, you have a range of ways to build models and an array of ways to train them, each representing a certain tradeoff between usability and flexibility. You could be using Keras like you would use scikit-learn—just calling `fit()` and letting the framework do its thing—or you could be using it like NumPy—taking full control of every little detail.

Because all these workflows are based on shared APIs, such as `Layer` and `Model`, components from any workflow can be used in any other workflow: they can all talk to each other. This means that everything you're learning now as you're getting started will still be relevant once you've become an expert. You can get started easily and then gradually dive into workflows where you're writing more and more logic from scratch. You won't have to switch to an entirely different framework as you go from student to researcher, or from data scientist to deep learning engineer.

This philosophy is not unlike that of Python itself! Some languages only offer one way to write programs—for instance, object-oriented programming or functional programming. Meanwhile, Python is a multiparadigm language: it offers a range of possible usage patterns, which all work nicely together. This makes Python suitable for a wide range of very different use cases: system administration, data science, machine learning engineering, web development, or just learning how to program. Likewise, you can think of Keras as the Python of deep learning: a user-friendly deep learning language that offers a variety of workflows for different user profiles.

7.2 Different ways to build Keras models

There are three APIs for building models in Keras, as shown in figure 7.1:

- The *Sequential model* is the most approachable API—it's basically a Python list. As such, it's limited to simple stacks of layers.
- The *Functional API*, which focuses on graph-like model architectures. It represents a nice mid-point between usability and flexibility, and as such, it's the most commonly used model-building API.
- *Model subclassing*, a low-level option where you write everything yourself from scratch. This is ideal if you want full control over every little thing. However, you won't get access to many built-in Keras features, and you will be more at risk of making mistakes.

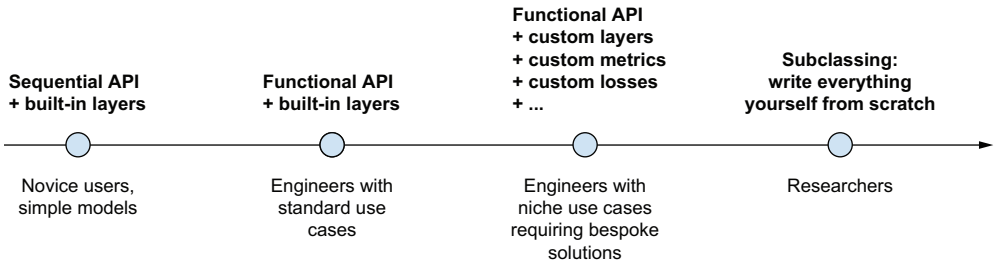


Figure 7.1 Progressive disclosure of complexity for model building

7.2.1 The Sequential model

The simplest way to build a Keras model is the Sequential model, which you already know about.

Listing 7.1 The `Sequential` class

```
import keras
from keras import layers

model = keras.Sequential(
    [
        layers.Dense(64, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)
```

Note that it's possible to build the same model incrementally via the `add()` method, similar to the `append()` method of a Python list.

Listing 7.2 Incrementally building a Sequential model

```
model = keras.Sequential()
model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(10, activation="softmax"))
```

You've seen in chapter 3 that layers only get built (which is to say, create their weights) when they are called for the first time. That's because the shape of the layers' weights depends on the shape of their input: until the input shape is known, they can't be created.

As such, the previous Sequential model does not have any weights until you actually call it on some data, or call its `build()` method with an input shape.

Listing 7.3 Models that aren't yet built have no weights

```
>>> model.weights
[]
```

At that point, the model isn't built yet.

Listing 7.4 Calling a model for the first time to build it

Builds the model. Now the model will expect samples of shape (3,). The None in the input shape signals that the batch size could be anything.

Now you can retrieve the model's weights.

```
>>> model.build(input_shape=(None, 3))
>>> model.weights
[<Variable shape=(3, 64), dtype=float32, path=sequential/dense_2/kernel ...>,
 <Variable shape=(64,), dtype=float32, path=sequential/dense_2/bias ...>,
 <Variable shape=(64, 10), dtype=float32, path=sequential/dense_3/kernel ...>,
 <Variable shape=(10,), dtype=float32, path=sequential/dense_3/bias ...>>]
```

After the model is built, you can display its contents via the `summary()` method, which comes in handy for debugging.

Listing 7.5 The summary method

```
>>> model.summary()
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 64)	256
dense_3 (Dense)	(None, 10)	650

```
Total params: 906 (3.54 KB)
Trainable params: 906 (3.54 KB)
Non-trainable params: 0 (0.00 B)
```

As you can see, your model happens to be named `sequential_1`. You can actually give names to everything in Keras—every model, every layer.

Listing 7.6 Naming models and layers with the `name` argument

```
>>> model = keras.Sequential(name="my_example_model")
>>> model.add(layers.Dense(64, activation="relu", name="my_first_layer"))
>>> model.add(layers.Dense(10, activation="softmax", name="my_last_layer"))
>>> model.build((None, 3))
>>> model.summary()
Model: "my_example_model"
```

Layer (type)	Output Shape	Param #
my_first_layer (Dense)	(None, 64)	256
my_last_layer (Dense)	(None, 10)	650

```
Total params: 906 (3.54 KB)
Trainable params: 906 (3.54 KB)
Non-trainable params: 0 (0.00 B)
```

When building a Sequential model incrementally, it's useful to be able to print a summary of what the current model looks like after you add each layer. But you can't print a summary until the model is built! There's actually a way to have your Sequential model get built on the fly: just declare the shape of the model's inputs in advance. You can do this via the `Input` class.

Listing 7.7 Specifying the input shape of your model in advance

```
model = keras.Sequential()
model.add(keras.Input(shape=(3,)))
model.add(layers.Dense(64, activation="relu"))
```

Use an Input to declare the shape of the inputs. Note that the shape argument must be the shape of each sample, not the shape of one batch.

Now you can use `summary()` to follow how the output shape of your model changes as you add more layers:

```
>>> model.summary()
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 64)	256

```
Total params: 256 (1.00 KB)
```

```
 Trainable params: 256 (1.00 KB)
 Non-trainable params: 0 (0.00 B)

>>> model.add(layers.Dense(10, activation="softmax"))
>>> model.summary()
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 64)	256
dense_5 (Dense)	(None, 10)	650

```
Total params: 906 (3.54 KB)
Trainable params: 906 (3.54 KB)
Non-trainable params: 0 (0.00 B)
```

This is a pretty common debugging workflow when dealing with layers that transform their inputs in complex ways, such as the convolutional layers you'll learn about in chapter 8.

7.2.2 The Functional API

The Sequential model is easy to use, but its applicability is extremely limited: it can only express models with a single input and a single output, applying one layer after the other in a sequential fashion. In practice, it's pretty common to encounter models with multiple inputs (say, an image and its metadata), multiple outputs (different things you want to predict about the data), or a nonlinear topology.

In such cases, you'd build your model using the Functional API. This is what most Keras models you'll encounter in the wild use. It's fun and powerful—it feels like playing with LEGO bricks.

A SIMPLE EXAMPLE

Let's start with something simple: the two-layer stack we used in the previous section. Its Functional API version looks like the following listing.

Listing 7.8 A simple Functional model with two `Dense` layers

```
inputs = keras.Input(shape=(3,), name="my_input")
features = layers.Dense(64, activation="relu")(inputs)
outputs = layers.Dense(10, activation="softmax")(features)
model = keras.Model(inputs=inputs, outputs=outputs, name="my_functional_model")
```

Let's go over this step by step. We started by declaring an `Input` (note that you can also give names to these input objects, like everything else):

```
inputs = keras.Input(shape=(3,), name="my_input")
```

This `inputs` object holds information about the shape and `dtype` of the data that the model will process:

```
>>> inputs.shape
(None, 3)
>>> inputs.dtype
"float32"
```

The model will process batches where each sample has shape (3,). The number of samples per batch is variable (indicated by the None batch size).

These batches will have dtype float32.

We call such an object a *symbolic tensor*. It doesn't contain any actual data, but it encodes the specifications of the actual tensors of data that the model will see when you use it. It *stands for* future tensors of data.

Next, we created a layer and called it on the input:

```
features = layers.Dense(64, activation="relu")(inputs)
```

All Keras layers can be called both on real tensors of data or on these symbolic tensors. In the latter case, they return a new symbolic tensor, with updated shape and dtype information:

```
>>> features.shape
(None, 64)
```

After obtaining the final outputs, we instantiated the model by specifying its inputs and outputs in the `Model` constructor:

```
outputs = layers.Dense(10, activation="softmax")(features)
model = keras.Model(inputs=inputs, outputs=outputs, name="my_functional_model")
```

Here's the summary of our model:

```
>>> model.summary()
Model: "my_functional_model"
```

Layer (type)	Output Shape	Param #
my_input (InputLayer)	(None, 3)	0
dense_8 (Dense)	(None, 64)	256
dense_9 (Dense)	(None, 10)	650

```
Total params: 906 (3.54 KB)
Trainable params: 906 (3.54 KB)
Non-trainable params: 0 (0.00 B)
```

MULTI-INPUT, MULTI-OUTPUT MODELS

Unlike this toy model, most deep learning models don't look like lists—they look like graphs. They may, for instance, have multiple inputs or multiple outputs. It's for this kind of model that the Functional API really shines.

Let's say you're building a system to rank customer support tickets by priority and route them to the appropriate department. Your model has three inputs:

- The title of the ticket (text input)
- The text body of the ticket (text input)
- Any tags added by the user (categorical input, assumed here to be multi-hot encoded)

We can encode the text inputs as arrays of 1s and 0s of size `vocabulary_size` (see chapter 14 for detailed information about text encoding techniques).

Your model also has two outputs:

- The priority score of the ticket, a scalar between 0 and 1 (sigmoid output)
- The department that should handle the ticket (a softmax over the set of departments)

You can build this model in a few lines with the Functional API.

Listing 7.9 A multi-input, multi-output Functional model

```
vocabulary_size = 10000
num_tags = 100
num_departments = 4

title = keras.Input(shape=(vocabulary_size,), name="title")
text_body = keras.Input(shape=(vocabulary_size,), name="text_body")
tags = keras.Input(shape=(num_tags,), name="tags")

features = layers.Concatenate()([title, text_body, tags])
features = layers.Dense(64, activation="relu", name="dense_features")(features)

priority = layers.Dense(1, activation="sigmoid", name="priority")(features)
department = layers.Dense(
    num_departments, activation="softmax", name="department"
)(features)
```

Applies intermediate layer to recombine input features into richer representations

Combines input features into a single tensor, features, by concatenating them

Defines model inputs

Defines model outputs

```
model = keras.Model(
    inputs=[title, text_body, tags],
    outputs=[priority, department],
)
```

Creates the model by specifying its inputs and outputs

The Functional API is a simple, LEGO-like, yet very flexible way to define arbitrary graphs of layers like these.

TRAINING A MULTI-INPUT, MULTI-OUTPUT MODEL

You can train your model in much the same way as you would train a Sequential model, by calling `fit()` with lists of input and output data. These lists of data should respect the same order as the inputs you passed to the `Model()` constructor.

Listing 7.10 Training a model by providing lists of input and target arrays

```
import numpy as np

num_samples = 1280

title_data = np.random.randint(0, 2, size=(num_samples, vocabulary_size))
text_body_data = np.random.randint(0, 2, size=(num_samples, vocabulary_size))
tags_data = np.random.randint(0, 2, size=(num_samples, num_tags))

priority_data = np.random.random(size=(num_samples, 1))
department_data = np.random.randint(0, num_departments, size=(num_samples, 1))

model.compile(
    optimizer="adam",
    loss=["mean_squared_error", "sparse_categorical_crossentropy"],
    metrics=[["mean_absolute_error"], ["accuracy"]],
)
model.fit(
    [title_data, text_body_data, tags_data],
    [priority_data, department_data],
    epochs=1,
)
model.evaluate(
    [title_data, text_body_data, tags_data], [priority_data, department_data]
)
priority_preds, department_preds = model.predict(
    [title_data, text_body_data, tags_data]
)
```

Dummy input data

Dummy target data

If you don't want to rely on input order (for instance, because you have many inputs or outputs), you can also use the names you gave to the `Input` objects and to the output layers, and pass data via dictionaries.

Listing 7.11 Training a model by providing dicts of input and target arrays

```
model.compile(
    optimizer="adam",
    loss={
        "priority": "mean_squared_error",
        "department": "sparse_categorical_crossentropy",
    },
    metrics={
        "priority": ["mean_absolute_error"],
        "department": ["accuracy"],
    },
)
model.fit(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data},
    {"priority": priority_data, "department": department_data},
    epochs=1,
)
model.evaluate(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data},
    {"priority": priority_data, "department": department_data},
)
priority_preds, department_preds = model.predict(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data}
)
```

The power of the Functional API: Access to layer connectivity

A Functional model is an explicit graph data structure. This makes it possible to *inspect how layers are connected* and *reuse previous graph nodes* (which are layer outputs) as part of new models. It also nicely fits the "mental model" that most researchers use when thinking about a deep neural network: a graph of layers.

This enables two important use cases: model visualization and feature extraction. Let's take a look.

Plotting layer connectivity

Let's visualize the connectivity of the model we just defined (the *topology* of the model). You can plot a Functional model as a graph with the `plot_model()` utility, as shown in figure 7.2:

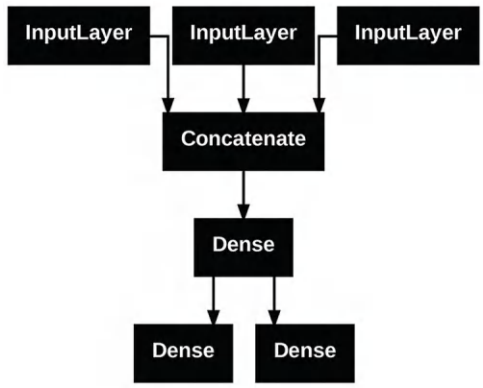


Figure 7.2 Plot generated by `plot_model()` on our ticket classifier model

```
keras.utils.plot_model(model, "ticket_classifier.png")
```

You can add to this plot the input and output shapes of each layer in the model, as well as layer names (rather than just layer types), which can be helpful during debugging (figure 7.3):

```
keras.utils.plot_model(
    model,
    "ticket_classifier_with_shape_info.png",
    show_shapes=True,
    show_layer_names=True,
)
```

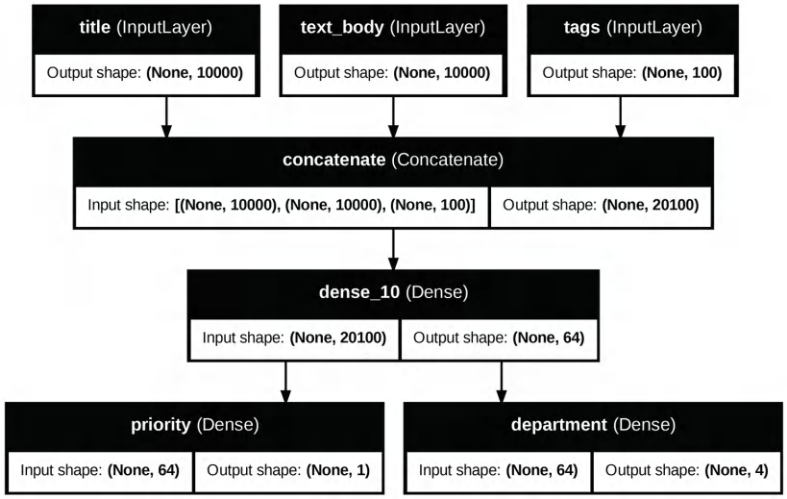


Figure 7.3 Model plot with shape information added

The `None` in the tensor shapes represents the batch size: this model allows batches of any size.

FEATURE EXTRACTION WITH A FUNCTIONAL MODEL

Access to layer connectivity also means that you can inspect and reuse individual nodes (layer calls) in the graph. The model property `model.layers` provides the list of layers that make up the model, and for each layer, you can query `layer.input` and `layer.output`.

Listing 7.12 Retrieving the inputs or outputs of a layer in a Functional model

```
>>> model.layers
[<InputLayer name=title, built=True>,
```

```
 <InputLayer name=text_body, built=True>,
 <InputLayer name=tags, built=True>,
 <Concatenate name=concatenate, built=True>,
 <Dense name=dense_10, built=True>,
 <Dense name=priority, built=True>,
 <Dense name=department, built=True>]

>>> model.layers[3].input
[<KerasTensor shape=(None, 10000), dtype=float32, sparse=None,
name=title>,
 <KerasTensor shape=(None, 10000), dtype=float32, sparse=None, name=text_body>,
 <KerasTensor shape=(None, 100), dtype=float32, sparse=None, name=tags>]

>>> model.layers[3].output
<KerasTensor shape=(None, 20100), dtype=float32, sparse=False>
```

This enables you to do *feature extraction*: creating models that reuse intermediate features from another model.

Let's say you want to add another output to the model we previously defined—you want to also predict an estimate of how long a given issue ticket will take to resolve, a kind of difficulty rating. You could do this via a classification layer over three categories—"quick," "medium," and "difficult." You don't need to recreate and retrain a model from scratch! You can just start from the intermediate features of your previous model, since you have access to them.

Listing 7.13 Creating a new model by reusing intermediate layer outputs

```
features = model.layers[4].output
difficulty = layers.Dense(3, activation="softmax", name="difficulty")(features)

new_model = keras.Model(
    inputs=[title, text_body, tags], outputs=[priority, department, difficulty]
)
```

layers[4] is our intermediate Dense layer.

Let's plot our new model, as shown in figure 7.4:

```
keras.utils.plot_model(
    new_model,
    "updated_ticket_classifier.png",
    show_shapes=True,
    show_layer_names=True,
)
```

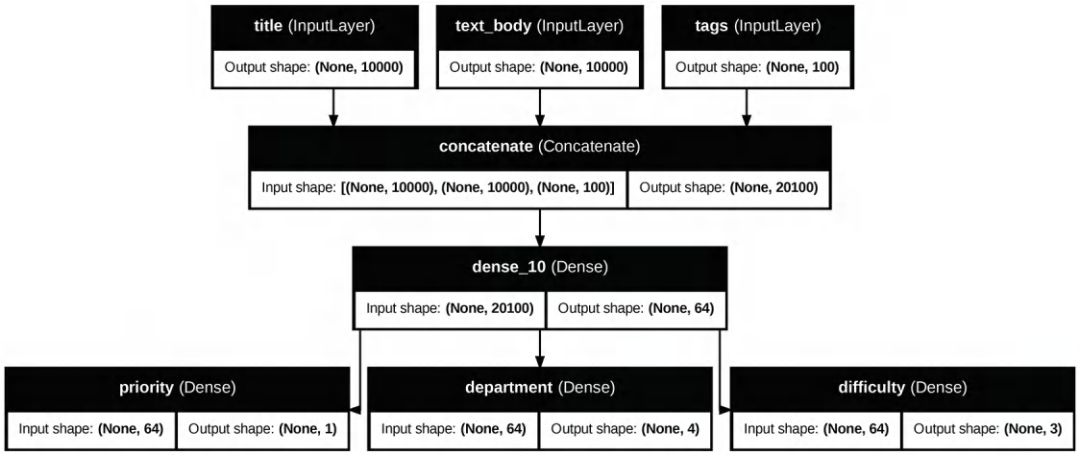


Figure 7.4 Plot of our new model

7.2.3 Subclassing the Model class

The last model-building pattern you should know about is the most advanced one: `Model` subclassing. You've already learned in chapter 3 how to subclass the `Layer` class to create custom layers. Subclassing `Model` is pretty similar:

- In the `__init__()` method, define the layers the model will use.
- In the `call` method, define the forward pass of the model, reusing the layers previously created.
- Instantiate your subclass and call it on data to create its weights.

REWRITING OUR PREVIOUS EXAMPLE AS A SUBCLASSED MODEL

Let's take a look at a simple example: we will reimplement the customer support ticket management model using a `Model` subclass.

Listing 7.14 A simple subclassed model

```
class CustomerTicketModel(keras.Model):
    def __init__(self, num_departments):
        super().__init__()
        self.concat_layer = layers.Concatenate()
        self.mixing_layer = layers.Dense(64, activation="relu")
        self.priority_scorer = layers.Dense(1, activation="sigmoid")
        self.department_classifier = layers.Dense(
            num_departments, activation="softmax"
        )

    def call(self, inputs):
        title = inputs["title"]
```

Don't forget to call the super constructor!

Defines sublayers in the constructor

Defines the forward pass in the call() method

```
        text_body = inputs["text_body"]
        tags = inputs["tags"]

        features = self.concat_layer([title, text_body, tags])
        features = self.mixing_layer(features)
        priority = self.priority_scorer(features)
        department = self.department_classifier(features)
        return priority, department
```

Once you've defined the model, you can instantiate it. Note that it will only create its weights the first time you call it on some data—much like `Layer` subclasses:

```
model = CustomerTicketModel(num_departments=4)

priority, department = model(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data}
)
```

So far, everything looks very similar to `Layer` subclassing, a workflow you've already encountered in chapter 3. What, then, is the difference between a `Layer` subclass and a `Model` subclass? It's simple: a *layer* is a building block you use to create models, and a *model* is the top-level object that you will actually train, export for inference, etc. In short, a `Model` has a `fit()`, `evaluate()`, and `predict()` method. Layers don't. Other than that, the two classes are virtually identical (another difference is that you can *save* a model to a file on disk—which we will cover in a few sections).

You can compile and train a `Model` subclass just like a Sequential or Functional model:

The structure of what you pass as the loss and metrics must match exactly what gets returned by call()—since we returned a list of two elements, so should loss and metrics be lists of two elements.

The structure of the input data must match exactly what is expected by the call() method, and the structure of the target data must match exactly what gets returned by the call() method. Here, the input data must be a dict with three keys (title, text_body, and tags) and the target data must be a list of two elements.

```
model.compile(
    optimizer="adam",
    loss=["mean_squared_error", "sparse_categorical_crossentropy"],
    metrics=[["mean_absolute_error"], ["accuracy"]],
)
model.fit(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data},
    [priority_data, department_data],
    epochs=1,
)
```

```
model.evaluate(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data},
    [priority_data, department_data],
)
priority_preds, department_preds = model.predict(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data}
)
```

The `Model` subclassing workflow is the most flexible way to build a model: it enables you to build models that cannot be expressed as directed acyclic graphs of layers—imagine, for instance, a model where the `call()` method uses layers inside a `for` loop, or even calls them recursively. Anything is possible—you're in charge.

BEWARE: WHAT SUBCLASSED MODELS DON'T SUPPORT

This freedom comes at a cost: with subclassed models, you are responsible for more of the model logic, which means your potential error surface is much larger. As a result, you will have more debugging work to do. You are developing a new Python object, not just snapping together LEGO bricks.

Functional and subclassed models are also substantially different in nature: a Functional model is an explicit data structure—a graph of layers, which you can view, inspect, and modify. Meanwhile, a subclassed model is a piece of bytecode—a Python class with a `call()` method that contains raw code. This is the source of the subclassing workflow's flexibility—you can just code up whatever functionality you like—but it introduces new limitations.

For instance, because the way layers are connected to each other is hidden inside the body of the `call()` method, you cannot access that information. Calling `summary()` will not display layer connectivity, and you cannot plot the model topology via `plot_model()`. Likewise, if you have a subclassed model, you cannot access the nodes of the graph of layers to do feature extraction—because there is simply no graph. Once the model is instantiated, its forward pass becomes a complete black box.

7.2.4 Mixing and matching different components

Crucially, choosing one of these patterns—the Sequential model, the Functional API, `Model` subclassing—does not lock you out of the others. All models in the Keras API can smoothly interoperate with each other, whether they're Sequential models, Functional models, or subclassed models written from scratch. They're all part of the same spectrum of workflows. For instance, you can use a subclassed layer or model in a Functional model.

Listing 7.15 Creating a Functional model that includes a subclassed model

```
class Classifier(keras.Model):
    def __init__(self, num_classes=2):
        super().__init__()
```

```
        if num_classes == 2:
            num_units = 1
            activation = "sigmoid"
        else:
            num_units = num_classes
            activation = "softmax"
        self.dense = layers.Dense(num_units, activation=activation)

    def call(self, inputs):
        return self.dense(inputs)

inputs = keras.Input(shape=(3,))
features = layers.Dense(64, activation="relu")(inputs)
outputs = Classifier(num_classes=10)(features)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Inversely, you can use a Functional model as part of a subclassed layer or model.

Listing 7.16 Creating a subclassed model that includes a Functional model

```
inputs = keras.Input(shape=(64,))
outputs = layers.Dense(1, activation="sigmoid")(inputs)
binary_classifier = keras.Model(inputs=inputs, outputs=outputs)

class MyModel(keras.Model):
    def __init__(self, num_classes=2):
        super().__init__()
        self.dense = layers.Dense(64, activation="relu")
        self.classifier = binary_classifier

    def call(self, inputs):
        features = self.dense(inputs)
        return self.classifier(features)

model = MyModel()
```

7.2.5 Remember: Use the right tool for the job

You've learned about the spectrum of workflows for building Keras models, from the simplest workflow—the Sequential model—to the most advanced one, model subclassing. When should you use one over the other? Each one has its pros and cons—pick the one most suitable for the job at hand.

In general, the Functional API provides you with a pretty good tradeoff between ease of use and flexibility. It also gives you direct access to layer connectivity, which is very powerful for use cases such as model plotting or feature extraction. If you *can* use the Functional API—that is, if your model can be expressed as a directed acyclic graph of layers—we recommend using it over model subclassing.

Going forward, all examples in this book will use the Functional API—simply because all of the models we will work with are expressible as graphs of layers. We will, however, make frequent use of subclassed layers. In general, using Functional models that include subclassed layers provides the best of both worlds: high development flexibility while retaining the advantages of the Functional API.

7.3 Using built-in training and evaluation loops

The principle of progressive disclosure of complexity—access to a spectrum of workflows that go from dead easy to arbitrarily flexible, one step at a time—also applies to model training. Keras provides you with different workflows for training models—it can be as simple as calling `fit()` on your data or as advanced as writing a new training algorithm from scratch.

You are already familiar with the `compile()`, `fit()`, `evaluate()`, `predict()` workflow. As a reminder, it looks like the following listing.

Listing 7.17 The standard workflow: `compile()`, `fit()`, `evaluate()`, `predict()`

```
from keras.datasets import mnist

def get_mnist_model():
    inputs = keras.Input(shape=(28 * 28,))
    features = layers.Dense(512, activation="relu")(inputs)
    features = layers.Dropout(0.5)(features)
    outputs = layers.Dense(10, activation="softmax")(features)
    model = keras.Model(inputs, outputs)
    return model

(images, labels), (test_images, test_labels) = mnist.load_data()
images = images.reshape((60000, 28 * 28)).astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28)).astype("float32") / 255
train_images, val_images = images[10000:], images[:10000]
train_labels, val_labels = labels[10000:], labels[:10000]

model = get_mnist_model()
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
model.fit(
    train_images,
    train_labels,
    epochs=3,
    validation_data=(val_images, val_labels),
)
test_metrics = model.evaluate(test_images, test_labels)
predictions = model.predict(test_images)
```

Creates a model. (We factor this into a separate function so as to reuse it later.)

Loads your data, reserving some for validation

Compiles the model by specifying its optimizer, the loss function to minimize, and metrics to monitor

Uses fit() to train the model, optionally providing validation data to monitor performance on unseen data

Uses evaluate() to compute the loss and metrics on new data

Uses predict() to compute classification probabilities on new data

There are a couple of ways you can customize this simple workflow:

- By providing your own custom metrics
- By passing *callbacks* to the `fit()` method to schedule actions to be taken at specific points during training

Let's take a look at these.

7.3.1 Writing your own metrics

Metrics are key to measuring the performance of your model—in particular, to measure the difference between its performance on the training data and its performance on the test data. Commonly used metrics for classification and regression are already part of the built-in `keras.metrics` module—most of the time, that's what you will use. But if you're doing anything out of the ordinary, you will need to be able to write your own metrics. It's simple!

A Keras metric is a subclass of the `keras.metrics.Metric` class. Similarly to layers, a metric has an internal state stored in Keras variables. Unlike layers, these variables aren't updated via backpropagation, so you have to write the state update logic yourself—which happens in the `update_state()` method. For example, here's a simple custom metric that measures the root mean squared error (RMSE).

Listing 7.18 Implementing a custom metric by subclassing the `Metric` class

```
from keras import ops

class RootMeanSquaredError(keras.metrics.Metric):
    def __init__(self, name="rmse", **kwargs):
        super().__init__(name=name, **kwargs)
        self.mse_sum = self.add_weight(name="mse_sum", initializer="zeros")
        self.total_samples = self.add_weight(
            name="total_samples", initializer="zeros"
        )

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_true = ops.one_hot(y_true, num_classes=ops.shape(y_pred)[1])
        mse = ops.sum(ops.square(y_true - y_pred))
        self.mse_sum.assign_add(mse)
        num_samples = ops.shape(y_pred)[0]
        self.total_samples.assign_add(num_samples)
```

Defines the state variables in the constructor. Like for layers, you have access to the add_weight() method.

Subclasses the Metric class

Implements the state update logic in update_state(). The y_true argument is the targets (or labels) for one batch, while y_pred represents the corresponding predictions from the model. To match our MNIST model, we expect categorical predictions and integer labels. You can ignore the sample_weight argument; we won't use it here.

You use the `result()` method to return the current value of the metric:

```
def result(self):
    return ops.sqrt(self.mse_sum / self.total_samples)
```

Meanwhile, you also need to expose a way to reset the metric state without having to reinstantiate it—this enables the same metric objects to be used across different epochs of training or across both training and evaluation. You do this in the `reset_state()` method:

```
def reset_state(self):
    self.mse_sum.assign(0.)
    self.total_samples.assign(0.)
```

Custom metrics can be used just like built-in ones. Let's test-drive our own metric:

```
model = get_mnist_model()
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy", RootMeanSquaredError()],
)
model.fit(
    train_images,
    train_labels,
    epochs=3,
    validation_data=(val_images, val_labels),
)
test_metrics = model.evaluate(test_images, test_labels)
```

You can now see the `fit()` progress bar display the RMSE of your model.

7.3.2 Using callbacks

Launching a training run on a large dataset for tens of epochs using `model.fit()` can be a bit like launching a paper airplane: past the initial impulse, you don't have any control over its trajectory or its landing spot. If you want to avoid bad outcomes (and thus wasted paper airplanes), it's smarter to use, not a paper plane, but a drone that can sense its environment, send data back to its operator, and automatically make steering decisions based on its current state. The Keras *callbacks* API will help you transform your call to `model.fit()` from a paper airplane into a smart, autonomous drone that can self-introspect and dynamically take action.

A *callback* is an object (a class instance implementing specific methods) that is passed to the model in the call to `fit()` and that is called by the model at various points during

training. It has access to all the available data about the state of the model and its performance, and it can take action: interrupt training, save a model, load a different weight set, or otherwise alter the state of the model.

Here are some examples of ways you can use callbacks:

- *Model checkpointing*—Saving the current state of the model at different points during training.
- *Early stopping*—Interrupting training when the validation loss is no longer improving (and of course, saving the best model obtained during training).
- *Dynamically adjusting the value of certain parameters during training*—Such as the learning rate of the optimizer.
- *Logging training and validation metrics during training, or visualizing the representations learned by the model as they're updated*—The `fit()` progress bar that you're familiar with is in fact a callback!

The `keras.callbacks` module includes a number of built-in callbacks (this is not an exhaustive list):

```
keras.callbacks.ModelCheckpoint
keras.callbacks.EarlyStopping
keras.callbacks.LearningRateScheduler
keras.callbacks.ReduceLROnPlateau
keras.callbacks.CSVLogger
```

Let's review two of them to give you an idea of how to use them: `EarlyStopping` and `ModelCheckpoint`.

The EarlyStopping and ModelCheckpoint callbacks

When you're training a model, there are many things you can't predict at the start. In particular, you can't tell how many epochs will be needed to get to an optimal validation loss. Our examples so far have adopted the strategy of training for enough epochs that you begin overfitting, using the first run to figure out the optimal number of epochs, and then finally launching a new training run from scratch using this optimal number. Of course, this approach is wasteful. A much better way to handle this is to stop training when you measure that the validation loss is no longer improving. This can be achieved using the `EarlyStopping` callback.

The `EarlyStopping` callback interrupts training once a target metric being monitored has stopped improving for a fixed number of epochs. For instance, this callback allows you to interrupt training as soon as you start overfitting, thus avoiding having to retrain your model for a smaller number of epochs. This callback is typically used in combination with `ModelCheckpoint`, which lets you continually save the model during training (and, optionally, save only the current best model so far: the version of the model that achieved the best performance at the end of an epoch).

Listing 7.19 Using the `callbacks` argument in the `fit()` method

```
callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor="accuracy",
        patience=1,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath="checkpoint_path.keras",
        monitor="val_loss",
        save_best_only=True,
    ),
]
model = get_mnist_model()
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
model.fit(
    train_images,
    train_labels,
    epochs=10,
    callbacks=callbacks_list,
    validation_data=(val_images, val_labels),
)
```

Callbacks are passed to the model via the callbacks argument in fit(), which takes a list of callbacks. You can pass any number of callbacks.

Interrupts training when improvement stops

Monitors the model's validation accuracy

Interrupts training when accuracy has stopped improving for more than one epoch (that is, two epochs)

Saves the current weights after every epoch

Path to the destination model file

These two arguments mean you won't overwrite the model file unless val_loss has improved, which allows you to keep the best model seen during training.

You monitor accuracy, so it should be part of the model's metrics.

Because the callback will monitor validation loss and validation accuracy, you need to pass validation_data to the call to fit().

Note that you can always save models manually after training as well—just call `model.save("my_checkpoint_path.keras")`. To reload the model you've saved, use

```
model = keras.models.load_model("checkpoint_path.keras")
```

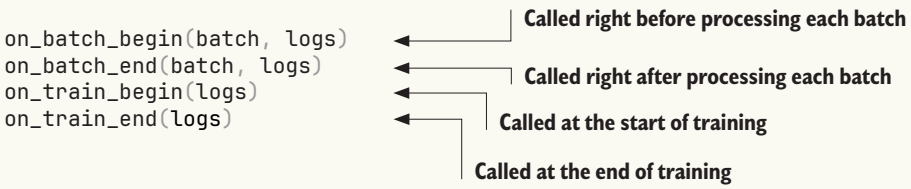
7.3.3 Writing your own callbacks

If you need to take a specific action during training that isn't covered by one of the built-in callbacks, you can write your own callback. Callbacks are implemented by subclassing the class `keras.callbacks.Callback`. You can then implement any number of the following transparently named methods, which are called at various points during training:

```
on_epoch_begin(epoch, logs)
on_epoch_end(epoch, logs)
```

Called at the start of every epoch

Called at the end of every epoch



These methods are all called with a `logs` argument, which is a dictionary containing information about the previous batch, epoch, or training run: training and validation metrics, and so on. The `on_epoch_*` and `on_batch_*` methods also take the epoch or batch index as first argument (an integer).

Here's a simple example callback that saves a list of per-batch loss values during training and plots these values at the end of each epoch.

Listing 7.20 Creating a custom callback by subclassing the `Callback` class

```
from matplotlib import pyplot as plt

class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs):
        self.per_batch_losses = []

    def on_batch_end(self, batch, logs):
        self.per_batch_losses.append(logs.get("loss"))

    def on_epoch_end(self, epoch, logs):
        plt.clf()
        plt.plot(
            range(len(self.per_batch_losses)),
            self.per_batch_losses,
            label="Training loss for each batch",
        )
        plt.xlabel(f"Batch (epoch {epoch})")
        plt.ylabel("Loss")
        plt.legend()
        plt.savefig(f"plot_at_epoch_{epoch}", dpi=300)
        self.per_batch_losses = []
```

Let's test-drive it:

```
model = get_mnist_model()
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
model.fit(
```

```
    train_images,
    train_labels,
    epochs=10,
    callbacks=[LossHistory()],
    validation_data=(val_images, val_labels),
)
```

We get plots that look like figure 7.5.

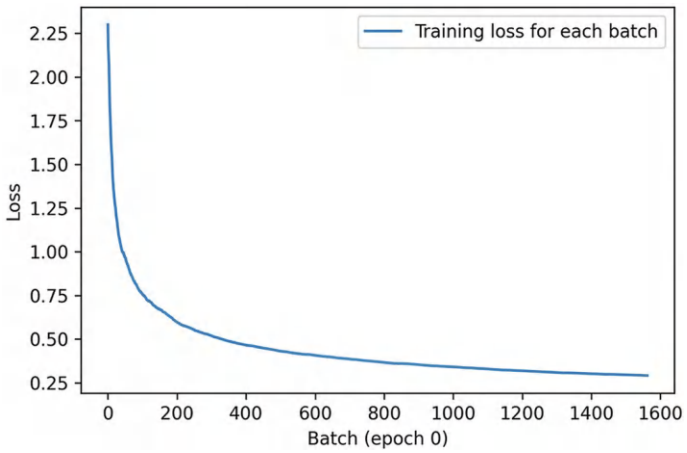


Figure 7.5 The output of our custom history–plotting callback

7.3.4 Monitoring and visualization with TensorBoard

To do good research or develop good models, you need rich, frequent feedback about what's going on inside your models during your experiments. That's the point of running experiments: to get information about how well a model performs—as much information as possible. Making progress is an iterative process, a loop: you start with an idea and express it as an experiment, attempting to validate or invalidate your idea. You run this experiment and process the information it generates, as shown in figure 7.6. This inspires your next idea. The more iterations of this loop you're able to run, the more refined and powerful your ideas become. Keras helps you go from idea to experiment in the least possible time, and fast GPUs can help you get from experiment to result as quickly as possible. But what about processing the experiment results? That's where TensorBoard comes in.

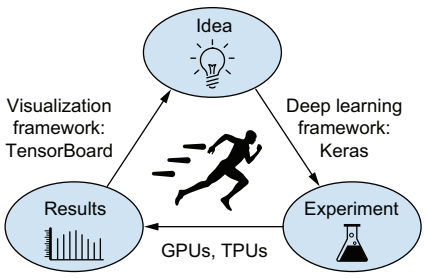


Figure 7.6 The loop of progress

TensorBoard is a browser-based application that you can run locally. It's the best way to monitor everything that goes on inside your model during training. With TensorBoard, you can

- Visually monitor metrics during training
- Visualize your model architecture
- Visualize histograms of activations and gradients
- Explore embeddings in 3D

If you're monitoring more information than just the model's final loss, you can develop a clearer vision of what the model does and doesn't do, and you can make progress more quickly.

The easiest way to use TensorBoard with a Keras model and the `fit()` method is the `keras.callbacks.TensorBoard` callback. In the simplest case, just specify where you want the callback to write logs, and you're good to go:

```
model = get_mnist_model()
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)

tensorboard = keras.callbacks.TensorBoard(
    log_dir="/full_path_to_your_log_dir",
)
model.fit(
    train_images,
    train_labels,
    epochs=10,
    validation_data=(val_images, val_labels),
    callbacks=[tensorboard],
)
```

Once the model starts running, it will write logs at the target location. If you are running your Python script on a local machine, you can then launch the local TensorBoard server using the following command (note that the `tensorboard` executable should already be available if you have installed TensorFlow via `pip`; if not, you can install TensorBoard manually via `pip install tensorboard`):

```
tensorboard --logdir /full_path_to_your_log_dir
```

You can then navigate to the URL that the command returns to access the TensorBoard interface.

If you are running your script in a Colab notebook, you can run an embedded TensorBoard instance as part of your notebook, using the following commands:

```
%load_ext tensorboard
%tensorboard --logdir /full_path_to_your_log_dir
```

In the TensorBoard interface, you will be able to monitor live graphs of your training and evaluation metrics, as shown in figure 7.7.

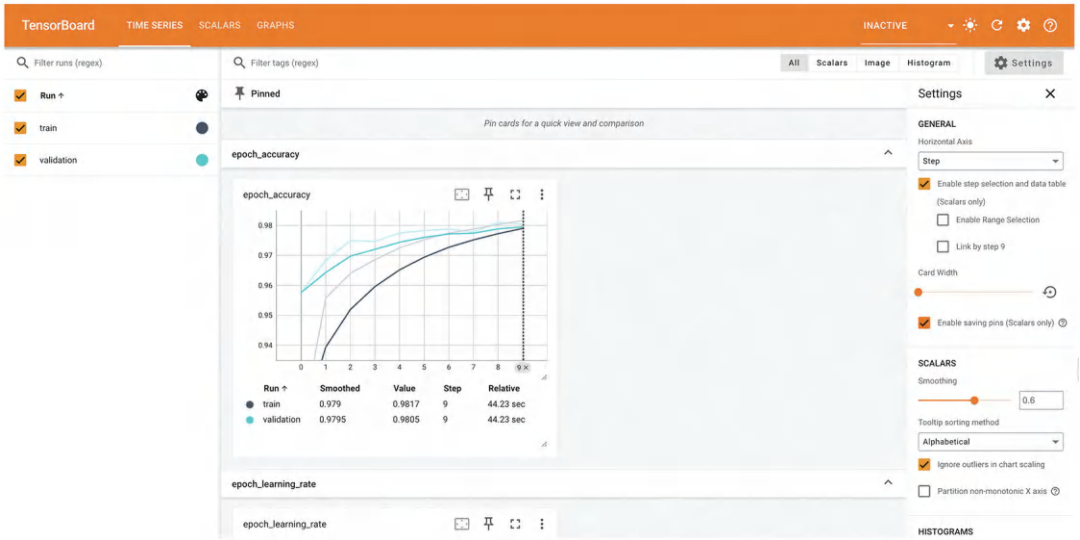


Figure 7.7 TensorBoard can be used for easy monitoring of training and evaluation metrics.

7.4 Writing your own training and evaluation loops

The `fit()` workflow strikes a nice balance between ease of use and flexibility. It's what you will use most of the time. However, it isn't meant to support everything a deep learning researcher may want to do—even with custom metrics, custom losses, and custom callbacks.

After all, the built-in `fit()` workflow is solely focused on *supervised learning*: a setup where there are known *targets* (also called *labels* or *annotations*) associated with your input data and where you compute your loss as a function of these targets and the model's predictions. However, not every form of machine learning falls into this category. There are other setups where no explicit targets are present, such as *generative learning* (which we will introduce in chapter 16), *self-supervised learning* (where targets are obtained from the inputs), or *reinforcement learning* (where learning is driven by occasional "rewards"—much like training a dog). And even if you're doing regular supervised learning, as a researcher, you may want to add some novel bells and whistles that require low-level flexibility.

Whenever you find yourself in a situation where the built-in `fit()` is not enough, you will need to write your own custom training logic. You've already seen simple examples of low-level training loops in chapters 2 and 3. As a reminder, the contents of a typical training loop look like this:

1. Run the "forward pass" (compute the model's output) to obtain a loss value for the current batch of data.
2. Retrieve the gradients of the loss with regard to the model's weights.
3. Update the model's weights so as to lower the loss value on the current batch of data.

These steps are repeated for as many batches as necessary. This is essentially what `fit()` does under the hood. In this section, you will learn to reimplement `fit()` from scratch, which will give you all the knowledge you need to write any training algorithm you may come up with.

Let's go over the details. Throughout the next few sections, you'll work your way up to writing a fully featured custom training loop in TensorFlow, PyTorch, and JAX.

7.4.1 Training vs. inference

In the low-level training loop examples you've seen so far, step 1 (the forward pass) was done via `predictions = model(inputs)`, and step 2 (retrieving the gradients computed by the gradient tape) was done via a backend-specific API, such as

- `gradients = tape.gradient(loss, model.weights)` in TensorFlow
- `loss.backward()` in PyTorch
- `jax.value_and_grad()` in JAX

In the general case, there are actually two subtleties you need to take into account.

Some Keras layers, such as the `Dropout` layer, have different behaviors during *training* and during *inference* (when you use them to generate predictions). Such layers expose a `training` Boolean argument in their `call()` method. Calling `dropout(inputs, training=True)` will drop some activation entries, while calling `dropout(inputs, training=False)` does nothing. By extension, Functional models and Sequential models also expose this `training` argument in their `call()` methods. Remember to pass `training=True` when you call a Keras model during the forward pass! Our forward pass thus becomes `predictions = model(inputs, training=True)`.

In addition, note that when you retrieve the gradients of the weights of your model, you should not use `model.weights`, but rather `model.trainable_weights`. Indeed, layers and models own two kinds of weights:

- *Trainable weights*, meant to be updated via backpropagation to minimize the loss of the model, such as the kernel and bias of a `Dense` layer.
- *Non-trainable weights*, which are meant to be updated during the forward pass by the layers that own them. For instance, if you wanted a custom layer to keep a counter of how many batches it has processed so far, that information would be

stored in a non-trainable weight, and at each batch, your layer would increment the counter by one.

Among Keras built-in layers, the only layer that features non-trainable weights is the `BatchNormalization` layer, which we will introduce in chapter 9. The `BatchNormalization` layer needs non-trainable weights to track information about the mean and standard deviation of the data that passes through it, so as to perform an online approximation of *feature normalization* (a concept you've learned about in chapters 4 and 6).

7.4.2 Writing custom training step functions

Taking into account these two details, a supervised learning training step ends up looking like this in pseudocode:

```
def train_step(inputs, targets):
    predictions = model(inputs, training=True)
    loss = loss_fn(targets, predictions)
    gradients = get_gradients_of(loss, wrt=model.trainable_weights)
    optimizer.apply(gradients, model.trainable_weights)
```

Runs the forward pass

Computes the loss for the current batch

Updates the model's trainable weights based on the gradients

Retrieves the gradients of the loss with regard to the model's trainable weights. This function doesn't actually exist!

This snippet is pseudocode rather than real code because it includes an imaginary function, `get_gradients_of()`. In reality, retrieving gradients is done in a way that is specific to your current backend—JAX, TensorFlow, or PyTorch.

Let's use what you learned about each framework in chapter 3 to implement a real version of this `train_step()` function. We'll start with TensorFlow and PyTorch because these two make the job relatively easy, so they're a good place to start. We'll end with JAX, which is quite a bit more complex.

A TensorFlow training step function

TensorFlow lets you write code that looks pretty much like our pseudocode snippet. The only difference is that your forward pass should take place inside a `GradientTape` scope. You can then use the `tape` object to retrieve the gradients:

```
import tensorflow as tf

model = get_mnist_model()
loss_fn = keras.losses.SparseCategoricalCrossentropy()
optimizer = keras.optimizers.Adam()
```

```
def train_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs, training=True)
        loss = loss_fn(targets, predictions)
    gradients = tape.gradient(loss, model.trainable_weights)
    optimizer.apply(gradients, model.trainable_weights)
    return loss
```

Opens a GradientTape

Runs the forward pass

Retrieves the gradients from the tape

Updates the model's trainable weights based on the gradients

Let's run it for a single step:

```
batch_size = 32
inputs = train_images[:batch_size]
targets = train_labels[:batch_size]
loss = train_step(inputs, targets)
```

Easy enough! Let's do PyTorch next.

A PyTorch training step function

When you use the PyTorch backend, all of your Keras layers and models inherit from the PyTorch `torch.nn.Module` class and expose the native `Module` API. As a result, your model, its trainable weights, and your loss tensor are all aware of each other and interact via three methods: `loss.backward()`, `weight.value.grad`, and `model.zero_grad()`.

As a reminder from chapter 3, the mental model you've got to keep in mind is this:

- With each forward pass, PyTorch builds up a one-time computation graph that keeps track of the computation that just happened.
- Calling `.backward()` on any given scalar node of this graph (like your loss) will run the graph backward starting from that node, automatically populating a `tensor.grad` attribute on all tensors involved (if they satisfy `requires_grad=True`), containing the gradient of the output node with respect to that tensor. In particular, it will populate the `grad` attribute of your trainable parameters.
- To clear the contents of that `tensor.grad` attribute, you should call `tensor.grad = None` on all your tensors. Because it would be a bit cumbersome to do this on all model variables individually, you can just do it at the model level via `model.zero_grad()`—the `zero_grad()` call will propagate to all variables tracked by the model. Clearing gradients is critical because calls to `backward()` are additive: if you don't clear the gradients at each step, the gradient values will accumulate and training won't proceed.

Let's chain these steps:

```
import torch

model = get_mnist_model()
loss_fn = keras.losses.SparseCategoricalCrossentropy()
optimizer = keras.optimizers.Adam()

def train_step(inputs, targets):
    predictions = model(inputs, training=True)
    loss = loss_fn(targets, predictions)
    loss.backward()
    gradients = [weight.value.grad for weight in model.trainable_weights]
    with torch.no_grad():
        optimizer.apply(gradients, model.trainable_weights)
    model.zero_grad()
    return loss
```

Runs the backward pass, populating gradient values

Runs the forward pass

Don't forget to clear the gradients!

Updates the model's trainable weights based on the gradients. This must be done in a no_grad() scope.

Recovers the gradient associated with each trainable variable. That weight.value is the PyTorch tensor that contains the variable's value.

Let's run it for a single step:

```
batch_size = 32
inputs = train_images[:batch_size]
targets = train_labels[:batch_size]
loss = train_step(inputs, targets)
```

That wasn't too difficult! Now, let's move on to JAX.

A JAX TRAINING STEP FUNCTION

When it comes to low-level training code, JAX tends to be the most complex of the three backends because of its fully stateless nature. Statelessness makes JAX highly performant and scalable, making it amenable to compilation and automatic performance optimizations. However, writing stateless code requires you to jump through some hoops.

Since the gradient function is obtained via metaprogramming, you first need to define the function that returns your loss. Further, this function needs to be stateless, so it needs to take as arguments all the variables it's going to be using, and it needs to return the value of any variable it has updated. Remember those non-trainable weights that can get modified during the forward pass? Those are the variables we need to return.

To make it easier to work with the stateless programming paradigm of JAX, Keras models make available a stateless forward pass method: the `stateless_call()` method. It behaves just like *`call`*, except that

- It takes as input the model's trainable weights and non-trainable weights, in addition to the `inputs` and `training` arguments.

- It returns the model's updated non-trainable weights, in addition to the model's outputs.

It works like this:

```
outputs, non_trainable_weights = model.stateless_call(
    trainable_weights, non_trainable_weights, inputs
)
```

We can use `stateless_call()` to implement our JAX loss function. Since the loss function also computes updates for all non-trainable variables, we name it `compute_loss_and_updates()`:

```
model = get_mnist_model()
loss_fn = keras.losses.SparseCategoricalCrossentropy()

def compute_loss_and_updates(
    trainable_variables, non_trainable_variables, inputs, targets
):
    outputs, non_trainable_variables = model.stateless_call(
        trainable_variables, non_trainable_variables, inputs, training=True
    )
    loss = loss_fn(targets, outputs)
    return loss, non_trainable_variables
```

Gradients are computed for the entries in the first argument (trainable_ variables here)

Calls stateless_call

Returns the scalar loss value and the updated non-trainable weights

Once we have this `compute_loss_and_updates()` function, we can pass it to `jax.value_and_grad` to obtain the gradient computation:

```
import jax

grad_fn = jax.value_and_grad(fn)
loss, gradients = grad_fn(...)
```

Now, there's just a small problem. Both `jax.grad()` and `jax.value_and_grad()` require `fn` to return a scalar value only. Our `compute_loss_and_updates()` function returns a scalar value as its first output, but it also returns the new value for the non-trainable weights. Remember what you learned in chapter 3? The solution is to pass a `has_aux` argument to `grad()` or `value_and_grad()`, like this:

```
import jax

grad_fn = jax.value_and_grad(compute_loss_and_updates, has_aux=True)
```

You would use it like this:

```
(loss, non_trainable_weights), gradients = grad_fn(
    trainable_variables, non_trainable_variables, inputs, targets
)
```

Okay, that was a lot of JAXiness. But now we've got almost everything we need to assemble our JAX training step. We just need the last piece of the puzzle: `optimizer.apply()`.

When you wrote your first basic training step in TensorFlow at the beginning of chapter 2, you wrote an update step function that looked like this:

```
learning_rate = 1e-3

def update_weights(gradients, weights):
    for g, w in zip(gradients, weights):
        w.assign(w - g * learning_rate)
```

This corresponds to what the optimizer `keras.optimizers.SGD` would do. However, every other optimizer in the Keras API is somewhat more complex than that and keeps track of auxiliary variables that help speed up training—in particular, most optimizers use some form of *momentum*, which you learned about in chapter 2. These extra variables get updated at each step of training, and in the JAX world, that means that you need to get your hands on a stateless function that takes these variables as arguments and returns their new value.

To make this easy, Keras makes available the `stateless_apply()` method on all optimizers. It works like this:

```
trainable_variables, optimizer_variables = optimizer.stateless_apply(
    optimizer_variables, grads, trainable_variables
)
```

Now, we have enough to assemble an end-to-end training step:

```
optimizer = keras.optimizers.Adam()
optimizer.build(model.trainable_variables)

def train_step(state, inputs, targets):
    (trainable_variables, non_trainable_variables, optimizer_variables) = state
    (loss, non_trainable_variables), grads = grad_fn(
        trainable_variables, non_trainable_variables, inputs, targets
    )
```

Unpacks the state

The state is part of the function arguments.

Computes gradients and updates to non-trainable variables

```
    trainable_variables, optimizer_variables = optimizer.stateless_apply(
        optimizer_variables, grads, trainable_variables
    )
    return loss, (
        trainable_variables,
        non_trainable_variables,
        optimizer_variables,
    )
```

Updates trainable variables and optimizer variables

Returns the updated state alongside the loss

Let's run it for a single step:

```
batch_size = 32
inputs = train_images[:batch_size]
targets = train_labels[:batch_size]

trainable_variables = [v.value for v in model.trainable_variables]
non_trainable_variables = [v.value for v in model.non_trainable_variables]
optimizer_variables = [v.value for v in optimizer.variables]

state = (trainable_variables, non_trainable_variables, optimizer_variables)
loss, state = train_step(state, inputs, targets)
```

It's definitely a bit more work than TensorFlow and PyTorch, but the speed and scalability benefits of JAX more than make up for it.

Next, let's take a look at another important element of a custom training loop: *metrics*.

7.4.3 Low-level usage of metrics

In a low-level training loop, you will probably want to use Keras metrics (whether custom ones or the built-in ones). You've already learned about the metrics API: simply call `update_state(y_true, y_pred)` for each batch of targets and predictions, and then use `result()` to query the current metric value:

```
from keras import ops

metric = keras.metrics.SparseCategoricalAccuracy()
targets = ops.array([0, 1, 2])
predictions = ops.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
metric.update_state(targets, predictions)
current_result = metric.result()
print(f"result: {current_result:.2f}")
```

You may also need to track the average of a scalar value, such as the model's loss. You can do this via the `keras.metrics.Mean` metric:

```
values = ops.array([0, 1, 2, 3, 4])
mean_tracker = keras.metrics.Mean()
for value in values:
    mean_tracker.update_state(value)
print(f"Mean of values: {mean_tracker.result():.2f}")
```

Remember to use `metric.reset_state()` when you want to reset the current results (at the start of a training epoch or at the start of evaluation).

Now, if you're using JAX, state-modifying methods like `update_state()` or `reset()` can't be used inside a stateless function. Instead, you can use the stateless metrics API, which is similar to the `model.stateless_call()` and `optimizer.stateless_apply()` methods you've already learned about. Here's how it works:

```
metric = keras.metrics.SparseCategoricalAccuracy()
targets = ops.array([0, 1, 2])
predictions = ops.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

metric_variables = metric.variables
metric_variables = metric.stateless_update_state(
    metric_variables, targets, predictions
)
current_result = metric.stateless_result(metric_variables)
print(f"result: {current_result:.2f}")

metric_variables = metric.stateless_reset_state()
```

Gets the metric's state variables

Gets updated values for the metric's state

Computes the metric value corresponding to the current state

Gets blank variable values for the metric

7.4.4 Using fit() with a custom training loop

In the previous sections, we were writing our own training logic entirely from scratch. Doing so provides you with the most flexibility, but you end up writing a lot of code, while simultaneously missing out on many convenient features of `fit()`, such as callbacks, performance optimizations, or built-in support for distributed training.

What if you need a custom training algorithm, but you still want to use the power of the built-in Keras training loop? There's actually a middle ground between `fit()` and a training loop written from scratch: you can provide a custom training step function and let the framework do the rest.

You can do this by overriding the `train_step()` method of the `Model` class. This is the function that is called by `fit()` for every batch of data. You will then be able to call `fit()` as usual—and it will be running your own learning algorithm under the hood.

Here's how it works:

- Create a new class that subclasses `keras.Model`.

- Override the `train_step()` method. Its contents are nearly identical to what we used in the previous section.
- Return a dictionary mapping metric names (including the loss) to their current value.

Note the following:

- This pattern does not prevent you from building models with the Functional API. You can do this whether you're building Sequential models, Functional API models, or subclassed models.
- You don't need to use a `@tf.function` or `@jax.jit` decorator when you override `train_step()`—the framework does it for you.

CUSTOMIZING FIT() WITH TENSORFLOW

Let's start by coding a custom TensorFlow train step.

Listing 7.21 Customizing `fit()`: TensorFlow version

```
import keras
from keras import layers

loss_fn = keras.losses.SparseCategoricalCrossentropy()
loss_tracker = keras.metrics.Mean(name="loss")

class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        with tf.GradientTape() as tape:
            predictions = self(inputs, training=True)
            loss = loss_fn(targets, predictions)
        gradients = tape.gradient(loss, self.trainable_weights)
        self.optimizer.apply(gradients, self.trainable_weights)

        loss_tracker.update_state(loss)
        return {"loss": loss_tracker.result()}

    @property
    def metrics(self):
        return [loss_tracker]
```

This metric object will be used to track the average of per-batch losses during training and evaluation.

Overrides the train_step() method

We use self(inputs, training=True) instead of model(inputs, training=True) since our model is the class itself.

Returns the average loss so far by querying the loss tracker metric

Updates the loss tracker metric that tracks the average of the loss

Listing the loss tracker metric in the model.metrics property enables the model to automatically call reset_state() on it at the start of each epoch and at the start of a call to evaluate()—so you don't have to do it by hand. Any metric you would like to reset across epochs should be listed here.

We can now instantiate our custom model, compile it (we only pass the optimizer, since the loss is already defined outside of the model), and train it using `fit()` as usual.

Let's put the model definition in its own reusable function:

```
def get_custom_model():
    inputs = keras.Input(shape=(28 * 28,))
    features = layers.Dense(512, activation="relu")(inputs)
    features = layers.Dropout(0.5)(features)
    outputs = layers.Dense(10, activation="softmax")(features)
    model = CustomModel(inputs, outputs)
    model.compile(optimizer=keras.optimizers.Adam())
    return model
```

Let's give it a whirl:

```
model = get_custom_model()
model.fit(train_images, train_labels, epochs=3)
```

CUSTOMIZING FIT() WITH PYTORCH

Next, the PyTorch version:

```
import keras
from keras import layers

loss_fn = keras.losses.SparseCategoricalCrossentropy()
loss_tracker = keras.metrics.Mean(name="loss")

class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        predictions = self(inputs, training=True)
        loss = loss_fn(targets, predictions)

        loss.backward()
        trainable_weights = [v for v in self.trainable_weights]
        gradients = [v.value.grad for v in trainable_weights]

        with torch.no_grad():
            self.optimizer.apply(gradients, trainable_weights)

        loss_tracker.update_state(loss)
        return {"loss": loss_tracker.result()}

    @property
    def metrics(self):
        return [loss_tracker]
```

Runs the forward pass

Retrieves the gradients

Updates weights

Updates loss tracker metric

Returns the average loss so far by querying the loss tracker metric

Let's try it:

```
model = get_custom_model()
model.fit(train_images, train_labels, epochs=3)
```

CUSTOMIZING FIT() WITH JAX

Finally, let's write the JAX version. First we need to define a `compute_loss_and_updates()` method, similar to the `compute_loss_and_updates()` function we used in our custom training step example:

```
import keras
from keras import layers

loss_fn = keras.losses.SparseCategoricalCrossentropy()

class CustomModel(keras.Model):
    def compute_loss_and_updates(
        self,
        trainable_variables,
        non_trainable_variables,
        inputs,
        targets,
        training=False,
    ):
        predictions, non_trainable_variables = self.stateless_call(
            trainable_variables,
            non_trainable_variables,
            inputs,
            training=training,
        )
        loss = loss_fn(targets, predictions)
        return loss, non_trainable_variables
```

Returns both the loss and the updated non-trainable variables

Note we aren't computing a moving average of the loss like we did for the other two backends. Instead we just return the per-batch loss value, which is less useful. We do this to simplify metric state management in the example: the code would get very verbose if we included it (you will learn about metric management in the next section):

```
    def train_step(self, state, data):
        (
            trainable_variables,
            non_trainable_variables,
            optimizer_variables,
            metrics_variables,
        ) = state
        inputs, targets = data

        grad_fn = jax.value_and_grad(
            self.compute_loss_and_updates, has_aux=True
        )

        (loss, non_trainable_variables), grads = grad_fn(
            trainable_variables,
            non_trainable_variables,
            inputs,
```

Unpacks the state. metrics_variables are part of it, although we won't use them here.

Gets the gradient function

Computes gradients and updates to non-trainable variables

```
            targets,
            training=True,
        )

        (
            trainable_variables,
            optimizer_variables,
        ) = self.optimizer.stateless_apply(
            optimizer_variables, grads, trainable_variables
        )

        logs = {"loss": loss}
        state = (
            trainable_variables,
            non_trainable_variables,
            optimizer_variables,
            metrics_variables,
        )
        return logs, state
```

Computes gradients and updates to non-trainable variables

Updates trainable variables and optimizer variables

We aren't computing a moving average of the loss, instead returning the per-batch value.

Returns metric logs and updated state variables

Let's try it out:

```
model = get_custom_model()
model.fit(train_images, train_labels, epochs=3)
```

7.4.5 Handling metrics in a custom train_step()

Finally, what about the `loss` and `metrics` that you can pass to `compile()`? After you've called `compile()`, you get access to

- `self.compute_loss`—This combines the loss function you passed to `compile()` together with regularization losses that may be added by certain layers.
- `self.metrics`—The list of metrics you passed to `compile()`. Note that it also includes a metric that tracks the loss.

TRAIN_STEP() METRICS HANDLING WITH TENSORFLOW

Here's what it looks like with TensorFlow:

```
import keras
from keras import layers

class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        with tf.GradientTape() as tape:
            predictions = self(inputs, training=True)
            loss = self.compute_loss(y=targets, y_pred=predictions)
```

Computes the loss via self.compute_loss

```
        gradients = tape.gradient(loss, self.trainable_weights)
        self.optimizer.apply(gradients, self.trainable_weights)

        for metric in self.metrics:
            if metric.name == "loss":
                metric.update_state(loss)
            else:
                metric.update_state(targets, predictions)

        return {m.name: m.result() for m in self.metrics}
```

Updates the model's metrics, including the one that tracks the loss

Returns a dict mapping metric names to their current value

Let's try it:

```
def get_custom_model():
    inputs = keras.Input(shape=(28 * 28,))
    features = layers.Dense(512, activation="relu")(inputs)
    features = layers.Dropout(0.5)(features)
    outputs = layers.Dense(10, activation="softmax")(features)
    model = CustomModel(inputs, outputs)
    model.compile(
        optimizer=keras.optimizers.Adam(),
        loss=keras.losses.SparseCategoricalCrossentropy(),
        metrics=[keras.metrics.SparseCategoricalAccuracy()],
    )
    return model

model = get_custom_model()
model.fit(train_images, train_labels, epochs=3)
```

TRAIN_STEP() METRICS HANDLING WITH PYTORCH

And here's what it looks like with PyTorch—it's exactly the same code change!

```
import keras
from keras import layers

class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        predictions = self(inputs, training=True)
        loss = self.compute_loss(y=targets, y_pred=predictions)

        loss.backward()
        trainable_weights = [v for v in self.trainable_weights]
        gradients = [v.value.grad for v in trainable_weights]
```

```
        with torch.no_grad():
            self.optimizer.apply(gradients, trainable_weights)

        for metric in self.metrics:
            if metric.name == "loss":
                metric.update_state(loss)
            else:
                metric.update_state(targets, predictions)

        return {m.name: m.result() for m in self.metrics}
```

Let's see how it runs:

```
def get_custom_model():
    inputs = keras.Input(shape=(28 * 28,))
    features = layers.Dense(512, activation="relu")(inputs)
    features = layers.Dropout(0.5)(features)
    outputs = layers.Dense(10, activation="softmax")(features)
    model = CustomModel(inputs, outputs)
    model.compile(
        optimizer=keras.optimizers.Adam(),
        loss=keras.losses.SparseCategoricalCrossentropy(),
        metrics=[keras.metrics.SparseCategoricalAccuracy()],
    )
    return model

model = get_custom_model()
model.fit(train_images, train_labels, epochs=3)
```

TRAIN_STEP() METRICS HANDLING WITH JAX

Finally, here's what it looks like with JAX. To start with, you can use `compute_loss()` in your `compute_loss_and_updates()` method to hit the loss passed to `compile()`:

```
import keras
from keras import layers

class CustomModel(keras.Model):
    def compute_loss_and_updates(
        self,
        trainable_variables,
        non_trainable_variables,
        inputs,
        targets,
        training=False,
    ):
        predictions, non_trainable_variables = self.stateless_call(
            trainable_variables,
            non_trainable_variables,
```

```
            inputs,
            training=training,
        )
        loss = self.compute_loss(y=targets, y_pred=predictions)
        return loss, (predictions, non_trainable_variables)
```

Next up: metric management. As usual, it's a tad more complicated due to JAX's statelessness requirements:

```
    def train_step(self, state, data):
        (
            trainable_variables,
            non_trainable_variables,
            optimizer_variables,
            metrics_variables,
        ) = state
        inputs, targets = data

        grad_fn = jax.value_and_grad(
            self.compute_loss_and_updates, has_aux=True
        )

        (loss, (predictions, non_trainable_variables)), grads = grad_fn(
            trainable_variables,
            non_trainable_variables,
            inputs,
            targets,
            training=True,
        )
        (
            trainable_variables,
            optimizer_variables,
        ) = self.optimizer.stateless_apply(
            optimizer_variables, grads, trainable_variables
        )

        new_metrics_vars = []
        logs = {}
        for metric in self.metrics:
            num_prev = len(new_metrics_vars)
            num_current = len(metric.variables)
            current_vars = metrics_variables[num_prev : num_prev + num_current]
            if metric.name == "loss":
                current_vars = metric.stateless_update_state(current_vars, loss)
            else:
                current_vars = metric.stateless_update_state(
                    current_vars, targets, predictions
                )
            logs[metric.name] = metric.stateless_result(current_vars)
```

Metric variables are part of the state.

Iterates over metrics

Grabs the variables of the current metrics

Stores the results in the logs dict

Updates the metric's state

```
            new_metrics_vars += current_vars

        state = (
            trainable_variables,
            non_trainable_variables,
            optimizer_variables,
            new_metrics_vars,
        )
        return logs, state
```

Returns the new metrics variables as part of the state

That was a lot of information, but by now you know enough to use Keras to do almost anything!

Summary

- Keras offers a spectrum of different workflows, based on the principle of *progressive disclosure of complexity*. They all smoothly interoperate.
- You can build models via the `Sequential` class, via the Functional API, or by subclassing the `Model` class. Most of the time, you'll be using the Functional API.
- The simplest way to train and evaluate a model is via the default `fit()` and `evaluate()` methods.
- Keras callbacks provide a simple way to monitor models during your call to `fit()` and automatically take action based on the state of the model.
- You can also fully control what `fit()` does by overriding the `train_step()` method, using APIs from your backend of choice—JAX, TensorFlow, or PyTorch.
- Beyond `fit()`, you can also write your own training loops entirely from scratch, in a backend-native way. This is useful for researchers implementing brand-new training algorithms.

8 Image classification

This chapter covers

- Understanding convolutional neural networks (ConvNets)
- Using data augmentation to mitigate overfitting
- Using a pretrained ConvNet for feature extraction
- Fine-tuning a pretrained ConvNet

Computer vision was the first big success story of deep learning. It led to the initial rise of deep learning between 2011 and 2015. A type of deep learning called *convolutional neural networks* started getting remarkably good results on image classification competitions around that time, first with Dan Ciresan winning two niche competitions (the ICDAR 2011 Chinese character recognition competition and the IJCNN 2011 German traffic signs recognition competition) and then, more notably, in fall 2012, with Hinton's group winning the high-profile ImageNet large-scale visual recognition challenge. Many more promising results quickly started bubbling up in other computer vision tasks.

Interestingly, these early successes weren't quite enough to make deep learning mainstream at the time—it took a few years. The computer vision research

community had spent many years investing in methods other than neural networks, and it wasn't quite ready to give up on them just because there was a new kid on the block. In 2013 and 2014, deep learning still faced intense skepticism from many senior computer vision researchers. It was only in 2016 that it finally became dominant. One author remembers exhorting an ex-professor, in February 2014, to pivot to deep learning. "It's the next big thing!" he would say. "Well, maybe it's just a fad," the professor would reply. By 2016, his entire lab was doing deep learning. There's no stopping an idea whose time has come.

Today, you're constantly interacting with deep learning–based vision models—via Google Photos, Google image search, the camera on your phone, YouTube, OCR software, and many more. These models are also at the heart of cutting-edge research in autonomous driving, robotics, AI-assisted medical diagnosis, autonomous retail checkout systems, and even autonomous farming.

This chapter introduces convolutional neural networks, also known as *ConvNets* or *CNNs*, the type of deep-learning model that is used by most computer vision applications. You'll learn to apply ConvNets to image classification problems—in particular, those involving small training datasets, which are the most common use case if you aren't a large tech company.

8.1 Introduction to ConvNets

We're about to dive into the theory of what ConvNets are and why they have been so successful at computer vision tasks. But first, let's take a practical look at a simple ConvNet example. It uses a ConvNet to classify MNIST digits, a task we performed in chapter 2 using a densely connected network (our test accuracy then was 97.8%). Even though the ConvNet will be basic, its accuracy will blow out of the water that of the densely connected model from chapter 2.

The following lines of code show you what a basic ConvNet looks like. It's a stack of `Conv2D` and `MaxPooling2D` layers. You'll see in a minute exactly what they do. We'll build the model using the Functional API, which we introduced in the previous chapter.

Listing 8.1 Instantiating a small ConvNet

```
import keras
from keras import layers

inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Importantly, a ConvNet takes as input tensors of shape `(image_height, image_width, image_channels)` (not including the batch dimension). In this case, we'll configure the ConvNet to process inputs of size `(28, 28, 1)`, which is the format of MNIST images.

Let's display the architecture of our ConvNet.

Listing 8.2 Displaying the model's summary

```
>>> model.summary()
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 26, 26, 64)	640
max_pooling2d (MaxPooling2D)	(None, 13, 13, 64)	0
conv2d_1 (Conv2D)	(None, 11, 11, 128)	73,856
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 128)	0
conv2d_2 (Conv2D)	(None, 3, 3, 256)	295,168
global_average_pooling2d (GlobalAveragePooling2D)	(None, 256)	0
dense (Dense)	(None, 10)	2,570

```
Total params: 372,234 (1.42 MB)
Trainable params: 372,234 (1.42 MB)
Non-trainable params: 0 (0.00 B)
```

You can see that the output of every `Conv2D` and `MaxPooling2D` layer is a 3D tensor of shape `(height, width, channels)`. The width and height dimensions tend to shrink as you go deeper in the model. The number of channels is controlled by the first argument passed to the `Conv2D` layers (64, 128, or 256).

Image data formats in deep learning frameworks

Some deep learning libraries flip the location of channels in image tensors to the first rank (notably much of the PyTorch ecosystem). Rather than passing images with shape `(height, width, channels)` you would pass `(channels, height, width)`.

This is purely a matter of convention, and in Keras is configurable. You can call `keras.config.set_image_data_format("channels_first")` to change Keras' default, or pass a `data_format` argument to any conv or pooling layer. In general, you can leave the default as is unless you have a specific need for `"channels_first"`.

After the last `Conv2D` layer, we end up with an output of shape `(3, 3, 256)`—a 3 × 3 feature map of 256 channels. The next step is to feed this output into a densely connected classifier like those you're already familiar with: a stack of `Dense` layers. These classifiers process vectors, which are 1D, whereas the current output is a rank-3 tensor. To bridge the gap, we flatten the 3D outputs to 1D with a `GlobalAveragePooling2D` layer before adding the `Dense` layers. This layer will take the average of each 3 × 3 feature map in the tensor of shape `(3, 3, 256)`, resulting in an output vector of shape `(256,)`. Finally, we'll do 10-way classification, so our last layer has 10 outputs and a softmax activation.

Now, let's train the ConvNet on the MNIST digits. We'll reuse a lot of the code from the MNIST example in chapter 2. Because we're doing 10-way classification with a softmax output, we'll use the categorical crossentropy loss, and because our labels are integers, we'll use the sparse version, `sparse_categorical_crossentropy`.

Listing 8.3 Training the ConvNet on MNIST images

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype("float32") / 255
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

Let's evaluate the model on the test data.

Listing 8.4 Evaluating the ConvNet

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print(f"Test accuracy: {test_acc:.3f}")
Test accuracy: 0.991
```

Whereas the densely connected model from chapter 2 had a test accuracy of 97.8%, the basic ConvNet has a test accuracy of 99.1%: we decreased the error rate by about 60% (relative). Not bad!

But why does this simple ConvNet work so well, compared to a densely connected model? To answer this, let's dive into what the `Conv2D` and `MaxPooling2D` layers do.

8.1.1 The convolution operation

The fundamental difference between a densely connected layer and a convolution layer is this: `Dense` layers learn global patterns in their input feature space (for example,

for a MNIST digit, patterns involving all pixels), whereas convolution layers learn local patterns (see figure 8.1): in the case of images, patterns found in small 2D windows of the inputs. In the previous example, these windows were all 3 × 3. This key characteristic gives ConvNets two interesting properties:

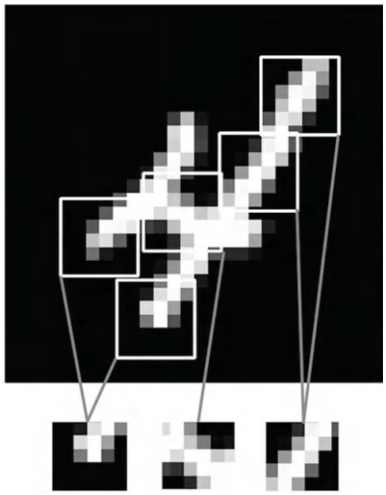


Figure 8.1 Images can be broken into local patterns such as edges, textures, and so on.

- *The patterns they learn are translation invariant.* After learning a certain pattern in the lower-right corner of a picture, a ConvNet can recognize it anywhere: for example, in the upper-left corner. A densely connected model would have to learn the pattern anew if it appeared at a new location. This makes ConvNets data efficient when processing images—because *the visual world is fundamentally translation invariant.* They need fewer training samples to learn representations that have generalization power.
- *They can learn spatial hierarchies of patterns (see figure 8.2).* A first convolution layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of the features of the first layers, and so on. This allows ConvNets to efficiently learn increasingly complex and abstract visual concepts—because *the visual world is fundamentally spatially hierarchical.*

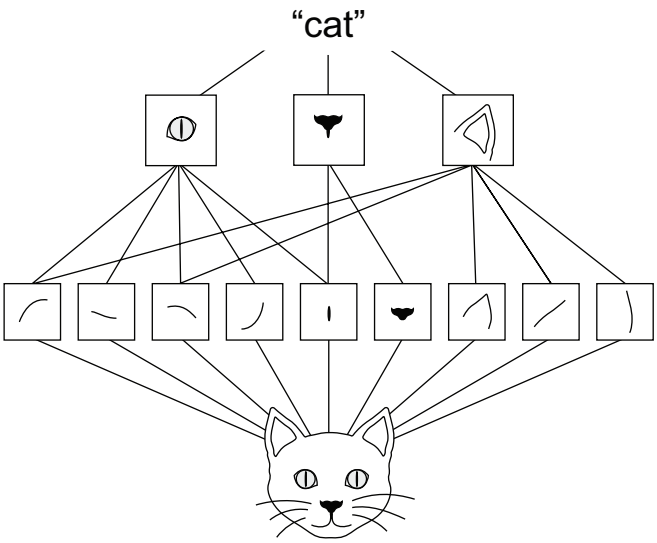


Figure 8.2 The visual world forms a spatial hierarchy of visual modules: elementary lines or textures combine into simple objects such as eyes or ears, which combine into high-level concepts such as "cat."

Convolutions operate over rank-3 tensors, called *feature maps*, with two spatial axes (*height* and *width*) as well as a *depth* axis (also called the *channels* axis). For an RGB image, the dimension of the depth axis is 3, because the image has three color channels: red, green, and blue. For a black-and-white picture, like the MNIST digits, the depth is 1 (levels of gray). The convolution operation extracts patches from its input feature map and applies the same transformation to all of these patches, producing an *output feature map*. This output feature map is still a rank-3 tensor: it has a width and a height. Its depth can be arbitrary because the output depth is a parameter of the layer, and the different channels in that depth axis no longer stand for specific colors as in RGB input; rather, they stand for *filters*. Filters encode specific aspects of the input data: at a high level, a single filter could encode the concept "presence of a face in the input," for instance.

In the MNIST example, the first convolution layer takes a feature map of size `(28, 28, 1)` and outputs a feature map of size `(26, 26, 64)`: it computes 64 filters over its input. Each of these 64 output channels contains a 26 × 26 grid of values, which is a *response map* of the filter over the input, indicating the response of that filter pattern at different locations in the input (see figure 8.3). That is what the term *feature map* means: every dimension in the depth axis is a feature (or filter), and the rank-2 tensor `output[:, :, n]` is the 2D spatial *map* of the response of this filter over the input.

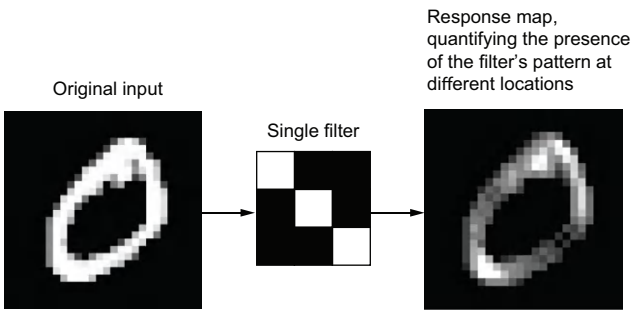


Figure 8.3 The concept of a response map: a 2D map of the presence of a pattern at different locations in an input

Convolutions are defined by two key parameters:

- *Size of the patches extracted from the inputs*—These are typically 3 × 3 or 5 × 5. In the example, they were 3 × 3, which is a common choice.
- *Depth of the output feature map*—The number of filters computed by the convolution. The example started with a depth of 32 and ended with a depth of 64.

In Keras `Conv2D` layers, these parameters are the first arguments passed to the layer: `Conv2D(output_depth, (window_height, window_width))`.

A convolution works by *sliding* these windows of size 3 × 3 or 5 × 5 over the 3D input feature map, stopping at every possible location, and extracting the 3D patch of surrounding features of shape `((window_height, window_width, input_depth))`. Each such 3D patch is then transformed into a 1D vector of shape `(output_depth,)`, which is

done via a tensor product with a learned weight matrix, called the *convolution kernel*—the same kernel is reused across every patch. All of these vectors (one per patch) are then spatially reassembled into a 3D output map of shape `(height, width, output_depth)`. Every spatial location in the output feature map corresponds to the same location in the input feature map (for example, the lower-right corner of the output contains information about the lower-right corner of the input). For instance, with 3 × 3 windows, the vector `output[i, j, :]` comes from the 3D patch `input[i-1:i+1, j-1:j+1, :]`. The full process is detailed in figure 8.4.

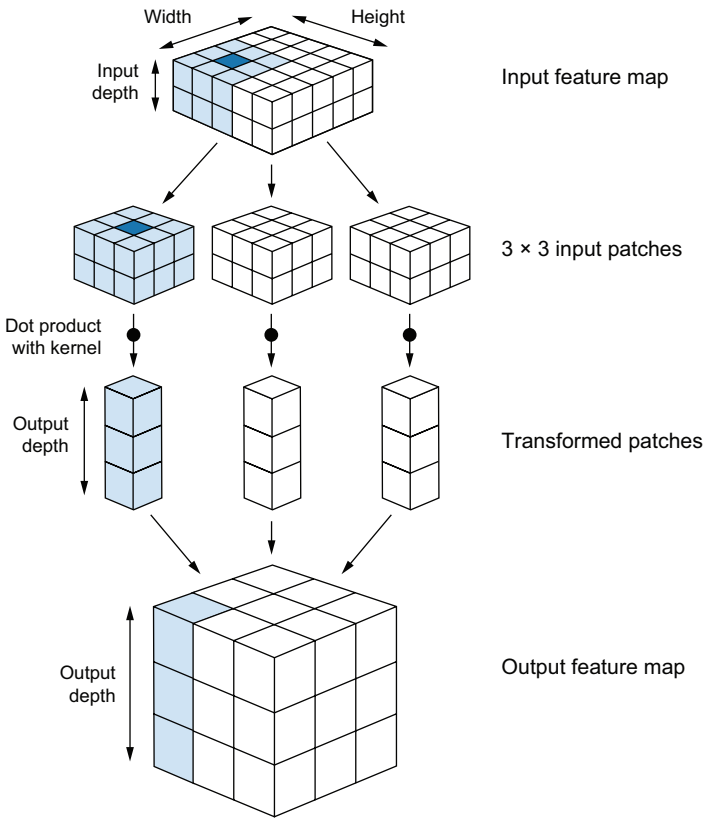


Figure 8.4 How convolution works

Note that the output width and height may differ from the input width and height. They may differ for two reasons:

- Border effects, which can be countered by padding the input feature map
- The use of *strides*, which we'll define in a second

Let's take a deeper look at these notions.

UNDERSTANDING BORDER EFFECTS AND PADDING

Consider a 5 × 5 feature map (25 tiles total). There are only 9 tiles around which you can center a 3 × 3 window, forming a 3 × 3 grid (see figure 8.5). Hence, the output feature map will be 3 × 3. It shrinks a little: by exactly two tiles alongside each dimension, in this case. You can see this border effect in action in the earlier example: you start with 28 × 28 inputs, which become 26 × 26 after the first convolution layer.

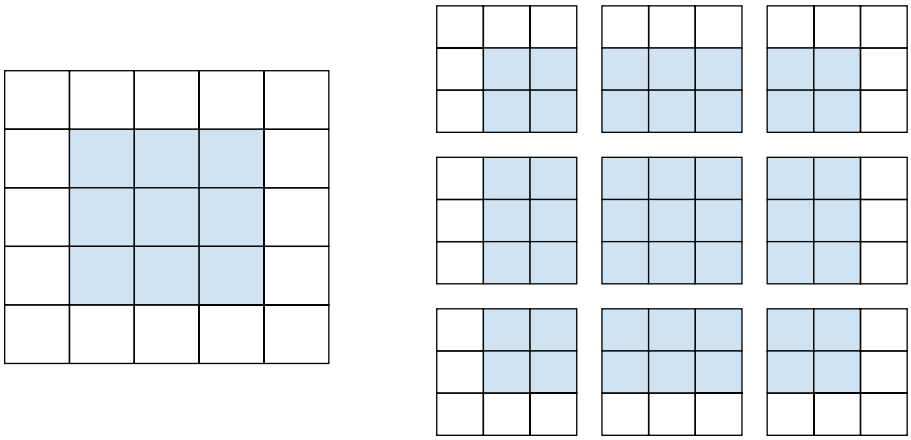


Figure 8.5 Valid locations of 3 × 3 patches in a 5 × 5 input feature map

If you want to get an output feature map with the same spatial dimensions as the input, you can use *padding*. Padding consists of adding an appropriate number of rows and columns on each side of the input feature map so as to make it possible to fit centered convolution windows around every input tile. For a 3 × 3 window, you add one column on the right, one column on the left, one row at the top, and one row at the bottom. For a 5 × 5 window, you add two rows (see figure 8.6).

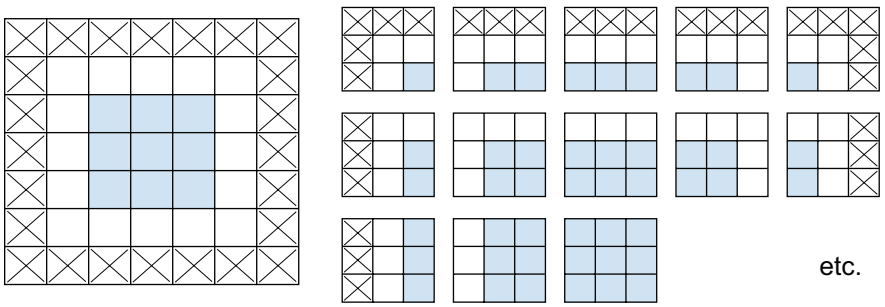


Figure 8.6 Padding a 5 × 5 input to be able to extract 25 3 × 3 patches

In `Conv2D` layers, padding is configurable via the `padding` argument, which takes two values: `"valid"`, which means no padding (only valid window locations will be used); and `"same"`, which means "pad in such a way as to have an output with the same width and height as the input." The `padding` argument defaults to `"valid"`.

UNDERSTANDING CONVOLUTION STRIDES

The other factor that can influence output size is the notion of *strides*. The description of convolution so far has assumed that the center tiles of the convolution windows are all contiguous. But the distance between two successive windows is a parameter of the convolution, called its *stride*, which defaults to 1. It's possible to have *strided convolutions*: convolutions with a stride higher than 1. In figure 8.7, you can see the patches extracted by a 3 × 3 convolution with stride 2 over a 5 × 5 input (without padding)

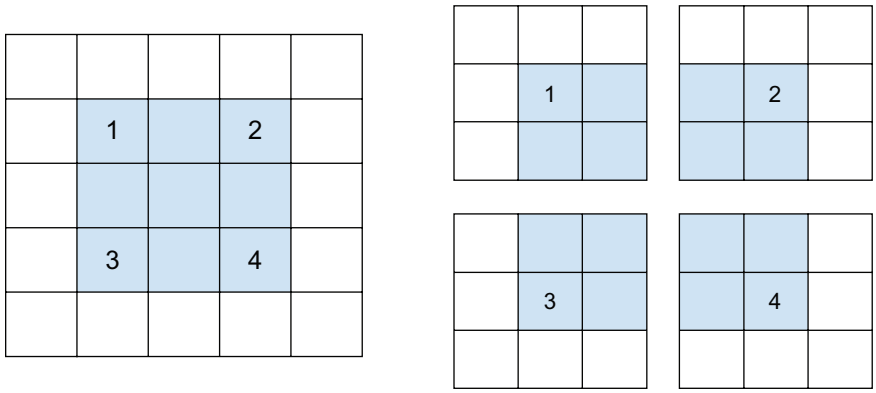


Figure 8.7 3 × 3 convolution patches with 2 × 2 strides

Using stride 2 means the width and height of the feature map are downsampled by a factor of 2 (in addition to any changes induced by border effects). Strided convolutions are rarely used in classification models, but they come in handy for some types of models, as you will find out in the next chapter.

In classification models, instead of strides, we tend to use the *max-pooling* operation to downsample feature maps—which you saw in action in our first ConvNet example. Let's look at it in more depth.

8.1.2 The max-pooling operation

In the ConvNet example, you may have noticed that the size of the feature maps is halved after every `MaxPooling2D` layer. For instance, before the first `MaxPooling2D` layers, the feature map is 26 × 26, but the max-pooling operation halves it to 13 × 13. That's the role of max pooling: to aggressively downsample feature maps, much like strided convolutions.

Max pooling consists of extracting windows from the input feature maps and outputting the max value of each channel. It's conceptually similar to convolution, except that instead of transforming local patches via a learned linear transformation (the convolution kernel), they're transformed via a hardcoded `max` tensor operation. A big difference from convolution is that max pooling is usually done with 2×2 windows and stride 2 to downsample the feature maps by a factor of 2. On the other hand, convolution is typically done with 3×3 windows and no stride (stride 1).

Why downsample feature maps this way? Why not remove the max-pooling layers and keep fairly large feature maps all the way up? Let's look at this option. Our model would then look like this.

Listing 8.5 An incorrectly structured ConvNet missing its max-pooling layers

```
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(inputs)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model_no_max_pool = keras.Model(inputs=inputs, outputs=outputs)
```

Here's a summary of the model:

```
>>> model_no_max_pool.summary()
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 28, 28, 1)	0
conv2d_3 (Conv2D)	(None, 26, 26, 64)	640
conv2d_4 (Conv2D)	(None, 24, 24, 128)	73,856
conv2d_5 (Conv2D)	(None, 22, 22, 256)	295,168
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2,570

```
Total params: 372,234 (1.42 MB)
Trainable params: 372,234 (1.42 MB)
Non-trainable params: 0 (0.00 B)
```

What's wrong with this setup? Two things:

- It isn't conducive to learning a spatial hierarchy of features. The 3×3 windows in the third layer will only contain information coming from 7×7 windows in

the initial input. The high-level patterns learned by the ConvNet will still be very small with regard to the initial input, which may not be enough to learn to classify digits (try recognizing a digit by only looking at it through windows that are 7×7 pixels!). We need the features from the last convolution layer to contain information about the totality of the input.

- The final feature map has dimensions 22×22. That's huge—when you take the average of each 22×22 feature map, you are going to be destroying a lot of information compared to when your feature maps were only 3×3.

In short, the reason to use downsampling is to reduce the size of the feature maps, making the information they contain increasingly less spatially distributed and increasingly contained in the channels, while also inducing spatial-filter hierarchies by making successive convolution layers "look" at increasingly large windows (in terms of the fraction of the original input image they cover).

Note that max pooling isn't the only way you can achieve such downsampling. As you already know, you can also use strides in the prior convolution layer. And you can use average pooling instead of max pooling, where each local input patch is transformed by taking the average value of each channel over the patch, rather than the max. But max pooling tends to work better than these alternative solutions. In a nutshell, the reason is that features tend to encode the spatial presence of some pattern or concept over the different tiles of the feature map (hence the term *feature map*), and it's more informative to look at the *maximal presence* of different features than at their *average presence*. So the most reasonable subsampling strategy is to first produce dense maps of features (via unstrided convolutions) and then look at the maximal activation of the features over small patches, rather than looking at sparser windows of the inputs (via strided convolutions) or averaging input patches, which could cause you to miss or dilute feature-presence information.

At this point, you should understand the basics of ConvNets—feature maps, convolution, and max pooling—and you know how to build a small ConvNet to solve a toy problem such as MNIST digits classification. Now let's move on to more useful, practical applications.

8.2 Training a ConvNet from scratch on a small dataset

Having to train an image-classification model using very little data is a common situation, which you'll likely encounter in practice if you ever do computer vision in a professional context. A "few" samples can mean anywhere from a few hundred to a few tens of thousands of images. As a practical example, we'll focus on classifying images as dogs or cats. We'll work with a dataset containing 5,000 pictures of cats and dogs (2,500 cats, 2,500 dogs), taken from the original Kaggle dataset. We'll use 2,000 pictures for training, 1,000 for validation, and 2,000 for testing.

In this section, we'll review one basic strategy to tackle this problem: training a new model from scratch using what little data we have. We'll start by naively training a small ConvNet on the 2,000 training samples, without any regularization, to set a baseline for what can be achieved. This will get us to a classification accuracy of about 80%. At that

point, the main issue will be overfitting. Then we'll introduce *data augmentation*, a powerful technique for mitigating overfitting in computer vision. By using data augmentation, we'll improve the model to reach a test accuracy of about 84%.

In the next section, we'll review two more essential techniques for applying deep learning to small datasets: *feature extraction with a pretrained model* and *fine-tuning a pretrained model* (which will get us to a final accuracy of 98.5%). Together, these three strategies—training a small model from scratch, doing feature extraction using a pretrained model, and fine-tuning a pretrained model—will constitute your future toolbox for tackling the problem of performing image classification with small datasets.

8.2.1 The relevance of deep learning for small-data problems

What qualifies as "enough samples" to train a model is relative—relative to the size and depth of the model you're trying to train, for starters. It isn't possible to train a ConvNet to solve a complex problem with just a few tens of samples, but a few hundred can potentially suffice if the model is small and well regularized and the task is simple. Because ConvNets learn local, translation-invariant features, they're highly data efficient on perceptual problems. Training a ConvNet from scratch on a very small image dataset will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering. You'll see this in action in this section.

What's more, deep learning models are by nature highly repurposable: you can take, say, an image-classification or speech-to-text model trained on a large-scale dataset and reuse it on a significantly different problem with only minor changes. Specifically, in the case of computer vision, many pretrained classification models are publicly available for download and can be used to bootstrap powerful vision models out of very little data. This is one of the greatest strengths of deep learning: feature reuse. You'll explore this in the next section.

Let's start by getting our hands on the data.

8.2.2 Downloading the data

The Dogs vs. Cats dataset that we will use isn't packaged with Keras. It was made available by Kaggle as part of a computer-vision competition in late 2013, back when ConvNets weren't mainstream. You can download the original dataset from `www.kaggle.com/c/dogs-vs-cats/data` (you'll need to create a Kaggle account if you don't already have one—don't worry, the process is painless). You can also use the Kaggle API to download the dataset in Colab.

Downloading a Kaggle dataset in Google Colaboratory

Kaggle makes available an easy-to-use API to programmatically download Kaggle hosted datasets. You can use it to download the Dogs vs. Cats dataset to a Colab notebook, for instance. This API is available via the `kagglehub` package, which is preinstalled on Colab.

Before we can download the dataset, we will need to do two things:

1. Go to https://www.kaggle.com/c/dogs-vs-cats/rules and click to Join the Competition.
2. Go to https://www.kaggle.com/settings and generate a Kaggle API key.

With that we are ready to download the data in our notebook. First, log in with your Kaggle API key:

```
import kagglehub

kagglehub.login()
```

Then, download the competition data:

```
download_path = kagglehub.competition_download("dogs-vs-cats")
```

This downloads two new files, `train.zip` (the training data) and `test1.zip` (the test data). We'll only use the training data here. Let's unzip it:

```
import zipfile

with zipfile.ZipFile(download_path + "/train.zip", "r") as zip_ref:
    zip_ref.extractall(".")
```

All done!

The pictures in our dataset are medium-resolution color JPEGs. Figure 8.8 shows some examples.

Figure 8.8 Samples from the Dogs vs. Cats dataset. Sizes weren't modified: the samples come in different sizes, colors, backgrounds, etc.

Unsurprisingly, the original dogs-versus-cats Kaggle competition, all the way back in 2013, was won by entrants who used ConvNets. The best entries achieved up to 95% accuracy. In this example, we will get fairly close to this accuracy (in the next section), even though we will train our models on less than 10% of the data that was available to the competitors.

This dataset contains 25,000 images of dogs and cats (12,500 from each class) and is 543 MB (compressed). After downloading and uncompressing the data, we'll create a new dataset containing three subsets: a training set with 1,000 samples of each class, a validation set with 500 samples of each class, and a test set with 1,000 samples of each class. Why do this? Because many of the image datasets you'll encounter in your career only contain a few thousand samples, not tens of thousands. Having more data available would make the problem easier—so it's good practice to learn with a small dataset.

The subsampled dataset we will work with will have the following directory structure:

```
dogs_vs_cats_small/
...train/
......cat/          <- Contains 1,000 cat images
......dog/          <- Contains 1,000 dog images
...validation/
......cat/          <- Contains 500 cat images
......dog/          <- Contains 500 dog images
...test/
......cat/          <- Contains 1,000 cat images
......dog/          <- Contains 1,000 dog images
```

Let's make it happen in a couple of calls to `shutil`, a Python library for running shell-like commands.

Listing 8.6 Copying images to training, validation, and test directories

```
import os, shutil, pathlib

original_dir = pathlib.Path("train")
new_base_dir = pathlib.Path("dogs_vs_cats_small")

def make_subset(subset_name, start_index, end_index):
    for category in ("cat", "dog"):
        dir = new_base_dir / subset_name / category
        os.makedirs(dir)
```

Utility function to copy cat (respectively, dog) images from index start_index to index end_index to the subdirectory new_base_dir/{subset_name}/cat (respectively, dog). "subset_name" will be either "train," "validation," or "test."

Path to the directory where the original dataset was uncompressed

Directory where we will store our smaller dataset

```
        fnames = [f"{category}.{i}.jpg" for i in range(start_index, end_index)]
        for fname in fnames:
            shutil.copyfile(src=original_dir / fname, dst=dir / fname)

make_subset("train", start_index=0, end_index=1000)
make_subset("validation", start_index=1000, end_index=1500)
make_subset("test", start_index=1500, end_index=2500)
```

Creates the test subset with the next 1,000 images of each category

Creates the validation subset with the next 500 images of each category

Creates the training subset with the first 1,000 images of each category

We now have 2,000 training images, 1,000 validation images, and 2,000 test images. Each split contains the same number of samples from each class: this is a balanced binary classification problem, which means classification accuracy will be an appropriate measure of success.

8.2.3 Building your model

We will reuse the same general model structure you saw in the first example: the ConvNet will be a stack of alternated `Conv2D` (with `relu` activation) and `MaxPooling2D` layers.

But because we're dealing with bigger images and a more complex problem, we'll make our model larger, accordingly: it will have two more `Conv2D MaxPooling2D` stages. This serves both to augment the capacity of the model and to further reduce the size of the feature maps so they aren't overly large when we reach the pooling layer. Here, because we start from inputs of size 180 × 180 pixels (a somewhat arbitrary choice), we end up with feature maps of size 7 × 7 just before the `GlobalAveragePooling2D` layer.

> **NOTE** The depth of the feature maps progressively increases in the model (from 32 to 512), whereas the size of the feature maps decreases (from 180 × 180 to 7 × 7). This is a pattern you'll see in almost all ConvNets.

Because we're looking at a binary classification problem, we'll end the model with a single unit (a `Dense` layer of size 1) and a `sigmoid` activation. This unit will encode the probability that the model is looking at one class or the other.

One last small difference: we will start the model with a `Rescaling` layer, which will rescale image inputs (whose values are originally in the [0, 255] range) to the [0, 1] range.

Listing 8.7 Instantiating a small ConvNet for dogs vs. cats classification

```
import keras
from keras import layers

inputs = keras.Input(shape=(180, 180, 3))
x = layers.Rescaling(1.0 / 255)(inputs)
```

The model expects RGB images of size 180 x 180.

Rescales inputs to the [0, 1] range by dividing them by 255

```
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=512, kernel_size=3, activation="relu")(x)
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Flattens the 3D activations with shape (height, width, 512) into 1D activations with shape (512,) by averaging them over spatial dimensions

Let's look at how the dimensions of the feature maps change with every successive layer:

```
>>> model.summary()
Model: "functional_2"
```

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 180, 180, 3)	0
rescaling (Rescaling)	(None, 180, 180, 3)	0
conv2d_6 (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 89, 89, 32)	0
conv2d_7 (Conv2D)	(None, 87, 87, 64)	18,496
max_pooling2d_3 (MaxPooling2D)	(None, 43, 43, 64)	0
conv2d_8 (Conv2D)	(None, 41, 41, 128)	73,856
max_pooling2d_4 (MaxPooling2D)	(None, 20, 20, 128)	0
conv2d_9 (Conv2D)	(None, 18, 18, 256)	295,168
max_pooling2d_5 (MaxPooling2D)	(None, 9, 9, 256)	0
conv2d_10 (Conv2D)	(None, 7, 7, 512)	1,180,160
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 512)	0
dense_2 (Dense)	(None, 1)	513

```
Total params: 1,569,089 (5.99 MB)
Trainable params: 1,569,089 (5.99 MB)
Non-trainable params: 0 (0.00 B)
```

For the compilation step, you'll go with the `adam` optimizer, as usual. Because you ended the model with a single sigmoid unit, you'll use binary crossentropy as the loss (as a reminder, check out table 6.1 in chapter 6 for a cheat sheet on what loss function to use in various situations).

Listing 8.8 Configuring the model for training

```
model.compile(
    loss="binary_crossentropy",
    optimizer="adam",
    metrics=["accuracy"],
)
```

8.2.4 Data preprocessing

As you know by now, data should be formatted into appropriately preprocessed floating-point tensors before being fed into the model. Currently, the data sits on a drive as JPEG files, so the steps for getting it into the model are roughly as follows:

1. Read the picture files.
2. Decode the JPEG content to RGB grids of pixels.
3. Convert these into floating-point tensors.
4. Resize them to a shared size (we'll use 180 x 180).
5. Pack them into batches (we'll use batches of 32 images).

This may seem a bit daunting, but fortunately Keras has utilities to take care of these steps automatically. In particular, Keras features the utility function `image_dataset_from_directory`, which lets you quickly set up a data pipeline that can automatically turn image files on disk into batches of preprocessed tensors. This is what you'll use here.

Calling `image_dataset_from_directory(directory)` will first list the subdirectories of `directory` and assume each one contains images from one of your classes. It will then index the image files in each subdirectory. Finally, it will create and return a `tf.data.Dataset` object configured to read these files, shuffle them, decode them to tensors, resize them to a shared size, and pack them into batches.

Listing 8.9 Using `image_dataset_from_directory` to read images from directories

```
from keras.utils import image_dataset_from_directory

batch_size = 64
```

```
image_size = (180, 180)
train_dataset = image_dataset_from_directory(
    new_base_dir / "train", image_size=image_size, batch_size=batch_size
)
validation_dataset = image_dataset_from_directory(
    new_base_dir / "validation", image_size=image_size, batch_size=batch_size
)
test_dataset = image_dataset_from_directory(
    new_base_dir / "test", image_size=image_size, batch_size=batch_size
)
```

UNDERSTANDING TENSORFLOW DATASET OBJECTS

TensorFlow makes available the `tf.data` API to create efficient input pipelines for machine learning models. Its core class is `tf.data.Dataset`.

The `Dataset` class can be used for data loading and preprocessing in any framework—not just TensorFlow. You can use it together with JAX or PyTorch. When you use it with a Keras model, it works the same, independently of the backend you're currently using.

A `Dataset` object is an iterator: you can use it in a `for` loop. It will typically return batches of input data and labels. You can pass a `Dataset` object directly to the `fit()` method of a Keras model.

The `Dataset` class handles many key features that would otherwise be cumbersome to implement yourself, in particular parallelization of the preprocessing logic across multiple CPU cores, as well as asynchronous data prefetching (preprocessing the next batch of data while the previous one is being handled by the model, which keeps execution flowing without interruptions).

The `Dataset` class also exposes a functional-style API for modifying datasets. Here's a quick example: let's create a `Dataset` instance from a NumPy array of random numbers. We'll consider 1,000 samples, where each sample is a vector of size 16.

Listing 8.10 Instantiating a `Dataset` from a NumPy array

```
import numpy as np
import tensorflow as tf

random_numbers = np.random.normal(size=(1000, 16))
dataset = tf.data.Dataset.from_tensor_slices(random_numbers)
```

The from_tensor_slices() class method can be used to create a Dataset from a NumPy array or a tuple or dict of NumPy arrays.

At first, our dataset just yields single samples.

Listing 8.11 Iterating on a dataset

```
>>> for i, element in enumerate(dataset):
>>>     print(element.shape)
>>>     if i >= 2:
>>>         break
(16,)
(16,)
(16,)
```

You can use the `.batch()` method to batch the data.

Listing 8.12 Batching a dataset

```
>>> batched_dataset = dataset.batch(32)
>>> for i, element in enumerate(batched_dataset):
>>>     print(element.shape)
>>>     if i >= 2:
>>>         break
(32, 16)
(32, 16)
(32, 16)
```

More broadly, you have access to a range of useful dataset methods, such as these:

- `.shuffle(buffer_size)` will shuffle elements within a buffer.
- `.prefetch(buffer_size)` will prefetch a buffer of elements in GPU memory to achieve better device utilization.
- `.map(callable)` will apply an arbitrary transformation to each element of the dataset (the function `callable`, expected to take as input a single element yielded by the dataset).

The method `.map(function, num_parallel_calls)` in particular is one that you will use often. Here's an example: let's use it to reshape the elements in our toy dataset from shape `(16,)` to shape `(4, 4)`.

Listing 8.13 Applying a transformation to `Dataset` elements using `map()`

```
>>> reshaped_dataset = dataset.map(
...     lambda x: tf.reshape(x, (4, 4)),
...     num_parallel_calls=8)
>>> for i, element in enumerate(reshaped_dataset):
...     print(element.shape)
...     if i >= 2:
...         break
(4, 4)
(4, 4)
(4, 4)
```

You're about to see more `map()` action over the next chapters.

FITTING THE MODEL

Let's look at the output of one of these `Dataset` objects: it yields batches of 180 × 180 RGB images (shape `(32, 180, 180, 3)`) and integer labels (shape `(32,)`). There are 32 samples in each batch (the batch size).

Listing 8.14 Displaying the shapes yielded by the `Dataset`

```
>>> for data_batch, labels_batch in train_dataset:
>>>     print("data batch shape:", data_batch.shape)
>>>     print("labels batch shape:", labels_batch.shape)
>>>     break
data batch shape: (32, 180, 180, 3)
labels batch shape: (32,)
```

Let's fit the model on our dataset. We use the `validation_data` argument in `fit()` to monitor validation metrics on a separate `Dataset` object.

Note that we also use a `ModelCheckpoint` callback to save the model after each epoch. We configure it with the path where to save the file, as well as the arguments `save_best_only=True` and `monitor="val_loss"`: they tell the callback to only save a new file (overwriting any previous one) when the current value of the `val_loss` metric is lower than at any previous time during training. This guarantees that your saved file will always contain the state of the model corresponding to its best-performing training epoch, in terms of its performance on the validation data. As a result, we won't have to retrain a new model for a lower number of epochs if we start overfitting: we can just reload our saved file.

Listing 8.15 Fitting the model using a `Dataset`

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch.keras",
        save_best_only=True,
        monitor="val_loss",
    )
]
history = model.fit(
    train_dataset,
    epochs=50,
    validation_data=validation_dataset,
    callbacks=callbacks,
)
```

Let's plot the loss and accuracy of the model over the training and validation data during training (see figure 8.9).

Listing 8.16 Displaying curves of loss and accuracy during training

```
import matplotlib.pyplot as plt

accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)

plt.plot(epochs, accuracy, "r--", label="Training accuracy")
plt.plot(epochs, val_accuracy, "b", label="Validation accuracy")
plt.title("Training and validation accuracy")
plt.legend()
plt.figure()

plt.plot(epochs, loss, "r--", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.legend()
plt.show()
```

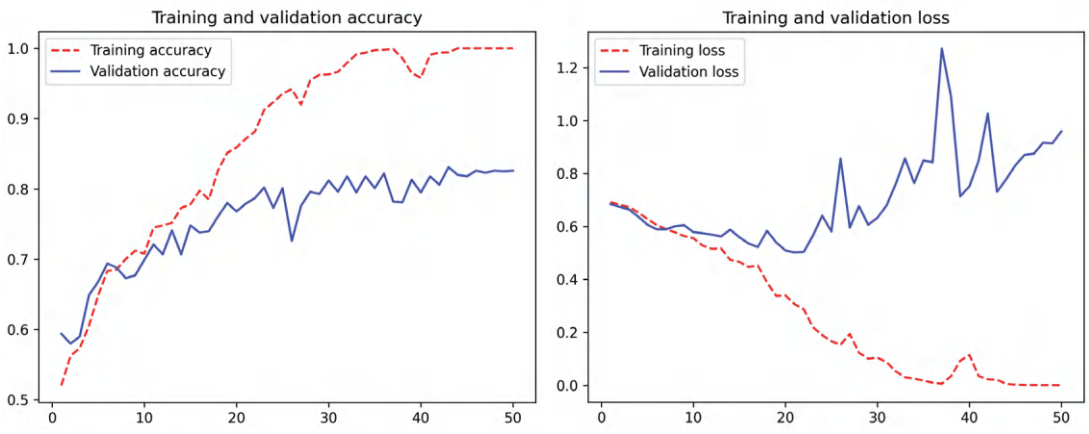


Figure 8.9 Training and validation metrics for a simple ConvNet

These plots are characteristic of overfitting. The training accuracy increases linearly over time, until it reaches nearly 100%, whereas the validation accuracy peaks around 80%. The validation loss reaches its minimum after only 10 epochs and then stalls, whereas the training loss keeps decreasing linearly as training proceeds.

Let's check the test accuracy. We'll reload the model from its saved file to evaluate it as it was before it started overfitting.

Listing 8.17 Evaluating the model on the test set

```
test_model = keras.models.load_model("convnet_from_scratch.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

We get a test accuracy of 78.6% (due to the randomness of neural network initializations, you may get numbers within 1 percentage point of that).

Because you have relatively few training samples (2,000), overfitting will be your number-one concern. You already know about a number of techniques that can help mitigate overfitting, such as dropout and weight decay (L2 regularization). We're now going to work with a new one, specific to computer vision and used almost universally when processing images with deep learning models: *data augmentation*.

8.2.5 Using data augmentation

Overfitting is caused by having too few samples to learn from, rendering you unable to train a model that can generalize to new data. Given infinite data, your model would be exposed to every possible aspect of the data distribution at hand: you would never overfit. Data augmentation takes the approach of generating more training data from existing training samples, by *augmenting* the samples via a number of random transformations that yield believable-looking images. The goal is that at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data and generalize better.

In Keras, this can be done via *data augmentation layers*. Such layers could be added in one of two ways:

- *At the start of the model—Inside* the model. In our case, the layers would come right before the `Rescaling` layer.
- *Inside the data pipeline—Outside* the model. In our case, we'd apply them to our `Dataset` via a `map()` call.

The main difference between these two options is that data augmentation done inside the model would be running on the GPU, just like the rest of the model. Meanwhile, data augmentation done in the data pipeline would be running on the CPU, typically in a parallel way on multiple CPU cores. Sometimes, there can be performance benefits to doing the former, but the latter is usually the better option. So let's go with that!

Listing 8.18 Defining a data augmentation stage

```
data_augmentation_layers = [
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.2),
]
```

Defines the transformations to apply as a list

```
def data_augmentation(images, targets):
    for layer in data_augmentation_layers:
        images = layer(images)
    return images, targets

augmented_train_dataset = train_dataset.map(
    data_augmentation, num_parallel_calls=8
)
augmented_train_dataset = augmented_train_dataset.prefetch(tf.data.AUTOTUNE)
```

Creates a function that applies them sequentially

Maps this function into the dataset

Enables prefetching of batches on GPU memory; important for best performance

These are just a few of the layers available (for more, see the Keras documentation). Let's quickly go over this code:

- `RandomFlip("horizontal")` will apply horizontal flipping to a random 50% of the images that go through it.
- `RandomRotation(0.1)` will rotate the input images by a random value in the range [–10%, +10%] (these are fractions of a full circle—in degrees the range would be [–36 degrees, +36 degrees]).
- `RandomZoom(0.2)` will zoom in or out of the image by a random factor in the range [–20%, +20%].

Let's look at the augmented images (see figure 8.10).

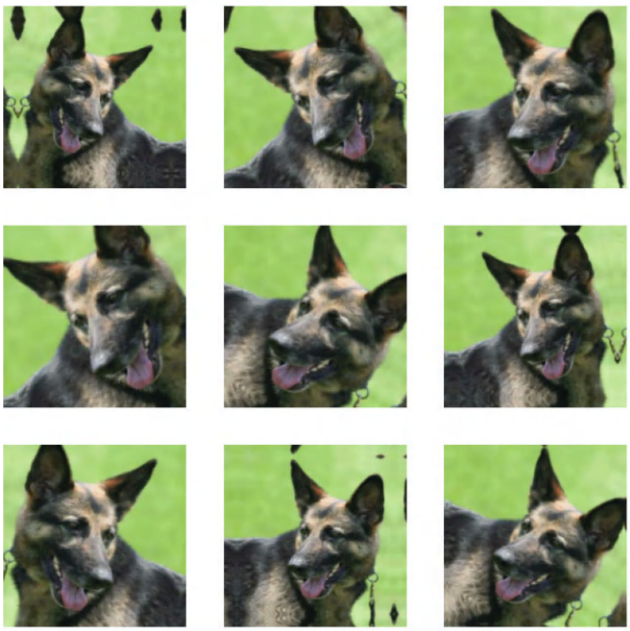


Figure 8.10 Generating variations of a very good boy via random data augmentation

Listing 8.19 Displaying some randomly augmented training images

```
plt.figure(figsize=(10, 10))
for image_batch, _ in train_dataset.take(1):
    image = image_batch[0]
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        augmented_image, _ = data_augmentation(image, None)
        augmented_image = keras.ops.convert_to_numpy(augmented_image)
        plt.imshow(augmented_image.astype("uint8"))
        plt.axis("off")
```

You can use take(N) to only sample N batches from the dataset. This is equivalent to inserting a break in the loop after the Nth batch.

Displays the first image in the output batch. For each of the nine iterations, this is a different augmentation of the same image.

If you train a new model using this data augmentation configuration, the model will never see the same input twice. But the inputs it sees are still heavily intercorrelated, because they come from a small number of original images—you can't produce new information; you can only remix existing information. As such, this may not be enough to completely get rid of overfitting. To further fight overfitting, you'll also add a `Dropout` layer to your model, right before the densely connected classifier.

Listing 8.20 Defining a new ConvNet that includes dropout

```
inputs = keras.Input(shape=(180, 180, 3))
x = layers.Rescaling(1.0 / 255)(inputs)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=512, kernel_size=3, activation="relu")(x)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dropout(0.25)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

model.compile(
    loss="binary_crossentropy",
    optimizer="adam",
    metrics=["accuracy"],
)
```

Let's train the model using data augmentation and dropout. Because we expect overfitting to occur much later during training, we will train for twice as many epochs—100. Note that we evaluate on images that aren't augmented—data augmentation is usually only performed at training time, as it is a regularization technique.

Listing 8.21 Training the regularized ConvNet on augmented images

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch_with_augmentation.keras",
        save_best_only=True,
        monitor="val_loss",
    )
]
history = model.fit(
    augmented_train_dataset,
    epochs=100,
    validation_data=validation_dataset,
    callbacks=callbacks,
)
```

Since we expect the model to overfit slower, we train for more epochs.

Let's plot the results again; see figure 8.11. Thanks to data augmentation and dropout, we start overfitting much later, around epochs 60–70 (compared to epoch 10 for the original model). The validation accuracy ends up peaking above 85%—a big improvement over our first try.

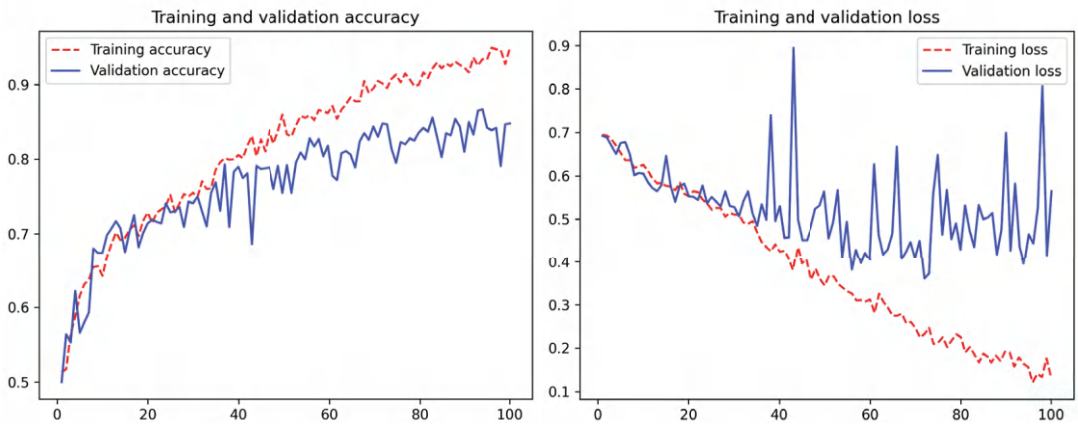


Figure 8.11 Training and validation metrics with data augmentation

Let's check the test accuracy.

Listing 8.22 Evaluating the model on the test set

```
test_model = keras.models.load_model(
    "convnet_from_scratch_with_augmentation.keras"
)
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

We get a test accuracy of 83.9%. It's starting to look good! If you're using Colab, make sure to download the saved file (`convnet_from_scratch_with_augmentation.keras`), as we will use it for some experiments in the next chapter.

By further tuning the model's configuration (such as the number of filters per convolution layer or the number of layers in the model), you may be able to get an even better accuracy, likely up to 90%. But it would prove difficult to go any higher just by training your own ConvNet from scratch because you have so little data to work with. As a next step to improve your accuracy on this problem, you'll have to use a pretrained model, which is the focus of the next two sections.

8.3 Using a pretrained model

A common and highly effective approach to deep learning on small image datasets is to use a pretrained model. A *pretrained model* is a model that was previously trained on a large dataset, typically on a large-scale image-classification task. If this original dataset is large enough and general enough, then the spatial hierarchy of features learned by the pretrained model can effectively act as a generic model of the visual world, and hence its features can prove useful for many different computer vision problems, even though these new problems may involve completely different classes than those of the original task. For instance, you might train a model on ImageNet (where classes are mostly animals and everyday objects) and then repurpose this trained model for something as remote as identifying furniture items in images. Such portability of learned features across different problems is a key advantage of deep learning compared to many older, shallow learning approaches, and it makes deep learning very effective for small-data problems.

In this case, let's consider a large ConvNet trained on the ImageNet dataset (1.4 million labeled images and 1,000 different classes). ImageNet contains many animal classes, including different species of cats and dogs, and you can thus expect it to perform well on the dogs-versus-cats classification problem.

We'll use the Xception architecture. This may be your first encounter with one of these cutesy model names—Xception, ResNet, EfficientNet, and so on; you'll get used to them if you keep doing deep learning for computer vision because they will come up frequently. You'll learn about the architectural details of Xception in the next chapter.

There are two ways to use a pretrained model: *feature extraction* and *fine-tuning*. We'll cover both of them. Let's start with feature extraction.

8.3.1 Feature extraction with a pretrained model

Feature extraction consists of using the representations learned by a previously trained model to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.

As you saw previously, ConvNets used for image classification comprise two parts: they start with a series of pooling and convolution layers, and they end with a densely connected classifier. The first part is called the *convolutional base* or *backbone* of the

model. In the case of ConvNets, feature extraction consists of taking the convolutional base of a previously trained network, running the new data through it, and training a new classifier on top of the output (see figure 8.12).

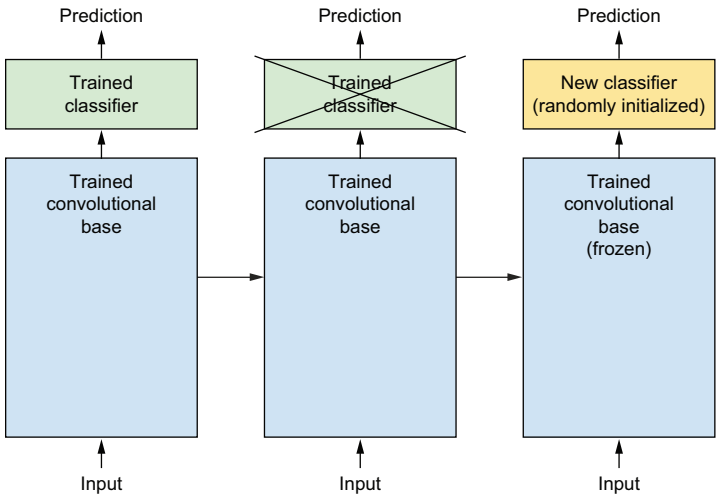


Figure 8.12 Swapping classifiers while keeping the same convolutional base

Why only reuse the convolutional base? Could you reuse the densely connected classifier as well? In general, doing so should be avoided. The reason is that the representations learned by the convolutional base are likely to be more generic and therefore more reusable: the feature maps of a ConvNet are presence maps of generic concepts over a picture, which is likely to be useful regardless of the computer vision problem at hand. But the representations learned by the classifier will necessarily be specific to the set of classes on which the model was trained—they will only contain information about the presence probability of this or that class in the entire picture. Additionally, representations found in densely connected layers no longer contain any information about where objects are located in the input image: these layers get rid of the notion of space, whereas the object location is still described by convolutional feature maps. For problems where object location matters, densely connected features are largely useless.

Note that the level of generality (and therefore reusability) of the representations extracted by specific convolution layers depends on the depth of the layer in the model. Layers that come earlier in the model extract local, highly generic feature maps (such as visual edges, colors, and textures), whereas layers that are higher up extract more abstract concepts (such as "cat ear" or "dog eye"). So if your new dataset differs a lot from the dataset on which the original model was trained, you may be better off using

only the first few layers of the model to do feature extraction, rather than using the entire convolutional base.

In this case, because the ImageNet class set contains multiple dog and cat classes, it's likely to be beneficial to reuse the information contained in the densely connected layers of the original model. But we'll choose not to, so we can cover the more general case where the class set of the new problem doesn't overlap the class set of the original model. Let's put this in practice by using the convolutional base of our pretrained model to extract interesting features from cat and dog images and then train a dogs-versus-cats classifier on top of these features.

We will use the *KerasHub* library to create all pretrained models used in this book. KerasHub contains Keras implementations of popular pretrained model architectures paired with pretrained weights that can be downloaded to your machine. It contains a number of ConvNets like Xception, ResNet, EfficientNet and MobileNet, as well as larger, generative models we will use in the later chapters of this book. Let's try using it to instantiate the Xception model trained on the ImageNet dataset.

NOTE KerasHub comes as a separate package from Keras. This package is pre-installed in Colab and Kaggle notebooks, but if you want to use it outside these environments you can install it yourself with `pip install keras-hub`.

Listing 8.23 Instantiating the Xception convolutional base

```
import keras_hub

conv_base = keras_hub.models.Backbone.from_preset("xception_41_imagenet")
```

You'll note a couple of things. First, KerasHub uses the term *backbone* to refer to the underlying feature extractor network without the classification head (it's a little easier to type than "convolutional base"). It also uses a special constructor called `from_preset()` that will download the configuration and weights for the Xception model.

What's that "41" in the name of the model we are using? Pretrained ConvNets are by convention often named by how "deep" they are. In this case, the 41 means that our Xception model has 41 trainable layers (conv and dense layers) stacked on top of each other. It's the "deepest" model we've used so far in the book by a good margin.

There's one more missing piece we need before we can use this model. Every pretrained ConvNet will do some rescaling and resizing of images before pretraining. It's important to make sure our input images *match*; otherwise, our model will need to relearn how to extract features from images with a totally different input range. Rather than keep track of which pretrained models use a `[0, 1]` input range for pixel values and which use a `[-1, 1]` range, we can use a KerasHub layer called `ImageConverter` that will rescale our images to match our pretrained checkpoint. It has the same special `from_preset()` constructor as the backbone class.

Listing 8.24 Instantiating the preprocessing paired with the Xception model

```
preprocessor = keras_hub.layers.ImageConverter.from_preset(
    "xception_41_imagenet",
    image_size=(180, 180),
)
```

At this point, there are two ways you could proceed:

- Running the convolutional base over your dataset, recording its output to a NumPy array on disk, and then using this data as input to a standalone, densely connected classifier similar to those you saw in chapters 4 and 5. This solution is fast and cheap to run, because it only requires running the convolutional base once for every input image, and the convolutional base is by far the most expensive part of the pipeline. But for the same reason, this technique won't allow you to use data augmentation.
- Extending the model you have (`conv_base`) by adding `Dense` layers on top and running the whole thing end to end on the input data. This will allow you to use data augmentation because every input image goes through the convolutional base every time it's seen by the model. But for the same reason, this technique is far more expensive than the first.

We'll cover both techniques. Let's walk through the code required to set up the first one: recording the output of `conv_base` on your data and using these outputs as inputs to a new model.

Fast feature extraction without data augmentation

We'll start by extracting features as NumPy arrays, by calling the `predict()` method of the `conv_base` model on our training, validation, and testing datasets. Let's iterate over our datasets to extract the pretrained model's features.

Listing 8.25 Extracting the image features and corresponding labels

```
def get_features_and_labels(dataset):
    all_features = []
    all_labels = []
    for images, labels in dataset:
        preprocessed_images = preprocessor(images)
        features = conv_base.predict(preprocessed_images, verbose=0)
        all_features.append(features)
        all_labels.append(labels)
    return np.concatenate(all_features), np.concatenate(all_labels)

train_features, train_labels = get_features_and_labels(train_dataset)
val_features, val_labels = get_features_and_labels(validation_dataset)
test_features, test_labels = get_features_and_labels(test_dataset)
```

Importantly, `predict()` only expects images, not labels, but our current dataset yields batches that contain both images and their labels.

The extracted features are currently of shape `(samples, 6, 6, 2048)`:

```
>>> train_features.shape
(2000, 6, 6, 2048)
```

At this point, you can define your densely connected classifier (note the use of dropout for regularization) and train it on the data and labels that you just recorded.

Listing 8.26 Defining and training the densely connected classifier

```
inputs = keras.Input(shape=(6, 6, 2048))
x = layers.GlobalAveragePooling2D()(inputs)
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.25)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(
    loss="binary_crossentropy",
    optimizer="adam",
    metrics=["accuracy"],
)

callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="feature_extraction.keras",
        save_best_only=True,
        monitor="val_loss",
    )
]
history = model.fit(
    train_features,
    train_labels,
    epochs=10,
    validation_data=(val_features, val_labels),
    callbacks=callbacks,
)
```

Averages spatial dimensions to flatten the feature map

Training is very fast because you only have to deal with two `Dense` layers—an epoch takes less than 1 second even on CPU.

Let's look at the loss and accuracy curves during training (see figure 8.13).

Listing 8.27 Plotting the results

```
import matplotlib.pyplot as plt

acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
```

```
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, "r--", label="Training accuracy")
plt.plot(epochs, val_acc, "b", label="Validation accuracy")
plt.title("Training and validation accuracy")
plt.legend()
plt.figure()
plt.plot(epochs, loss, "r--", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.legend()
plt.show()
```

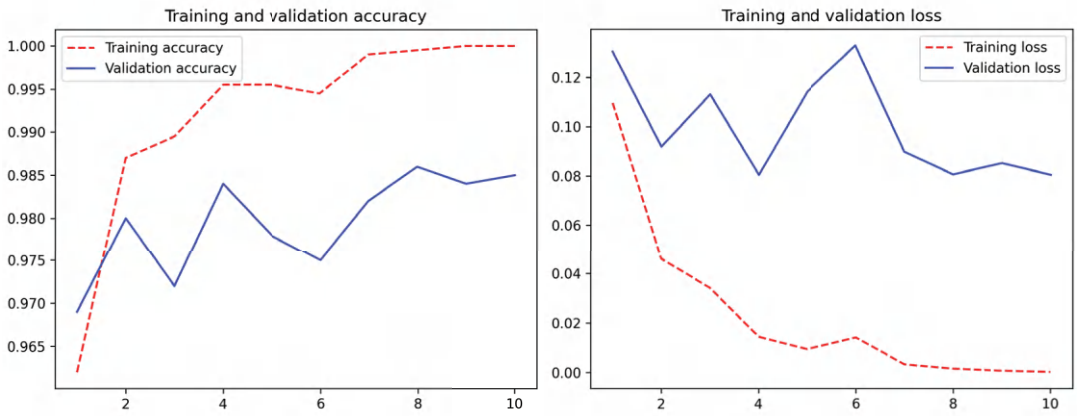


Figure 8.13 Training and validation metrics for plain feature extraction

You reach a validation accuracy of slightly over 98%—much better than you achieved in the previous section with the small model trained from scratch. This is a bit of an unfair comparison, however, because ImageNet contains many dog and cat instances, which means that our pretrained model already has the exact knowledge required for the task at hand. This won't always be the case when you use pretrained features.

However, the plots also indicate that you're overfitting almost from the start—despite using dropout with a fairly large rate. That's because this technique doesn't use data augmentation, which is essential for preventing overfitting with small image datasets.

Let's check the test accuracy:

```
test_model = keras.models.load_model("feature_extraction.keras")
test_loss, test_acc = test_model.evaluate(test_features, test_labels)
print(f"Test accuracy: {test_acc:.3f}")
```

We get test accuracy of 98.1%—a very nice improvement over training a model from scratch!

FEATURE EXTRACTION TOGETHER WITH DATA AUGMENTATION

Now, let's review the second technique we mentioned for doing feature extraction, which is much slower and more expensive but allows you to use data augmentation during training: creating a model that chains the `conv_base` with a new dense classifier and training it end to end on the inputs.

To do this, we will first freeze the convolutional base. *Freezing* a layer or set of layers means preventing their weights from being updated during training. Here, if you don't do this, then the representations that were previously learned by the convolutional base will be modified during training. Because the `Dense` layers on top are randomly initialized, very large weight updates would be propagated through the network, effectively destroying the representations previously learned.

In Keras, you freeze a layer or model by setting its `trainable` attribute to `False`.

Listing 8.28 Creating the frozen convolutional base

```
import keras_hub

conv_base = keras_hub.models.Backbone.from_preset(
    "xception_41_imagenet",
    trainable=False,
)
```

Setting `trainable` to `False` empties the list of trainable weights of the layer or model.

Listing 8.29 Printing the list of trainable weights before and after freezing

```
>>> conv_base.trainable = True
>>> len(conv_base.trainable_weights)
26
>>> conv_base.trainable = False
>>> len(conv_base.trainable_weights)
0
```

The number of trainable weights before freezing the conv base

The number of trainable weights after freezing the conv base

Now, we can just create a new model that chains together our frozen convolutional base and a dense classifier, like this:

```
inputs = keras.Input(shape=(180, 180, 3))
x = preprocessor(inputs)
x = conv_base(x)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(256)(x)
x = layers.Dropout(0.25)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
```

```
model = keras.Model(inputs, outputs)
model.compile(
    loss="binary_crossentropy",
    optimizer="adam",
    metrics=["accuracy"],
)
```

With this setup, only the weights from the two `Dense` layers that you added will be trained. That's a total of four weight tensors: two per layer (the main weight matrix and the bias vector). Note that for these changes to take effect, you must first compile the model. If you ever modify weight trainability after compilation, you should then recompile the model, or these changes will be ignored.

Let's train our model. We'll reuse our augmented dataset `augmented_train_dataset`. Thanks to data augmentation, it will take much longer for the model to start overfitting, so we can train for more epochs—let's do 30:

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="feature_extraction_with_data_augmentation.keras",
        save_best_only=True,
        monitor="val_loss",
    )
]
history = model.fit(
    augmented_train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks,
)
```

NOTE This technique is expensive enough that you should only attempt it if you have access to a GPU (such as the free GPU available in Colab)—it's intractable on CPU. If you can't run your code on GPU, then the previous technique is the way to go.

Let's plot the results again (see figure 8.14). This model reaches a validation accuracy of 98.2%.

Let's check the test accuracy.

Listing 8.30 Evaluating the model on the test set

```
test_model = keras.models.load_model(
    "feature_extraction_with_data_augmentation.keras"
)
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

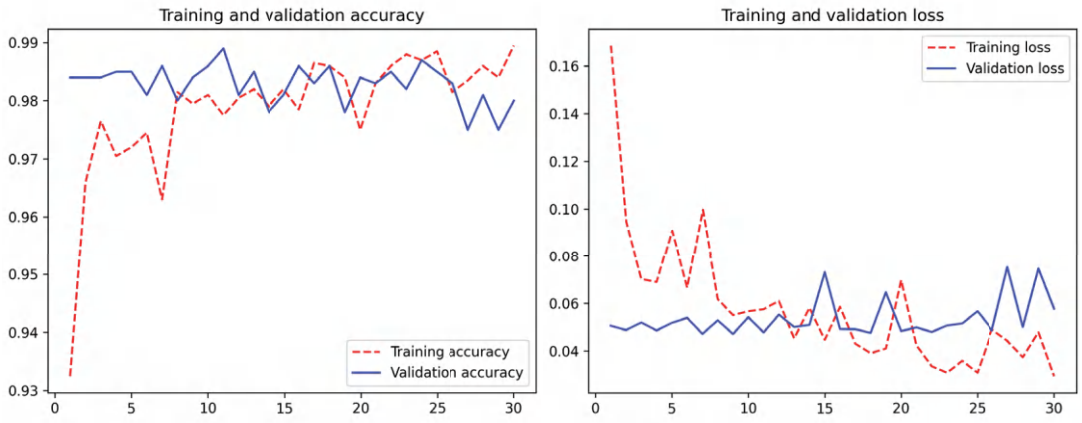


Figure 8.14 Training and validation metrics for feature extraction with data augmentation

We get a test accuracy of 98.4%. This is not an improvement over the previous model, which is a bit disappointing. This could be a sign that our data augmentation configuration does not exactly match the distribution of the test data. Let's see if we can do better with our latest attempt.

8.3.2 Fine-tuning a pretrained model

Another widely used technique for model reuse, complementary to feature extraction, is *fine-tuning* (see figure 8.15). Fine-tuning consists of unfreezing the frozen model base used for feature extraction and jointly training both the newly added part of the model (in this case, the fully connected classifier) and the base model. This is called *fine-tuning* because it slightly adjusts the more abstract representations of the model being reused to make them more relevant for the problem at hand.

We stated earlier that it's necessary to freeze the pretrained convolution base first to be able to train a randomly initialized classifier on top. For the same reason, it's only possible to fine-tune the convolutional base once the classifier on top has already been trained. If the classifier isn't already trained, then the error signal propagating through the network during training will be too large, and the representations previously learned by the layers being fine-tuned will be destroyed. Thus, the steps for fine-tuning a network are as follows:

1. Add your custom network on top of an already trained base network.
2. Freeze the base network.
3. Train the part you added.
4. Unfreeze the base network.
5. Jointly train both these layers and the part you added.

Note that you should not unfreeze "batch normalization" layers (`BatchNormalization`). Batch normalization and its effect on fine-tuning is explained in the next chapter.

You already completed the first three steps when doing feature extraction. Let's proceed with step 4: you'll unfreeze your `conv_base`.

Partial fine-tuning

In this case, we chose to unfreeze and fine-tune all of the Xception convolutional base. However, when dealing with large pretrained models, you may sometimes only unfreeze some of the top layers of the convolutional base, and leave the lower layers frozen. You're probably wondering, why only fine-tune some of the layers? Why the top ones specifically? Here's why:

- Earlier layers in the convolutional base encode more-generic, reusable features, whereas layers higher up encode more-specialized features. It's more useful to fine-tune the more specialized features because these are the ones that need to be repurposed on your new problem. There would be fast-decreasing returns in fine-tuning lower layers.
- The more parameters you're training, the more you're at risk of overfitting. The convolutional base has 15 million parameters, so it would be risky to attempt to train it on your small dataset.

Thus, it can be a good strategy to fine-tune only the top three or four layers in the convolutional base. You'd do something like this:

```
conv_base.trainable = True
for layer in conv_base.layers[:-4]:
    layer.trainable = False
```

Let's start fine-tuning the model using a very low learning rate. The reason for using a low learning rate is that you want to limit the magnitude of the modifications you make to the representations of the layers you're fine-tuning. Updates that are too large may harm these representations.

Listing 8.31 Fine-tuning the model

```
model.compile(
    loss="binary_crossentropy",
    optimizer=keras.optimizers.Adam(learning_rate=1e-5),
    metrics=["accuracy"],
)

callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="fine_tuning.keras",
        save_best_only=True,
        monitor="val_loss",
    )
]
```

```
history = model.fit(
    augmented_train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks,
)
```

You can now finally evaluate this model on the test data (see figure 8.15):

```
model = keras.models.load_model("fine_tuning.keras")
test_loss, test_acc = model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

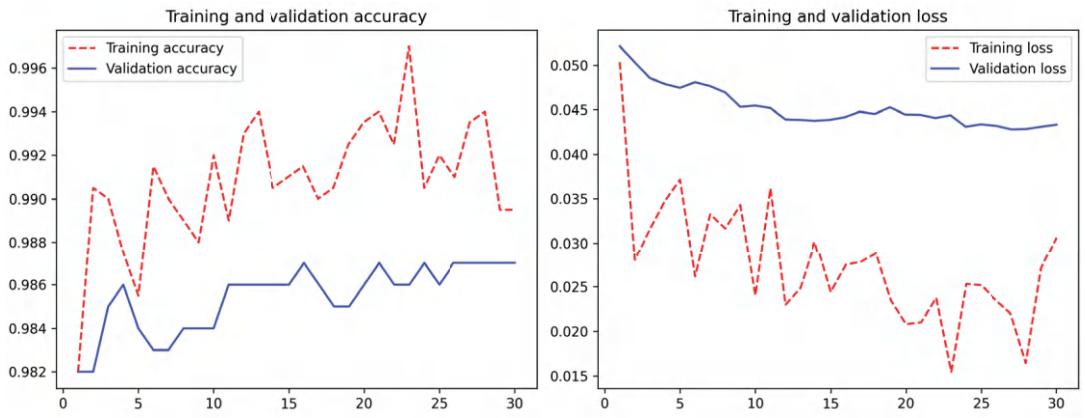


Figure 8.15 Training and validation metrics for fine-tuning

Here, you get a test accuracy of 98.6% (again, your own results may be within half a percentage point). In the original Kaggle competition around this dataset, this would have been one of the top results. It's not quite a fair comparison, however, since you used pretrained features that already contained prior knowledge about cats and dogs, which competitors couldn't use at the time.

On the positive side, by using modern deep learning techniques, you managed to reach this result using only a small fraction of the training data that was available for the competition (about 10%). There is a huge difference between being able to train on 20,000 samples compared to 2,000 samples!

Now you have a solid set of tools for dealing with image-classification problems—in particular, with small datasets.

Summary

- ConvNets excel at computer vision tasks. It's possible to train one from scratch, even on a very small dataset, with decent results.
- ConvNets work by learning a hierarchy of modular patterns and concepts to represent the visual world.
- On a small dataset, overfitting will be the main issue. Data augmentation is a powerful way to fight overfitting when you're working with image data.
- It's easy to reuse an existing ConvNet on a new dataset via feature extraction. This is a valuable technique for working with small image datasets.
- As a complement to feature extraction, you can use fine-tuning, which adapts to a new problem some of the representations previously learned by an existing model. This pushes performance a bit further.

9 ConvNet architecture patterns

This chapter covers

- The modularity-hierarchy-reuse formula for model architecture
- An overview of standard best practices for building ConvNets: residual connections, batch normalization, and depthwise separable convolutions
- Ongoing design trends for computer vision models

A model's "architecture" is the sum of the choices that went into creating it: which layers to use, how to configure them, in what arrangement to connect them. These choices define the *hypothesis space* of your model: the space of possible functions that gradient descent can search over, parameterized by the model's weights. Like feature engineering, a good hypothesis space encodes *prior knowledge* that you have about the problem at hand and its solution. For instance, using convolution layers means that you know in advance that the relevant patterns present in your input

images are translation-invariant. To effectively learn from data, you need to make assumptions about what you're looking for.

Model architecture is often the difference between success and failure. If you make inappropriate architecture choices, your model may be stuck with suboptimal metrics, and no amount of training data will save it. Inversely, a good model architecture will accelerate learning and will enable your model to make efficient use of the training data available, reducing the need for large datasets. A good model architecture is one that *reduces the size of the search space* or otherwise *makes it easier to converge to a good point of the search space*. Just like feature engineering and data curation, model architecture is all about *making the problem simpler* for gradient descent to solve—and remember that gradient descent is a pretty stupid search process, so it needs all the help it can get.

Model architecture is more an art than a science. Experienced machine learning engineers are able to intuitively cobble together high-performing models on their first try, while beginners often struggle to create a model that trains at all. The keyword here is *intuitively*: no one can give you a clear explanation of what works and what doesn't. Experts rely on pattern matching, an ability that they acquire through extensive practical experience. You'll develop your own intuition throughout this book. However, it's not *all* about intuition either—there isn't much in the way of actual science, but like in any engineering discipline, there are best practices.

In the following sections, we'll review a few essential ConvNet architecture best practices, in particular, *residual connections*, *batch normalization*, and *separable convolution*. Once you master how to use them, you will be able to build highly effective image models. We will demonstrate how to apply them on our dogs-versus-cats classification problem.

Let's start from the bird's-eye view: the modularity-hierarchy-reuse (MHR) formula for system architecture.

9.1 Modularity, hierarchy, and reuse

If you want to make a complex system simpler, there's a universal recipe you can apply: just structure your amorphous soup of complexity into *modules*, organize the modules into a *hierarchy*, and start *reusing* the same modules in multiple places as appropriate ("reuse" is another word for *abstraction*). That's the modularity-hierarchy-reuse (MHR) formula (see figure 9.1), and it underlies system architecture across pretty much every domain where the term *architecture* is used. It's at the heart of the organization of any system of meaningful complexity, whether it's a cathedral, your own body, the US Navy, or the Keras codebase.

If you're a software engineer, you're already keenly familiar with these principles: an effective codebase is one that is modular, hierarchical, and where you don't reimplement the same thing twice but instead rely on reusable classes and functions. If you factor your code by following these principles, you could say you're doing "software architecture."

Deep learning itself is simply the application of this recipe to continuous optimization via gradient descent: you take a classic optimization technique (gradient descent

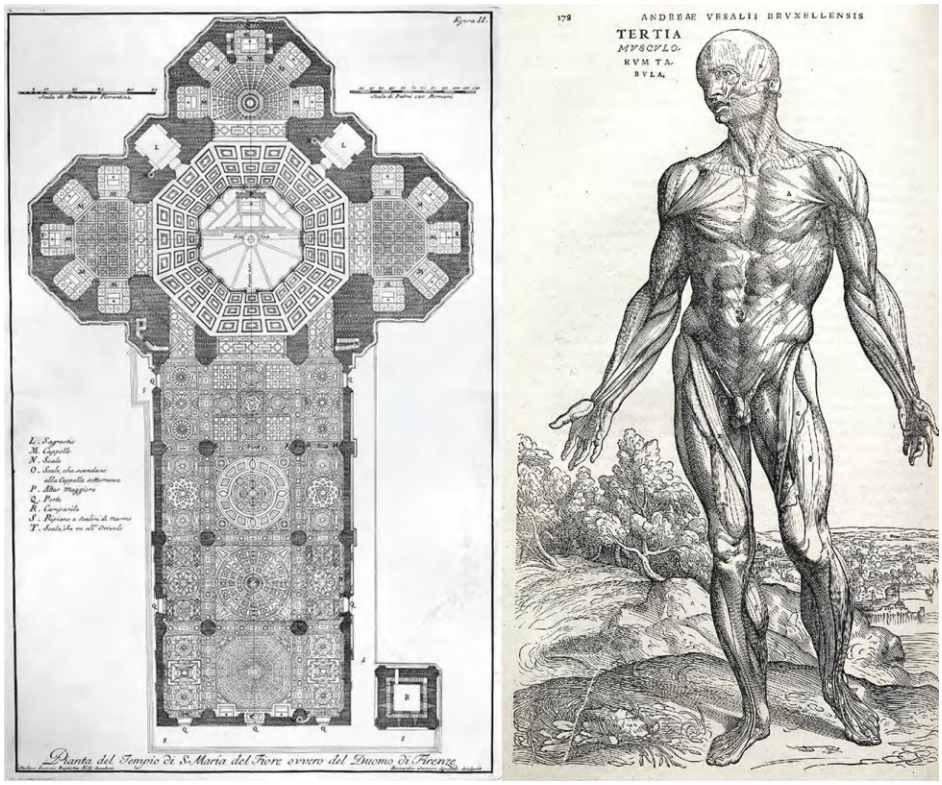


Figure 9.1 Complex systems follow a hierarchical structure and are organized into distinct modules, which are reused multiple times (such as your 4 limbs, which are all variants of the same blueprint, or your 20 fingers).

over a continuous function space), and you structure the search space into modules (layers), organized into a deep hierarchy (often just a stack, the simplest kind of hierarchy), where you reuse whatever you can (for instance, convolutions are all about reusing the same information in different spatial locations).

Likewise, deep learning model architecture is primarily about making clever use of modularity, hierarchy, and reuse. You'll notice that all popular ConvNet architectures are not only structured into layers, they're structured into repeated groups of layers (called `_blocks_` or `_modules_`). For instance, Xception architecture (used in the previous chapter) is structured into repeated `SeparableConv` - `SeparableConv` - `MaxPooling` blocks (see figure 9.2).

Further, most ConvNets often feature pyramid-like structures (*feature hierarchies*). Recall, for example, the progression in the number of convolution filters we used in the first ConvNet we built in the previous chapter: 32, 64, 128. The number of filters grows with layer depth, while the size of the feature maps shrinks accordingly. You'll notice the same pattern in the blocks of the Xception model (see figure 9.2).

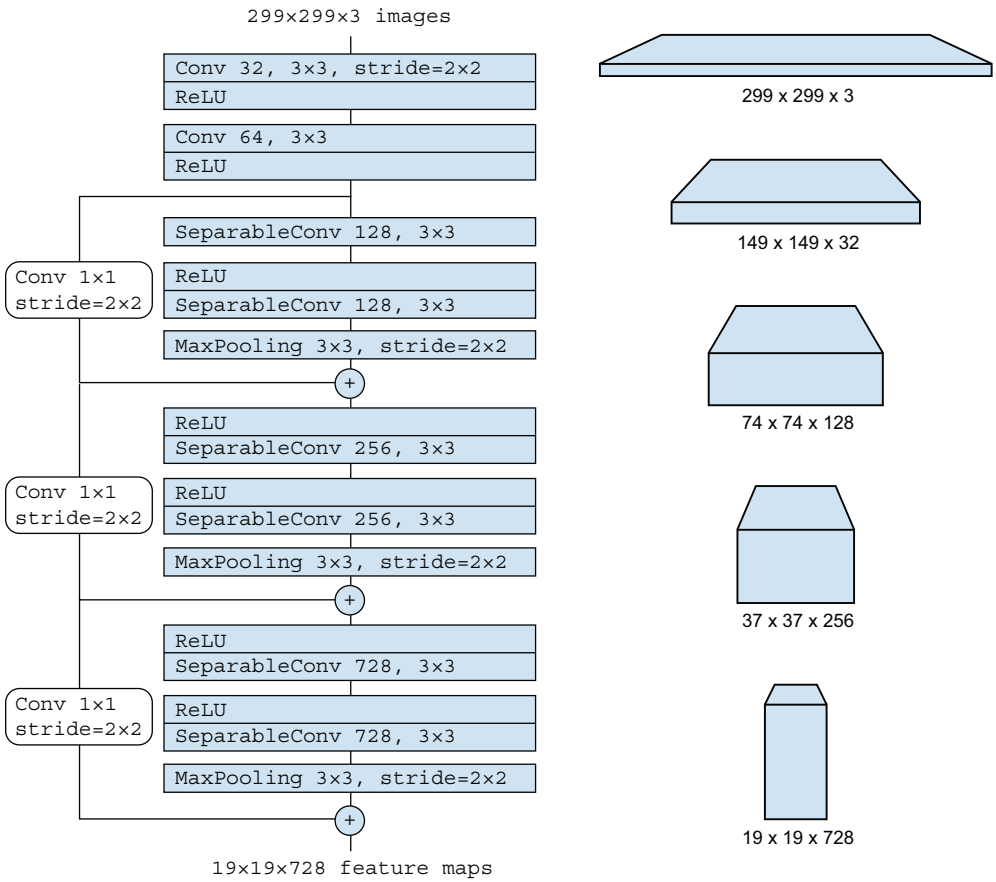


Figure 9.2 The "entry flow" of the Xception architecture: note the repeated layer blocks and the gradually shrinking and deepening feature maps, going from 299 x 299 x 3 to 19 x 19 x 728.

Deeper hierarchies are intrinsically good because they encourage feature reuse and, therefore, abstraction. In general, a deep stack of narrow layers performs better than a shallow stack of large layers. However, there's a limit to how deep you can stack layers: the problem of *vanishing gradients*. This leads us to our first essential model architecture pattern: residual connections.

On the importance of ablation studies in deep learning research

Deep learning architectures are often more *evolved* than designed—they were developed by repeatedly trying things and selecting what seemed to work. Much like in biological systems, if you take any complicated experimental deep learning setup,

(continued)

chances are you can remove a few modules (or replace some trained features with random ones) with no loss of performance.

This is made worse by the incentives that deep learning researchers face: by making a system more complex than necessary, they can make it appear more interesting or more novel and thus increase their chances of getting a paper through the peer review process. If you read lots of deep learning papers, you will notice that they're often optimized for peer review in both style and content in ways that actively hurt clarity of explanation and reliability of results. For instance, mathematics in deep learning papers is rarely used for clearly formalizing concepts or deriving unobvious results—rather, it gets used as a *signal of seriousness*, like an expensive suit on a salesperson.

The goal of research shouldn't be merely to publish but to *generate reliable knowledge*. Crucially, *understanding causality* in your system is the most straightforward way to generate reliable knowledge. And there's a very low-effort way to look into causality: *ablation studies*. Ablation studies consist of systematically trying to remove parts of a system—that is, make it simpler—to identify where its performance actually comes from. If you find that X + Y + Z gives you good results, also try X, Y, Z, X + Y, X + Z, Y + Z and see what happens.

If you become a deep learning researcher, cut through the noise in the research process: do ablation studies for your models. Always ask, could there be a simpler explanation? Is this added complexity really necessary? Why?

9.2 Residual connections

You probably know about the game of *telephone*, also called *Chinese whispers* in the UK and *téléphone arabe* in France, where an initial message is whispered in the ear of a player, who then whispers it in the ear of the next player, and so on. The final message ends up bearing little resemblance to its original version. It's a fun metaphor for the cumulative errors that occur in sequential transmission over a noisy channel.

As it happens, backpropagation in a sequential deep learning model is pretty similar to the game of telephone. You've got a chain of functions, like this one:

```
y = f4(f3(f2(f1(x))))
```

The name of the game is to adjust the parameters of each function in the chain based on the error recorded on the output of `f4` (the loss of the model). To adjust `f1`, you'll need to percolate error information through `f2`, `f3`, and `f4`. However, each successive function in the chain introduces some amount of noise in the process. If your function chain is too deep, this noise starts overwhelming gradient information, and backpropagation stops working. Your model won't train at all. This is called the *vanishing gradients* problem.

The fix is simple: just force each function in the chain to be nondestructive—to retain a noiseless version of the information contained in the previous input. The easiest way to implement this is called a *residual connection*. It's dead easy: just add the input of a layer or block of layers back to its output (see figure 9.3). The residual connection acts as an *information shortcut* around destructive or noisy blocks (such as blocks that contain ReLU activations or dropout layers), enabling error gradient information from early layers to propagate noiselessly through a deep network. This technique was introduced in 2015 with the ResNet family of models (developed by He et al. at Microsoft).[1]

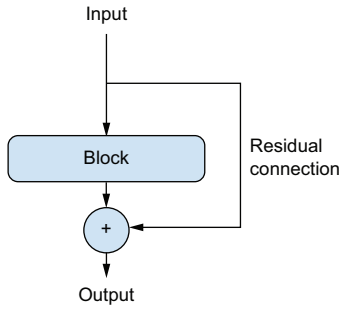


Figure 9.3 A residual connection around a processing block

In practice, you'd implement a residual connection like the following listing.

Listing 9.1 A residual connection in pseudocode

```
x = ...
residual = x
x = block(x)
x = add([x, residual])
```

Some input tensor

Saves a reference to the original input. This is called the residual.

This computation block can potentially be destructive or noisy, and that's fine.

Adds the original input to the layer's output. The final output will thus always preserve full information about the original input.

Note that adding the input back to the output of a block implies that the output should have the same shape as the input. This is not the case if your block includes convolutional layers with an increased number of filters or a max pooling layer. In such cases, use a 1 × 1 `Conv2D` layer with no activation to linearly project the residual to the desired output shape. You'd typically use `padding="same"` in the convolution layers in your target block to avoid spatial downsampling due to padding, and you'd use strides in the residual projection to match any downsampling caused by a max pooling layer.

Listing 9.2 The target block changing the number of output filters

```
import keras
from keras import layers

inputs = keras.Input(shape=(32, 32, 3))
```

[1] Kaiming He et al., "Deep Residual Learning for Image Recognition," Conference on Computer Vision and Pattern Recognition (2015), https://arxiv.org/abs/1512.03385.

```
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

Sets aside the residual

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. We use padding="same" to avoid downsampling due to padding.

Now the block output and the residual have the same shape and can be added.

The residual only had 32 filters, so we use a 1 x 1 Conv2D to project it to the correct shape.

Listing 9.3 The target block including a max pooling layer

This is the block of two layers around which we create a residual connection: it includes a 2 x 2 max pooling layer. We use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
```

Sets aside the residual

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

Now the block output and the residual have the same shape and can be added.

To make these ideas more concrete, here's an example of a simple ConvNet structured into a series of blocks, each made of two convolution layers and one optional max pooling layer, with a residual connection around each block:

If we use max pooling, we add a strided convolution to project the residual to the expected shape.

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1.0 / 255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
```

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

```
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

If we don't use max pooling, we only project the residual if the number of channels has changed.

First block

Second block. Note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.

Let's take a look at the model summary:

```
>>> model.summary()
Model: "functional"
```

Layer (type)	Output Shape	Param #	Connected to
input_layer_2 (InputLayer)	(None, 32, 32, 3)	0	-
rescaling (Rescaling)	(None, 32, 32, 3)	0	input_layer_2[0][0]
conv2d_6 (Conv2D)	(None, 32, 32, 32)	896	rescaling[0][0]
conv2d_7 (Conv2D)	(None, 32, 32, 32)	9,248	conv2d_6[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 32)	0	conv2d_7[0][0]
conv2d_8 (Conv2D)	(None, 16, 16, 32)	128	rescaling[0][0]
add_2 (Add)	(None, 16, 16, 32)	0	max_pooling2d_1[0]… conv2d_8[0][0]
conv2d_9 (Conv2D)	(None, 16, 16, 64)	18,496	add_2[0][0]
conv2d_10 (Conv2D)	(None, 16, 16, 64)	36,928	conv2d_9[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 8, 8, 64)	0	conv2d_10[0][0]

conv2d_11 (Conv2D)	(None, 8, 8, 64)	2,112	add_2[0][0]
add_3 (Add)	(None, 8, 8, 64)	0	max_pooling2d_2[0]… conv2d_11[0][0]
conv2d_12 (Conv2D)	(None, 8, 8, 128)	73,856	add_3[0][0]
conv2d_13 (Conv2D)	(None, 8, 8, 128)	147,584	conv2d_12[0][0]
conv2d_14 (Conv2D)	(None, 8, 8, 128)	8,320	add_3[0][0]
add_4 (Add)	(None, 8, 8, 128)	0	conv2d_13[0][0], conv2d_14[0][0]
global_average_pool… (GlobalAveragePooli…	(None, 128)	0	add_4[0][0]
dense (Dense)	(None, 1)	129	global_average_poo…

```
Total params: 297,697 (1.14 MB)
Trainable params: 297,697 (1.14 MB)
Non-trainable params: 0 (0.00 B)
```

With residual connections, you can build networks of arbitrary depth, without having to worry about vanishing gradients. Now, let's move on to the next essential ConvNet architecture pattern: *batch normalization*.

9.3 Batch normalization

Normalization in machine learning is a broad category of methods that seek to make different samples seen by a machine learning model more similar to each other, which helps the model learn and generalize well to new data. The most common form of data normalization is one you've seen several times in this book already: centering the data on zero by subtracting the mean from the data and giving the data a unit standard deviation by dividing the data by its standard deviation. In effect, this makes the assumption that the data follows a normal (or Gaussian) distribution and makes sure this distribution is centered and scaled to unit variance:

```
normalized_data = (data - np.mean(data, axis=...)) / np.std(data, axis=...)
```

Previous examples you saw in this book normalized data before feeding it into models. But data normalization may be a concern after every transformation performed by the network: even if the data entering a `Dense` or `Conv2D` network has a 0 mean and unit variance, there's no reason to expect a priori that this will be the case for the data coming out. Could normalizing intermediate activations help?

Batch normalization does just that. It's a type of layer (`BatchNormalization` in Keras) introduced in 2015 by Ioffe and Szegedy;[2] it can adaptively normalize data even as the mean and variance change over time during training. During training, it uses the mean and variance of the current batch of data to normalize samples, and during inference (when a big enough batch of representative data may not be available), it uses an exponential moving average of the batchwise mean and variance of the data seen during training.

Although Ioffe and Szegedy's original paper suggested that batch normalization operates by "reducing internal covariate shift," no one really knows for sure why batch normalization helps. There are various hypotheses but no certitudes. You'll find that this is true of many things in deep learning—deep learning is not an exact science but a set of ever-changing, empirically derived engineering best practices, woven together by unreliable narratives. You will sometimes feel like the book you have in hand tells you *how* to do something but doesn't quite satisfactorily say *why* it works: that's because we know the how but we don't know the why. Whenever a reliable explanation is available, we make sure to mention it. Batch normalization isn't one of those cases.

In practice, the main effect of batch normalization appears to be that it helps with gradient propagation—much like residual connections—and thus allows for deeper networks. Some very deep networks can only be trained if they include multiple `BatchNormalization` layers. For instance, batch normalization is used liberally in many of the advanced ConvNet architectures that come packaged with Keras, such as ResNet50, EfficientNet, and Xception.

The `BatchNormalization` layer can be used after any layer—`Dense`, `Conv2D`, and so on:

```
x = ...
x = layers.Conv2D(32, 3, use_bias=False)(x)
x = layers.BatchNormalization()(x)
```

Because the output of the Conv2D layer gets normalized, the layer doesn't need its own bias vector.

> **NOTE** Both `Dense` and `Conv2D` involve a "bias vector," a learned variable whose purpose is to make the layer *affine* rather than purely linear. For instance, `Conv2D` returns, schematically, `y = conv(x, kernel) + bias`, and `Dense` returns `y = dot(x, kernel) + bias`. Because the normalization step will take care of centering the layer's output on zero, the bias vector is no longer needed when using `BatchNormalization`, and the layer can be created without it via the option `use_bias=False`. This makes the layer slightly leaner.

Importantly, I generally recommend placing the previous layer's activation *after* the batch normalization layer (although this is still a subject of debate). So instead of doing

[2] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *Proceedings of the 32nd International Conference on Machine Learning* (2015), https://arxiv.org/abs/1502.03167.

```
x = layers.Conv2D(32, 3, activation="relu")(x)
x = layers.BatchNormalization()(x)
```

you would actually do the following:

```
x = layers.Conv2D(32, 3, use_bias=False)(x)
x = layers.BatchNormalization()(x)
x = layers.Activation("relu")(x)
```

Note the lack of activation here.

We place the activation after the BatchNormalization layer.

The intuitive reason why is that batch normalization will center your inputs on zero, while your ReLU activation uses zero as a pivot for keeping or dropping activated channels: doing normalization before the activation maximizes the utilization of the ReLU. That said, this ordering best practice is not exactly critical, so if you do convolution-activation-batch normalization, your model will still train, and you won't necessarily see worse results.

> **NOTE** Batch normalization has many quirks. One of the main ones relates to fine-tuning: when fine-tuning a model that includes `BatchNormalization` layers, I recommend leaving these layers frozen (set their `trainable` attribute to `False`). Otherwise, they will keep updating their internal mean and variance, which can interfere with the very small updates applied to the surrounding `Conv2D` layers.

Now, let's take a look at the last architecture pattern in our series: depthwise separable convolutions.

9.4 Depthwise separable convolutions

What if we told you that there's a layer you can use as a drop-in replacement for `Conv2D` that will make your model smaller (fewer trainable weight parameters), leaner (fewer floating-point operations), and cause it to perform a few percentage points better on its task? That is precisely what the *depthwise separable convolution* layer does (`SeparableConv2D` in Keras). This layer performs a spatial convolution on each channel of its input, independently, before mixing output channels via a pointwise convolution (a 1×1 convolution), as shown in figure 9.4.

This is equivalent to separating the learning of spatial features and the learning of channel-wise features. In much the same way that convolution relies on the assumption that the patterns in images are not tied to specific locations, depthwise separable convolution relies on the assumption that *spatial locations* in intermediate activations are *highly correlated*, but *different channels* are *highly independent*. Because this assumption is generally true for the image representations learned by deep

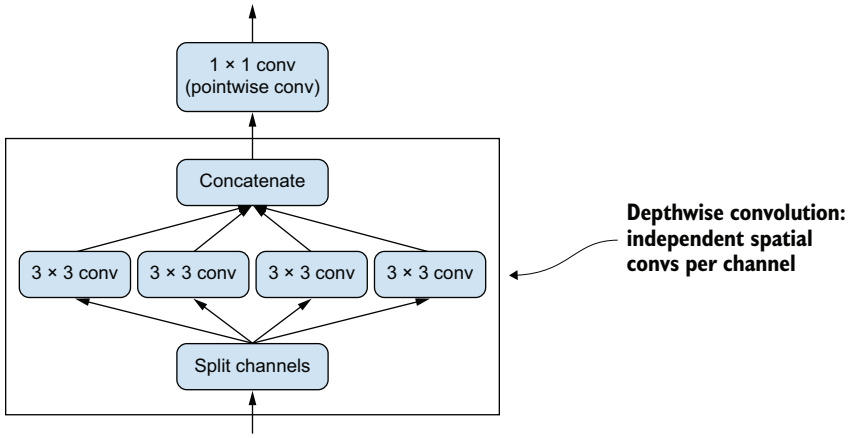


Figure 9.4 Depthwise separable convolution: a depthwise convolution followed by a pointwise convolution

neural networks, it serves as a useful prior that helps the model make more efficient use of its training data. A model with stronger priors about the structure of the information it will have to process is a better model—as long as the priors are accurate.

Depthwise separable convolution requires significantly fewer parameters and involves fewer computations compared to regular convolution, while having comparable representational power. They result in smaller models that converge faster and are less prone to overfitting. These advantages become especially important when you're training small models from scratch on limited data.

When it comes to larger-scale models, depthwise separable convolutions are the basis of the Xception architecture, a high-performing ConvNet that comes packaged with Keras. You can read more about the theoretical grounding for depthwise separable convolutions and Xception in the paper "Xception: Deep Learning with Depthwise Separable Convolutions."[3]

The co-evolution of hardware, software, and algorithms

Consider a regular convolution operation with a 3 x 3 window, 64 input channels, and 64 output channels. It uses 3 × 3 × 64 × 64 = 36,864 trainable parameters, and when you apply it to an image, it runs a number of floating-point operations that is proportional to this parameter count. Meanwhile, consider an equivalent depthwise separable convolution: it only involves 3 × 3 × 64 + 64 × 64 = 4,672 trainable parameters and proportionally fewer floating-point operations. This efficiency improvement only increases as the number of filters or the size of the convolution windows gets larger.

[3] François Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," Conference on Computer Vision and Pattern Recognition (2017), https://arxiv.org/abs/1610.02357.

(continued)

As a result, you would expect depthwise separable convolutions to be dramatically faster, right? Hold on. This would be true if you were writing simple CUDA or C++ implementations of these algorithms—in fact, you do see a meaningful speedup when running on CPU, where the underlying implementation is parallelized C++. But in practice, you're probably using a GPU, and what you're executing on it is far from a "simple" CUDA implementation: it's a *cuDNN kernel*, a piece of code that has been extraordinarily optimized, down to each machine instruction. It certainly makes sense to spend a lot of effort optimizing this code, since cuDNN convolutions on NVIDIA hardware are responsible for many exaflops of computation every day. But a side effect of this extreme micro-optimization is that alternative approaches have little chance to compete on performance—even approaches that have significant intrinsic advantages, like depthwise separable convolutions.

Despite repeated requests to NVIDIA, depthwise separable convolutions have not benefited from nearly the same level of software and hardware optimization as regular convolutions, and as a result they remain only about as fast as regular convolutions, even though they're using quadratically fewer parameters and floating-point operations. Note, though, that using depthwise separable convolutions remains a good idea even if it does not result in a speedup: their lower parameter count means that you are less at risk of overfitting, and their assumption that channels should be uncorrelated leads to faster model convergence and more robust representations.

What is a slight inconvenience in this case can become an impassable wall in other situations: because the entire hardware and software ecosystem of deep learning has been micro-optimized for a very specific set of algorithms (in particular, ConvNets trained via backpropagation), there's an extremely high cost to steering away from the beaten path. If you were to experiment with alternative algorithms, such as gradient-free optimization or spiking neural networks, the first few parallel C++ or CUDA implementations you'd come up with would be orders of magnitude slower than a good old ConvNet—no matter how clever and efficient your ideas were. Convincing other researchers to adopt your method would be a tough sell, even if it were just plain better.

You could say that modern deep learning is the product of a co-evolution process between hardware, software, and algorithms: the availability of NVIDIA GPUs and CUDA led to the early success of backpropagation-trained ConvNets, which led NVIDIA to optimize its hardware and software for these algorithms, which in turn led to consolidation of the research community behind these methods. At this point, figuring out a different path would require a multiyear reengineering of the entire ecosystem.

9.5 Putting it together: A mini Xception-like model

As a reminder, here are the ConvNet architecture principles you've learned so far:

- Your model should be organized into repeated *blocks* of layers, usually made of multiple convolution layers and a max pooling layer.
- The number of filters in your layers should increase as the size of the spatial feature maps decreases.

- Deep and narrow is better than broad and shallow.
- Introducing residual connections around blocks of layers helps you train deeper networks.
- It can be beneficial to introduce batch normalization layers after your convolution layers.
- It can be beneficial to replace `Conv2D` layers with `SeparableConv2D` layers, which are more parameter efficient.

Let's bring all of these ideas together into a single model. Its architecture resembles a smaller version of Xception. We'll apply it to the dogs-versus-cats task from last chapter. For data loading and model training, simply reuse the exact same setup as what we used in chapter 8, section 8.2—but replace the model definition with the following ConvNet:

We apply a series of convolutional blocks with increasing feature depth. Each block consists of two batch-normalized depthwise separable convolution layers and a max pooling layer, with a residual connection around the entire block.

The assumption that underlies separable convolution, "Feature channels are largely independent," does not hold for RGB images! Red, green, and blue color channels are actually highly correlated in natural images. As such, the first layer in our model is a regular Conv2D layer. We'll start using SeparableConv2D afterward.

Don't forget input rescaling!

```
import keras

inputs = keras.Input(shape=(180, 180, 3))
x = layers.Rescaling(1.0 / 255)(inputs)
x = layers.Conv2D(filters=32, kernel_size=5, use_bias=False)(x)

for size in [32, 64, 128, 256, 512]:
    residual = x

    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(size, 3, padding="same", use_bias=False)(x)

    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(size, 3, padding="same", use_bias=False)(x)

    x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

    residual = layers.Conv2D(
        size, 1, strides=2, padding="same", use_bias=False
    )(residual)
    x = layers.add([x, residual])

x = layers.GlobalAveragePooling2D()(x)
```

In the original model, we used a Flatten layer before the Dense layer. Here, we go with a GlobalAveragePooling2D layer.

```
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Like in the original model, we add a dropout layer for regularization.

This ConvNet has a trainable parameter count of 721,857, significantly lower than the 1,569,089 trainable parameters of the model from the previous chapter, yet it achieves better results. Figure 9.5 shows the training and validation curves.

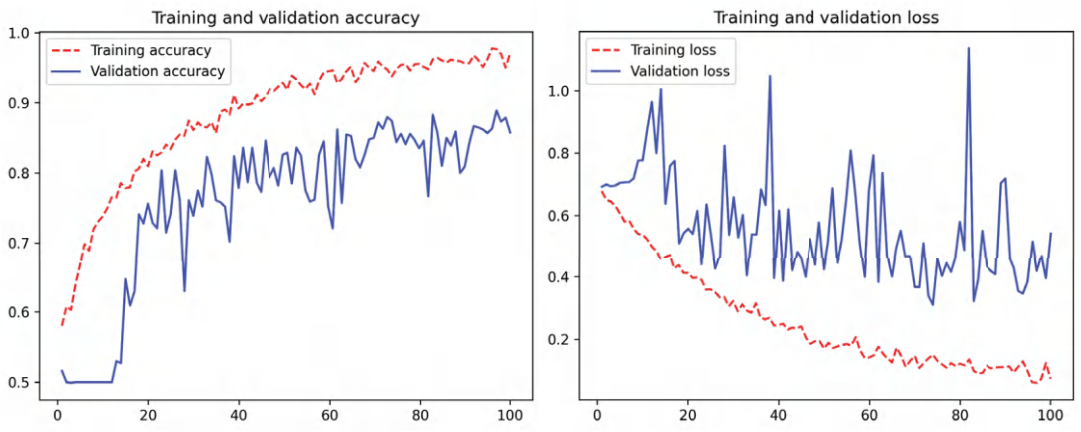


Figure 9.5 Training and validation metrics with a Xception-like architecture

You'll find that our new model achieves a test accuracy of 90.8%—compared to 83.9% for the previous model. As you can see, following architecture best practices does have an immediate, sizeable effect on model performance!

At this point, if you want to further improve performance, you should start systematically tuning the hyperparameters of your architecture—a topic we cover in detail in chapter 18. We haven't gone through this step here, so the configuration of the previous model is purely from the best practices we outlined, plus, when it comes to gauging model size, a small amount of intuition.

9.6 Beyond convolution: Vision Transformers

While ConvNets have been dominating the field of computer vision since the mid-2010s, they've been recently competing with an alternative architecture: Vision Transformers (or ViTs for short). It may well be that ViTs will end up replacing ConvNets in the long term—though, for now, ConvNets remain your best option in most cases.

You don't yet know what Transformers are because we'll cover them in chapter 15. In short, the Transformer architecture was developed to process text—it's fundamentally

a sequence-processing architecture. And Transformers are very good at it, which has led to the question: could we also use them for images?

Because ViTs are a type of Transformer, they also process sequences: they split up an image into a 1D sequence of patches, turn each patch into a flat vector, and process the vector sequence. The Transformer architecture allows ViTs to capture long-range relationships between different parts of the image, something ConvNets can sometimes struggle with.

Our general experience with Transformers is that they're a great choice if you're working with a massive dataset. They're simply better at utilizing large amounts of data. However, for smaller datasets, they tend to be suboptimal for two reasons. First, they lack the spatial prior of ConvNets—the 2D patch-based architecture of ConvNets incorporates more assumptions about the local structure of the visual space, making them more data efficient. Second, for ViTs to shine, they need to be really large. They end up being unwieldy for anything smaller than ImageNet.

The battle for image recognition supremacy is far from over, but ViTs have undoubtedly opened a new and exciting chapter. You'll probably work with this architecture in the context of large-scale generative image models—a topic we'll cover in chapter 17. For your small-scale image classification needs, however, ConvNets remain your best bet.

This concludes our introduction to essential ConvNet architecture best practices. With these principles in hand, you'll be able to develop higher-performing models across a wide range of computer vision tasks. You're now well on your way to becoming a proficient computer vision practitioner. To further deepen your expertise, there's one last important topic we need to cover: interpreting how a model arrives at its predictions.

Summary

- The architecture of a deep learning model encodes key assumptions about the nature of the problem at hand.
- The modularity-hierarchy-reuse formula underpins the architecture of nearly all complex systems, including deep learning models.
- Key architecture patterns for computer vision include residual connections, batch normalization, and depthwise separable convolutions.
- Vision Transformers are an up-and-coming alternative to ConvNets for large-scale computer vision tasks.

10 Interpreting what ConvNets learn

This chapter covers

- Interpreting how ConvNets decompose an input image
- Visualizing the filters learned by ConvNets
- Visualizing areas in an image responsible for a certain classification decision

A fundamental problem when building a computer vision application is that of *interpretability*: *Why* did your classifier think a particular image contained a fridge, when all you can see is a truck? This is especially relevant to use cases where deep learning is used to complement human expertise, such as medical imaging use cases. This chapter will get you familiar with a range of different techniques for visualizing what ConvNets learn and understanding the decisions they make.

It's often said that deep learning models are "black boxes": they learn representations that are difficult to extract and present in a human-readable form. Although this is partially true for certain types of deep learning models, it's definitely not true for ConvNets. The representations learned by ConvNets are highly amenable to visualization, in large part because they're *representations of visual concepts*. Since 2013, a

wide array of techniques has been developed for visualizing and interpreting these representations. We won't survey all of them, but we'll cover three of the most accessible and useful ones:

- *Visualizing intermediate ConvNet outputs (intermediate activations)*—Useful for understanding how successive ConvNet layers transform their input, and for getting a first idea of the meaning of individual ConvNet filters
- *Visualizing ConvNets filters*—Useful for understanding precisely what visual pattern or concept each filter in a ConvNet is receptive to
- *Visualizing heatmaps of class activation in an image*—Useful for understanding which parts of an image were identified as belonging to a given class, thus allowing you to localize objects in images
- *Visualizing the way ConvNets learn semantically organized manifolds of images*—Useful for detecting outliers and getting a sense for what images are seen by your model as semantically ambiguous, thus allowing you to clean and improve your training dataset as well as refine your evaluation dataset

For the first method—activation visualization—you'll use the small ConvNet that you trained from scratch on the dogs-versus-cats classification problem in chapter 8. For the next two methods, you'll use a pretrained Xception model.

10.1 Visualizing intermediate activations

Visualizing intermediate activations consists of displaying the values returned by various convolution and pooling layers in a model, given a certain input (the output of a layer is often called its *activation*, the output of the activation function). This gives a view into how an input is decomposed into the different filters learned by the network. You want to visualize feature maps with three dimensions: width, height, and depth (channels). Each channel encodes relatively independent features, so the proper way to visualize these feature maps is by independently plotting the contents of every channel as a 2D image. Let's start by loading the model that you saved in section 8.2:

```
>>> import keras
>>> model = keras.models.load_model(
...     "convnet_from_scratch_with_augmentation.keras"
... )
>>> model.summary()
Model: "functional_3"
```

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 180, 180, 3)	0
rescaling_1 (Rescaling)	(None, 180, 180, 3)	0
conv2d_11 (Conv2D)	(None, 178, 178, 32)	896

max_pooling2d_6 (MaxPooling2D)	(None, 89, 89, 32)	0
conv2d_12 (Conv2D)	(None, 87, 87, 64)	18,496
max_pooling2d_7 (MaxPooling2D)	(None, 43, 43, 64)	0
conv2d_13 (Conv2D)	(None, 41, 41, 128)	73,856
max_pooling2d_8 (MaxPooling2D)	(None, 20, 20, 128)	0
conv2d_14 (Conv2D)	(None, 18, 18, 256)	295,168
max_pooling2d_9 (MaxPooling2D)	(None, 9, 9, 256)	0
conv2d_15 (Conv2D)	(None, 7, 7, 512)	1,180,160
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 512)	0
dropout (Dropout)	(None, 512)	0
dense_3 (Dense)	(None, 1)	513

```
Total params: 4,707,269 (17.96 MB)
Trainable params: 1,569,089 (5.99 MB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 3,138,180 (11.97 MB)
```

Next, you'll get an input image—a picture of a cat, not part of the images the network was trained on.

Listing 10.1 Preprocessing a single image

```
import keras
import numpy as np

img_path = keras.utils.get_file(
    fname="cat.jpg", origin="https://img-datasets.s3.amazonaws.com/cat.jpg"
)

def get_img_array(img_path, target_size):
    img = keras.utils.load_img(img_path, target_size=target_size)
    array = keras.utils.img_to_array(img)
    array = np.expand_dims(array, axis=0)
    return array

img_tensor = get_img_array(img_path, target_size=(180, 180))
```

Downloads a test image

Opens the image file and resizes it

Turns the image into a float32 NumPy array of shape (180, 180, 3)

We add a dimension to transform our array into a "batch" of a single sample. Its shape is now (1, 180, 180, 3).

Let's display the picture (see figure 10.1).

Listing 10.2 Displaying the test picture

```
import matplotlib.pyplot as plt

plt.axis("off")
plt.imshow(img_tensor[0].astype("uint8"))
plt.show()
```

Figure 10.1 The test cat picture

To extract the feature maps you want to look at, you'll create a Keras model that takes batches of images as input and outputs the activations of all convolution and pooling layers.

Listing 10.3 Instantiating a model that returns layer activations

```
from keras import layers

layer_outputs = []
layer_names = []
for layer in model.layers:
    if isinstance(layer, (layers.Conv2D, layers.MaxPooling2D)):
        layer_outputs.append(layer.output)
        layer_names.append(layer.name)
activation_model = keras.Model(inputs=model.input, outputs=layer_outputs)
```

Extracts the outputs of all Conv2D and MaxPooling2D layers and put them in a list

Creates a model that will return these outputs, given the model input

Saves the layer names for later

When fed an image input, this model returns the values of the layer activations in the original model, as a list. This is the first time you've encountered a multi-output model in this book in practice since you learned about them in chapter 7: until now, the models you've seen have had exactly one input and one output. This one has one input and nine outputs—one output per layer activation.

Listing 10.4 Using the model to compute layer activations

```
activations = activation_model.predict(img_tensor)
```

Returns a list of nine NumPy arrays—one array per layer activation

For instance, this is the activation of the first convolution layer for the cat image input:

```
>>> first_layer_activation = activations[0]
>>> print(first_layer_activation.shape)
(1, 178, 178, 32)
```

It's a 178 × 178 feature map with 32 channels. Let's try plotting the sixth channel of the activation of the first layer of the original model (see figure 10.2).

Listing 10.5 Visualizing the sixth channel

```
import matplotlib.pyplot as plt

plt.matshow(first_layer_activation[0, :, :, 5], cmap="viridis")
```

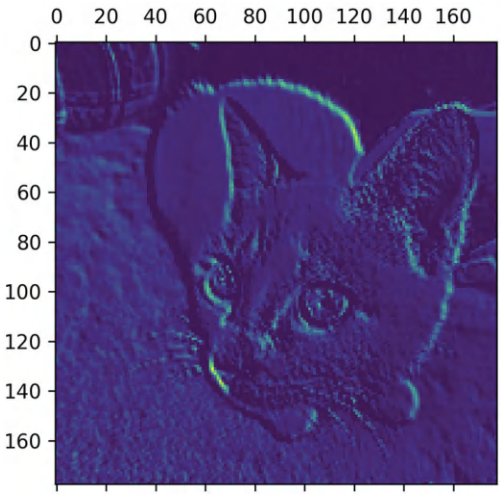


Figure 10.2 Sixth channel of the activation of the first layer on the test cat picture

This channel appears to encode a diagonal edge detector, but note that your own channels may vary because the specific filters learned by convolution layers aren't deterministic.

Now let's plot a complete visualization of all the activations in the network (see figure 10.3). We'll extract and plot every channel in each of the layer activations, and we'll stack the results in one big grid, with channels stacked side by side.

Listing 10.6 Visualizing every channel in every intermediate activation

```
images_per_row = 16
for layer_name, layer_activation in zip(layer_names, activations):
    n_features = layer_activation.shape[-1]
    size = layer_activation.shape[1]
    n_cols = n_features // images_per_row
    display_grid = np.zeros(
        ((size + 1) * n_cols - 1, images_per_row * (size + 1) - 1)
    )
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_index = col * images_per_row + row
            channel_image = layer_activation[0, :, :, channel_index].copy()
            if channel_image.sum() != 0:
                channel_image -= channel_image.mean()
                channel_image /= channel_image.std()
                channel_image *= 64
                channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype("uint8")
            display_grid[
                col * (size + 1) : (col + 1) * size + col,
                row * (size + 1) : (row + 1) * size + row,
            ] = channel_image
    scale = 1.0 / size
    plt.figure(
        figsize=(scale * display_grid.shape[1], scale * display_grid.shape[0])
    )
    plt.title(layer_name)
    plt.grid(False)
    plt.axis("off")
    plt.imshow(display_grid, aspect="auto", cmap="viridis")
```

Prepares an empty grid for displaying all the channels in this activation

The layer activation has shape (1, size, size, n_features).

Iterates over the activations (and the names of the corresponding layers)

This is a single channel (or feature).

Normalizes channel values within the [0, 255] range. All-zero channels are kept at zero.

Places the channel matrix in the empty grid we prepared

Displays the grid for the layer

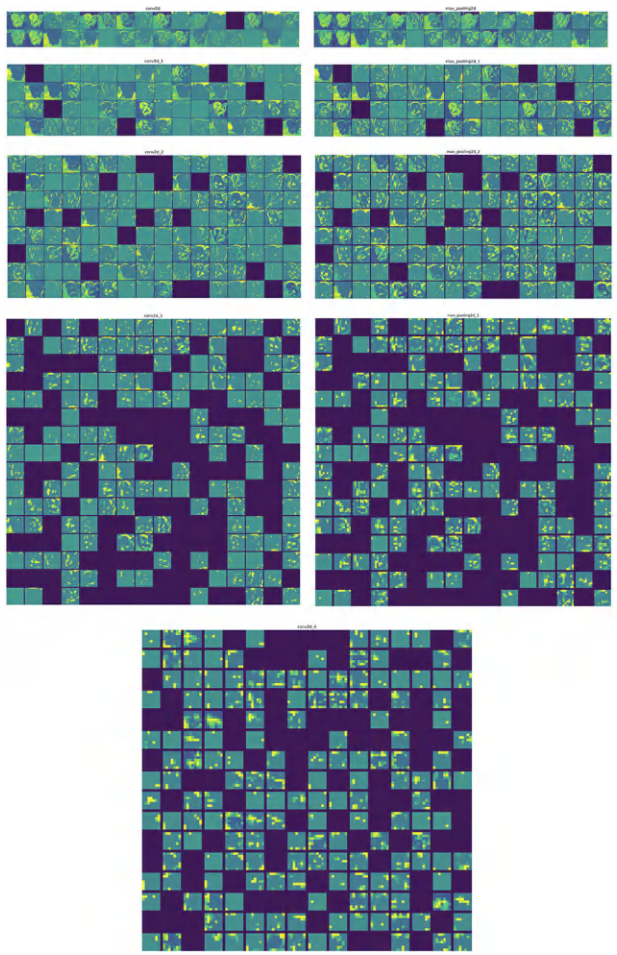


Figure 10.3 Every channel of every layer activation on the test cat picture

There are a few things to note here:

- The first layer acts as a collection of various edge detectors. At that stage, the activations retain almost all of the information present in the initial picture.
- As you go higher, the activations become increasingly abstract and less visually interpretable. They begin to encode higher-level concepts such as "cat ear" and "cat eye." Higher representations carry increasingly less information about the visual contents of the image and increasingly more information related to the class of the image.
- The sparsity of the activations increases with the depth of the layer: in the first layer, all filters are activated by the input image, but in the following layers, more and more filters are blank. This means the pattern encoded by the filter isn't found in the input image.

We have just observed an important universal characteristic of the representations learned by deep neural networks: the features extracted by a layer become increasingly abstract with the depth of the layer. The activations of higher layers carry less and less information about the specific input being seen and more and more information about the target (in this case, the class of the image: cat or dog). A deep neural network effectively acts as an *information distillation pipeline*, with raw data going in (in this case, RGB pictures) and being repeatedly transformed so that irrelevant information is filtered out (for example, the specific visual appearance of the image) and useful information is magnified and refined (for example, the class of the image).

This is analogous to the way humans and animals perceive the world: after observing a scene for a few seconds, a human can remember which abstract objects were present in it (bicycle, tree) but can't remember the specific appearance of these objects. In fact, if you tried to draw a generic bicycle from memory, chances are you couldn't get it even remotely right, even though you've seen thousands of bicycles in your lifetime (see, for example, figure 10.4). Try it right now: this effect is absolutely real. Your brain has learned to completely abstract its visual input—to transform it into high-level visual concepts while filtering out irrelevant visual details—making it tremendously difficult to remember how things around you look.

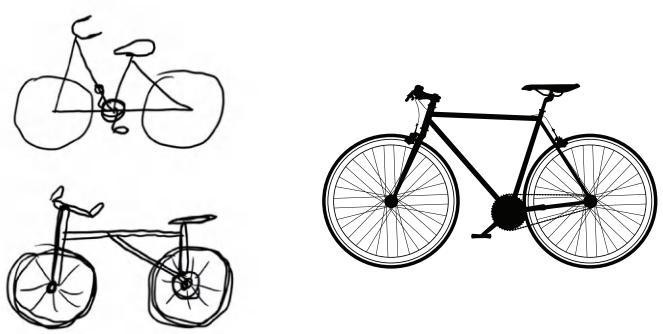


Figure 10.4 Left: Attempts to draw a bicycle from memory. Right: What a schematic bicycle should look like.

10.2 Visualizing ConvNet filters

Another easy way to inspect the filters learned by ConvNets is to display the visual pattern that each filter is meant to respond to. This can be done with *gradient ascent in input space*, applying *gradient descent* to the value of the input image of a ConvNet so as to *maximize* the response of a specific filter, starting from a blank input image. The resulting input image will be one that the chosen filter is maximally responsive to.

Let's try this with the filters of the Xception model. The process is simple: we'll build a loss function that maximizes the value of a given filter in a given convolution layer, and then we'll use stochastic gradient descent to adjust the values of the input image so as to maximize this activation value. This will be your second example of a low-level

gradient descent loop (the first one was in chapter 2). We will show it for TensorFlow, PyTorch, and Jax.

First, let's instantiate the Xception model trained on the ImageNet dataset. We can once again use the KerasHub library, exactly as we did in chapter 8.

Listing 10.7 Instantiating the Xception convolutional base

```
import keras_hub

model = keras_hub.models.Backbone.from_preset(
    "xception_41_imagenet",
)
preprocessor = keras_hub.layers.ImageConverter.from_preset(
    "xception_41_imagenet",
    image_size=(180, 180),
)
```

Instantiates the feature extractor network from pretrained weights

Loads the matching preprocessing to scale our input images

We're interested in the convolutional layers of the model—the `Conv2D` and `SeparableConv2D` layers. We'll need to know their names so we can retrieve their outputs. Let's print their names, in order of depth.

Listing 10.8 Printing the names of all convolutional layers in Xception

```
for layer in model.layers:
    if isinstance(layer, (keras.layers.Conv2D, keras.layers.SeparableConv2D)):
        print(layer.name)
```

You'll notice that the `SeparableConv2D` layers here are all named something like `block6_sepconv1`, `block7_sepconv2`, etc.—Xception is structured into blocks, each containing several convolutional layers.

Now let's create a second model that returns the output of a specific layer—a "feature extractor" model. Because our model is a Functional API model, it is inspectable: you can query the `output` of one of its layers and reuse it in a new model. No need to copy the entire Xception code.

Listing 10.9 A feature extractor model returning a specific output

```
layer_name = "block3_sepconv1"
layer = model.get_layer(name=layer_name)
feature_extractor = keras.Model(inputs=model.input, outputs=layer.output)
```

You could replace this with the name of any layer in the Xception convolutional base.

This is the layer object we're interested in.

We use model.input and layer.output to create a model that, given an input image, returns the output of our target layer.

To use this model, we can simply call it on some input data, but we should be careful to apply our model-specific image preprocessing so that our images are scaled to the same range as the Xception pretraining data.

Listing 10.10 Using the feature extractor

```
activation = feature_extractor(preprocessor(img_tensor))
```

Let's use our feature extractor model to define a function that returns a scalar value quantifying how much a given input image "activates" a given filter in the layer. This is the loss function that we'll maximize during the gradient ascent process:

```
from keras import ops

def compute_loss(image, filter_index):
    activation = feature_extractor(image)
    filter_activation = activation[:, 2:-2, 2:-2, filter_index]
    return ops.mean(filter_activation)
```

The loss function takes an image tensor and the index of the filter we consider (an integer).

We avoid border artifacts by only involving nonborder pixels in the loss: we discard the first 2 pixels along the sides of the activation.

Returns the mean of the activation values for the filter

The difference between model.predict(x) and model(x)

In the previous chapter, we used `predict(x)` for feature extraction. Here, we're using `model(x)`. What gives?

Both `y = model.predict(x)` and `y = model(x)` (where `x` is an array of input data) mean "run the model on `x` and retrieve the output `y`." Yet, they aren't exactly the same thing.

`predict()` loops over the data in batches (in fact, you can specify the batch size via `predict(x, batch_size=64)`) and extracts the NumPy value of the outputs. It's schematically equivalent to

```
def predict(x):
    y_batches = []
    for x_batch in get_batches(x):
        y_batch = model(x).numpy()
        y_batches.append(y_batch)
    return np.concatenate(y_batches)
```

This means that `predict()` calls can scale to very large arrays. Meanwhile, `model(x)` happens in-memory and doesn't scale. On the other hand, `predict()` is not differentiable: TensorFlow, PyTorch, and JAX cannot backpropagate through it.

(continued)

You should use `model(x)` when you need to retrieve the gradients of the model call. And you should use `predict()` if you just need the output value. In other words, always use `predict()`, unless you're in the middle of writing a low-level gradient descent loop (as we are now).

A non-obvious trick to help the gradient-ascent process go smoothly is to normalize the gradient tensor by dividing it by its L2 norm (the square root of the sum of the squares of the values in the tensor). This ensures that the magnitude of the updates done to the input image is always within the same range.

Let's set up the gradient ascent step function. Anything that involves gradients requires calling backend-level APIs, such as `GradientTape` in TensorFlow, `.backward()` in PyTorch, and `jax.grad()` in JAX. Let's line up all the code snippets for each of the three backends, starting with TensorFlow.

10.2.1 Gradient ascent in TensorFlow

For TensorFlow, we can just open a `GradientTape` scope and compute the loss inside of it to retrieve the gradients we need. We'll use a `@tf.function` decorator to speed up computation:

```
import keras
import keras_hub
from keras import ops

model = keras_hub.models.Backbone.from_preset("xception_41_imagenet")

layer_name = "block3_sepconv1"
layer = model.get_layer(name=layer_name)
feature_extractor = keras.Model(inputs=model.input, outputs=layer.output)

def compute_loss(image, filter_index):
    activation = feature_extractor(image)
    filter_activation = activation[:, 2:-2, 2:-2, filter_index]
    return ops.mean(filter_activation)
```

Listing 10.11 Loss maximization via stochastic gradient ascent: TensorFlow

```
import tensorflow as tf

@tf.function
def gradient_ascent_step(image, filter_index, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(image)
```

Explicitly watches the image tensor, since it isn't a TensorFlow Variable (only Variables are automatically watched in a gradient tape)

```
        loss = compute_loss(image, filter_index)
    grads = tape.gradient(loss, image)
    grads = ops.normalize(grads)
    image += learning_rate * grads
    return image
```

Computes the loss scalar, indicating how much the current image activates the filter

Computes the gradients of the loss with respect to the image

Applies the "gradient normalization trick"

Moves the image a little bit in a direction that activates our target filter more strongly

Returns the updated image, so we can run the step function in a loop

10.2.2 Gradient ascent in PyTorch

In the case of PyTorch, we use `loss.backward()` and `image.grad` to obtain the gradients of the loss with respect to the input image, like this.

Listing 10.12 Loss maximization via stochastic gradient ascent: PyTorch

```
import torch

def gradient_ascent_step(image, filter_index, learning_rate):
    image = image.clone().detach().requires_grad_(True)
    loss = compute_loss(image, filter_index)
    loss.backward()
    grads = image.grad
    grads = ops.normalize(grads)
    image = image + learning_rate * grads
    return image
```

Creates a copy of "image" that we can get gradients for

No need to reset the gradients since the image tensor is recreated at each iteration.

10.2.3 Gradient ascent in JAX

In the case of JAX, we use `jax.grad()` to obtain a function that returns the gradients of the loss with respect to the input image.

Listing 10.13 Loss maximization via stochastic gradient ascent: JAX

```
import jax

grad_fn = jax.grad(compute_loss)

@jax.jit
def gradient_ascent_step(image, filter_index, learning_rate):
    grads = grad_fn(image, filter_index)
    grads = ops.normalize(grads)
    image += learning_rate * grads
    return image
```

10.2.4 The filter visualization loop

Now you have all the pieces. Let's put them together into a Python function that takes a filter index as input and returns a tensor representing the pattern that maximizes the activation of the specified filter in our target layer.

Listing 10.14 Function to generate filter visualizations

```
img_width = 200
img_height = 200

def generate_filter_pattern(filter_index):
    iterations = 30
    learning_rate = 10.0
    image = keras.random.uniform(
        minval=0.4, maxval=0.6, shape=(1, img_width, img_height, 3)
    )
    for i in range(iterations):
        image = gradient_ascent_step(image, filter_index, learning_rate)
    return image[0]
```

Initialize an image tensor with random values. (The Xception model expects input values in the [0, 1] range, so here we pick a range centered on 0.5.)

The number of gradient ascent steps to apply

The amplitude of a single step

Repeatedly updates the values of the image tensor to maximize our loss function

The resulting image tensor is a floating-point array of shape `(200, 200, 3)`, with values that may not be integers within `[0, 255]`. Hence, you need to post-process this tensor to turn it into a displayable image. You do so with the following straightforward utility function.

Listing 10.15 Utility function to convert a tensor into a valid image

```
def deprocess_image(image):
    image -= ops.mean(image)
    image /= ops.std(image)
    image *= 64
    image += 128
    image = ops.clip(image, 0, 255)
    image = image[25:-25, 25:-25, :]
    image = ops.cast(image, dtype="uint8")
    return ops.convert_to_numpy(image)
```

Normalizes image values within the [0, 255] range

Center crop to avoid border artifacts

Let's try it (see figure 10.5):

```
>>> plt.axis("off")
>>> plt.imshow(deprocess_image(generate_filter_pattern(filter_index=2)))
```

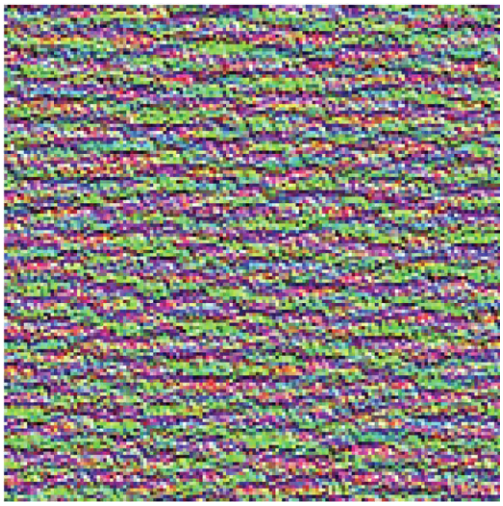


Figure 10.5 Pattern that the second channel in layer `block3_sepconv1` responds to maximally

It seems that filter 2 in layer `block3_sepconv1` is responsive to a horizontal lines pattern, somewhat water-like or fur-like.

Now the fun part: you can start visualizing every filter in the layer—and even every filter in every layer in the model (see figure 10.6).

Listing 10.16 Generating a grid of all filter response patterns

```
all_images = []
for filter_index in range(64):
    print(f"Processing filter {filter_index}")
    image = deprocess_image(generate_filter_pattern(filter_index))
    all_images.append(image)

margin = 5
n = 8
box_width = img_width - 25 * 2
box_height = img_height - 25 * 2
full_width = n * box_width + (n - 1) * margin
full_height = n * box_height + (n - 1) * margin
stitched_filters = np.zeros((full_width, full_height, 3))

for i in range(n):
    for j in range(n):
        image = all_images[i * n + j]
        stitched_filters[
            (box_width + margin) * i : (box_width + margin) * i + box_width,
            (box_height + margin) * j : (box_height + margin) * j + box_height,
            :,
        ] = image

keras.utils.save_img(f"filters_for_layer_{layer_name}.png", stitched_filters)
```

Generates and saves visualizations for the first 64 filters in the layer

Prepares a blank canvas for us to paste filter visualizations

Fills the picture with our saved filters

Saves the canvas to disk

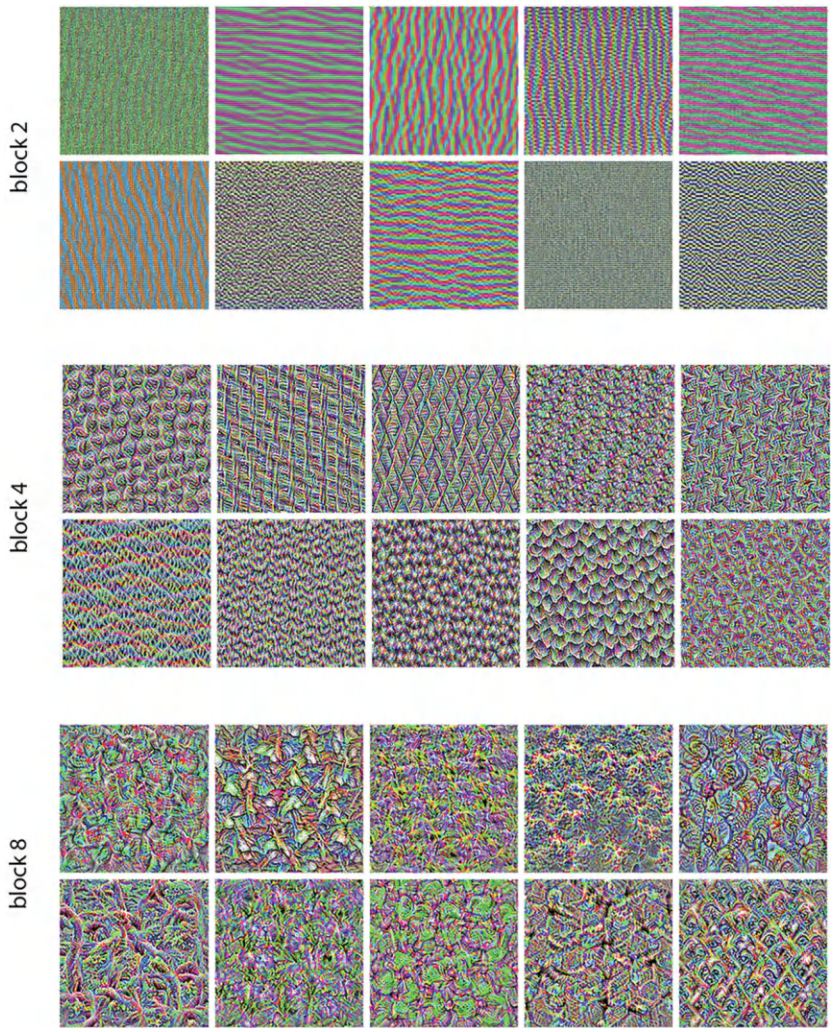


Figure 10.6 Some filter patterns for layers `block2_sepconv1`, `block4_sepconv1`, and `block8_sepconv1`

These filter visualizations tell you a lot about how ConvNet layers see the world: each layer in a ConvNet learns a collection of filters such that their inputs can be expressed as a combination of the filters. This is similar to how the Fourier transform decomposes signals onto a bank of cosine functions. The filters in these ConvNet filter banks get increasingly complex and refined as you go higher in the model:

- The filters from the first layers in the model encode simple directional edges and colors (or colored edges, in some cases).
- The filters from layers a bit further up the stack, such as `block4_sepconv1`, encode simple textures made from combinations of edges and colors.

- The filters in higher layers begin to resemble textures found in natural images: feathers, eyes, leaves, and so on.

10.3 Visualizing heatmaps of class activation

Here's one last visualization technique—one that is useful for understanding which parts of a given image led a ConvNet to its final classification decision. This is helpful for "debugging" the decision process of a ConvNet, particularly in the case of a classification mistake (a problem domain called *model interpretability*). It can also allow you to locate specific objects in an image.

This general category of techniques is called *class activation map* (CAM) visualization, and it consists of producing heatmaps of class activation over input images. A class activation heatmap is a 2D grid of scores associated with a specific output class, computed for every location in any input image, indicating how important each location is with respect to the class under consideration. For instance, given an image fed into a dogs-versus-cats ConvNet, CAM visualization would allow you to generate a heatmap for the class "cat," indicating how cat-like different parts of the image are, and also a heatmap for the class "dog," indicating how dog-like parts of the image are. The specific implementation we'll use is the one described in Selvaraju et al.[1]

Grad-CAM consists of taking the output feature map of a convolution layer, given an input image, and weighting every channel in that feature map by the gradient of the class with respect to the channel. Intuitively, one way to understand this trick is that you're weighting a spatial map of "how intensely the input image activates different channels" by "how important each channel is with regard to the class," resulting in a spatial map of "how intensely the input image activates the class."

Let's demonstrate this technique using the pretrained Xception model. Consider the image of two African elephants shown in figure 10.7, possibly a mother and her calf, strolling in the savanna. We can start by downloading this image and converting it to a NumPy array, as shown in figure 10.7.

Listing 10.17 Preprocessing an input image for Xception

Downloads the image and stores it locally under the path img_path

```
img_path = keras.utils.get_file(
    fname="elephant.jpg",
    origin="https://img-datasets.s3.amazonaws.com/elephant.jpg",
)
img = keras.utils.load_img(img_path)
img_array = np.expand_dims(img, axis=0)
```

Returns a Python Imaging Library (PIL) image

Converts to NumPy and adds a batch dimension

[1] Ramprasaath R. Selvaraju, et al., "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization," arxiv (2019), https://arxiv.org/abs/1610.02391.

Figure 10.7 Test picture of African elephants

So far, we have only used KerasHub to instantiate a pretrained feature extractor network using the backbone class. For Grad-CAM, we need the entire Xception model including the classification head—recall that Xception was trained on the ImageNet dataset with ~1 million labeled images belonging to 1,000 different classes.

KerasHub provides a high-level *task* API for common end-to-end workflows like image classification, text classification, image generation, and so on. A task wraps preprocessing, a feature extraction network, and a task-specific head into a single class that is easy to use. Let's try it out:

```
>>> model = keras_hub.models.ImageClassifier.from_preset(
...     "xception_41_imagenet",
...     activation="softmax",
... )
>>> preds = model.predict(img_array)
>>> preds.shape
(1, 1000)
>>> keras_hub.utils.decode_imagenet_predictions(preds)
[[("African_elephant", 0.90331),
  ("tusker", 0.05487),
  ("Indian_elephant", 0.01637),
```

We can configure the final activation of the classifier. Here, we use a softmax activation so our outputs are probabilities.

ImageNet has 1,000 classes, so each prediction from our classifier has 1,000 entries.

```
    ("triceratops", 0.00029),
    ("Mexican_hairless", 0.00018)]]
```

The top five classes predicted for this image are as follows:

- African elephant (with 90% probability)
- Tusker (with 5% probability)
- Indian elephant (with 2% probability)
- Triceratops and Mexican hairless dog with less than 0.1% probability

The network has recognized the image as containing an undetermined quantity of African elephants. The entry in the prediction vector that was maximally activated is the one corresponding to the "African elephant" class, at index 386:

```
>>> np.argmax(preds[0])
386
```

To visualize which parts of the image are the most African elephant–like, let's set up the Grad-CAM process.

You will note that we didn't need to preprocess our image before calling the task model. That's because the KerasHub `ImageClassifier` is preprocessing inputs for us as part of `predict()`. Let's preprocess the image ourselves so we can use the preprocessed inputs directly:

```
img_array = model.preprocessor(img_array)
```

KerasHub tasks like ImageClassifier have a preprocessor layer.

First, we create a model that maps the input image to the activations of the last convolutional layer.

Listing 10.18 Returning the last convolutional output

```
last_conv_layer_name = "block14_sepconv2_act"
last_conv_layer = model.backbone.get_layer(last_conv_layer_name)
last_conv_layer_model = keras.Model(model.inputs, last_conv_layer.output)
```

Second, we create a model that maps the activations of the last convolutional layer to the final class predictions.

Listing 10.19 Going from the last convolutional output to final predictions

```
classifier_input = last_conv_layer.output
x = classifier_input
```

```
for layer_name in ["pooler", "predictions"]:
    x = model.get_layer(layer_name)(x)
classifier_model = keras.Model(classifier_input, x)
```

Then, we compute the gradient of the top predicted class for our input image with respect to the activations of the last convolution layer. Once again, having to compute gradients means we have to use backend APIs.

10.3.1 Getting the gradient of the top class: TensorFlow version

Let's start with the TensorFlow version, once again using `GradientTape`.

Listing 10.20 Computing the top class gradients with TensorFlow

```
import tensorflow as tf

def get_top_class_gradients(img_array):
    last_conv_layer_output = last_conv_layer_model(img_array)
    with tf.GradientTape() as tape:
        tape.watch(last_conv_layer_output)
        preds = classifier_model(last_conv_layer_output)
        top_pred_index = ops.argmax(preds[0])
        top_class_channel = preds[:, top_pred_index]

    grads = tape.gradient(top_class_channel, last_conv_layer_output)
    return grads, last_conv_layer_output

grads, last_conv_layer_output = get_top_class_gradients(img_array)
grads = ops.convert_to_numpy(grads)
last_conv_layer_output = ops.convert_to_numpy(last_conv_layer_output)
```

Retrieves the activation channel corresponding to the top predicted class

Computes activations of the last conv layer and makes the tape watch it

Gets the gradient of the top predicted class with regard to the output feature map of the last convolutional layer

10.3.2 Getting the gradient of the top class: PyTorch version

Next, here's the PyTorch version, using `.backward()` and `.grad`.

Listing 10.21 Computing the top class gradients with PyTorch

```
def get_top_class_gradients(img_array):
    last_conv_layer_output = last_conv_layer_model(img_array)
```

Computes activations of the last conv layer

```
    last_conv_layer_output = (
        last_conv_layer_output.clone().detach().requires_grad_(True)
    )
    preds = classifier_model(last_conv_layer_output)
    top_pred_index = ops.argmax(preds[0])
    top_class_channel = preds[:, top_pred_index]
    top_class_channel.backward()
    grads = last_conv_layer_output.grad
    return grads, last_conv_layer_output

grads, last_conv_layer_output = get_top_class_gradients(img_array)
grads = ops.convert_to_numpy(grads)
last_conv_layer_output = ops.convert_to_numpy(last_conv_layer_output)
```

Creates a copy of last_conv_layer_output that we can get gradients for

Gets the gradient of the top predicted class with regard to the output feature map of the last convolutional layer

Retrieves the activation channel corresponding to the top predicted class

10.3.3 Getting the gradient of the top class: JAX version

Finally, let's do JAX. We define a separate loss computation function that takes the final layer's output and returns the activation channel corresponding to the top predicted class. We use this activation value as our loss, allowing us to compute the gradient.

Listing 10.22 Computing the top class gradients with Jax

```
import jax

def loss_fn(last_conv_layer_output):
    preds = classifier_model(last_conv_layer_output)
    top_pred_index = ops.argmax(preds[0])
    top_class_channel = preds[:, top_pred_index]
    return top_class_channel[0]

grad_fn = jax.grad(loss_fn)

def get_top_class_gradients(img_array):
    last_conv_layer_output = last_conv_layer_model(img_array)
    grads = grad_fn(last_conv_layer_output)
    return grads, last_conv_layer_output

grads, last_conv_layer_output = get_top_class_gradients(img_array)
grads = ops.convert_to_numpy(grads)
last_conv_layer_output = ops.convert_to_numpy(last_conv_layer_output)
```

Defines a separate loss function

Returns the activation value of the top-class channel

Creates a gradient function

Now retrieving the gradient of the top-class channel is just a matter of calling the gradient function!

10.3.4 Displaying the class activation heatmap

Now, we apply pooling and importance weighting to the gradient tensor to obtain our heatmap of class activation.

Listing 10.23 Gradient pooling and channel importance weighting

```
pooled_grads = np.mean(grads, axis=(0, 1, 2))
last_conv_layer_output = last_conv_layer_output[0].copy()
for i in range(pooled_grads.shape[-1]):
    last_conv_layer_output[:, :, i] *= pooled_grads[i]
heatmap = np.mean(last_conv_layer_output, axis=-1)
```

This is a vector where each entry is the mean intensity of the gradient for a given channel. It quantifies the importance of each channel with regard to the top predicted class.

Multiplies each channel in the output of the last convolutional layer by how important this channel is

The channel-wise mean of the resulting feature map is our heatmap of class activation.

For visualization purposes, you'll also normalize the heatmap between 0 and 1. The result is shown in figure 10.8.

Listing 10.24 Heatmap post-processing

```
heatmap = np.maximum(heatmap, 0)
heatmap /= np.max(heatmap)
plt.matshow(heatmap)
```

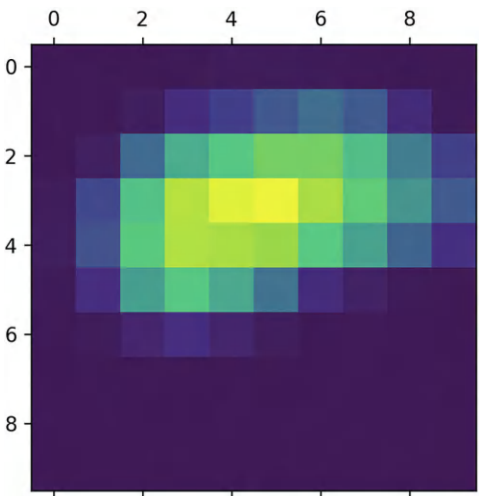


Figure 10.8 Standalone class activation heatmap

Finally, let's generate an image that superimposes the original image on the heatmap you just obtained (see figure 10.9).

Figure 10.9 African elephant class activation heatmap over the test picture

Listing 10.25 Superimposing the heatmap with the original picture

```
import matplotlib.cm as cm

img = keras.utils.load_img(img_path)
img = keras.utils.img_to_array(img)

heatmap = np.uint8(255 * heatmap)

jet = cm.get_cmap("jet")
jet_colors = jet(np.arange(256))[:, :3]
jet_heatmap = jet_colors[heatmap]

jet_heatmap = keras.utils.array_to_img(jet_heatmap)
jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0]))
jet_heatmap = keras.utils.img_to_array(jet_heatmap)
```

Loads the original image

Rescales the heatmap to the range 0–255

Uses the "jet" colormap to recolorize the heatmap

Creates an image that contains the recolorized heatmap

```
superimposed_img = jet_heatmap * 0.4 + img
superimposed_img = keras.utils.array_to_img(superimposed_img)

save_path = "elephant_cam.jpg"
superimposed_img.save(save_path)
```

Saves the superimposed image

Superimposes the heatmap and the original image, with the heatmap at 40% opacity

This visualization technique answers two important questions:

- Why did the network think this image contained an African elephant?
- Where is the African elephant located in the picture?

In particular, it's interesting to note that the ears of the elephant calf are strongly activated: this is probably how the network can tell the difference between African and Indian elephants.

10.4 Visualizing the latent space of a ConvNet

I'll introduce one last model inspection technique: latent space visualization. All deep learning models work by embedding their inputs in a *latent space*, a manifold (or curve) where points that are spatially close to each other represent semantically similar inputs. On this manifold, different classes of inputs become nicely separable, which is what enables classification.

Can we visualize in two dimensions what this manifold looks like?

The general idea is to pick a certain layer in the model, typically the one right before the classification layer, and record its activations for a set of inputs. The activations for each input represent the coordinates of the corresponding point on the latent manifold. They're points on a curve. To visualize that curve—the latent manifold—all we need to do is to plot these points.

Note that this process can be done with any layer in your model—each layer encodes a different latent space, and these latent spaces get gradually more semantically organized as your go up the model. Remember our analogy about how deep learning models are machines for uncrumpling paper balls? The latent space of each layer is a snapshot of the paper ball at different stages of its uncrumpling. This is why it's most interesting to target the second-before-last layer, when the paper ball is almost fully flattened into a nice sheet.

Now there's just one obstacle. The human visual system is very much limited to low-dimensional objects. We can interpret 2D, 3D, maybe 4D visualizations. We're generally happiest with simple 2D spaces. Meanwhile, the point coordinates we'd like to plot are in a space that has typically hundreds of dimensions, and possibly thousands. To visualize them, we're going to need to project them on a 2D surface, while trying to retain as much of the organization of the original manifold as feasible.

This task is not as impossible as it may sound. You see, while the points you're working with may be, say, 2048-dimensional, the curve they represent has a much lower intrinsic dimensionality. That's precisely why we refer to it as a *manifold*—a manifold is a lower-dimensional curve inside a higher-dimensional space. For instance, while planet Earth is a three-dimensional object, its surface is actually 2D, and so it possible to project it on a 2D plane. That's what world maps are all about.

In fact, world maps illustrate both the feasibility and the challenges of lower-dimensional projections of higher-dimensional manifolds. All projections distort reality. The widely used Mercator map projection distorts distances, in particular by making areas near the equator look smaller than they actually are, and inversely by making areas near the poles stretch towards infinity. And that's even though the surface you're projecting is in fact 2D—whereas deep learning latent manifolds may be much higher-dimensional than that.

There's no projection that won't destroy at least some information.

Summary

- ConvNets process images by applying a set of learned filters. Filters from earlier layers detect edges and basic textures, while filters from later layers detect increasingly abstract concepts.
- You can visualize both the pattern that a filter detects and a filter's response map across an image.
- You can use the Grad-CAM technique to visualize what area(s) in an image were responsible for a classifier's decision.
- Together, these techniques make ConvNets highly interpretable.

11 Image segmentation

This chapter covers

- The different branches of computer vision: image classification, image segmentation, and object detection
- Building a segmentation model from scratch
- Using the pretrained Segment Anything Model

Chapter 8 gave you a first introduction to deep learning for computer vision via a simple use case: binary image classification. But there's more to computer vision than image classification! This chapter dives deeper into another essential computer vision application—image segmentation.

11.1 Computer vision tasks

So far, we've focused on image classification models: an image goes in, a label comes out. "This image likely contains a cat; this other one likely contains a dog." But image classification is only one of several possible applications of deep learning in computer vision. In general, there are three essential computer vision tasks you need to know about:

- *Image classification*, where the goal is to assign one or more labels to an image. It may be either single-label classification (meaning categories are mutually exclusive) or multilabel classification (tagging all categories that an image belongs to, as shown in figure 11.1). For example, when you search for a keyword on the Google Photos app, behind the scenes you're querying a very large multilabel classification model—one with over 20,000 different classes, trained on millions of images.
- *Image segmentation*, where the goal is to "segment" or "partition" an image into different areas, with each area usually representing a category (as shown in figure 11.1). For instance, when Zoom or Google Meet displays a custom background behind you in a video call, it's using an image segmentation model to distinguish your face from what's behind it, with pixel-level precision.
- *Object detection*, where the goal is to draw rectangles (called *bounding boxes*) around objects of interest in an image and associate each rectangle with a class. A self-driving car could use an object detection model to monitor cars, pedestrians, and signs in view of its cameras, for instance.

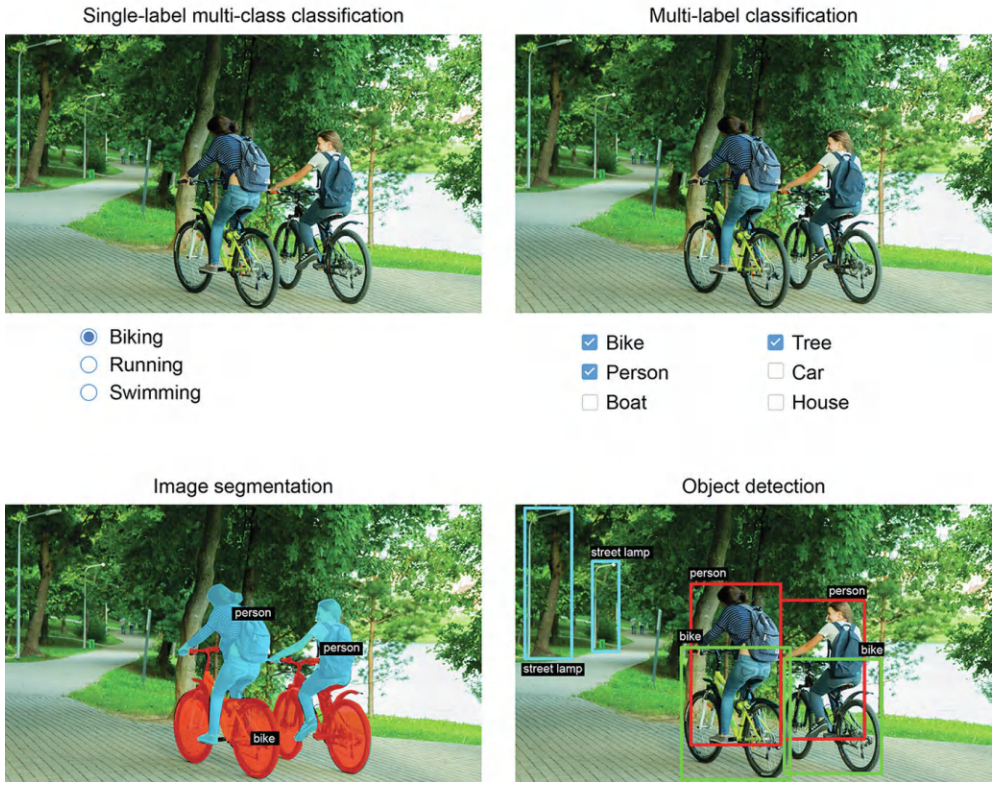


Figure 11.1 The three main computer vision tasks: classification, segmentation, and detection

Deep learning for computer vision also encompasses a number of somewhat more niche tasks besides these three, such as image similarity scoring (estimating how visually similar two images are), keypoint detection (pinpointing attributes of interest in an image, such as facial features), pose estimation, 3D mesh estimation, depth estimation, and so on. But to start with, image classification, image segmentation, and object detection form the foundation that every machine learning engineer should be familiar with. Almost all computer vision applications boil down to one of these three.

You've seen image classification in action in chapter 8. Next, let's dive into image segmentation. It's a very useful and very versatile technique, and you can straightforwardly approach it with what you've already learned so far. Then, in the next chapter, you'll learn about object detection in detail.

11.1.1 Types of image segmentation

Image segmentation with deep learning is about using a model to assign a class to each pixel in an image, thus *segmenting* the image into different zones (such as "background" and "foreground" or "road," "car," and "sidewalk"). This general category of techniques can be used to power a considerable variety of valuable applications in image and video editing, autonomous driving, robotics, medical imaging, and so on.

There are three different flavors of image segmentation that you should know about:

- *Semantic segmentation*, where each pixel is independently classified into a semantic category, like "cat." If there are two cats in the image, the corresponding pixels are all mapped to the same generic "cat" category (see figure 11.2).
- *Instance segmentation*, which seeks to parse out individual object instances. In an image with two cats in it, instance segmentation would distinguish between pixels belonging to "cat 1" and pixels belonging to "cat 2" (see figure 11.2).
- *Panoptic segmentation*, which combines semantic segmentation and instance segmentation by assigning to each pixel in an image both a semantic label (like "cat") and an instance label (like "cat 2"). This is the most informative of all three segmentation types.

Figure 11.2 Semantic segmentation vs. instance segmentation

To get more familiar with segmentation, let's get started with training a small segmentation model from scratch on your own data.

11.2 Training a segmentation model from scratch

In this first example, we'll focus on semantic segmentation. We'll be looking once again at images of cats and dogs, and this time we'll be learning to tell apart the main subject and its background.

11.2.1 Downloading a segmentation dataset

We'll work with the Oxford-IIIT Pets dataset (https://www.robots.ox.ac.uk/~vgg/data/pets/), which contains 7,390 pictures of various breeds of cats and dogs, together with foreground-background *segmentation masks* for each picture. A segmentation mask is the image segmentation equivalent of a label: it's an image the same size as the input image, with a single color channel where each integer value corresponds to the class of the corresponding pixel in the input image. In our case, the pixels of our segmentation masks can take one of three integer values:

- 1 (foreground)
- 2 (background)
- 3 (contour)

Let's start by downloading and uncompressing our dataset, using the `wget` and `tar` shell utilities:

```
!wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
!wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
!tar -xf images.tar.gz
!tar -xf annotations.tar.gz
```

The input pictures are stored as JPG files in the `images/` folder (such as `images/Abyssinian_1.jpg`), and the corresponding segmentation mask is stored as a PNG file with the same name in the `annotations/trimaps/` folder (such as `annotations/trimaps/Abyssinian_1.png`).

Let's prepare the list of input file paths, as well as the list of the corresponding mask file paths:

```
import pathlib

input_dir = pathlib.Path("images")
target_dir = pathlib.Path("annotations/trimaps")

input_img_paths = sorted(input_dir.glob("*.jpg"))
target_paths = sorted(target_dir.glob("[!.]*.png"))
```

Ignores some spurious files in the trimaps directory that start with a "."

Now, what does one of these inputs and its mask look like? Let's take a quick look (see figure 11.3).

```
import matplotlib.pyplot as plt
from keras.utils import load_img, img_to_array, array_to_img

plt.axis("off")
plt.imshow(load_img(input_img_paths[9]))
```

Displays input image number 9

Figure 11.3
An example image

Let's look at its target mask as well (see figure 11.4):

The original labels are 1, 2, and 3. We subtract 1 so that the labels range from 0 to 2, and then we multiply by 127 so that the labels become 0 (black), 127 (gray), 254 (near-white).

```
def display_target(target_array):
    normalized_array = (target_array.astype("uint8") - 1) * 127
    plt.axis("off")
    plt.imshow(normalized_array[:, :, 0])

img = img_to_array(load_img(target_paths[9], color_mode="grayscale"))
display_target(img)
```

We use color_mode='grayscale' so that the image we load is treated as having a single color channel.

Figure 11.4 The corresponding target mask

Next, let's load our inputs and targets into two NumPy arrays. Since the dataset is very small, we can load everything into memory:

```
import numpy as np
import random

img_size = (200, 200)
num_imgs = len(input_img_paths)

random.Random(1337).shuffle(input_img_paths)
random.Random(1337).shuffle(target_paths)

def path_to_input_image(path):
    return img_to_array(load_img(path, target_size=img_size))

def path_to_target(path):
    img = img_to_array(
        load_img(path, target_size=img_size, color_mode="grayscale")
    )
    img = img.astype("uint8") - 1
    return img
```

Shuffles the file paths (they were originally sorted by breed). We use the same seed (1337) in both statements to ensure that the input paths and target paths stay in the same order.

We resize everything to 200 x 200 for this example.

Total number of samples in the data

Subtracts 1 so that our labels become 0, 1, and 2

```
input_imgs = np.zeros((num_imgs,) + img_size + (3,), dtype="float32")
targets = np.zeros((num_imgs,) + img_size + (1,), dtype="uint8")
for i in range(num_imgs):
    input_imgs[i] = path_to_input_image(input_img_paths[i])
    targets[i] = path_to_target(target_paths[i])
```

Loads all images in the input_imgs float32 array and their masks in the targets uint8 array (same order). The inputs have three channels (RGB values), and the targets have a single channel (which contains integer labels).

As always, let's split the arrays into a training and a validation set:

```
num_val_samples = 1000
train_input_imgs = input_imgs[:-num_val_samples]
train_targets = targets[:-num_val_samples]
val_input_imgs = input_imgs[-num_val_samples:]
val_targets = targets[-num_val_samples:]
```

Reserves 1,000 samples for validation

Splits the data into a training and a validation set

11.2.2 Building and training the segmentation model

Now, it's time to define our model:

We use padding="same" everywhere to avoid the influence of border padding on feature map size.

```
import keras
from keras.layers import Rescaling, Conv2D, Conv2DTranspose

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = Rescaling(1.0 / 255)(inputs)

    x = Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = Conv2D(64, 3, activation="relu", padding="same")(x)
    x = Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = Conv2D(128, 3, activation="relu", padding="same")(x)
    x = Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = Conv2D(256, 3, activation="relu", padding="same")(x)

    x = Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = Conv2DTranspose(256, 3, strides=2, activation="relu", padding="same")(x)
    x = Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = Conv2DTranspose(128, 3, strides=2, activation="relu", padding="same")(x)
    x = Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = Conv2DTranspose(64, 3, strides=2, activation="relu", padding="same")(x)
```

Don't forget to rescale input images to the [0–1] range.

```
    outputs = Conv2D(num_classes, 3, activation="softmax", padding="same")(x)
    return keras.Model(inputs, outputs)
model = get_model(img_size=img_size, num_classes=3)
```

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

The first half of the model closely resembles the kind of ConvNet you'd use for image classification: a stack of `Conv2D` layers, with gradually increasing filter sizes. We downsample our images three times by a factor of two each—ending up with activations of size `(25, 25, 256)`. The purpose of this first half is to encode the images into smaller feature maps, where each spatial location (or "pixel") contains information about a large spatial chunk of the original image. You can understand it as a kind of compression.

One important difference between the first half of this model and the classification models you've seen before is the way we do downsampling: in the classification ConvNets from chapter 8, we used `MaxPooling2D` layers to downsample feature maps. Here, we downsample by adding *strides* to every other convolution layer (if you don't remember the details of how convolution strides work, see chapter 8, section 8.1.1). We do this because, in the case of image segmentation, we care a lot about the spatial location of information in the image since we need to produce per-pixel target masks as output of the model. When you do 2 × 2 max pooling, you are completely destroying location information within each pooling window: you return one scalar value per window, with zero knowledge of which of the four locations in the windows the value came from.

So, while max pooling layers perform well for classification tasks, they would hurt us quite a bit for a segmentation task. Meanwhile, strided convolutions do a better job at downsampling feature maps while retaining location information. Throughout this book, you'll notice that we tend to use strides instead of max pooling in any model that cares about feature location, such as the generative models in chapter 17.

The second half of the model is a stack of `Conv2DTranspose` layers. What are those? Well, the output of the first half of the model is a feature map of shape `(25, 25, 256)`, but we want our final output to predict a class for each pixel, matching the original spatial dimensions. The final model output will have shape `(200, 200, num_classes)`, which is `(200, 200, 3)` here. Therefore, we need to apply a kind of *inverse* of the transformations we've applied so far, something that will *upsample* the feature maps instead of downsampling them. That's the purpose of the `Conv2DTranspose` layer: you can think of it as a kind of convolution layer that *learns to upsample*. If you have an input of shape `(100, 100, 64)` and you run it through the layer `Conv2D(128, 3, strides=2, padding="same")`, you get an output of shape `(50, 50, 128)`. If you run this output through the layer `Conv2DTranspose(64, 3, strides=2, padding="same")`, you get back an output of shape `(100, 100, 64)`, the same as the original. So after compressing our inputs into feature maps of shape `(25, 25, 256)` via a stack of `Conv2D` layers, we can

simply apply the corresponding sequence of `Conv2DTranspose` layers followed by a final `Conv2D` layer to produce outputs of shape `(200, 200, 3)`.

To evaluate the model, we'll use a metric named *Intersection over Union* (IoU). It's a measure of the match between the ground truth segmentation masks and the predicted masks. It can be computed separately for each class or averaged over multiple classes. Here's how it works:

1. Compute the *intersection* between the masks, the area where the prediction and ground truth overlap.
2. Compute the *union* of the masks, the total area covered by both masks combined. This is the whole space we're interested in—the target object and any extra bits your model might have included by mistake.
3. Divide the intersection area by the union area to get the IoU. It's a number between 0 and 1, where 1 denotes a perfect match, and 0 denotes a complete miss.

We can simply use a built-in Keras metric rather than building this ourselves:

```
foreground_iou = keras.metrics.IoU(
    num_classes=3,
    target_class_ids=(0,),
    name="foreground_iou",
    sparse_y_true=True,
    sparse_y_pred=False,
)
```

Specifies the total number of classes

Specifies the class to compute IoU for (0 = foreground)

Our targets are sparse (integer class IDs).

But our model's predictions are a dense softmax!

We can now compile and fit our model:

```
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=[foreground_iou],
)
callbacks = [
    keras.callbacks.ModelCheckpoint(
        "oxford_segmentation.keras",
        save_best_only=True,
    ),
]
history = model.fit(
    train_input_imgs,
    train_targets,
    epochs=50,
    callbacks=callbacks,
```

```
    batch_size=64,
    validation_data=(val_input_imgs, val_targets),
)
```

Let's display our training and validation loss (see figure 11.5):

```
epochs = range(1, len(history.history["loss"]) + 1)
loss = history.history["loss"]
val_loss = history.history["val_loss"]
plt.figure()
plt.plot(epochs, loss, "r--", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.legend()
```

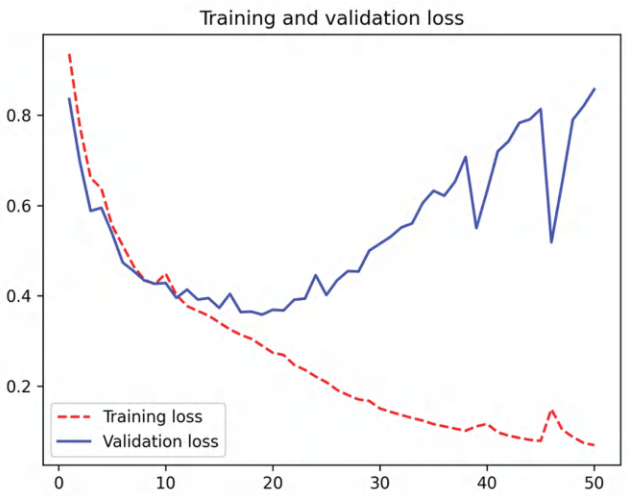


Figure 11.5 Displaying training and validation loss curves

You can see that we start overfitting midway, around epoch 25. Let's reload our best-performing model according to validation loss and demonstrate how to use it to predict a segmentation mask (see figure 11.6):

```
model = keras.models.load_model("oxford_segmentation.keras")

i = 4
test_image = val_input_imgs[i]
plt.axis("off")
plt.imshow(array_to_img(test_image))
```

```
mask = model.predict(np.expand_dims(test_image, 0))[0]

def display_mask(pred):
    mask = np.argmax(pred, axis=-1)
    mask *= 127
    plt.axis("off")
    plt.imshow(mask)

display_mask(mask)
```

Utility to display a model's prediction

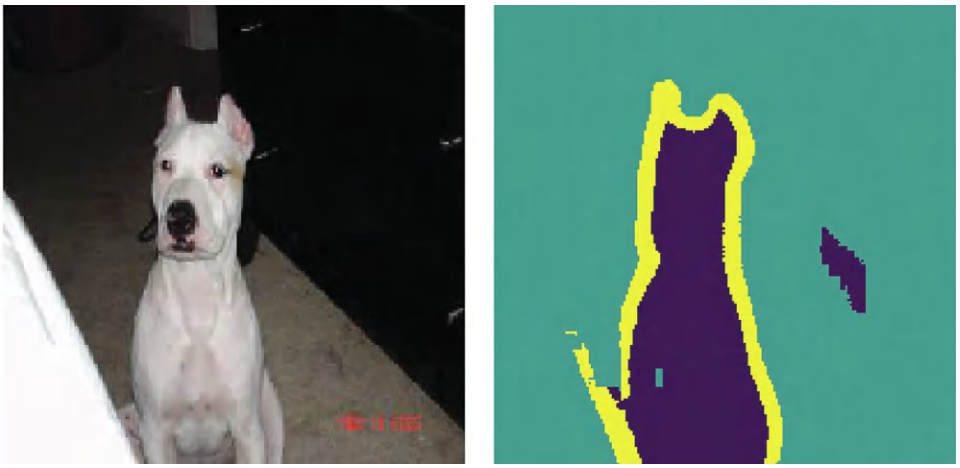


Figure 11.6 A test image and its predicted segmentation mask

There are a couple of small artifacts in our predicted mask, caused by geometric shapes in the foreground and background. Nevertheless, our model appears to work nicely.

11.3 Using a pretrained segmentation model

In the image classification example from chapter 8, you saw how using a pretrained model could significantly boost your accuracy—especially when you only have a few samples to train on. Image segmentation is no different.

The *Segment Anything Model*,[1] or SAM for short, is a powerful pretrained segmentation model you can use for, well, almost anything. It was developed by Meta AI and released in April 2023. It was trained on 11 million images and their segmentation masks, covering over 1 billion object instances. This massive amount of training data

[1] Kirillov et al., "Segment Anything," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, arXiv (2023), https://arxiv.org/abs/2304.02643.

provides the model with built-in knowledge of virtually any object that appears in natural images.

The main innovation of SAM is that it's not limited to a predefined set of object classes. You can use it for segmenting new objects simply by providing an example of what you're looking for. You don't even need to fine-tune the model first. Let's see how that works.

11.3.1 Downloading the Segment Anything Model

First, let's instantiate SAM and download its weights. Once again, we can use the KerasHub package to use this pretrained model without needing to implement it ourselves from scratch.

Remember the `ImageClassifier` task we used in the previous chapter? We can use another KerasHub task `ImageSegmenter` for wrapping pretrained image segmentation models into a high-level model with standard inputs and outputs. Here, we'll use the `sam_huge_sa1b` pretrained model, where `sam` stands for the model, `huge` refers to the number of parameters in the model, and `sa1b` stands for the SA-1B dataset released along with the model, with 1 billion annotated masks. Let's download it now:

```
import keras_hub

model = keras_hub.models.ImageSegmenter.from_preset("sam_huge_sa1b")
```

One thing we can note off the bat is that our model is, indeed, huge:

```
>>> model.count_params()
641090864
```

At 641 million parameters, SAM is the largest model we have used so far in this book. The trend of pretrained models getting larger and larger and using more and more data will be discussed in more detail in chapter 16.

11.3.2 How Segment Anything works

Before we try running some segmentation with the model, let's talk a little more about how SAM works. Much of the capability of the model comes from the scale of the pretraining dataset. Meta developed the SA-1B dataset along with the model, where the partially trained model was used to assist with the data labeling process. That is, the dataset and model were developed together in a feedback loop of sorts.

The goal with the SA-1B dataset is to create fully segmented images, where every object in an image is given a unique segmentation mask. See figure 11.7 as an example. Each image in the dataset has ~100 masks on average, and some images have over 500 individually masked objects. This was done through a pipeline of increasingly automated data collection. At first, human experts manually segmented a small example

dataset of images, which was used to train an initial model. This model was used to help drive a semiautomated stage of data collection, where images were first segmented by SAM and improved by human correction and further annotation.

Figure 11.7 An example image from the SA-1B dataset

The model is trained on `(image, prompt, mask)` triples. `image` and `prompt` are the model inputs. The image can be any input image, and the prompt can take a couple of forms:

- A point inside the object to mask
- A box around the object to mask

Given the `image` and `prompt` input, the model is expected to produce an accurate predicted mask for the object indicated by the prompt, which is compared with a ground truth `mask` label.

The model consists of a few separate components. An image encoder, similar to the Xception model we used in previous chapters, will take an input image and output a much smaller image embedding. This is something we already know how to build.

Next, we add a prompt encoder, which is responsible for mapping prompts in any of the previously mentioned forms to an embedded vector, and a mask decoder, which takes in both the image embedding and prompt embedding and outputs a few possible predicted masks. We won't get into the details of the prompt encoder and mask decoder here, as they use some modeling techniques we won't see until later chapters. We can compare these predicted masks with our ground truth mask much like we did in the earlier section of this chapter (see figure 11.8).

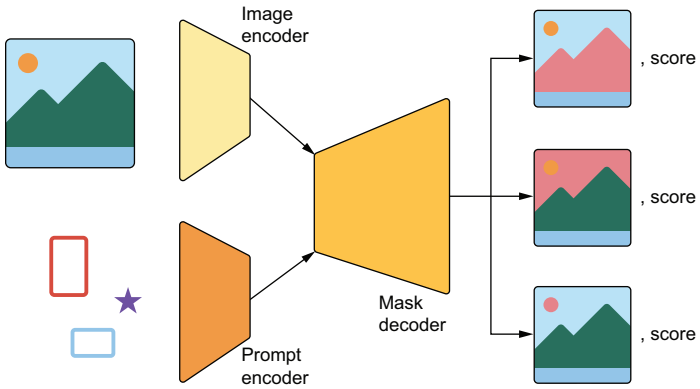


Figure 11.8 The Segment Anything high-level architecture overview

All of these subcomponents are trained simultaneously by forming batches of new `(image, prompt, mask)` triples to train on from the SA-1B image and mask data. The process here is actually quite simple. For a given input image, choose a random mask in the input. Next, randomly choose whether to create a box prompt or a point prompt. To create a point prompt, choose a random pixel inside the mask label. To create a box prompt, draw a box around all points inside the mask label. We can repeat this process indefinitely, sampling a number of `(image, prompt, mask)` triples from each image input.

11.3.3 Preparing a test image

Let's make this a little more concrete by trying the model out. We can start by loading a test image for our segmentation work. We'll use a picture of a bowl of fruits (see figure 11.9):

```
path = keras.utils.get_file(
    origin="https://s3.amazonaws.com/keras.io/img/book/fruits.jpg"
)
pil_image = keras.utils.load_img(path)
image_array = keras.utils.img_to_array(pil_image)

plt.imshow(image_array.astype("uint8"))
plt.axis("off")
plt.show()
```

Downloads the image and returns the local file path

Loads the image as a Python Imaging Library (PIL) object

Turns the PIL object into a NumPy matrix

Displays the NumPy matrix

Figure 11.9 Our test image

SAM expects inputs that are 1024 × 1024. However, forcibly resizing arbitrary images to 1024 × 1024 would distort their aspect ratio—for instance, our image isn't square. It's better to first resize the image so that its longest side becomes 1,024 pixels long and then pad the remaining pixels with a filler value, such as 0. We can achieve this with the `pad_to_aspect_ratio` argument in the `keras.ops.image.resize()` operation, like this:

```
from keras import ops

image_size = (1024, 1024)

def resize_and_pad(x):
    return ops.image.resize(x, image_size, pad_to_aspect_ratio=True)

image = resize_and_pad(image_array)
```

Next, let's define a few utilities that will come in handy when using the model. We're going to need to

- Display images.
- Display segmentation masks overlaid on an image.
- Highlight specific points on an image.
- Display boxes overlaid on an image.

All our utilities take a Matplotlib `axis` object (noted `ax`) so that they can all write to the same figure:

```
import matplotlib.pyplot as plt
from keras import ops

def show_image(image, ax):
    ax.imshow(ops.convert_to_numpy(image).astype("uint8"))

def show_mask(mask, ax):
    color = np.array([30 / 255, 144 / 255, 255 / 255, 0.6])
    h, w, _ = mask.shape
    mask_image = mask.reshape(h, w, 1) * color.reshape(1, 1, -1)
    ax.imshow(mask_image)

def show_points(points, ax):
    x, y = points[:, 0], points[:, 1]
    ax.scatter(x, y, c="green", marker="*", s=375, ec="white", lw=1.25)

def show_box(box, ax):
    box = box.reshape(-1)
    x0, y0 = box[0], box[1]
    w, h = box[2] - box[0], box[3] - box[1]
    ax.add_patch(plt.Rectangle((x0, y0), w, h, ec="red", fc="none", lw=2))
```

11.3.4 Prompting the model with a target point

To use SAM, you need to prompt it. This means we need one of the following:

- *Point prompts*—Select a point in an image and let the model segment the object that the point belongs to.
- *Box prompts*—Draw an approximate box around an object (it does not need to be particularly precise) and let the model segment the object in the box.

Let's start with a point prompt. Points are labeled, with 1 indicating the foreground (the object you want to segment) and 0 indicating the background (everything around the object). In ambiguous cases, to improve your results, you could pass multiple labeled points, instead of a single point, to refine your definition of what should be included (points labeled 1) and what should be excluded (points labeled 0).

We try a single foreground point (see figure 11.10). Here's a test point:

```
import numpy as np

input_point = np.array([[580, 450]])
input_label = np.array([1])

plt.figure(figsize=(10, 10))
show_image(image, plt.gca())
show_points(input_point, plt.gca())
plt.show()
```

Coordinates of our point

1 means foreground, and 0 means background.

"gca" means "get current axis"—the current figure.

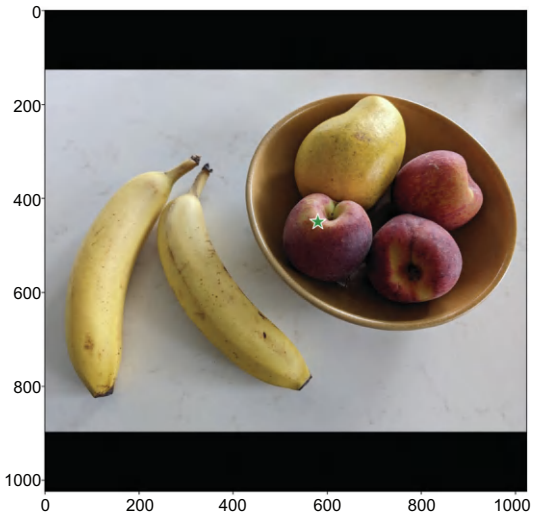


Figure 11.10 A prompt point, landing on a peach

Let's prompt SAM with it:

```
outputs = model.predict(
    {
        "images": ops.expand_dims(image, axis=0),
        "points": ops.expand_dims(input_point, axis=0),
        "labels": ops.expand_dims(input_label, axis=0),
    }
)
```

The return value `outputs` has a `"masks"` field which contains four 256 × 256 candidate masks for the target object, ranked by decreasing match quality. The quality scores of the masks are available under the `"iou_pred"` field as part of the model's output:

```
>>> outputs["masks"].shape
(1, 4, 256, 256)
```

Let's overlay the first mask on the image (see figure 11.11):

```
def get_mask(sam_outputs, index=0):
    mask = sam_outputs["masks"][0][index]
    mask = np.expand_dims(mask, axis=-1)
    mask = resize_and_pad(mask)
    return ops.convert_to_numpy(mask) > 0.0

mask = get_mask(outputs, index=0)
```

```
plt.figure(figsize=(10, 10))
show_image(image, plt.gca())
show_mask(mask, plt.gca())
show_points(input_point, plt.gca())
plt.show()
```

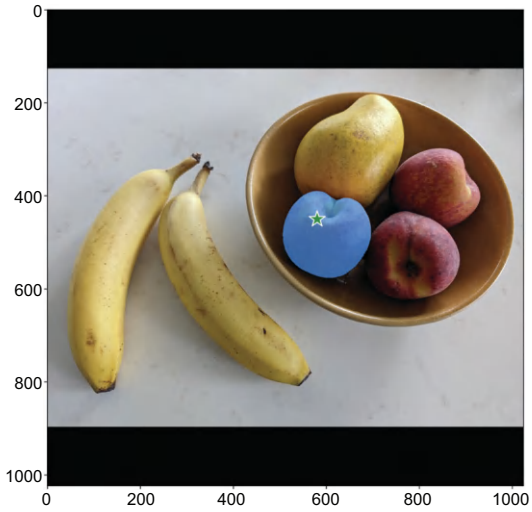


Figure 11.11
Segmented peach

Pretty good!

Next, let's try a banana. We'll prompt the model with coordinates `(300, 550)`, which land on the second banana from the left (see figure 11.12):

```
input_point = np.array([[300, 550]])
input_label = np.array([1])

outputs = model.predict(
    {
        "images": ops.expand_dims(image, axis=0),
        "points": ops.expand_dims(input_point, axis=0),
        "labels": ops.expand_dims(input_label, axis=0),
    }
)
mask = get_mask(outputs, index=0)

plt.figure(figsize=(10, 10))
show_image(image, plt.gca())
show_mask(mask, plt.gca())
show_points(input_point, plt.gca())
plt.show()
```

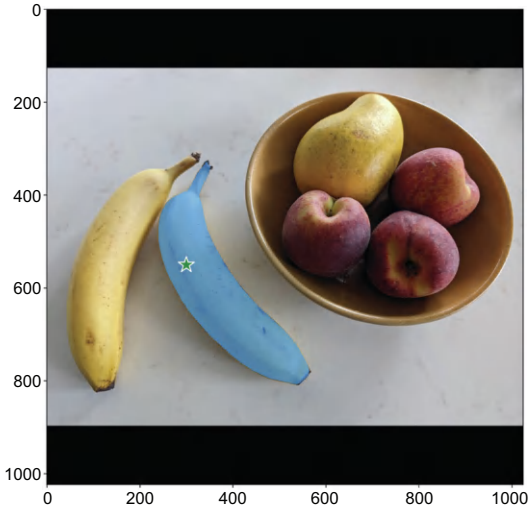


Figure 11.12
Segmented banana

Now, what about the other mask candidates? Those can come in handy for ambiguous prompts. Let's try to plot the other three masks (see figure 11.13):

```
fig, axes = plt.subplots(1, 3, figsize=(20, 60))
masks = outputs["masks"][0][1:]
for i, mask in enumerate(masks):
    show_image(image, axes[i])
    show_points(input_point, axes[i])
    mask = get_mask(outputs, index=i + 1)
    show_mask(mask, axes[i])
    axes[i].set_title(f"Mask {i + 1}", fontsize=16)
    axes[i].axis("off")
plt.show()
```

Figure 11.13 Alternative segmentation masks for the banana prompt

As you can see here, an alternative segmentation found by the model includes both bananas.

11.3.5 Prompting the model with a target box

Besides providing one or more target points, you can also provide boxes approximating the location of the object to segment. These boxes should be passed via the coordinates of their top-left and bottom-right corners. Here's a box around the mango (see figure 11.14):

```
input_box = np.array(
    [
        [520, 180],
        [770, 420],
    ]
)

plt.figure(figsize=(10, 10))
show_image(image, plt.gca())
show_box(input_box, plt.gca())
plt.show()
```

Top-left corner

Bottom-right corner

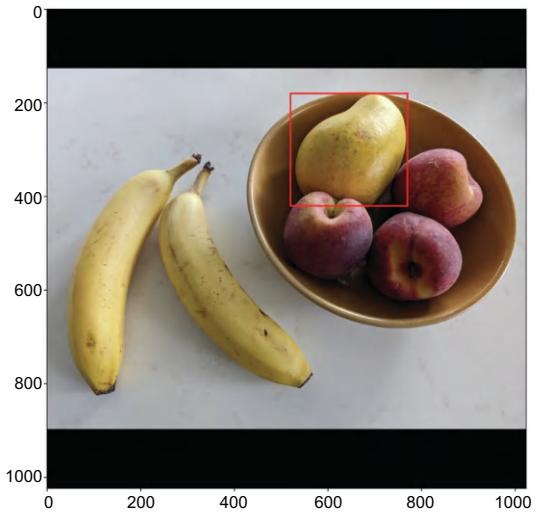


Figure 11.14 Box prompt around the mango

Let's prompt SAM with it (see figure 11.15):

```
outputs = model.predict(
    {
```

```
        "images": ops.expand_dims(image, axis=0),
        "boxes": ops.expand_dims(input_box, axis=(0, 1)),
    }
)
mask = get_mask(outputs, 0)
plt.figure(figsize=(10, 10))
show_image(image, plt.gca())
show_mask(mask, plt.gca())
show_box(input_box, plt.gca())
plt.show()
```

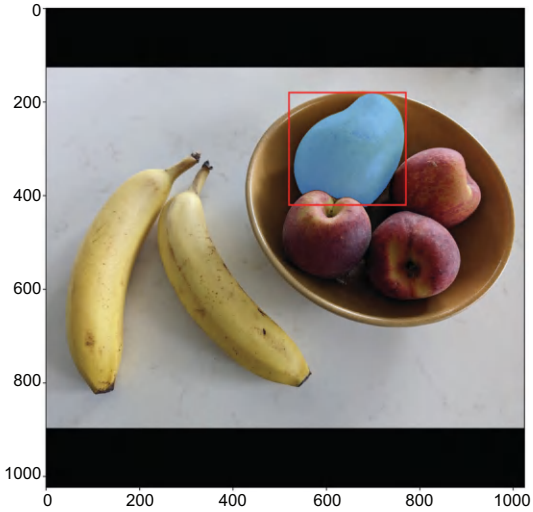


Figure 11.15 Segmented mango

SAM can be a powerful tool to quickly create large datasets of images annotated with segmentation masks.

Summary

- Image segmentation is one of the main categories of computer vision tasks. It consists of computing segmentation masks that describe the contents of an image at the pixel level.
- To build your own segmentation model, use a stack of strided `Conv2D` layers to "compress" the input image into a smaller feature map, followed by a stack of corresponding `Conv2DTranspose` layers to "expand" the feature map into a segmentation mask the same size as the input image.
- You can also use a pretrained segmentation model. Segment Anything, included in KerasHub, is a powerful model that supports image prompting, text prompting, point prompting, and box prompting.

12 Object detection

This chapter covers

- Understanding the object detection problem
- Two-stage and single-stage object detectors
- Training a simple single-stage detector from scratch
- Using a pretrained object detector

Object detection is all about drawing boxes (called *bounding boxes*) around objects of interest in a picture (see figure 12.1). This enables you to know not just which objects are in a picture, but also where they are. Some of its most common applications are

- *Counting*—Find out how many instances of an object are in an image.
- *Tracking*—Track how objects move in a scene over time by performing object detection on every frame of a movie.
- *Cropping*—Identify the area of an image that contains an object of interest to crop it and send a higher-resolution version of the image patch to a classifier or an Optical Character Recognition (OCR) model.

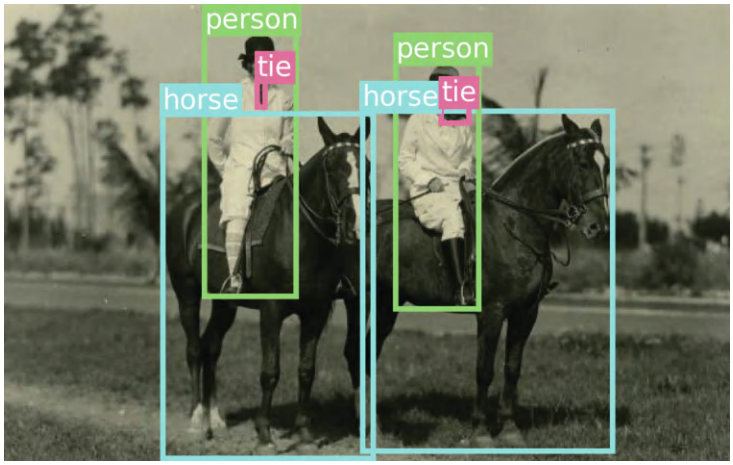


Figure 12.1 Object detectors draw boxes around objects in an image and label them.

You might be thinking, if I have a segmentation mask for an object instance, I can already compute the coordinates of the smallest box that contains the mask. So couldn't we just use image segmentation all the time? Do we need object detection models at all?

Indeed, segmentation is a strict superset of detection. It returns all the information that could be returned by a detection model—plus a lot more. This increased wealth of information has a significant computational cost: a good object detection model will typically run much faster than an image segmentation model. It also has a data labeling cost: to train a segmentation model, you need to collect pixel-precise masks, which are much more time-consuming to produce than the mere bounding boxes required by object detection models.

As a result, you will always want to use an object detection model if you have no need for pixel-level information—for instance, if all you want is to count objects in an image.

12.1 Single-stage vs. two-stage object detectors

There are two broad categories of object detection architectures:

- Two-stage detectors, which first extract region proposals, known as Region-based Convolutional Neural Networks (R-CNN) models
- Single-stage detectors, such as RetinaNet or the You Only Look Once family of models

Here's how they work.

12.1.1 Two-stage R-CNN detectors

A region-based ConvNet, or R-CNN model, is a two-stage model. The first stage takes an image and produces a few thousand partially overlapping bounding boxes around

areas that look object-like. These boxes are called *region proposals*. This stage isn't very smart, so at that point we aren't quite sure whether the proposed regions do contain objects and, if so, what objects they contain.

That's the job of the second stage—a ConvNet that looks at each region proposal and classifies it into a number of predetermined classes, just like the models you've seen in chapter 9 (see figure 12.2). Region proposals that have a low score across all classes considered are discarded. We are then left with a much smaller set of boxes, each with a high class-presence score for one particular class. Finally, bounding boxes around each object are further refined to eliminate duplicates and make each bounding box as precise as possible.

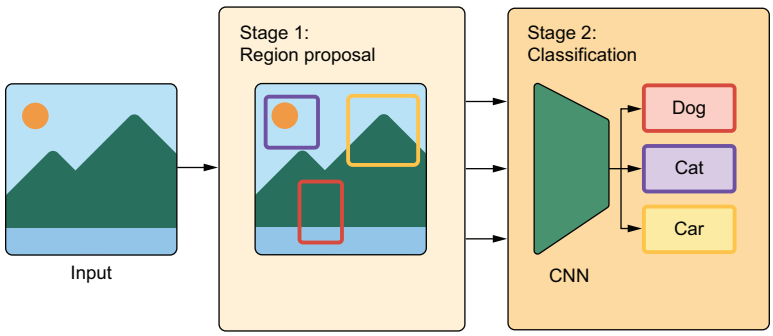


Figure 12.2 An R-CNN first extracts region proposals and then classifies the proposals with a ConvNet (a CNN).

In early R-CNN versions, the first stage was a *heuristic model* called *Selective Search* that used some definition of spatial consistency to identify object-like areas. *Heuristic* is a term you'll hear quite a lot in machine learning—it simply means "a bundle of hard-coded rules someone made up." It's usually used in opposition to learned models (where the rules are automatically derived) or theory-derived models. In later versions of R-CNN, such as Faster-R-CNN, the box generation stage became a deep learning model, called a Region Proposal Network.

The two-stage approach of R-CNN works very well in practice, but it's quite computationally expensive, most notably because it requires you to classify thousands of patches—for every single image you process. That makes it unsuitable for most real-time applications and for embedded systems. My take is that, in practical applications, you generally don't ever need a computationally expensive object detection system like R-CNN because if you're doing server-side inference with a beefy GPU, then you'll probably be better off using a segmentation model instead, like the Segment Anything model we saw in the previous chapter. And if you're resource-constrained, then you're going to want to use a more computationally efficient object detection architecture—a single-stage detector.

12.1.2 Single-stage detectors

Around 2015, researchers and practitioners began experimenting with using a single deep learning model to jointly predict bounding box coordinates together with their labels, an architecture called a *single-stage detector*. The main families of single-stage detectors are RetinaNet, Single Shot MultiBox Detectors (SSD), and the You Only Look Once family, abbreviated as YOLO. Yes, like the meme. That's on purpose.

Single-stage detectors, especially recent YOLO iterations, boast significantly faster speeds and greater efficiency than their two-stage counterparts, albeit with a minor potential tradeoff in accuracy. Nowadays, YOLO is arguably the most popular object detection model out there, especially when it comes to real-time applications. There's usually a new version of it that comes out every year—interestingly, each new version tends to be developed by a separate organization.

In the next section, we will build a simplified YOLO model from scratch.

12.2 Training a YOLO model from scratch

Overall, building an object detector can be a bit of an undertaking—not that there's anything theoretically complex about it. There's just a lot of code needed to handle manipulating bounding boxes and predicted output. To keep things simple, we will recreate the very first YOLO model from 2015. There are 12 YOLO versions as of this writing, but the original is a bit simpler to work with.

12.2.1 Downloading the COCO dataset

Before we start creating our model, we need data to train with. The COCO dataset,[1] short for *Common Objects in Context*, is one of the best-known and most commonly used object detection datasets. It consists of real-world photos from a number of different sources plus human-created annotations. This includes object labels, bounding box annotations, and full segmentation masks. We will disregard the segmentation masks and just use bounding boxes.

Let's download the 2017 version of the COCO dataset. While not a large dataset by today's standards, this 18 GB dataset will be the largest dataset we use in the book. If you are running the code as you read, this is a good chance to take a breather.

Listing 12.1 Downloading the 2017 COCO dataset

```
import keras
import keras_hub

images_path = keras.utils.get_file(
    "coco",
    "http://images.cocodataset.org/zips/train2017.zip",
    extract=True,
```

[1] The COCO 2017 detection dataset can be explored at https://cocodataset.org/. Most images in this chapter are from the dataset.

```
)
annotations_path = keras.utils.get_file(
    "annotations",
    "http://images.cocodataset.org/annotations/annotations_trainval2017.zip",
    extract=True,
)
```

We need to do some input massaging before we are ready to use this data. The first download gives us an unlabeled directory of all the COCO images. The second download includes all image metadata via a JSON file. COCO associates each image file with an ID, and each bounding box is paired with one of these IDs. We need to collate all box and image data together.

Each bounding box comes with `x, y, width, height` pixel coordinates starting at the top left corner of the image. As we load our data, we can rescale all bounding box coordinates so they are points in a `[0, 1]` unit square. This will make it easier to manipulate these boxes without needing to check the size of each input image.

Listing 12.2 Parsing the COCO data

```
import json

with open(f"{annotations_path}/annotations/instances_train2017.json", "r") as f:
    annotations = json.load(f)

images = {image["id"]: image for image in annotations["images"]}

def scale_box(box, width, height):
    scale = 1.0 / max(width, height)
    x, y, w, h = [v * scale for v in box]
    x += (height - width) * scale / 2 if height > width else 0
    y += (width - height) * scale / 2 if width > height else 0
    return [x, y, w, h]

metadata = {}
for annotation in annotations["annotations"]:
    id = annotation["image_id"]
    if id not in metadata:
        metadata[id] = {"boxes": [], "labels": []}
    image = images[id]
    box = scale_box(annotation["bbox"], image["width"], image["height"])
    metadata[id]["boxes"].append(box)
    metadata[id]["labels"].append(annotation["category_id"])
    metadata[id]["path"] = images_path + "/train2017/" + image["file_name"]
metadata = list(metadata.values())
```

Sorts image metadata by ID

Converts bounding box to coordinates on a unit square

Aggregates all bounding box annotations by image ID

Let's take a look at the data we just loaded.

Listing 12.3 Inspecting the COCO data

```
>>> len(metadata)
117266
>>> min([len(x["boxes"]) for x in metadata])
1
>>> max([len(x["boxes"]) for x in metadata])
63
>>> max(max(x["labels"]) for x in metadata) + 1
91
>>> metadata[435]
{"boxes": [[0.12, 0.27, 0.57, 0.33],
  [0.0, 0.15, 0.79, 0.69],
  [0.0, 0.12, 1.0, 0.75]],
 "labels": [17, 15, 2],
 "path": "/root/.keras/datasets/coco/train2017/000000171809.jpg"}
>>> [keras_hub.utils.coco_id_to_name(x) for x in metadata[435]["labels"]]
["cat", "bench", "bicycle"]
```

We have 117,266 images. Each image can have anywhere from 1 to 63 objects with an associated bounding box. There are only 91 possible labels for objects, chosen by the COCO dataset creators.

We can use a KerasHub utility `keras_hub.utils.coco_id_to_name(id)` to map these integer labels to human-readable names, similar to the utility we used to decode ImageNet predictions to text labels back in chapter 8.

Let's visualize an example image to make this a little more concrete. We can define a function to draw an image with Matplotlib and another function to draw a labeled bounding box on this image. We will need both of these throughout the chapter. We can use the HSV colorspace as a simple trick to generate new colors for each new label we see. By fixing the saturation and brightness of the color and only updating its hue, we can generate bright new colors that stand out clearly from our image.

Listing 12.4 Visualizing a COCO image with box annotations

```
import matplotlib.pyplot as plt
from matplotlib.colors import hsv_to_rgb
from matplotlib.patches import Rectangle

color_map = {0: "gray"}

def label_to_color(label):
    if label not in color_map:
        h, s, v = (len(color_map) * 0.618) % 1, 0.5, 0.9
        color_map[label] = hsv_to_rgb((h, s, v))
    return color_map[label]

def draw_box(ax, box, text, color):
    x, y, w, h = box
    ax.add_patch(Rectangle((x, y), w, h, lw=2, ec=color, fc="none"))
```

Uses the golden ratio to generate new hues of a bright color with the HSV colorspace

```
    textbox = dict(fc=color, pad=1, ec="none")
    ax.text(x, y, text, c="white", size=10, va="bottom", bbox=textbox)

def draw_image(ax, image):
    ax.set(xlim=(0, 1), ylim=(1, 0), xticks=[], yticks=[], aspect="equal")
    image = plt.imread(image)
    height, width = image.shape[:2]
    hpad = (1 - height / width) / 2 if width > height else 0
    wpad = (1 - width / height) / 2 if height > width else 0
    extent = [wpad, 1 - wpad, 1 - hpad, hpad]
    ax.imshow(image, extent=extent)
```

Pads the image so it fits inside the unit cube

Draws the image on a unit cube with (0, 0) at the top left

Let's use our new visualization to look at the sample image[2] we were inspecting earlier (see figure 12.3):

```
sample = metadata[435]
ig, ax = plt.subplots(dpi=300)
draw_image(ax, sample["path"])
for box, label in zip(sample["boxes"], sample["labels"]):
    label_name = keras_hub.utils.coco_id_to_name(label)
    draw_box(ax, box, label_name, label_to_color(label))
plt.show()
```

Figure 12.3 YOLO outputs a bounding box prediction and class label for each image region.

[2] Image from the COCO 2017 dataset, https://cocodataset.org/. Image from Flickr, http://farm8.staticflickr.com/7250/7520201840_3e01349e3f_z.jpg, CC BY 2.0 https://creativecommons.org/licenses/by/2.0/.

While it would be fun to train on all 18 gigabytes of our input data, we want to keep the examples in this book easily runnable on modest hardware. If we limit ourselves to only images with four or fewer boxes, we will make our training problem easier and a halve the data size. Let's do this and shuffle our data—the images are grouped by object type, which would be terrible for training:

```
import random

metadata = list(filter(lambda x: len(x["boxes"]) <= 4, metadata))
random.shuffle(metadata)
```

That's it for data loading! Let's start creating our YOLO model.

12.2.2 Creating a YOLO model

As mentioned previously, the YOLO model is a single stage detector. Rather than first attempting to identify all candidate objects in a scene and then classifying the object regions, YOLO will propose bounding boxes and object labels in one go.

Our model will divide an image up into a grid and predict two separate outputs at each grid location—a bounding box and a class label. In the original paper by Redmon et al.,[3] the model actually predicted multiple boxes per grid location, but we keep things simple and just predict one box in each grid square.

Most images will not have objects evenly distributed across a grid, and to account for this, the model will output a *confidence score* along with each box, as shown in figure 12.4. We'd like this confidence to be high when an object is detected at a location, and zero when there's no object. Most grid locations will have no object and should report a near-zero confidence.

Like many models in computer vision, the YOLO model uses a ConvNet *backbone* to obtain interesting high-level features for an input image, a concept we first explored in chapter 8. In their paper, the authors created their own backbone model and pretrained it with ImageNet for classification. Rather than do this ourselves, we can instead use KerasHub to load a pretrained backbone.

Instead of using the Xception backbone we've used so far in this book, we will switch to ResNet, a family of models we first mentioned in chapter 9. The structure is quite similar to Xception, but ResNet uses strides instead of pooling layers to downsample the image. As we mentioned in chapter 11, strided convolutions are better when we care about the *spatial location* of the input.

Let's load up our pretrained model and matching preprocessing (to rescale the image). We will resize our images to 448 × 448; image input size is quite important for the object detection task.

[3] Redmon et al., "You Only Look Once: Unified, Real-Time Object Detection," CoRR (2015), https://arxiv.org/abs/1506.02640.

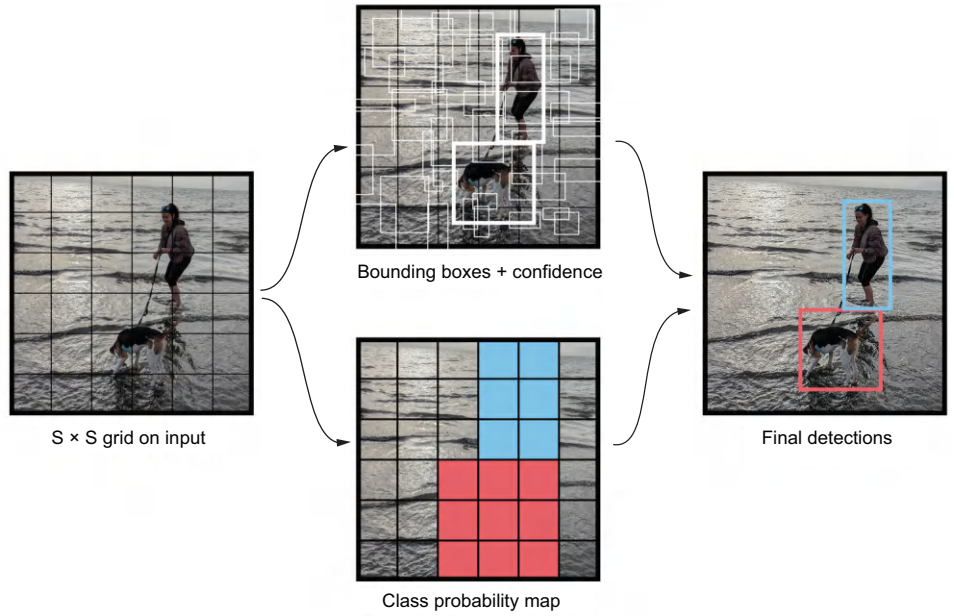


Figure 12.4 YOLO outputs as visualized in the first YOLO paper

Listing 12.5 Loading the ResNet model

```
image_size = 448

backbone = keras_hub.models.Backbone.from_preset(
    "resnet_50_imagenet",
)
preprocessor = keras_hub.layers.ImageConverter.from_preset(
    "resnet_50_imagenet",
    image_size=(image_size, image_size),
)
```

Next, we can turn our backbone into a detection model by adding new layers for outputting box and class predictions. The setup proposed in the YOLO paper is quite simple. Take the output of a ConvNet backbone and feed it through two densely connected layers with an activation in the middle. Then, split the output. The first five numbers will be used for bounding box prediction (four for the box and one for the box confidence). The rest will be used for the *class probability map* shown in figure 12.4—a classification prediction for each grid location over all possible 91 labels.

Let's write this out.

Listing 12.6 Attaching a YOLO prediction head

```
from keras import layers

grid_size = 6
num_labels = 91

inputs = keras.Input(shape=(image_size, image_size, 3))
x = backbone(inputs)
x = layers.Conv2D(512, (3, 3), strides=(2, 2))(x)
x = keras.layers.Flatten()(x)
x = layers.Dense(2048, activation="relu", kernel_initializer="glorot_normal")(x)
x = layers.Dropout(0.5)(x)
x = layers.Dense(grid_size * grid_size * (num_labels + 5))(x)
x = layers.Reshape((grid_size, grid_size, num_labels + 5))(x)
box_predictions = x[..., :5]
class_predictions = layers.Activation("softmax")(x[..., 5:])
outputs = {"box": box_predictions, "class": class_predictions}
model = keras.Model(inputs, outputs)
```

Passes our flattened feature maps through two densely connected layers

Makes our backbone outputs smaller and then flattens the output features

Split box and class predictions

Reshapes outputs to a 6 × 6 grid

We can get a better sense of the model by looking at the model summary:

```
>>> model.summary()
Model: "functional_3"
```

Layer (type)	Output Shape	Param #	Connected to
input_layer_7 (InputLayer)	(None, 448, 448, 3)	0	-
res_net_backbone_12 (ResNetBackbone)	(None, 14, 14, 2048)	23,580,512	input_layer_7[0][…
conv2d_3 (Conv2D)	(None, 6, 6, 512)	9,437,696	res_net_backbone_…
flatten_3 (Flatten)	(None, 18432)	0	conv2d_3[0][0]
dense_6 (Dense)	(None, 2048)	37,750,784	flatten_3[0][0]
dropout_3 (Dropout)	(None, 2048)	0	dense_6[0][0]
dense_7 (Dense)	(None, 3456)	7,081,344	dropout_3[0][0]
reshape_3 (Reshape)	(None, 6, 6, 96)	0	dense_7[0][0]
get_item_7 (GetItem)	(None, 6, 6, 91)	0	reshape_3[0][0]

get_item_6 (GetItem)	(None, 6, 6, 5)	0	reshape_3[0][0]
activation_33 (Activation)	(None, 6, 6, 91)	0	get_item_7[0][0]

```
Total params: 77,850,336 (296.98 MB)
Trainable params: 77,797,088 (296.77 MB)
Non-trainable params: 53,248 (208.00 KB)
```

Our backbone outputs have shape `(batch_size, 14, 14, 2048)`. That is 401,408 output floats per image, a bit too many to feed into our dense layers. We downscale the feature maps with a strided conv layer to `(batch_size, 6, 6, 512)` with a more manageable 18,432 floats per image.

Next, we can add our two densely connected layers. We flatten the entire feature map, pass it through a `Dense` with a `relu` activation and then pass it through a final `Dense` with our exact number of output predictions—5 for the bounding box and confidence and 91 for each object class at each grid location.

Finally, we reshape the outputs back to a 6×6 grid and split our box and class predictions. As usual for our classification outputs, we apply a softmax. The box outputs will need more special consideration; we will cover this later.

Looking good! Note that because we flatten the entire feature map through the classification layer, every grid detector can use the entire image's features; there's no locality constraint. This is by design—large objects will not stay contained to a single grid cell.

12.2.3 Readying the COCO data for the YOLO model

Our model is relatively simple, but we still need to preprocess our inputs to align them with the prediction grid. Each grid detector will be responsible for detecting any boxes whose center falls inside the grid box. Our model will output five floats for the box `(x, y, w, h, confidence)`. The `x` and `y` will represent the object's center relative to the bounds of the grid cell (from 0 to 1). The `w` and `h` will represent the object's size relative to the image size.

We already have the right `w` and `h` values in our training data. However, we need to translate our `x` and `y` values to and from the grid. Let's define two utilities:

```
def to_grid(box):
    x, y, w, h = box
    cx, cy = (x + w / 2) * grid_size, (y + h / 2) * grid_size
    ix, iy = int(cx), int(cy)
    return (ix, iy), (cx - ix, cy - iy, w, h)

def from_grid(loc, box):
    (xi, yi), (x, y, w, h) = loc, box
```

```
    x = (xi + x) / grid_size - w / 2
    y = (yi + y) / grid_size - h / 2
    return (x, y, w, h)
```

Let's rework our training data so it conforms to this new grid structure. We can create two arrays as long as our dataset with our grid:

- The first will contain our class probability map. We will mark all grid cells that intersect with a bounding box with the correct label. To keep our code simple, we won't worry about overlapping boxes.
- The second will contain the boxes themselves. We will translate all boxes to the grid and label the correct grid cell with the coordinates for the box. The confidence for an actual box in our label data will always be one, and the confidence for all other locations will be zero.

Listing 12.7 Creating the YOLO targets

```
import numpy as np
import math

class_array = np.zeros((len(metadata), grid_size, grid_size))
box_array = np.zeros((len(metadata), grid_size, grid_size, 5))

for index, sample in enumerate(metadata):
    boxes, labels = sample["boxes"], sample["labels"]
    for box, label in zip(boxes, labels):
        (x, y, w, h) = box
        left, right = math.floor(x * grid_size), math.ceil((x + w) * grid_size)
        bottom, top = math.floor(y * grid_size), math.ceil((y + h) * grid_size)
        class_array[index, bottom:top, left:right] = label

for index, sample in enumerate(metadata):
    boxes, labels = sample["boxes"], sample["labels"]
    for box, label in zip(boxes, labels):
        (xi, yi), (grid_box) = to_grid(box)
        box_array[index, yi, xi] = [*grid_box, 1.0]
        class_array[index, yi, xi] = label
```

Finds all grid cells whose center falls inside the box

Transforms the box to the grid coordinate system

Makes sure the class label for the box's center location matches the box

Let's visualize our YOLO training data with our box drawing helpers (figure 12.5). We will draw the entire class activation map over our first input image[4] and add the confidence score of a box along with its label.

[4] Image from the COCO 2017 dataset, https://cocodataset.org/. Image from Flickr, http://farm9.staticflickr.com/8081/8387882360_5b97a233c4_z.jpg, CC BY 2.0 https://creativecommons.org/licenses/by/2.0/.

Listing 12.8 Visualizing a YOLO target

```
def draw_prediction(image, boxes, classes, cutoff=None):
    fig, ax = plt.subplots(dpi=300)
    draw_image(ax, image)
    for yi, row in enumerate(classes):                    # Draws the YOLO output grid
        for xi, label in enumerate(row):                  # and class probability map
            color = label_to_color(label) if label else "none"
            x, y, w, h = (v / grid_size for v in (xi, yi, 1.0, 1.0))
            r = Rectangle((x, y), w, h, lw=2, ec="black", fc=color, alpha=0.5)
            ax.add_patch(r)
    for yi, row in enumerate(boxes):                      # Draws all boxes at each grid
        for xi, box in enumerate(row):                    # location above our cutoff
            box, confidence = box[:4], box[4]
            if not cutoff or confidence >= cutoff:
                box = from_grid((xi, yi), box)
                label = classes[yi, xi]
                color = label_to_color(label)
                name = keras_hub.utils.coco_id_to_name(label)
                draw_box(ax, box, f"{name} {max(confidence, 0):.2f}", color)
    plt.show()

draw_prediction(metadata[0]["path"], box_array[0], class_array[0], cutoff=1.0)
```

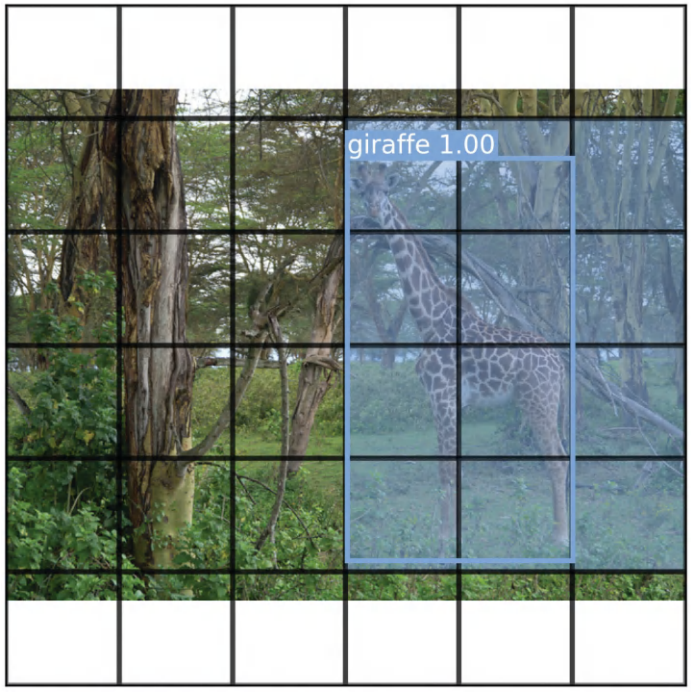


Figure 12.5 YOLO outputs a bounding box prediction and class label for each image region.

Lastly, let's use `tf.data` to load our image data. We will load our images from disk, apply our preprocessing, and batch them. We should also split a validation set to monitor training.

Listing 12.9 Creating a dataset to train on

```
import tensorflow as tf

def load_image(path):                                  # Loads and resizes the model with tf.data
    x = tf.io.read_file(path)
    x = tf.image.decode_jpeg(x, channels=3)
    return preprocessor(x)

images = tf.data.Dataset.from_tensor_slices([x["path"] for x in metadata])
images = images.map(load_image, num_parallel_calls=8)
labels = {"box": box_array, "class": class_array}
labels = tf.data.Dataset.from_tensor_slices(labels)    # Creates a merged dataset and batches it

dataset = tf.data.Dataset.zip(images, labels).batch(16).prefetch(2)
val_dataset, train_dataset = dataset.take(500), dataset.skip(500)   # Splits off some validation data
```

With that, our data is ready for training.

> **NOTE** This training example shows clearly why a streaming library like `tf.data` is helpful. Loading all the images in this large dataset in one go would overwhelm our system memory (remember an image tensor is much larger than a compressed JPEG file). With `tf.data`, we can load our image data in batch by batch and release the memory when we are done, only mapping in the particular parts of the dataset we need at a given moment. The `prefetch(2)` call will cause `tf.data` to keep two batches buffered and ready before they are used so we don't interrupt training each batch to load and resize more images.

12.2.4 Training the YOLO model

We have our model and our training data ready, but there's one last element we need before we can actually run `fit()`: the loss function. Our model outputs predicted boxes and predicted grid labels. We saw in chapter 7 how we can define multiple losses for each output—Keras will simply sum the losses together during training. We can handle the classification loss with `sparse_categorical_crossentropy` as usual.

The box loss, however, needs some special consideration. The basic loss proposed by the YOLO authors is fairly simple. They use the sum-squared error of the difference between the target box parameters and the predicted ones. We will only compute this error for grid cells with actual boxes in the labeled data.

The tricky part of the loss is the box confidence output. The authors wanted the confidence output to reflect not just the presence of an object, but also how good the predicted

box is. To create a smooth estimate of how good a box prediction is, the authors propose using the *Intersection over Union* (IoU) metric we saw last chapter. If a grid cell is empty, the predicted confidence at the location should be zero. However, if a grid cell contains an object, we can use the IoU score between the current box prediction and the actual box as the target confidence value. This way, as the model becomes better at predicting box locations, the IoU score and the learned confidence values will go up.

This calls for a custom loss function. We can start be defining a utility to compute IoU scores for target and predicted boxes.

Listing 12.10 Computing IoU for two boxes

```
from keras import ops

def unpack(box):
    return box[..., 0], box[..., 1], box[..., 2], box[..., 3]

def intersection(box1, box2):
    cx1, cy1, w1, h1 = unpack(box1)
    cx2, cy2, w2, h2 = unpack(box2)
    left = ops.maximum(cx1 - w1 / 2, cx2 - w2 / 2)
    bottom = ops.maximum(cy1 - h1 / 2, cy2 - h2 / 2)
    right = ops.minimum(cx1 + w1 / 2, cx2 + w2 / 2)
    top = ops.minimum(cy1 + h1 / 2, cy2 + h2 / 2)
    return ops.maximum(0.0, right - left) * ops.maximum(0.0, top - bottom)

def intersection_over_union(box1, box2):
    cx1, cy1, w1, h1 = unpack(box1)
    cx2, cy2, w2, h2 = unpack(box2)
    intersection_area = intersection(box1, box2)
    a1 = ops.maximum(w1, 0.0) * ops.maximum(h1, 0.0)
    a2 = ops.maximum(w2, 0.0) * ops.maximum(h2, 0.0)
    union_area = a1 + a2 - intersection_area
    return ops.divide_no_nan(intersection_area, union_area)
```

Unpacks a tensor of boxes

Computes the intersection area between two box tensors

Computes the IoU between two box tensors

Let's use this utility to define our custom loss. Redmon et al. propose a couple of loss scaling tricks to improve the quality of training:

- They scale up the box placement loss by a factor of five, so it becomes a more important part of overall training.
- Since most grid cells are empty, they also scale down the confidence loss in empty locations by a factor of two. This keeps these zero confidence predictions from overwhelming the loss.
- They take the square root of the width and height before computing the loss. This is to stop large boxes from mattering disproportionately more than small boxes. We will use a `sqrt` function that preserves the sign of the input, since our model might predict negative widths and heights at the start of training.

Let's write this out.

Listing 12.11 Defining the YOLO bounding box loss

```
def signed_sqrt(x):
    return ops.sign(x) * ops.sqrt(ops.absolute(x) + keras.config.epsilon())

def box_loss(true, pred):
    xy_true, wh_true, conf_true = true[..., :2], true[..., 2:4], true[..., 4:]
    xy_pred, wh_pred, conf_pred = pred[..., :2], pred[..., 2:4], pred[..., 4:]
    no_object = conf_true == 0.0
    xy_error = ops.square(xy_true - xy_pred)
    wh_error = ops.square(signed_sqrt(wh_true) - signed_sqrt(wh_pred))
    iou = intersection_over_union(true, pred)
    conf_target = ops.where(no_object, 0.0, ops.expand_dims(iou, -1))
    conf_error = ops.square(conf_target - conf_pred)
    error = ops.concatenate(
        (
            ops.where(no_object, 0.0, xy_error * 5.0),
            ops.where(no_object, 0.0, wh_error * 5.0),
            ops.where(no_object, conf_error * 0.5, conf_error),
        ),
        axis=-1,
    )
    return ops.sum(error, axis=(1, 2, 3))
```

If confidence_true is 0.0, there is no object in this grid cell.

Unpacks values

Concatenates the errors weith scaling hacks

Computes confidence error

Computes box placement errors

Returns one loss value per sample; Keras will sum over the batch.

We are finally ready to start training our YOLO model. Purely to keep this example short, we will skip over metrics. In a real-world setting, you'd want quite a few metrics here—such as the accuracy of the model at different confidence cutoff levels.

Listing 12.12 Training the YOLO model

```
model.compile(
    optimizer=keras.optimizers.Adam(2e-4),
    loss={"box": box_loss, "class": "sparse_categorical_crossentropy"},
)
model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=4,
)
```

Training takes over an hour on the Colab free GPU runtime, and our model is still undertrained (validation loss is still falling!). Let's try visualizing an output from our model (figure 12.6). We will use a low-confidence cutoff, as our model is not a very good object detector quite yet.

Listing 12.13 Training the YOLO model

```
x, y = next(iter(val_dataset.rebatch(1)))
preds = model.predict(x)
boxes = preds["box"][0]
classes = np.argmax(preds["class"][0], axis=-1)
path = metadata[0]["path"]
draw_prediction(path, boxes, classes, cutoff=0.1)
```

Rebatches our dataset to get a single sample instead of 16

Uses argmax to find the most likely label at each grid location

Loads the image from disk to view it a full size

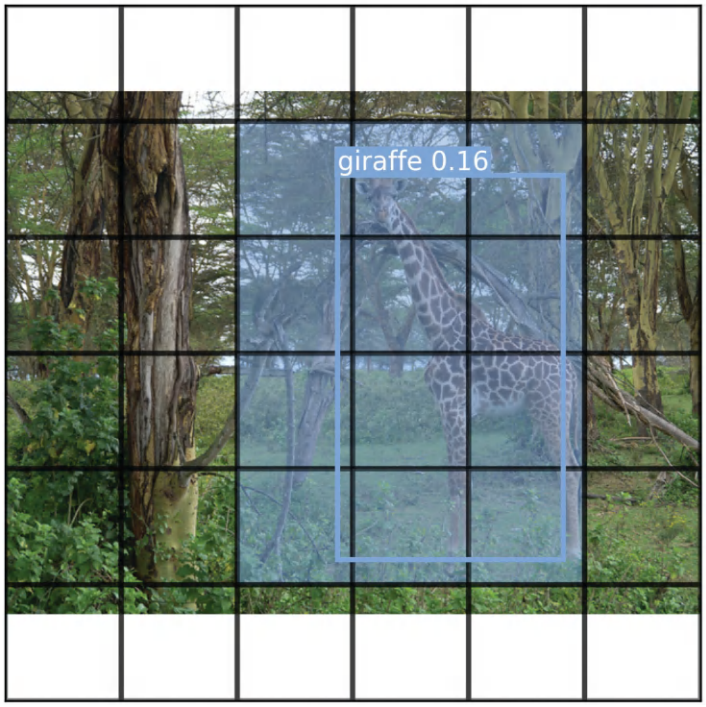


Figure 12.6 Predictions for our sample image

We can see our model is starting to understand box locations and class labels, though it is still not very accurate. Let's visualize every box predicted by the model (figure 12.7), even those with zero confidence:

```
draw_prediction(path, boxes, classes, cutoff=None)
```

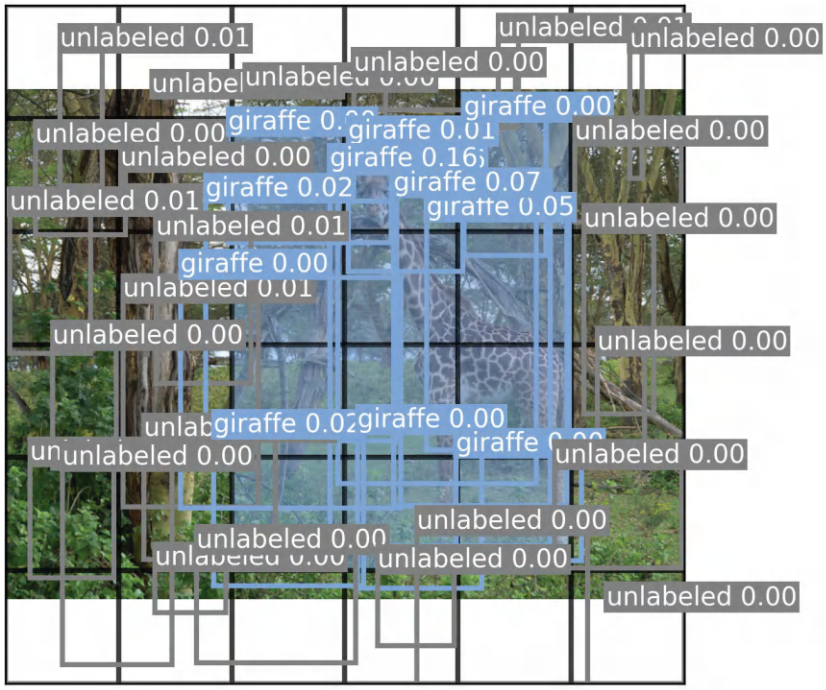


Figure 12.7 Every bounding box predicted by the YOLO model

Our model learns very low-confidence values because it has not yet learned to consistently locate objects in a scene. To further improve the model, we should try a number of things:

- Train for more epochs
- Use the whole COCO dataset
- Data augmentation (e.g., translating and rotating input images and boxes)
- Improve our class probability map for overlapping boxes
- Predict multiple boxes per grid location using a bigger output grid

All of these would positively affect the model performance and get us closer to the original YOLO training recipe. However, this example is really just to get a feel for object detection training—training an accurate COCO detection model from scratch would take a large amount of compute and time. Instead, to get a sense of a better-performing detection model, let's try using a pretrained object detection model called RetinaNet.

12.3 Using a pretrained RetinaNet detector

RetinaNet is also a single-stage object detector and operates on the same basic principles as the YOLO model. The biggest conceptual difference between our model and

RetinaNet is that RetinaNet uses its underlying ConvNet differently to better handle both small and large objects simultaneously.

In our YOLO model, we simply took the final outputs of our ConvNet and used them to build our object detector. These output features map to large areas on our input image—as a result, they are not very effective at finding small objects in the scene.

One option to solve this scale issue would be to directly use the output of earlier layers in our ConvNet. This would extract high-resolution features that map to small localized areas of our input image. However, the output of these early layers is not very *semantically interesting*. It might map to different types of simple features like edges and curves, but only later in the ConvNet layers do we start building latent representations for entire objects.

The solution used by RetinaNet is called a feature pyramid network. The final features from the ConvNet base model are upsampled with progressive `Conv2DTranspose` layers, just like we saw in the previous chapter. But critically, we also include *lateral connections* where we sum these upsampled feature maps with the feature maps of the same size from the original ConvNet. This combines the semantically interesting, low-resolution features at the end of the ConvNet with the high-resolution, small-scale features from the beginning of the ConvNet. A rough sketch of this architecture is shown in figure 12.8.

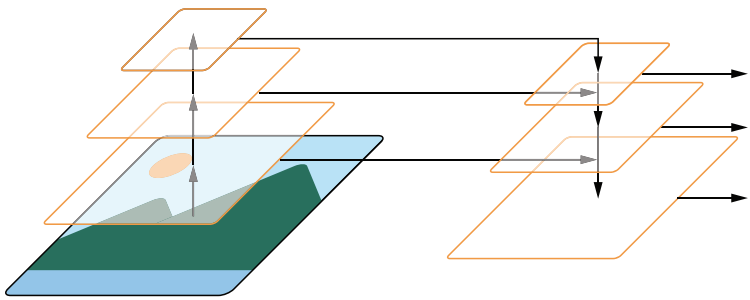


Figure 12.8 A feature pyramid network creates semantically interesting feature maps at different scales.

Feature pyramid networks can substantially boost performance by building effective features for both small and large objects in terms of pixel footprint. Recent versions of YOLO also use the same setup.

Let's go ahead and try out the RetinaNet model, which was also trained on the COCO dataset. To make this a little more interesting, let's try an image that is out-of-distribution for the model, the Pointillist painting *A Sunday Afternoon on the Island of La Grande Jatte*.

We can start by downloading the image and converting it to a NumPy array:

```
url = (
    "https://upload.wikimedia.org/wikipedia/commons/thumb/7/7d/"
    "A_Sunday_on_La_Grande_Jatte%2C_Georges_Seurat%2C_1884.jpg/"
    "1280px-A_Sunday_on_La_Grande_Jatte%2C_Georges_Seurat%2C_1884.jpg"
)
path = keras.utils.get_file(origin=url)
image = np.array([keras.utils.load_img(path)])
```

Next, let's download the model and make a prediction. As we did in the previous chapter, we can use the high-level task API in KerasHub to create an `ObjectDetector` and use it—preprocessing included.

Listing 12.14 Creating the ResNet model

```
detector = keras_hub.models.ObjectDetector.from_preset(
    "retinanet_resnet50_fpn_v2_coco",
    bounding_box_format="rel_xywh",
)
predictions = detector.predict(image)
```

You'll note we pass an extra argument to specify the bounding box format. We can do this for most Keras models and layers that support bounding boxes. We pass `"rel_xywh"` to use the same format as we did for the YOLO model, so we can use the same box drawing utilities. Here, `rel` stands for relative to the image size (e.g., from [0, 1]). Let's inspect the prediction we just made:

```
>>> [(k, v.shape) for k, v in predictions.items()]
[("boxes", (1, 100, 4)),
 ("confidence", (1, 100)),
 ("labels", (1, 100)),
 ("num_detections", (1,))]
>>> predictions["boxes"][0][0]
array([0.53, 0.00, 0.81, 0.29], dtype=float32)
```

We have four different model outputs: bounding boxes, `confidences`, labels, and the total number of detections. This is overall quite similar to our YOLO model. The model can predict a total of 100 objects for each input model.

Let's try displaying the prediction with our box-drawing utilities (figure 12.9).

Listing 12.15 Running inference with RetinaNet

```
fig, ax = plt.subplots(dpi=300)
draw_image(ax, path)
num_detections = predictions["num_detections"][0]
for i in range(num_detections):
```

```
    box = predictions["boxes"][0][i]
    label = predictions["labels"][0][i]
    label_name = keras_hub.utils.coco_id_to_name(label)
    draw_box(ax, box, label_name, label_to_color(label))
plt.show()
```

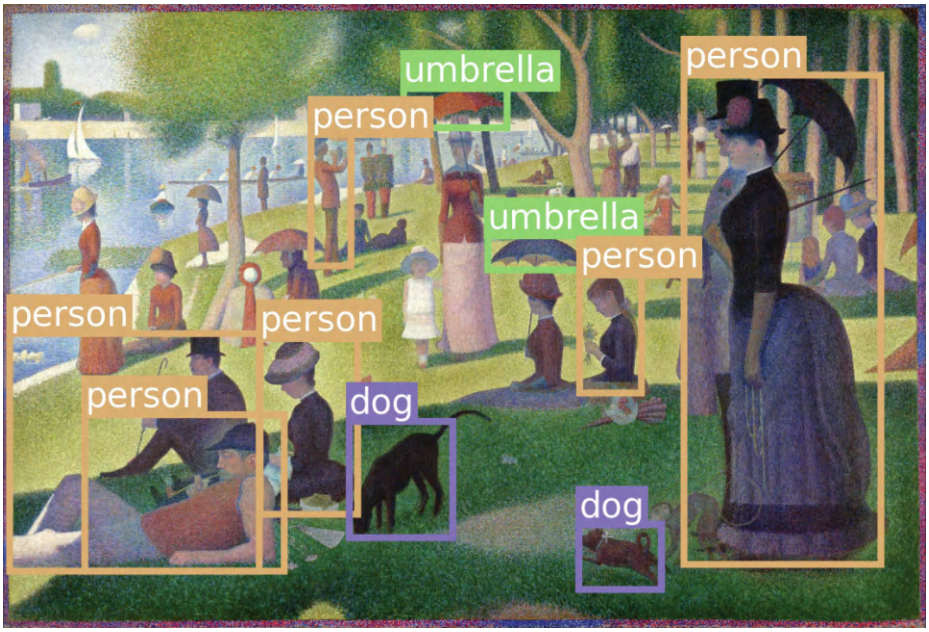


Figure 12.9 Predictions on a test image from the RetinaNet model

The RetinaNet model is able to generalize to a pointillist painting with ease, despite no training on this style of input! This is actually one of the advantages of single-stage object detectors. Paintings and photographs are very different at a pixel level but share a similar structure at a high level. Two-stage detectors like R-CNNs, in contrast, are forced to classify small patches of an input image in isolation, which is extra difficult when small patches of pixels look very different than training data. Single-stage detectors can draw on features from the entire input and are more robust to novel test-time inputs.

With that, you have reached the end of the computer vision section of this book! We have trained image classifiers, segmenters, and object detectors from scratch. We've developed a good intuition for how ConvNets work, the first major success of the deep learning era. We aren't quite done with images yet; you will see them again in chapter 17 when we start generating image output.

Summary

- Object detection identifies and locates objects within an image using bounding boxes. It's basically a weaker version of image segmentation, but one that can be run much more efficiently.
- There are two primary approaches to object detection:
  - Region-based Convolutional Neural Networks (R-CNNs), which are two-stage models that first propose regions of interest and then classify them with a ConvNet.
  - Single-stage detectors (like RetinaNet and YOLO), which perform both tasks in a single step. Single-stage detectors are generally faster and more efficient, making them suitable for real-time applications (e.g., self-driving cars).
- YOLO computes two separate outputs simultaneously during training—possible bounding boxes and a class probability map:
  - Each candidate bounding box is paired with a confidence score, which is trained to target the *Intersection over Union* of the predicted box and the ground truth box.
  - The class probability map classifies different regions of an image as belonging to different objects.
- RetinaNet builds on this idea by using a feature pyramid network (FPN), which combines features from multiple ConvNet layers to create feature maps at different scales, allowing it to more accurately detect objects of different sizes.

13 Timeseries forecasting

This chapter covers

- An overview of machine learning for timeseries
- Understanding recurrent neural networks (RNNs)
- Applying RNNs to a temperature forecasting example

This chapter tackles timeseries, where temporal order is everything. We'll focus on the most common and valuable timeseries task: forecasting. Using the recent past to predict the near future is a powerful capability, whether you're trying to anticipate energy demand, manage inventory, or simply forecast the weather.

13.1 Different kinds of timeseries tasks

A *timeseries* can be any data obtained via measurements at regular intervals, like the daily price of a stock, the hourly electricity consumption of a city, or the weekly sales of a store. Timeseries are everywhere, whether we're looking at natural phenomena (like seismic activity, the evolution of fish populations in a river, or the weather at a location) or human activity patterns (like visitors to a website, a country's GDP, or credit card transactions). Unlike the types of data you've encountered so far,

working with timeseries involves understanding the *dynamics* of a system—its periodic cycles, how it trends over time, its regular regime, and its sudden spikes.

By far, the most common timeseries-related task is *forecasting*: predicting what happens next in the series. Forecast electricity consumption a few hours in advance so you can anticipate demand, forecast revenue a few months in advance so you can plan your budget, forecast the weather a few days in advance so you can plan your schedule. Forecasting is what this chapter focuses on. But there's actually a wide range of other things you can do with timeseries, such as

- *Anomaly detection*—Detect anything unusual happening within a continuous data stream. Unusual activity on your corporate network? Might be an attacker. Unusual readings on a manufacturing line? Time for a human to go take a look. Anomaly detection is typically done via unsupervised learning, because you often don't know what kind of anomaly you're looking for, and thus you can't train on specific anomaly examples.
- *Classification*—Assign one or more categorical labels to a timeseries. For instance, given the timeseries of activity of a visitor on a website, classify whether the visitor is a bot or a human.
- *Event detection*—Identify the occurrence of a specific, expected event within a continuous data stream. A particularly useful application is "hotword detection," where a model monitors an audio stream and detects utterances like "OK, Google" or "Hey, Alexa."

In this chapter, you'll learn about recurrent neural networks (RNNs) and how to apply them to timeseries forecasting.

13.2 A temperature forecasting example

Throughout this chapter, all of our code examples will target a single problem: predicting the temperature 24 hours in the future, given a timeseries of hourly measurements of quantities such as atmospheric pressure and humidity, recorded over the recent past by a set of sensors on the roof of a building. As you will see, it's a fairly challenging problem!

We'll use this temperature forecasting task to highlight what makes timeseries data fundamentally different from the kinds of datasets you've encountered so far, to show that densely connected networks and convolutional networks aren't well equipped to deal with it, and to demonstrate a new kind of machine learning technique that really shines on this type of problem: recurrent neural networks (RNNs).

We'll work with a weather timeseries dataset recorded at the weather station at the Max Planck Institute for Biogeochemistry in Jena, Germany.[1] In this dataset, 14 different quantities (such as temperature, atmospheric pressure, humidity, wind direction, and so on) were recorded every 10 minutes, over several years. The original data goes back to 2003, but the subset of the data we'll download is limited to 2009–2016.

Let's start by downloading and uncompressing the data:

[1] Adam Erickson and Olaf Kolle, https://www.bgc-jena.mpg.de/wetter.

```
!wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
!unzip jena_climate_2009_2016.csv.zip
```

Let's look at the data.

Listing 13.1 Inspecting the data of the Jena weather dataset

```
import os

fname = os.path.join("jena_climate_2009_2016.csv")

with open(fname) as f:
    data = f.read()

lines = data.split("\n")
header = lines[0].split(",")
lines = lines[1:]
print(header)
print(len(lines))
```

This outputs a count of 420,551 lines of data (each line is a timestep: a record of a date and 14 weather-related values), as well as the following header:

```
["Date Time",
 "p (mbar)",
 "T (degC)",
 "Tpot (K)",
 "Tdew (degC)",
 "rh (%)",
 "VPmax (mbar)",
 "VPact (mbar)",
 "VPdef (mbar)",
 "sh (g/kg)",
 "H2OC (mmol/mol)",
 "rho (g/m**3)",
 "wv (m/s)",
 "max. wv (m/s)",
 "wd (deg)"]
```

Now, convert all 420,551 lines of data into NumPy arrays: one array for the temperature (in degrees Celsius), and another one for the rest of the data—the features we will use to predict future temperatures. Note that we discard the "Date Time" column.

Listing 13.2 Parsing the data

```
import numpy as np

temperature = np.zeros((len(lines),))
```

```
raw_data = np.zeros((len(lines), len(header) - 1))

for i, line in enumerate(lines):
    values = [float(x) for x in line.split(",")[1:]]
    temperature[i] = values[1]
    raw_data[i, :] = values[:]
```

We store column 1 in the temperature array.

We store all columns (including the temperature) in the raw_data array.

Figure 10.1 shows the plot of temperature (in degrees Celsius) over time. On this plot, you can clearly see the yearly periodicity of temperature—the data spans eight years.

Listing 13.3 Plotting the temperature timeseries

```
from matplotlib import pyplot as plt

plt.plot(range(len(temperature)), temperature)
```

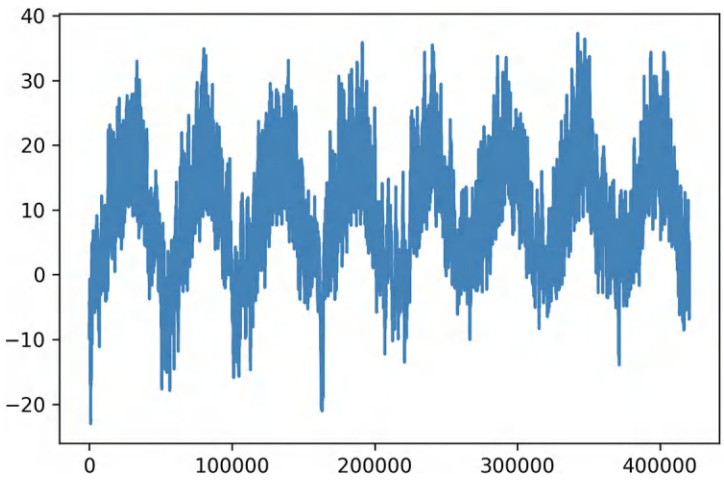


Figure 13.1 Temperature over the full temporal range of the dataset (°C)

Figure 10.2 shows a more narrow plot of the first 10 days of temperature data. Because the data is recorded every 10 minutes, you get 24 × 6 = 144 data points per day.

Listing 13.4 Plotting the first 10 days of the temperature timeseries

```
plt.plot(range(1440), temperature[:1440])
```

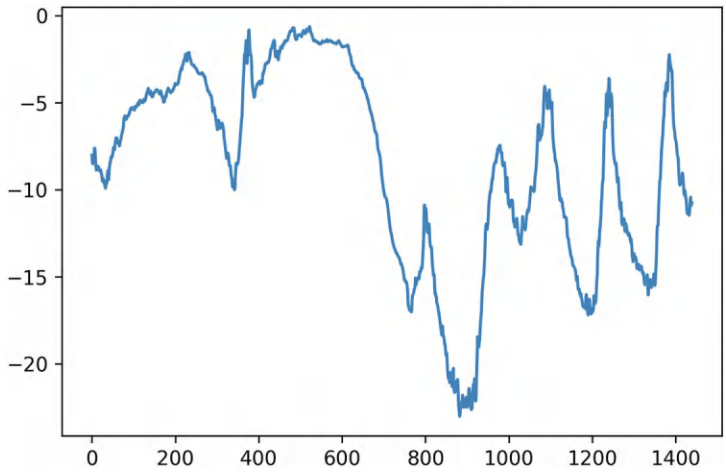


Figure 13.2 Temperature over the first 10 days of the dataset (°C)

On this plot, you can see daily periodicity, especially evident for the last four days. Also note that this 10-day period must be coming from a fairly cold winter month.

> **NOTE** Periodicity over multiple timescales is an important and very common property of timeseries data. Whether you're looking at the weather, mall parking occupancy, traffic to a website, sales of a grocery store, or steps logged in a fitness tracker, you'll see daily cycles and yearly cycles (human-generated data also tends to feature weekly cycles). When exploring your data, make sure to look for these patterns.

With our dataset, if you were trying to predict average temperature for the next month given a few months of past data, the problem would be easy, due to the reliable year-scale periodicity of the data. But looking at the data over a scale of days, the temperature looks a lot more chaotic. Is this timeseries predictable at a daily scale? Let's find out.

In all our experiments, we'll use the first 50% of the data for training, the following 25% for validation, and the last 25% for testing. When working with timeseries data, it's important to use validation and test data that is more recent than the training data because you're trying to predict the future given the past, not the reverse, and your validation/test splits should reflect this temporal ordering. Some problems happen to be considerably simpler if you reverse the time axis!

Listing 13.5 Computing the number of samples for each data split

```
>>> num_train_samples = int(0.5 * len(raw_data))
>>> num_val_samples = int(0.25 * len(raw_data))
>>> num_test_samples = len(raw_data) - num_train_samples - num_val_samples
```

```
>>> print("num_train_samples:", num_train_samples)
>>> print("num_val_samples:", num_val_samples)
>>> print("num_test_samples:", num_test_samples)
num_train_samples: 210225
num_val_samples: 105112
num_test_samples: 105114
```

13.2.1 Preparing the data

The exact formulation of the problem will be as follows: given data covering the previous five days and sampled once per hour, can we predict the temperature in 24 hours?

First, let's preprocess the data to a format a neural network can ingest. This is easy: the data is already numerical, so you don't need to do any vectorization. But each timeseries in the data is on a different scale (for example, atmospheric pressure, measured in mbar, is around 1,000, while H2OC, measured in millimoles per mole, is around 3). We'll normalize each timeseries independently so that they all take small values on a similar scale. We're going to use the first 210,225 timesteps as training data, so we'll compute the mean and standard deviation only on this fraction of the data.

Listing 13.6 Normalizing the data

```
mean = raw_data[:num_train_samples].mean(axis=0)
raw_data -= mean
std = raw_data[:num_train_samples].std(axis=0)
raw_data /= std
```

Next, let's create a `Dataset` object that yields batches of data from the past five days along with a target temperature 24 hours in the future. Because the samples in the dataset are highly redundant (sample `N` and sample `N + 1` will have most of their timesteps in common), it would be wasteful to explicitly allocate memory for every sample. Instead, we'll generate the samples on the fly while only keeping in memory the original `raw_data` and `temperature` arrays, and nothing more.

We could easily write a Python generator to do this, but there's a built-in dataset utility in Keras that does just that (`timeseries_dataset_from_array()`), so we can save ourselves some work by using it. You can generally use it for any kind of timeseries forecasting task.

Understanding timeseries_dataset_from_array()

To understand what `timeseries_dataset_from_array()` does, let's take a look at a simple example. The general idea is that you provide an array of timeseries data (the `data` argument), and `timeseries_dataset_from_array` gives you windows extracted from the original timeseries (we'll call them "sequences").

Let's say you're using `data = [0 1 2 3 4 5 6]` and `sequence_length=3`; then `timeseries_dataset_from_array` will generate the following samples: `[0 1 2]`, `[1 2 3]`, `[2 3 4]`, `[3 4 5]`, `[4 5 6]`.

You can also pass a `target` array to `timeseries_dataset_from_array()`. The first entry of the `targets` array should match the desired target for the first sequence that will be generated from the `data` array. So if you're doing timeseries forecasting, simply use as `targets` the same array as for `data`, offset by some amount.

For instance, with `data = [0 1 2 3 4 5 6 ...]` and `sequence_length=3`, you could create a dataset to predict the next step in the series by passing `targets = [3 4 5 6 ...]`. Let's try it:

```
import numpy as np
import keras

int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)

for inputs, targets in dummy_dataset:
    for i in range(inputs.shape[0]):
        print([int(x) for x in inputs[i]], int(targets[i]))
```

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

Generate an array of sorted integers from 0 to 9.

The sequences will be 3 steps long.

The sequences will be batched in batches of size 2.

The target for the sequence that starts at data[N] will be data[N + 3].

This bit of code prints the following results:

```
[0, 1, 2] 3
[1, 2, 3] 4
[2, 3, 4] 5
[3, 4, 5] 6
[4, 5, 6] 7
```

We'll use `timeseries_dataset_from_array` to instantiate three datasets: one for training, one for validation, and one for testing.

We'll use the following parameter values:

- `sampling_rate = 6` — Observations will be sampled at one data point per hour: we will only keep one data point out of six.
- `sequence_length = 120` — Observations will go back five days (120 hours).
- `delay = sampling_rate * (sequence_length + 24 - 1)` — The target for a sequence will be the temperature 24 hours after the end of the sequence.

- `start_index = 0` and `end_index = num_train_samples`—For the training dataset, to only use the first 50% of the data.
- `start_index = num_train_samples` and `end_index = num_train_samples + num_val_samples`—For the validation dataset, to only use the next 25% of the data.
- `start_index = num_train_samples + num_val_samples`—For the test dataset, to use the remaining samples.

Listing 13.7 Instantiating datasets for training, validation, and testing

```
sampling_rate = 6
sequence_length = 120
delay = sampling_rate * (sequence_length + 24 - 1)
batch_size = 256

train_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=0,
    end_index=num_train_samples,
)

val_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples,
    end_index=num_train_samples + num_val_samples,
)

test_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples + num_val_samples,
)
```

Each dataset yields a tuple `(samples, targets)`, where `samples` is a batch of 256 samples, each containing 120 consecutive hours of input data, and `targets` is the corresponding array of 256 target temperatures. Note that the samples are randomly shuffled, so two

consecutive sequences in a batch (like `samples[0]` and `samples[1]`) aren't necessarily temporally close.

Listing 13.8 Inspecting the dataset

```
>>> for samples, targets in train_dataset:
>>>     print("samples shape:", samples.shape)
>>>     print("targets shape:", targets.shape)
>>>     break
samples shape: (256, 120, 14)
targets shape: (256,)
```

13.2.2 A commonsense, non-machine-learning baseline

Before you start using black box, deep learning models to solve the temperature prediction problem, let's try a simple, commonsense approach. It will serve as a sanity check, and it will establish a baseline that you'll have to beat to demonstrate the usefulness of more advanced, machine learning models. Such commonsense baselines can be useful when you're approaching a new problem for which there is no known solution (yet). A classic example is that of unbalanced classification tasks, where some classes are much more common than others. If your dataset contains 90% instances of class A and 10% instances of class B, then a commonsense approach to the classification task is to always predict "A" when presented with a new sample. Such a classifier is 90% accurate overall, and any learning-based approach should therefore beat this 90% score to demonstrate usefulness. Sometimes, such elementary baselines can prove surprisingly hard to beat.

In this case, the temperature timeseries can safely be assumed to be continuous (the temperatures tomorrow are likely to be close to the temperatures today) as well as periodic with a daily period. Thus, a commonsense approach is to always predict that the temperature 24 hours from now will be equal to the temperature right now. Let's evaluate this approach, using the mean absolute error (MAE) metric, defined as follows:

```
np.mean(np.abs(preds - targets))
```

Here's the evaluation loop.

Listing 13.9 Computing the commonsense baseline MAE

```
def evaluate_naive_method(dataset):
    total_abs_err = 0.0
    samples_seen = 0
    for samples, targets in dataset:
```

```
        preds = samples[:, -1, 1] * std[1] + mean[1]
        total_abs_err += np.sum(np.abs(preds - targets))
        samples_seen += samples.shape[0]
    return total_abs_err / samples_seen

print(f"Validation MAE: {evaluate_naive_method(val_dataset):.2f}")
print(f"Test MAE: {evaluate_naive_method(test_dataset):.2f}")
```

The temperature feature is at column 1, so samples[:, -1, 1] is the last temperature measurement in the input sequence. Recall that we normalized our features to retrieve a temperature in Celsius degrees, we need to un-normalize it, by multiplying it by the standard deviation and adding back the mean.

This commonsense baseline achieves a validation MAE of 2.44 degrees Celsius, and a test MAE of 2.62 degrees Celsius. So if you always assume that the temperature 24 hours in the future will be the same as it is now, you will be off by two and a half degrees on average. It's not too bad, but you probably won't launch a weather forecasting service based on this heuristic. Now, the game is to use your knowledge of deep learning to do better.

13.2.3 Let's try a basic machine learning model

In the same way that it's useful to establish a commonsense baseline before trying machine learning approaches, it's useful to try simple, cheap, machine learning models (such as small, densely connected networks) before looking into complicated and computationally expensive models such as RNNs. This is the best way to make sure any further complexity you throw at the problem is legitimate and delivers real benefits.

Listing 13.10 shows a fully connected model that starts by flattening the data and then runs it through two `Dense` layers. Note the lack of activation function on the last `Dense` layer, which is typical for a regression problem. We use mean squared error (MSE) as the loss, rather than MAE, because unlike MAE, it's smooth around zero, a useful property for gradient descent. We will monitor MAE by adding it as a metric in `compile()`.

Listing 13.10 Training and evaluating a densely connected model

```
import keras
from keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras", save_best_only=True)
]
```

We use a callback to save the best-performing model.

```
]
model.compile(optimizer="adam", loss="mse", metrics=["mae"])
history = model.fit(
    train_dataset,
    epochs=10,
    validation_data=val_dataset,
    callbacks=callbacks,
)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

Reloads the best model and evaluates it on the test data

Let's display the loss curves for validation and training (see figure 13.3).

Listing 13.11 Plotting results

```
import matplotlib.pyplot as plt

loss = history.history["mae"]
val_loss = history.history["val_mae"]
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, "r--", label="Training MAE")
plt.plot(epochs, val_loss, "b", label="Validation MAE")
plt.title("Training and validation MAE")
plt.legend()
plt.show()
```

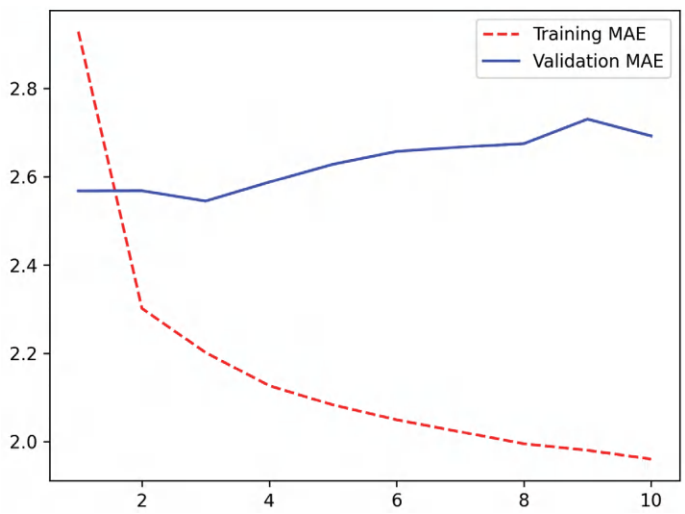


Figure 13.3 Training and validation MAE on the Jena temperature forecasting task with a simple, densely connected network

Some of the validation losses are close to the no-learning baseline, but not reliably. This goes to show the merit of having this baseline in the first place: it turns out to be not easy to outperform. Your common sense contains a lot of valuable information to which a machine learning model doesn't have access.

You may wonder, if a simple, well-performing model exists to go from the data to the targets (the commonsense baseline), why doesn't the model you're training find it and improve on it? Well, the space of models in which you're searching for a solution—that is, your hypothesis space—is the space of all possible two-layer networks with the configuration you defined. The commonsense heuristic is just one model among millions that can be represented in this space. It's like looking for a needle in a haystack. Just because a good solution technically exists in your hypothesis space doesn't mean you'll be able to find it via gradient descent.

That's a pretty significant limitation of machine learning in general: unless the learning algorithm is hardcoded to look for a specific kind of simple model, it can sometimes fail to find a simple solution to a simple problem. That's why using good feature engineering and relevant architecture priors is essential: you need to be precisely telling your model what it should be looking for.

13.2.4 Let's try a 1D convolutional model

Speaking of using the right architecture priors: since our input sequences feature daily cycles, perhaps a convolutional model could work? A temporal ConvNet could reuse the same representations across different days, much like a spatial ConvNet can reuse the same representations across different locations in an image.

You already know about the `Conv2D` and `SeparableConv2D` layers, which see their inputs through small windows that swipe across 2D grids. There are also 1D and even 3D versions of these layers: `Conv1D`, `SeparableConv1D`, and `Conv3D`.[2] The `Conv1D` layer relies on 1D windows that slide across input sequences, and the `Conv3D` layer relies on cubic windows that slide across input volumes.

You can thus build 1D ConvNets, strictly analogous to 2D ConvNets. They're a great fit for any sequence data that follows the translation invariance assumption (meaning that if you slide a window over the sequence, the content of the window should follow the same properties independently of the location of the window).

Let's try one on our temperature forecasting problem. We'll pick an initial window length of 24, so that we look at 24 hours of data at a time (one cycle). As we downsample the sequences (via `MaxPooling1D` layers), we'll reduce the window size accordingly (figure 13.4):

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
```

[2] There isn't a `SeparableConv3D` layer, not for any theoretical reason, but simply because we haven't implemented it.

```
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras", save_best_only=True)
]
model.compile(optimizer="adam", loss="mse", metrics=["mae"])
history = model.fit(
    train_dataset,
    epochs=10,
    validation_data=val_dataset,
    callbacks=callbacks,
)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

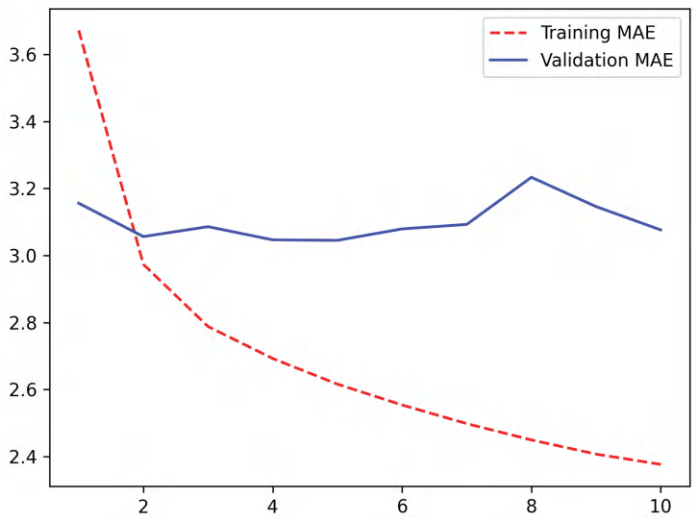


Figure 13.4 Training and validation MAE on the Jena temperature forecasting task with a 1D ConvNet

As it turns out, this model performs even worse than the densely connected one, only achieving a validation MAE of about 2.9 degrees, far from the commonsense baseline. What went wrong here? Two things:

- First, weather data doesn't quite respect the translation invariance assumption. While the data does feature daily cycles, data from a morning follows different

properties than data from an evening or from the middle of the night. Weather data is only translation-invariant for a very specific timescale.

- Second, order in our data matters—a lot. The recent past is far more informative for predicting the next day's temperature than data from five days ago. A 1D ConvNet is not able to make use of this fact. In particular, our max pooling and global average pooling layers are largely destroying order information.

13.3 Recurrent neural networks

Neither the fully connected approach nor the convolutional approach did well, but that doesn't mean machine learning isn't applicable to this problem. The densely connected approach first flattened the timeseries, which removed the notion of time from the input data. The convolutional approach treated every segment of the data in the same way, even applying pooling, which destroyed order information. Let's instead look at the data as what it is: a sequence, where causality and order matter.

There's a family of neural network architectures that were designed specifically for this use case: recurrent neural networks. Among them, the Long Short-Term Memory (LSTM) layer in particular has long been very popular. We'll see in a minute how these models work—but let's start by giving the LSTM layer a try.

Listing 13.12 A simple LSTM-based model

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm.keras", save_best_only=True)
]
model.compile(optimizer="adam", loss="mse", metrics=["mae"])
history = model.fit(
    train_dataset,
    epochs=10,
    validation_data=val_dataset,
    callbacks=callbacks,
)

model = keras.models.load_model("jena_lstm.keras")
print("Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

Figure 13.5 shows the results. Much better! We achieve a validation MAE as low as 2.39 degrees and a test MAE of 2.55 degrees. The LSTM-based model can finally beat the commonsense baseline (albeit just by a bit, for now), demonstrating the value of machine learning on this task.

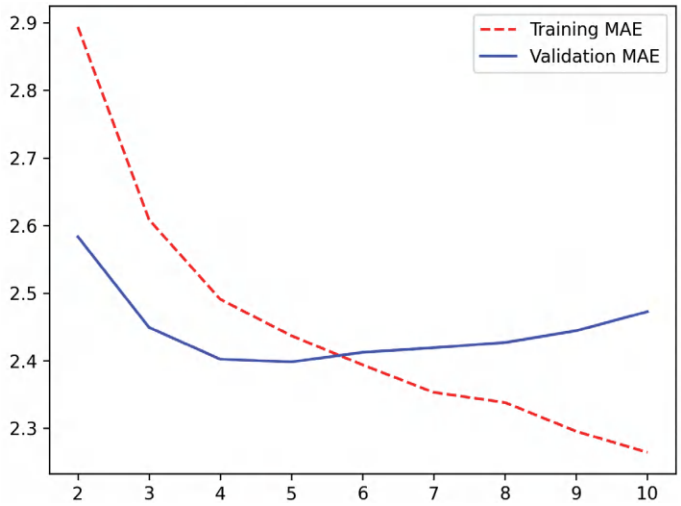


Figure 13.5 Training and validation MAE on the Jena temperature forecasting task with an LSTM-based model. (Note that we omit epoch 1 on this graph because the high training MAE (7.75) at epoch 1 would distort the scale.)

But why did the LSTM model perform markedly better than the densely connected one or the ConvNet? And how can we further refine the model? To answer this, let's take a closer look at recurrent neural networks.

13.3.1 Understanding recurrent neural networks

A major characteristic of all neural networks you've seen so far, such as densely connected networks and ConvNets, is that they have no memory. Each input shown to them is processed independently, with no state kept between inputs. With such networks, to process a sequence or a temporal series of data points, you have to show the entire sequence to the network at once: turn it into a single data point. For instance, this is what we did in the densely connected network example: we flattened our five days of data into a single large vector and processed it in one go. Such networks are called *feedforward networks*.

In contrast, as you're reading the present sentence, you're processing it word by word—or rather, eye saccade by eye saccade—while keeping memories of what came before; this gives you a fluid representation of the meaning conveyed by this sentence. Biological intelligence processes information incrementally while maintaining an internal model of what it's processing, built from past information and constantly updated as new information comes in.

A *recurrent neural network* (RNN) adopts the same principle, albeit in an extremely simplified version: it processes sequences by iterating through the sequence elements and maintaining a *state* containing information relative to what it has seen so far. In effect, an RNN is a type of neural network that has an internal *loop* (see figure 13.6).

The state of the RNN is reset between processing two different, independent sequences (such as two samples in a batch), so you still consider one sequence to be a single data point: a single input to the network. What changes is that this data point is no longer processed in a single step; rather, the network internally loops over sequence elements.

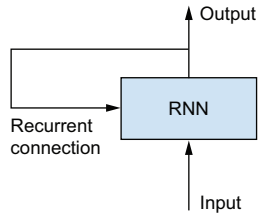


Figure 13.6 A recurrent network: a network with a loop

To make these notions of *loop* and *state* clear, let's implement the forward pass of a toy RNN. This RNN takes as input a sequence of vectors, which we'll encode as a rank-2 tensor of size `(timesteps, input_features)`. It loops over timesteps, and at each timestep, it considers its current state at `t` and the input at `t` (of shape `(input_features,)`), and combines them to obtain the output at `t`. We'll then set the state for the next step to be this previous output. For the first timestep, the previous output isn't defined; hence, there is no current state. So we'll initialize the state as an all-zero vector called the *initial* state of the network.

In pseudocode, this is the RNN.

Listing 13.13 Pseudocode RNN

```
state_t = 0
for input_t in input_sequence:
    output_t = f(input_t, state_t)
    state_t = output_t
```

The state at t

Iterates over sequence elements

The previous output becomes the state for the next iteration.

You can even flesh out the function `f`: the transformation of the input and state into an output will be parameterized by two matrices, `W` and `U`, and a bias vector. It's similar to the transformation operated by a densely connected layer in a feedforward network.

Listing 13.14 More detailed pseudocode for the RNN

```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

To make these notions absolutely unambiguous, let's write a naive NumPy implementation of the forward pass of the simple RNN.

Listing 13.15 NumPy implementation of a simple RNN

```
import numpy as np

timesteps = 100
input_features = 32
```

Number of timesteps in the input sequence

Dimensionality of the input feature space

```
output_features = 64
inputs = np.random.random((timesteps, input_features))
state_t = np.zeros((output_features,))
W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))
successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
    successive_outputs.append(output_t)
    state_t = output_t
final_output_sequence = np.concatenate(successive_outputs, axis=0)
```

Initial state: an all-zero vector

Dimensionality of the output feature space

Input data: random noise for the sake of the example

Creates random weight matrices

input_t is a vector of shape (input_features,).

Combines the input with the current state (the previous output) to obtain the current output. We use tanh to add nonlinearity (we could use any other activation function).

The final output is a rank-2 tensor of shape (timesteps, output_features).

Updates the state of the network for the next timestep

Stores this output in a list

Easy enough: in summary, an RNN is a `for` loop that reuses quantities computed during the previous iteration of the loop, nothing more. Of course, there are many different RNNs fitting this definition that you could build—this example is one of the simplest RNN formulations. RNNs are characterized by their step function, such as the following function in this case (see figure 13.7):

```
output_t = tanh(matmul(input_t, W) + matmul(state_t, U) + b)
```

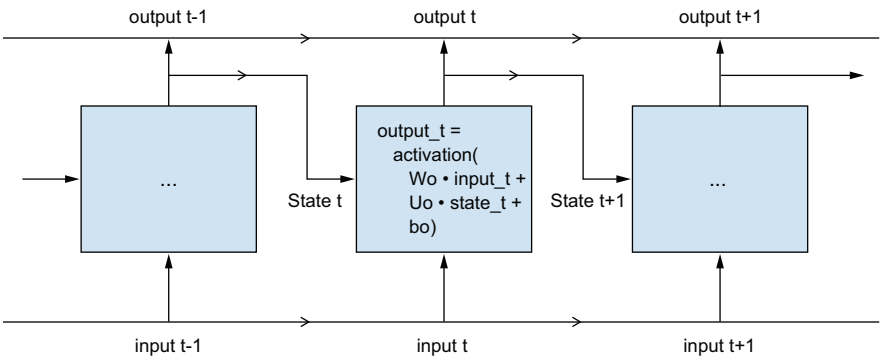


Figure 13.7 A simple RNN, unrolled over time

NOTE In this example, the final output is a rank-2 tensor of shape `(timesteps, output_features)`, where each timestep is the output of the loop at time `t`. Each timestep `t` in the output tensor contains information about timesteps `0` to `t` in the input sequence—about the entire past. For this reason, in many cases, you don't need this full sequence of outputs; you just need the last output (`output_t` at the end of the loop), because it already contains information about the entire sequence.

13.3.2 A recurrent layer in Keras

The process you just naively implemented in NumPy corresponds to an actual Keras layer—the `SimpleRNN` layer.

There is one minor difference: `SimpleRNN` processes batches of sequences, like all other Keras layers, not a single sequence as in the NumPy example. This means it takes inputs of shape `(batch_size, timesteps, input_features)` rather than `(timesteps, input_features)`. When specifying the `shape` argument of your initial `Input()`, note that you can set the `timesteps` entry to `None`, which enables your network to process sequences of arbitrary length.

Listing 13.16 An RNN layer that can process sequences of any length

```
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
```

This is especially useful if your model is meant to process sequences of variable length. However, if all of your sequences have the same length, I recommend specifying a complete input shape, since it enables `model.summary()` to display output length information, which is always nice, and it can unlock some performance optimizations (see the sidebar "On RNN runtime performance" in section 13.3.4).

All recurrent layers in Keras (`SimpleRNN`, `LSTM`, and `GRU`) can be run in two different modes: they can return either full sequences of successive outputs for each timestep (a rank-3 tensor of shape `(batch_size, timesteps, output_features)`) or only the last output for each input sequence (a rank-2 tensor of shape `(batch_size, output_features)`). These two modes are controlled by the `return_sequences` constructor argument. Let's look at an example that uses `SimpleRNN` and returns only the output at the last timestep.

Listing 13.17 An RNN layer that returns only its last output step

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)
>>> print(outputs.shape)
(None, 16)
```

Note that return_sequences=False is the default.

The following example returns the full output sequence.

Listing 13.18 An RNN layer that returns its full output sequence

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)
>>> print(outputs.shape)
(None, 120, 16)
```

Sets return_sequences to True

It's sometimes useful to stack several recurrent layers one after the other to increase the representational power of a network. In such a setup, you have to get all of the intermediate layers to return the full sequence of outputs.

Listing 13.19 Stacking RNN layers

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs)
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
```

Now, in practice, you'll rarely work with the `SimpleRNN` layer. It's generally too simplistic to be of real use. In particular, `SimpleRNN` has a major issue: although it should theoretically be able to retain at time `t` information about inputs seen many timesteps before, in practice, such long-term dependencies prove impossible to learn. This is due to the *vanishing gradients problem*, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable. The theoretical reasons for this effect were studied by Hochreiter, Schmidhuber, and Bengio in the early 1990s.[3]

Thankfully, `SimpleRNN` isn't the only recurrent layer available in Keras. There are two others: `LSTM` and `GRU`, which were designed to address these issues.

Let's consider the `LSTM` layer. The underlying Long Short-Term Memory (LSTM) algorithm was developed by Hochreiter and Schmidhuber in 1997;[4] it was the culmination of their research on the vanishing gradients problem.

This layer is a variant of the `SimpleRNN` layer you already know about; it adds a way to carry information across many timesteps. Imagine a conveyor belt running parallel to the sequence you're processing. Information from the sequence can jump onto the conveyor belt at any point, be transported to a later timestep, and jump off, intact, when you need it. This is essentially what LSTM does: it saves information for later, thus preventing older signals from gradually vanishing during processing. This should remind

[3] See, for example, Yoshua Bengio, Patrice Simard, and Paolo Frasconi, "Learning Long-Term Dependencies with Gradient Descent Is Difficult," *IEEE Transactions on Neural Networks* 5, no. 2 (1994).

[4] Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory," *Neural Computation* 9, no. 8 (1997).

you of *residual connections*, which you learned about in chapter 9: it's pretty much the same idea.

To understand this process in detail, let's start from the `SimpleRNN` cell (see figure 13.8). Because you'll have a lot of weight matrices, index the `W` and `U` matrices in the cell with the letter `o` (`Wo` and `Uo`) for *output*.

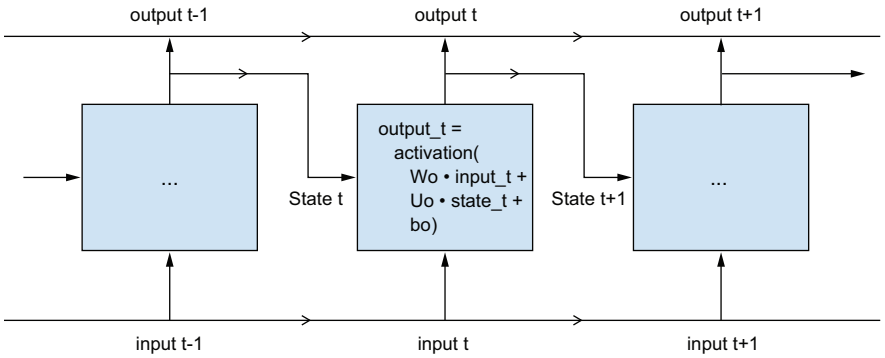


Figure 13.8 The starting point of an `LSTM` layer: a `SimpleRNN`

Let's add to this picture an additional data flow that carries information across timesteps. Call its values at different timesteps `Ct`, where *C* stands for *carry*. This information will have the following effect on the cell: it will be combined with the input connection and the recurrent connection (via a dense transformation: a dot product with a weight matrix followed by a bias add and the application of an activation function), and it will affect the state being sent to the next timestep (via an activation function and a multiplication operation). Conceptually, the carry dataflow is a way to modulate the next output and the next state (see figure 13.9). Simple so far.

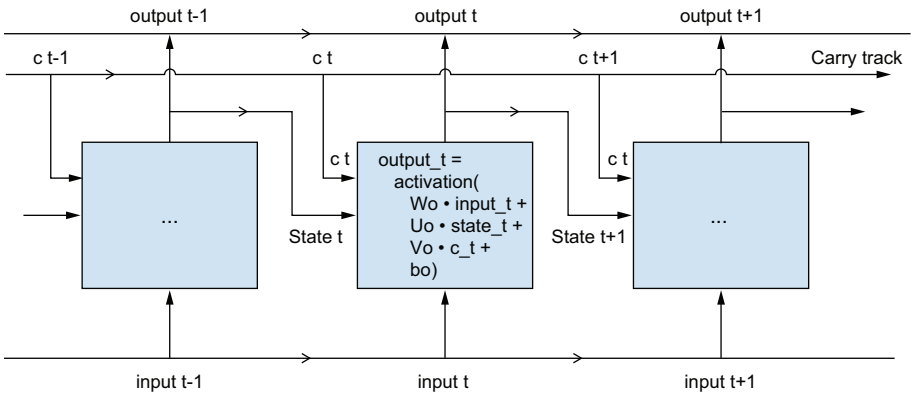


Figure 13.9 Going from a SimpleRNN to an LSTM: adding a carry track

Now the subtlety: the way the next value of the carry dataflow is computed. It involves three distinct transformations. All three have the form of a `SimpleRNN` cell:

```
y = activation(dot(state_t, U) + dot(input_t, W) + b)
```

But all three transformations have their own weight matrices, which you'll index with the letters `i`, `f`, and `k`. Here's what you have so far (it may seem a bit arbitrary, but bear with us).

Listing 13.20 Pseudocode details of the LSTM architecture (1/2)

```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(C_t, Vo) + bo)
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
```

You obtain the new carry state (the next `c_t`) by combining `i_t`, `f_t`, and `k_t`.

Listing 13.21 Pseudocode details of the LSTM architecture (2/2)

```
c_t+1 = i_t * k_t + c_t * f_t
```

Add this as shown in figure 13.10. And that's it. Not so complicated—merely a tad complex.

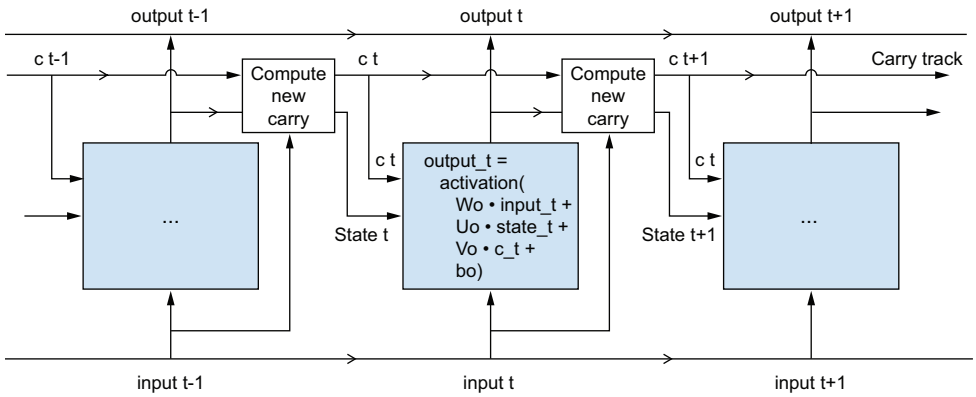


Figure 13.10 Anatomy of an `LSTM`

If you want to get philosophical, you can interpret what each of these operations is meant to do. For instance, you can say that multiplying `c_t` and `f_t` is a way to

deliberately forget irrelevant information in the carry dataflow. Meanwhile, `i_t` and `k_t` provide information about the present, updating the carry track with new information. But at the end of the day, these interpretations don't mean much because what these operations *actually* do is determined by the contents of the weights parameterizing them, and the weights are learned in an end-to-end fashion, starting over with each training round, making it impossible to credit this or that operation with a specific purpose. The specification of an RNN cell (as just described) determines your hypothesis space—the space in which you'll search for a good model configuration during training—but it doesn't determine what the cell does; that is up to the cell weights. The same cell with different weights can be doing very different things. So the combination of operations making up an RNN cell is better interpreted as a set of *constraints* on your search, not as a *design* in an engineering sense.

Arguably, the choice of such constraints—the question of how to implement RNN cells—is better left to optimization algorithms (like genetic algorithms or reinforcement learning processes) than to human engineers. In the future, that's how we'll build our models. In summary, you don't need to understand anything about the specific architecture of an LSTM cell; as a human, it shouldn't be your job to understand it. Just keep in mind what the LSTM cell is meant to do: allow past information to be reinjected at a later time, thus fighting the vanishing gradients problem.

13.3.3 Getting the most out of recurrent neural networks

By this point, you've learned

- What RNNs are and how they work
- What an LSTM is and why it works better on long sequences than a naive RNN
- How to use Keras RNN layers to process sequence data

Next, we'll review a number of more advanced features of RNNs, which can help you get the most out of your deep learning sequence models. By the end of the section, you'll know most of what there is to know about using recurrent networks with Keras.

We'll cover the following:

- *Recurrent dropout*—This is a variant of dropout, used to fight overfitting in recurrent layers.
- *Stacking recurrent layers*—This increases the representational power of the model (at the cost of higher computational loads).
- *Bidirectional recurrent layers*—These present the same information to a recurrent network in different ways, increasing accuracy and mitigating forgetting issues.

We'll use these techniques to refine our temperature forecasting RNN.

13.3.4 Using recurrent dropout to fight overfitting

Let's go back to the LSTM-based model we used earlier in the chapter—our first model able to beat the commonsense baseline. If you look at the training and validation

curves, it's evident that the model is quickly overfitting, despite only having very few units: the training and validation losses start to diverge considerably after a few epochs. You're already familiar with a classic technique for fighting this phenomenon: dropout, which randomly zeros out input units of a layer to break happenstance correlations in the training data that the layer is exposed to. But how to correctly apply dropout in recurrent networks isn't a trivial question.

It has long been known that applying dropout before a recurrent layer hinders learning rather than helping with regularization. In 2015, Yarin Gal, as part of his PhD thesis on Bayesian deep learning,[5] determined the proper way to use dropout with a recurrent network: the same dropout mask (the same pattern of dropped units) should be applied at every timestep, instead of a dropout mask that varies randomly from timestep to timestep. What's more, to regularize the representations formed by the recurrent gates of layers such as `GRU` and `LSTM`, a temporally constant dropout mask should be applied to the inner recurrent activations of the layer (a recurrent dropout mask). Using the same dropout mask at every timestep allows the network to properly propagate its learning error through time; a temporally random dropout mask would disrupt this error signal and be harmful to the learning process.

Yarin Gal did his research using Keras and helped build this mechanism directly into Keras recurrent layers. Every recurrent layer in Keras has two dropout-related arguments: `dropout`, a float specifying the dropout rate for input units of the layer, and `recurrent_dropout`, specifying the dropout rate of the recurrent units. Let's add recurrent dropout to the `LSTM` layer of our first LSTM example and see how doing so affects overfitting.

Thanks to dropout, we won't need to rely as much on network size for regularization, so we'll use an `LSTM` layer with twice as many units, which should hopefully be more expressive (without dropout, this network would have started overfitting right away—try it). Because networks being regularized with dropout always take much longer to fully converge, we'll train the model for five times as many epochs.

Listing 13.22 Training and evaluating a dropout-regularized LSTM

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint(
        "jena_lstm_dropout.keras", save_best_only=True
    )
]
```

To regularize the Dense layer, we also add a Dropout layer after the LSTM.

[5] See Yarin Gal, "Uncertainty in Deep Learning (PhD Thesis)," October 13, 2016, https://www.cs.ox.ac.uk/people/yarin.gal/website/blog_2248.html.

```
model.compile(optimizer="adam", loss="mse", metrics=["mae"])
history = model.fit(
    train_dataset,
    epochs=50,
    validation_data=val_dataset,
    callbacks=callbacks,
)
```

Figure 13.11 shows the results. Success! We're no longer overfitting during the first 20 epochs. We achieve a validation MAE as low as 2.27 degrees (7% improvement over the no-learning baseline) and a test MAE of 2.45 degrees (6.5% improvement over the baseline). Not too bad.

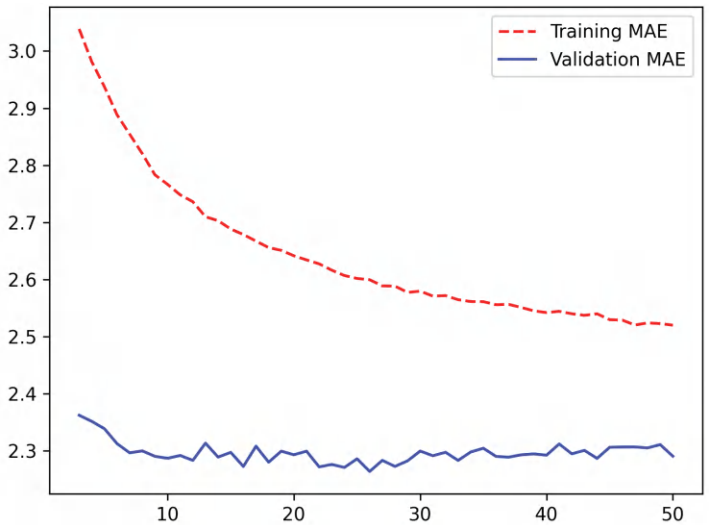


Figure 13.11 Training and validation loss on the Jena temperature forecasting task with a dropout-regularized LSTM

On RNN runtime performance

Recurrent models with very few parameters, like the ones in this chapter, tend to be significantly faster on a multicore CPU than on GPU because they only involve small matrix multiplications, and the chain of multiplications is not well parallelizable due to the presence of a `for` loop. But larger RNNs can greatly benefit from a GPU runtime.

When using a Keras `LSTM` or `GRU` layer on GPU with default keyword arguments, your layer will be using a *cuDNN kernel*, a highly optimized, low-level, NVIDIA-provided implementation of the underlying algorithm (we've mentioned those in the previous chapter). As usual, cuDNN kernels are a mixed blessing: they're fast, but inflexible—if you try doing anything not supported by the default kernel, you will suffer a dramatic

slowdown, which more or less forces you stick to what NVIDIA happens to provide. For instance, recurrent dropout isn't supported by the LSTM and GRU cuDNN kernels, so adding it to your layers forces the runtime to fall back to the regular TensorFlow implementation, which is generally two to five times slower on GPU (even though its computational cost is the same).

As a way to speed up your RNN layer when you can't use cuDNN, you can try *unrolling* it. Unrolling a `for` loop consists of removing the loop and simply in-lining its content *N* times. In the case of the `for` loop of an RNN, unrolling can help TensorFlow optimize the underlying computation graph. However, it will also considerably increase the memory consumption of your RNN—as such, it's only viable for relatively small sequences (around 100 steps or fewer). Also, you can only do this if the number of timesteps in the data is known in advance by the model (that is, if you pass a `shape` without any `None` entries to your initial `Input()`). It works like this:

sequence_length cannot be None.

```
inputs = keras.Input(shape=(sequence_length, num_features))
x = layers.LSTM(32, recurrent_dropout=0.2, unroll=True)(inputs)
```

Passes unroll=True to enable unrolling

13.3.5 Stacking recurrent layers

Because you're no longer overfitting, but seem to have hit a performance bottleneck, you should consider increasing the capacity and expressive power of the network. Recall the description of the universal machine learning workflow: it's generally a good idea to increase the capacity of your model until overfitting becomes the primary obstacle (assuming you're already taking basic steps to mitigate overfitting, such as using dropout). As long as you aren't overfitting too badly, you're likely under capacity.

Increasing network capacity is typically done by increasing the number of units in the layers or adding more layers. Recurrent layer stacking is a classic way to build more powerful recurrent networks: for instance, not too long ago the Google Translate algorithm was powered by a stack of seven large `LSTM` layers—that's huge.

To stack recurrent layers on top of each other in Keras, all intermediate layers should return their full sequence of outputs (a rank-3 tensor) rather than their output at the last timestep. As you've already learned, this is done by specifying `return_sequences=True`.

In the following example, we'll try a stack of two dropout-regularized recurrent layers. For a change, we'll use `GRU` layers instead of `LSTM`. A Gated Recurrent Unit (GRU) is very similar to an LSTM—you can think of it as a slightly simpler, streamlined version of the LSTM architecture. It was introduced in 2014 by Cho et al. just when recurrent networks were starting to gain interest anew in the then-tiny research community.[6]

[6] See Cho et al., "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches," 2014, https://arxiv.org/abs/1409.1259.

Listing 13.23 Training and evaluating a dropout-regularized, stacked GRU model

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint(
        "jena_stacked_gru_dropout.keras", save_best_only=True
    )
]
model.compile(optimizer="adam", loss="mse", metrics=["mae"])
history = model.fit(
    train_dataset,
    epochs=50,
    validation_data=val_dataset,
    callbacks=callbacks,
)
model = keras.models.load_model("jena_stacked_gru_dropout.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

Figure 13.12 shows the results. We achieve a test MAE of 2.39 degrees (an 8.8% improvement over the baseline). You can see that the added layer does improve the results a bit, though not dramatically. You may be seeing diminishing returns from increasing network capacity at this point.

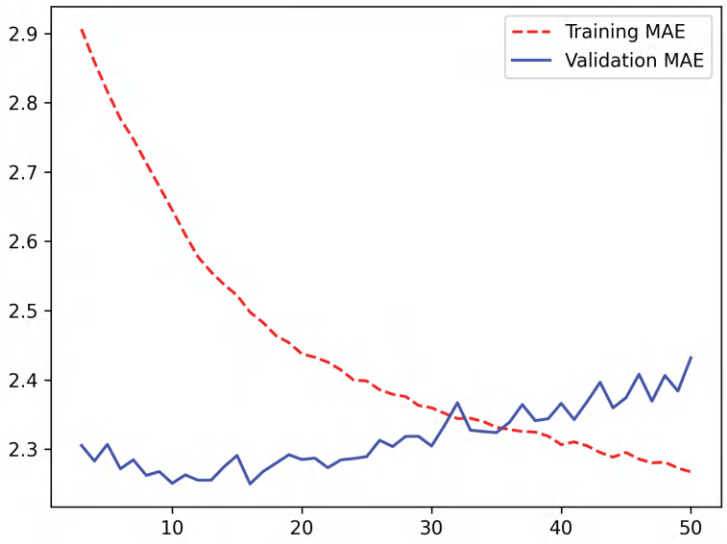


Figure 13.12 Training and validation loss on the Jena temperature forecasting task with a stacked GRU network

13.3.6 Using bidirectional RNNs

The last technique introduced in this section is called *bidirectional RNNs*. A bidirectional RNN is a common RNN variant that can offer greater performance than a regular RNN on certain tasks. It's frequently used in natural language processing—you could call it the Swiss Army knife of deep learning for natural language processing.

RNNs are notably order dependent: they process the timesteps of their input sequences in order, and shuffling or reversing the timesteps can completely change the representations the RNN extracts from the sequence. This is precisely the reason they perform well on problems where order is meaningful, such as the temperature forecasting problem. A bidirectional RNN exploits the order sensitivity of RNNs: it consists of using two regular RNNs, such as the `GRU` and `LSTM` layers you're already familiar with, each of which processes the input sequence in one direction (chronologically and antichronologically), and then merging their representations. By processing a sequence both ways, a bidirectional RNN can catch patterns that may be overlooked by a unidirectional RNN.

Remarkably, the fact that the RNN layers in this section have processed sequences in chronological order (older timesteps first) may have been an arbitrary decision. At least, it's a decision we made no attempt to question so far. Could the RNNs have performed well enough if they processed input sequences in antichronological order, for instance, newer timesteps first? Let's try this in practice and see what happens. All you need to do is write a variant of the data generator where the input sequences are reverted along the time dimension (replace the last line with `yield samples[:, ::-1, :], targets`).

When training the same LSTM-based model that you used in the first experiment in this section, you would find that such a reversed-order LSTM strongly underperforms even the commonsense baseline. This indicates that, in this case, chronological processing is important to the success of the approach. This makes perfect sense: the underlying `LSTM` layer will typically be better at remembering the recent past than the distant past, and, naturally, the more recent weather data points are more predictive than older data points for the problem (that's what makes the commonsense baseline fairly strong). Thus the chronological version of the layer is bound to outperform the reversed-order version.

However, this isn't true for many other problems, including natural language: intuitively, the importance of a word in understanding a sentence isn't strongly dependent on its position in the sentence. On text data, reversed-order processing works just as well as chronological processing—you can read text backwards just fine (try it!). Although word order does matter in understanding language, *which order* you use isn't crucial.

Importantly, an RNN trained on reversed sequences will learn different representations than one trained on the original sequences, much as you would have different mental models if time flowed backward in the real world—if you lived a life where you died on your first day and were born on your last day. In machine learning, representations that are *different* yet *useful* are always worth exploiting, and the more they differ,

the better: they offer a new angle from which to look at your data, capturing aspects of the data that were missed by other approaches, and thus they can help boost performance on a task. This is the intuition behind *ensembling*, a concept we'll explore in chapter 18.

A bidirectional RNN exploits this idea to improve on the performance of chronological-order RNNs. It looks at its input sequence both ways (see figure 13.13), obtaining potentially richer representations and capturing patterns that may have been missed by the chronological-order version alone.

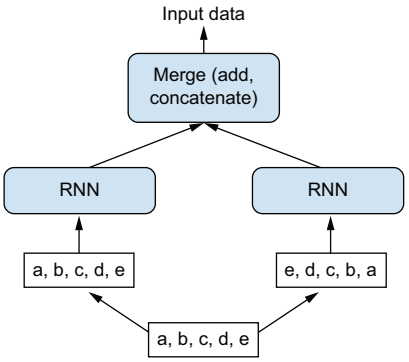


Figure 13.13 How a bidirectional RNN layer works

To instantiate a bidirectional RNN in Keras, you use the `Bidirectional` layer, which takes as its first argument a recurrent layer instance. `Bidirectional` creates a second, separate instance of this recurrent layer and uses one instance for processing the input sequences in chronological order and the other instance for processing the input sequences in reversed order. You can try it on our temperature forecasting task.

Listing 13.24 Training and evaluating a bidirectional LSTM

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

model.compile(optimizer="adam", loss="mse", metrics=["mae"])
history = model.fit(
    train_dataset,
    epochs=10,
    validation_data=val_dataset,
)
```

You'll find that it doesn't perform as well as the plain `LSTM` layer. It's easy to understand why: all the predictive capacity must come from the chronological half of the network because the antichronological half is known to be severely underperforming on this task (again, because the recent past matters much more than the distant past in this case). At the same time, the presence of the antichronological half doubles the network's capacity and causes it to start overfitting much earlier.

However, bidirectional RNNs are a great fit for text data—or any other kind of data where order matters, yet where *which* order you use doesn't matter. In fact, for a while in 2016, bidirectional LSTMs were considered the state of the art on many natural language processing tasks (before the rise of the Transformer architecture, which you will learn about in chapter 15).

13.4 Going even further

There are many other things you could try to improve performance on the temperature forecasting problem:

- Adjust the number of units in each recurrent layer in the stacked setup, as well as the amount of dropout. The current choices are largely arbitrary and thus probably suboptimal.
- Adjust the learning rate used by the `Adam` optimizer or try a different optimizer.
- Try using a stack of `Dense` layers as the regressor on top of the recurrent layer, instead of a single `Dense` layer.
- Improve the input to the model: try using longer or shorter sequences or a different sampling rate, or start doing feature engineering.

As always, deep learning is more an art than a science. We can provide guidelines that suggest what is likely to work or not work on a given problem, but, ultimately, every dataset is unique; you'll have to evaluate different strategies empirically. There is currently no theory that will tell you in advance precisely what you should do to optimally solve a problem. You must iterate.

In our experience, improving on the no-learning baseline by about 10% is likely the best you can do with this dataset. This isn't so great, but these results make sense: while near-future weather is highly predictable if you have access to data from a wide grid of different locations, it's not very predictable if you only have measurements from a single location. The evolution of the weather where you are depends on current weather patterns in surrounding locations.

Markets and machine learning

Some readers are bound to want to take the techniques we've introduced here and try them on the problem of forecasting the future price of securities on the stock market (or currency exchange rates, and so on). However, markets have very different statistical characteristics than natural phenomena such as weather patterns. When it comes to markets, past performance is *not* a good predictor of future returns—looking in the rear-view mirror is a bad way to drive. Machine learning, on the other hand, is applicable to datasets where the past *is* a good predictor of the future, like weather, electricity consumption, or foot traffic at a store.

Always remember that all trading is fundamentally *information arbitrage*: gaining an advantage by using data or insights that other market participants are missing. Trying to use well-known machine learning techniques and publicly available data to beat the markets is effectively a dead end, since you won't have any information advantage compared to everyone else. You're likely to waste your time and resources with nothing to show for it.

Summary

- As you first learned in chapter 6, when approaching a new problem, it's good to first establish commonsense baselines for your metric of choice. If you don't have a baseline to beat, you can't tell whether you're making real progress.
- Try simple models before expensive ones, to justify the additional expense. Sometimes a simple model will turn out to be your best option.
- When you have data where ordering matters—in particular, for timeseries data—*recurrent networks* are a great fit and easily outperform models that first flatten the temporal data. The two essential RNN layers available in Keras are the `LSTM` layer and the `GRU` layer.
- To use dropout with recurrent networks, you should use a time-constant dropout mask and recurrent dropout mask. These are built into Keras recurrent layers, so all you have to do is use the `recurrent_dropout` arguments of recurrent layers.
- Stacked RNNs provide more representational power than a single RNN layer. They're also much more expensive and thus not always worth it. Although they offer clear gains on complex problems (such as machine translation), they may not always be relevant to smaller, simpler problems.

14 Text classification

This chapter covers

- An introduction to the field of natural language processing (NLP)
- Preprocessing text input into numeric input
- Building simple text classification models

This chapter will lay the foundation for working with text input that we will build on in the next two chapters of this book. By the end of this chapter, you will be able to build a simple text classifier in a number of different ways. This will set the stage for building more complicated models, like the Transformer, in the next chapter.

14.1 A brief history of natural language processing

In computer science, we refer to human languages, like English or Mandarin, as "natural" languages to distinguish them from languages that were designed for machines, like LISP, Assembly, and XML. Every machine language was designed: its starting point was an engineer writing down a set of formal rules to describe what statements you can make and what they mean. The rules came first, and people only started using the language once the rule set was complete. With human language,

it's the reverse: usage comes first, and rules arise later. Natural language was shaped by an evolutionary process, much like biological organisms—that's what makes it "natural." Its "rules," like the grammar of English, were formalized after the fact and are often ignored or broken by its users. As a result, while machine-readable language is highly structured and rigorous, natural language is messy—ambiguous, chaotic, sprawling, and constantly in flux.

Computer scientists have long fixated on the potential of systems that can ingest or produce natural language. Language, particularly written text, underpins most of our communications and cultural production. Centuries of human knowledge are stored via text; the internet is mostly text, and even our thoughts are based on language! The practice of using computers to interpret and manipulate language is called natural language processing, or NLP for short. It was first proposed as a field of study immediately following World War II, where some thought we could view understanding language as a form of "code cracking," where natural language is the "code" used to transmit information.

In the early days of the field, many people naively thought that you could write down the "rule set of English," much like one can write down the rule set of LISP. In the early 1950s, researchers at IBM and Georgetown demonstrated a system that could translate Russian into English. The system used a grammar with six hardcoded rules and a lookup table with a couple of hundred elements (words and suffixes) to translate 60 handpicked Russian sentences accurately. The goal was to drum up excitement and funding for machine translation, and in that sense, it was a huge success. Despite the limited nature of the demo, the authors claimed that within five years, translation would be a solved problem. Funding poured in for the better part of a decade. However, generalizing such a system proved to be maddeningly difficult. Words change their meaning dramatically depending on context. Any grammar rules needed countless exceptions. Developing a program that could shine on a few handpicked examples was simple enough, but building a robust system that could compete with human translators was another matter. An influential US report a decade later picked apart the lack of progress, and funding dried up.

Despite these setbacks and repeated swings from excitement to disillusionment, handcrafted rules held out as the dominant approach well into the 1990s. The problems were obvious, but there was simply no viable alternative to writing down symbolic rules to describe grammar. However, as faster computers and greater quantities of data became available in the late 1980s, research began to head in a new direction. When you find yourself building systems that are big piles of ad hoc rules, as a clever engineer, you're likely to start asking, "Could I use a corpus of data to automate the process of finding these rules? Could I search for the rules within some rule space, instead of having to come up with them myself?" And just like that, you've graduated to doing machine learning.

In the late 1980s, we started seeing machine learning approaches to natural language processing. The earliest ones were based on decision trees—the intent was literally to

automate the development of the kind of if/then/else rules of hardcoded language systems. Then, statistical approaches started gaining speed, starting with logistic regression. Over time, learned parametric models took over, and linguistics came to be seen by some as a hindrance when baked directly into a model. Frederick Jelinek, an early speech recognition researcher, joked in the 1990s, "Every time I fire a linguist, the performance of the speech recognizer goes up."

Much as computer vision is pattern recognition applied to pixels, the modern field of NLP is all about pattern recognition applied to words in text. There's no shortage of practical applications:

- Given the text of an email, what is the probability that it is spam? (*text classification*)
- Given an English sentence, what is the most likely Russian translation? (*translation*)
- Given an incomplete sentence, what word will likely come next? (*language modeling*)

The text-processing models you will train in this book won't possess a human-like understanding of language; rather, they simply look for statistical regularities in their input data, which turns out to be sufficient to perform well on a wide array of real-world tasks.

In the last decade, NLP researchers and practitioners have discovered just how shockingly effective it can be to learn the answer to narrow statistical questions about text. In the 2010s, researchers began applying LSTM models to text, dramatically increasing the number of parameters in NLP models and the compute resources required to train them. The results were encouraging—LSTMs could generalize to unseen examples with far greater accuracy than previous approaches, but they eventually hit limits. LSTMs struggled to track dependencies in long chains of text with many sentences and paragraphs, and compared to computer vision models, they were slow and unwieldy to train.

Toward the end of the 2010s, researchers at Google discovered a new architecture called the Transformer that solved many scalability issues plaguing LSTMs. As long as you increased the size of a model and its training data together, Transformers appeared to perform more and more accurately. Better yet, the computations needed for training a Transformer could be effectively parallelized, even for long sequences. If you doubled the number of machines doing training, you could roughly halve the time you need to wait for a result.

The discovery of the Transformer architecture, along with ever-faster GPUs and CPUs, has led to a dramatic explosion of investment and interest in NLP models over the past few years. Chat systems like ChatGPT have captivated public attention with their ability to produce fluent and natural text on seemingly arbitrary topics and questions. The raw text used to train these models is a significant portion of all written language available on the internet, and the compute to train individual models can cost tens of millions of dollars. Some hype is worth cutting down to size—these are pattern

recognition machines. Despite our persistent human tendency to find intelligence in "things that talk," these models copy and synthesize training data in a way that is wholly distinct (and much less efficient!) than human intelligence. However, it is also fair to say that the emergence of complex behaviors from incredibly simple "guess the missing word" training setups has been one of the most shocking empirical results in the last decade of machine learning.

In the following three chapters, we will look at a range of techniques for machine learning with text data. We will skip discussion of the hardcoded linguistic features that prevailed until the 1990s, but we will look at everything from running logistic regressions for classifying text to training LSTMs for machine translation. We will closely examine the Transformer model and discuss what makes it so scalable and effective at generalizing in the text domain. Let's dig in.

14.2 Preparing text data

Let's consider an English sentence:

```
The quick brown fox jumped over the lazy dog.
```

There is an obvious blocker before we can start applying any of the deep learning techniques of previous chapters—our input is not numeric! Before beginning any modeling, we need to translate the written word into tensors of numbers. Unlike images, which have a relatively natural numeric representation, you could build a numeric representation of text in several ways.

A simple approach would be to borrow from standard text file formats for text and use something like an ASCII encoding. We could chop the input into a sequence of characters and assign each a unique index. Another intuitive approach would be building a representation based on words, first breaking sentences apart on all spaces and punctuation and then mapping each word to a unique numeric representation.

These are both good approaches to try, and in general, all text preprocessing will include a *splitting* step, where text is split into small individual units, called *tokens*. A powerful tool for splitting text is regular expressions, which can flexibly match patterns of characters in text.

Let's look at how to use a regular expression to split a string into a sequence of characters. The most basic regex we can apply is `"."`, which matches any character in the input text:

```
import regex as re

def split_chars(text):
    return re.findall(r".", text)
```

We can apply the function to our example input string:

```
>>> chars = split_chars("The quick brown fox jumped over the lazy dog.")
>>> chars[:12]
["T", "h", "e", " ", "q", "u", "i", "c", "k", " ", "b", "r"]
```

Regex can easily be applied to split our text into words instead. The `"[\w]+"` regular expression will grab consecutive non-whitespace characters, and the `"[.,!?;]"` can match the punctuation marks between the brackets. We can combine the two to achieve a regular expression that splits each word and punctuation mark into a token:

```
def split_words(text):
    return re.findall(r"[\w]+|[.,!?;]", text)
```

Here's what it does to a test sentence:

```
>>> split_words("The quick brown fox jumped over the dog.")
["The", "quick", "brown", "fox", "jumped", "over", "the", "dog", "."]
```

Splitting takes us from a single string to a token sequence, but we still need to transform our string tokens into numeric inputs. By far the most common approach is to map each token to a unique integer index, often called *indexing* our input. This is a flexible and reversible representation of our tokenized input that can work with a wide range of modeling approaches. Later on, we can decide how to map from token indices into a latent space ingested by the model.

For character tokens, we could use ASCII lookups to index each token—for example, `ord('A')` → `65` and `ord('z')` → `122`. However, this can scale poorly when you start to consider other languages—there are over a million characters in the Unicode specification! A more robust technique is to build a mapping from specific tokens in our training data to indices that occur in the data we care about, which in NLP is called a *vocabulary*. This has the nice property of working for word-level tokens as easily as for character-level tokens.

Let's take a look at how we might use a vocabulary to transform text. We will build a simple Python dictionary that maps tokens to indices, split our input into tokens, and finally index our tokens:

```
vocabulary = {
    "[UNK]": 0,
    "the": 1,
    "quick": 2,
    "brown": 3,
    "fox": 4,
    "jumped": 5,
```

```
    "over": 6,
    "dog": 7,
    ".": 8,
}
words = split_words("The quick brown fox jumped over the lazy dog.")
indices = [vocabulary.get(word, 0) for word in words]
```

This outputs the following:

```
[0, 2, 3, 4, 5, 6, 1, 0, 7, 8]
```

We introduce a special token called `"[UNK]"` to our vocabulary, which represents a token that is unknown to the vocabulary. This way, we can index all input we come across, even if some terms only occur in our test data. In the previous example `"lazy"` maps to the `"[UNK]"` index 0, as it was not included in our vocabulary.

With these simple text transformations, we are well on our way to building a text preprocessing pipeline. However, there is one more common type of text manipulation we should consider—standardization.

Consider these two sentences:

- "sunset came. i was staring at the Mexico sky. Isnt nature splendid??"
- "Sunset came; I stared at the México sky. Isn't nature splendid?"

They are very similar—in fact, they are almost identical. Yet, if you were to convert them to indices as previously described, you would end up with very different representations because "i" and "I" are two distinct characters, "Mexico" and "México" are two distinct words, "isnt" isn't "isn't," and so on. A machine learning model doesn't know a priori that "i" and "I" are the same letter, that "é" is an "e" with an accent, or that "staring" and "stared" are two forms of the same verb. *Standardizing* text is a basic form of feature engineering that aims to erase encoding differences that you don't want your model to have to deal with. It's not exclusive to machine learning, either—you'd have to do the same thing if you were building a search engine.

One simple and widespread standardization scheme is to convert to lowercase and remove punctuation characters. Our two sentences would become

- "sunset came i was staring at the mexico sky isnt nature splendid"
- "sunset came i stared at the méxico sky isnt nature splendid"

Much closer already. We could get even closer if we removed accent marks on all characters.

There's a lot you can do with standardization, and it used to be one of the most critical areas to improve model performance. For many decades in NLP, it was common practice to use regular expressions to attempt to map words to a common root (e.g. "tired" → "tire" and "trophies" → "trophy"), called *stemming* or *lemmatization*. But

as models have grown more expressive, this type of standardization tends to do more harm than good. The tense and plurality of a word are necessary signals to its meaning. For the larger models used today, most standardization is as light as possible—for example, converting all inputs to a standard character encoding before further processing.

With standardization, we have now seen three distinct stages for preprocessing text (figure 14.1):

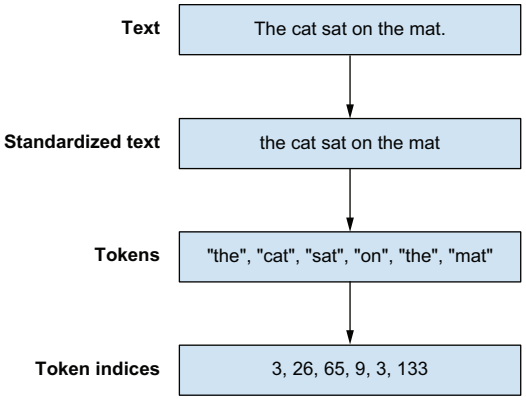


Figure 14.1 The text preprocessing pipeline

1. *Standardization*—Where we normalize input with basic text-to-text transformations
2. *Splitting*—Where we split our text into sequences of *tokens*
3. *Indexing*—Where we map our tokens to indices using a *vocabulary*

People often refer to the entire process as *tokenization*, and to an object that maps text to sequence of token indices as a *tokenizer*. Let's try building a few.

14.2.1 Character and word tokenization

To start, let's build a character-level tokenizer that maps each character in an input string to an integer. To keep things simple, we will use only one standardization step—we lowercase all input.

Listing 14.1 A basic character-level tokenizer

```
class CharTokenizer:
    def __init__(self, vocabulary):
        self.vocabulary = vocabulary
        self.unk_id = vocabulary["[UNK]"]

    def standardize(self, inputs):
        return inputs.lower()

    def split(self, inputs):
        return re.findall(r".", inputs)

    def index(self, tokens):
        return [self.vocabulary.get(t, self.unk_id) for t in tokens]

    def __call__(self, inputs):
        inputs = self.standardize(inputs)
        tokens = self.split(inputs)
```

```
        indices = self.index(tokens)
        return indices
```

Pretty simple. Before using this, we also need to build a function that computes a vocabulary of tokens based on some input text. Rather than simply mapping all characters to a unique index, let's give ourselves the ability to limit our vocabulary size to only the most common tokens in our input data. When we get into the modeling side of things, limiting the vocabulary size will be an important way to limit the number of parameters in a model.

Listing 14.2 Computing a character-level vocabulary

```
import collections

def compute_char_vocabulary(inputs, max_size):
    char_counts = collections.Counter()
    for x in inputs:
        x = x.lower()
        tokens = re.findall(r".", x)
        char_counts.update(tokens)
    vocabulary = ["[UNK]"]
    most_common = char_counts.most_common(max_size - len(vocabulary))
    for token, count in most_common:
        vocabulary.append(token)
    return dict((token, i) for i, token in enumerate(vocabulary))
```

We can now do the same for a word-level tokenizer. We can use the same code as our character-level tokenizer with a different splitting step.

Listing 14.3 A basic word-level tokenizer

```
class WordTokenizer:
    def __init__(self, vocabulary):
        self.vocabulary = vocabulary
        self.unk_id = vocabulary["[UNK]"]

    def standardize(self, inputs):
        return inputs.lower()

    def split(self, inputs):
        return re.findall(r"[\w]+|[.,!?;]", inputs)

    def index(self, tokens):
        return [self.vocabulary.get(t, self.unk_id) for t in tokens]

    def __call__(self, inputs):
        inputs = self.standardize(inputs)
        tokens = self.split(inputs)
```

```
        indices = self.index(tokens)
        return indices
```

We can also substitute this new split rule into our vocabulary function.

Listing 14.4 Computing a word-level vocabulary

```
def compute_word_vocabulary(inputs, max_size):
    word_counts = collections.Counter()
    for x in inputs:
        x = x.lower()
        tokens = re.findall(r"[\w]+|[.,!?;]", x)
        word_counts.update(tokens)
    vocabulary = ["[UNK]"]
    most_common = word_counts.most_common(max_size - len(vocabulary))
    for token, count in most_common:
        vocabulary.append(token)
    return dict((token, i) for i, token in enumerate(vocabulary))
```

Let's try out our tokenizers on some real-world input—the full text of *Moby Dick* by Herman Melville. We will first build a vocabulary for both tokenizers and then use it to tokenize some text:

```
import keras

filename = keras.utils.get_file(
    origin="https://www.gutenberg.org/files/2701/old/moby10b.txt",
)
moby_dick = list(open(filename, "r"))

vocabulary = compute_char_vocabulary(moby_dick, max_size=100)
char_tokenizer = CharTokenizer(vocabulary)
```

Let's inspect what our character-level tokenizer has computed:

```
>>> print("Vocabulary length:", len(vocabulary))
Vocabulary length: 64
>>> print("Vocabulary start:", list(vocabulary.keys())[:10])
Vocabulary start: ["[UNK]", " ", "e", "t", "a", "o", "n", "i", "s", "h"]
>>> print("Vocabulary end:", list(vocabulary.keys())[-10:])
Vocabulary end: ["@", "$", "%", "#", "=", "~", "&", "+", "<", ">"]>>>
print("Line length:", len(char_tokenizer(
...     "Call me Ishmael. Some years ago--never mind how long precisely."
... )))
Line length: 63
```

Now what about the word-level tokenizer?

```
vocabulary = compute_word_vocabulary(moby_dick, max_size=2_000)
word_tokenizer = WordTokenizer(vocabulary)
```

We can print out the same data for our word-level tokenizer:

```
>>> print("Vocabulary length:", len(vocabulary))
Vocabulary length: 2000
>>> print("Vocabulary start:", list(vocabulary.keys())[:5])
Vocabulary start: ["[UNK]", ",", "the", ".", "of"]
>>> print("Vocabulary end:", list(vocabulary.keys())[-5:])
Vocabulary end: ["tambourine", "subtle", "perseus", "elevated", "repose"]
>>> print("Line length:", len(word_tokenizer(
...     "Call me Ishmael. Some years ago--never mind how long precisely."
... )))
Line length: 13
```

We can already see some of the strengths and weaknesses of both tokenization techniques. A character-level tokenizer needs only 64 vocabulary terms to cover the entire book but will encode each input as a very long sequence. A word-level tokenizer quickly fills a 2,000-term vocabulary (you would need a dictionary with 17,000 terms to index every word in the book!), but the outputs of the word-level tokenizer are much shorter.

As machine learning practitioners have scaled models up with more and more data and parameters, the downsides of both word and character tokenization have become apparent. The "compression" offered by word-level tokenization turns out to be very important—it allows feeding longer sequences into a model. However, if you attempt to build a word-level vocabulary for a large dataset (today, you might see a dataset with trillions of words), you would have an unworkably large vocabulary with hundreds of millions of terms. If you aggressively restrict your word-level vocabulary size, you will encode a lot of text to the `"[UNK]"` token, throwing out valuable information.

These issues have led to the rise in popularity of a third type of tokenization, called *subword tokenization*, which attempts to bridge the gap between word- and character-level approaches.

14.2.2 Subword tokenization

Subword tokenization aims to combine the best of both character- and word-level encoding techniques. We want the `WordTokenizer`'s ability to produce concise output and the `CharTokenizer`'s ability to encode a wide range of inputs with a small vocabulary.

We can think of the search for the ideal tokenizer as the hunt for an ideal compression of the input data. Reducing token length compresses the overall length of our examples. A small vocabulary reduces the number of bytes we would need to represent each token. If we achieve both, we will be able to feed short, information-rich sequences to our deep learning model.

This analogy between compression and tokenization was not always obvious, but it turns out to be powerful. One of the most practically effective tricks found in the last decade of NLP research was repurposing a 1990s algorithm for lossless compression called *byte-pair encoding*[1] for tokenization. It is used by ChatGPT and many other models to this day. In this section, we will build a tokenizer that uses the byte-pair encoding algorithm.

The idea with byte-pair encoding is to start with a basic vocabulary of characters and progressively "merge" common pairings into longer and longer sequences of characters. Let's say we start with the following input text:

```
data = [
    "the quick brown fox",
    "the slow brown fox",
    "the quick brown foxhound",
]
```

Like the `WordTokenizer`, we will start by computing word counts for all the words in the text. As we create our dictionary of word counts, we will split all our text into characters and join the characters with a space. This will make it easier to consider pairs of characters in our next step.

Listing 14.5 Initializing state for the byte-pair encoding algorithm

```
def count_and_split_words(data):
    counts = collections.Counter()
    for line in data:
        line = line.lower()
        for word in re.findall(r"[\w]+|[.,!?;]", line):
            chars = re.findall(r".", word)
            split_word = " ".join(chars)
            counts[split_word] += 1
    return dict(counts)

counts = count_and_split_words(data)
```

Let's try this out on our data:

```
>>> counts
{"t h e": 3,
 "q u i c k": 2,
 "b r o w n": 3,
 "f o x": 2,
 "s l o w": 1,
 "f o x h o u n d": 1}
```

[1] Phillip Gage, "A New Algorithm for Data Compression," *The C Users Journal Archive* (1994), https://dl.acm.org/doi/10.5555/177910.177914.

To apply byte-pair encoding to our split word counts, we will find two characters and merge them into a new symbol. We consider all pairs of characters in all words and only merge the most common one we find. In the previous example, the most common character pair is `("o", "w")`, in both the word `"brown"` (which occurs three times in our data) and `"slow"` (which occurs once). We combine this pair into a new symbol `"ow"` and merge all occurrences of `"o w"`.

Then we continue, counting pairs and merging pairs, except now `"ow"` will be a single unit that could merge with, say, `"l"` to form `"low"`. By progressively merging the most frequent symbol pair, we build up a vocabulary of larger and larger subwords.

Let's try this out on our toy dataset.

Listing 14.6 Running a few steps of byte-pair merging

```
def count_pairs(counts):
    pairs = collections.Counter()
    for word, freq in counts.items():
        symbols = word.split()
        for pair in zip(symbols[:-1], symbols[1:]):
            pairs[pair] += freq
    return pairs

def merge_pair(counts, first, second):
    split = re.compile(f"(?<!\S){first} {second}(?!\S)")
    merged = f"{first}{second}"
    return {split.sub(merged, word): count for word, count in counts.items()}

for i in range(10):
    pairs = count_pairs(counts)
    first, second = max(pairs, key=pairs.get)
    counts = merge_pair(counts, first, second)
    print(list(counts.keys()))
```

Matches an unmerged pair

Replaces all occurances with a merged version

Here's what we get:

```
["t h e", "q u i c k", "b r ow n", "f o x", "s l ow", "f o x h o u n d"]
["th e", "q u i c k", "b r ow n", "f o x", "s l ow", "f o x h o u n d"]
["the", "q u i c k", "b r ow n", "f o x", "s l ow", "f o x h o u n d"]
["the", "q u i c k", "br ow n", "f o x", "s l ow", "f o x h o u n d"]
["the", "q u i c k", "brow n", "f o x", "s l ow", "f o x h o u n d"]
["the", "q u i c k", "brown", "f o x", "s l ow", "f o x h o u n d"]
["the", "q u i c k", "brown", "fo x", "s l ow", "fo x h o u n d"]
["the", "q u i c k", "brown", "fox", "s l ow", "fox h o u n d"]
["the", "qu i c k", "brown", "fox", "s l ow", "fox h o u n d"]
["the", "qui c k", "brown", "fox", "s l ow", "fox h o u n d"]
```

We can see how common words are merged entirely, whereas less common words are only partially merged.

We can now extend this to a full function for computing a byte-pair encoding vocabulary. We start our vocabulary with all characters found in the input text, and we will progressively add merged symbols (larger and larger subwords) to our vocabulary until it reaches our desired length. We also keep a separate dictionary of our merge rules, including a rank order in which we applied them. Next, we will see how to use these merge rules to tokenize new input text.

Listing 14.7 Computing a byte-pair encoding vocabulary

```
def compute_sub_word_vocabulary(dataset, vocab_size):
    counts = count_and_split_words(dataset)

    char_counts = collections.Counter()
    for word in counts:
        for char in word.split():
            char_counts[char] += counts[word]
    most_common = char_counts.most_common()
    vocab = ["[
            <
            <!\S)\S+ \S+(?!\S)", word, overlapped=True)
            if not pairs:
                break
            best = min(pairs, key=lambda pair: self.merges.get(pair, 1e9))
            if best not in self.merges:
                break
            first, second = best.split()
            split = re.compile(f"(?<!\S){first} {second}(?!\S)")
            merged = f"{first}{second}"
            word = split.sub(merged, word)
        return word

    def split(self, inputs):
        tokens = []
        for word in re.findall(r"[

            word=self.bpe_merge(word)

            tokens.extend(word.split())
        return tokens

    def index(self, tokens):
        return [self.vocabulary.get(t, self.unk_id) for t in tokens]

    def __call__(self, inputs):
        inputs = self.standardize(inputs)
        tokens = self.split(inputs)
        indices = self.index(tokens)
        return indices
```

Matches all symbol pairs in the text

We apply merge rules in "rank" order. More frequent pairs are merged first.

Split words

Joins all characters with a space

Applies byte-pair encoding merge rules

Let's try out our tokenizer on the full text of *Moby Dick*:

```
vocabulary, merges = compute_sub_word_vocabulary(moby_dick, 2_000)
sub_word_tokenizer = SubWordTokenizer(vocabulary, merges)
```

We can take a look at our vocabulary and try a test sentence on our tokenizer, as we did with `WordTokenizer` and `CharTokenizer`:

```
>>> print("Vocabulary length:", len(vocabulary))
Vocabulary length: 2000
>>> print("Vocabulary start:", list(vocabulary.keys())[:10])
Vocabulary start: ["[UNK]", "e", "t", "a", "o", "n", "i", "s", "h", "r"]
>>> print("Vocabulary end:", list(vocabulary.keys())[-7:])
Vocabulary end: ["bright", "pilot", "sco", "ben", "dem", "gale", "ilo"]
>>> print("Line length:", len(sub_word_tokenizer(
...     "Call me Ishmael. Some years ago--never mind how long precisely."
... )))
Line length: 16
```

The `SubWordTokenizer` has a slightly longer length for our test sentence than the `WordTokenizer` (16 versus 13 tokens), but unlike the `WordTokenizer`, it can tokenize every word in *Moby Dick* without using the `"[UNK]"` token. The vocabulary contains every character in our source text, so the worst-case performance will be tokenizing a word into individual characters. We have achieved a short *average* token length while handling rare words with a small vocabulary. This is the advantage of subword tokenizers.

You might notice that running this code is noticeably slower than the word and character tokenizers; it takes about a minute on our reference hardware. Learning merge rules is much more complex than simply counting the words in an input dataset. While this is a downside to subword tokenization, it is rarely an important concern in practice. You only need to learn a vocabulary once per model, and the cost of learning a subword vocabulary is generally negligible compared to model training.

We have now seen three separate approaches for tokenizing input. Now that we can translate from text to numeric input, we can move on to training a model.

One final note on tokenization—while it is quite important to understand how tokenizers work, it is rarely the case that you will need to build one yourself. Keras comes with utilities for tokenizing text input, as do most deep learning frameworks. For the rest of the chapter, we will make use of the built-in functionality in Keras for tokenization.

Which tokenization technique should I use?

When approaching a new text-modeling problem, one of the first questions you will need to answer is how to tokenize your input. As we will see at the end of this chapter, the question is trivial for a given pretrained model. You have to preserve the exact

tokenization used during pretraining or throw out the useful representations of input tokens contained in the model weights.

If you are building a model from scratch, you can tailor your tokenization to the problem at hand. In general, the compression offered by word and subword tokenizers is too important to pass up. The shorter the length of your inputs on average, the better a model will be able to track long-range dependencies in the text, improving its overall performance. This has made subword the dominantly popular choice for modern language models. They can handle rare or misspelled words without inflating token length for common inputs.

However, there is no one-size-fits-all approach. Some problems in NLP, such as spelling correction, might benefit from low-level character tokenization of input text. On the other hand, a word-level approach is both simple to work with and easily understandable—each model input corresponds to a word a human would read. This would make ranking tokens by importance to a prediction easy to interpret.

We will use all three types of tokenizers throughout the text chapters of this book.

14.3 Sets vs. sequences

How a machine learning model should represent individual tokens is a relatively uncontroversial question: they're categorical features (values from a predefined set), and we know how to handle those. They should be encoded as dimensions in a feature space or as category vectors (token vectors in this case). A much more problematic question, however, is how to encode the ordering of tokens in text.

The problem of order in natural language is an interesting one: unlike the steps of a timeseries, words in a sentence don't have a natural, canonical order. Different languages order similar words in very different ways. For instance, the sentence structure of English is quite different from that of Japanese. Even within a given language, you can typically say the same thing in different ways by reshuffling the words a bit. If you were to fully randomize the words in a short sentence, you could still sometimes figure out what it was saying—though, in many cases, significant ambiguity would arise. Order is clearly important, but its relationship to meaning isn't straightforward.

How to represent word order is the pivotal question from which different kinds of NLP architectures spring. The simplest thing you could do is discard order and treat text as an unordered set of words—this gives you bag-of-words models. You could also decide that words should be processed strictly in the order in which they appear, one at a time, like steps in a timeseries—you could then use the recurrent models from the previous chapter. Finally, a hybrid approach is also possible: the Transformer architecture is technically order-agnostic, yet it injects word-position information into the representations it processes, which enables it to simultaneously look at different parts of a sentence (unlike RNNs) while still being order-aware. Because they take into account word order, both RNNs and Transformers are called *sequence models*.

Historically, most early applications of machine learning to NLP just involved bag-of-words models that discarded sequence data. Interest in sequence models only started rising in 2015, with the rebirth of RNNs. Today, both approaches remain relevant. Let's see how they work and when to use which.

We will demonstrate each approach on a well-known text classification benchmark: the IMDb movie review sentiment-classification dataset. In chapters 4 and 5, you worked with a prevectorized version of the IMDb dataset; now let's process the raw IMDb text data, just like you would do when approaching a new text-classification problem in the real world.

14.3.1 Loading the IMDb classification dataset

To begin, let's download and extract our dataset.

Listing 14.8 Downloading the IMDb movie review dataset

```
import os, pathlib, shutil, random

zip_path = keras.utils.get_file(
    origin="https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz",
    fname="imdb",
    extract=True,
)

imdb_extract_dir = pathlib.Path(zip_path) / "aclImdb"
```

Let's list out our directory structure:

```
>>> for path in imdb_extract_dir.glob("*/*"):
...     if path.is_dir():
...         print(path)
~/.keras/datasets/aclImdb/train/pos
~/.keras/datasets/aclImdb/train/unsup
~/.keras/datasets/aclImdb/train/neg
~/.keras/datasets/aclImdb/test/pos
~/.keras/datasets/aclImdb/test/neg
```

We can see both a train and test set with positive and negative examples. Movie reviews with a low user rating on the IMDb site were sorted into the `neg/` directory and those with a high rating into the `pos/` directory. We can also see an `unsup/` directory, which is short for unsupervised. These are reviews deliberately left unlabeled by the dataset creator; they could be negative or positive reviews.

Let's look at the content of a few of these text files. Whether you're working with text or image data, remember to inspect what your data looks like before you dive into modeling. It will ground your intuition about what your model is actually doing.

Listing 14.9 Previewing a single IMDb review

```
>>> print(open(imdb_extract_dir / "train" / "pos" / "4077_10.txt", "r").read())
I first saw this back in the early 90s on UK TV, i did like it then but i missed
the chance to tape it, many years passed but the film always stuck with me and i
lost hope of seeing it TV again, the main thing that stuck with me was the end,
the hole castle part really touched me, its easy to watch, has a great story,
great music, the list goes on and on, its OK me saying how good it is but
everyone will take there own best bits away with them once they have seen it,
yes the animation is top notch and beautiful to watch, it does show its age in a
very few parts but that has now become part of it beauty, i am so glad it has
came out on DVD as it is one of my top 10 films of all time. Buy it or rent it
just see it, best viewing is at night alone with drink and food in reach so you
don't have to stop the film.<br/>
                                                                    <br/>Enjoy
```

Before we begin tokenizing our input text, we will make a copy of our training data with a few important modifications. We can ignore the unsupervised reviews for now and create a separate validation set to monitor our accuracy while training. We do this by splitting 20% of the training text files into a new directory.

Listing 14.10 Splitting validation from the IMDb dataset

```
train_dir = pathlib.Path("imdb_train")
test_dir = pathlib.Path("imdb_test")
val_dir = pathlib.Path("imdb_val")

shutil.copytree(imdb_extract_dir / "test", test_dir)

val_percentage = 0.2
for category in ("neg", "pos"):
    src_dir = imdb_extract_dir / "train" / category
    src_files = os.listdir(src_dir)
    random.Random(1337).shuffle(src_files)
    num_val_samples = int(len(src_files) * val_percentage)

    os.makedirs(val_dir / category)
    for file in src_files[:num_val_samples]:
        shutil.copy(src_dir / file, val_dir / category / file)
    os.makedirs(train_dir / category)
    for file in src_files[num_val_samples:]:
        shutil.copy(src_dir / file, train_dir / category / file)
```

Moves the test data unaltered

Splits the training data into a train set and a validation set

We are now ready to load the data. Remember how, in chapter 8, we used the `image_dataset_from_directory` utility to create a `Dataset` of images and their labels for a directory structure? You can do the exact same thing for text files using the `text_dataset_from_directory` utility. Let's create three `Dataset` objects for training, validation, and testing.

Listing 14.11 Loading the IMDb dataset for use with Keras

```
from keras.utils import text_dataset_from_directory

batch_size = 32
train_ds = text_dataset_from_directory(train_dir, batch_size=batch_size)
val_ds = text_dataset_from_directory(val_dir, batch_size=batch_size)
test_ds = text_dataset_from_directory(test_dir, batch_size=batch_size)
```

Originally we had 25,000 training and testing examples each, and after our validation split, we have 20,000 reviews to train on and 5,000 for validation. Let's try learning something from this data.

14.4 Set models

The simplest approach we can take regarding the ordering of tokens in text is to discard it. We still tokenize our input reviews normally as a sequence of token IDs, but immediately after tokenization, we convert the entire training example to a set—a simple unordered "bag" of tokens that are either present or absent in a movie review.

The idea here is to use these sets to build a very simple model that assigns a weight to every individual word in a review. The presence of the word `"terrible"` would probably (though not always) indicate a bad review, and `"riveting"` might indicate a good review. We can build a small model that can learn these weights—called a bag-of-words model.

For example, let's say you had a simple input sentence and vocabulary:

```
"this movie made me cry"

{"[UNK]": 0, "movie": 1, "film": 2, "made": 3, "laugh": 4, "cry": 5}
```

We would tokenize this tiny review as

```
[0, 1, 3, 0, 5]
```

Discarding order, we can turn this into a set of token IDs:

```
{0, 1, 3, 5}
```

Finally, we could use a multi-hot encoding to transform the set to a fixed-sized vector with the same length as a vocabulary:

```
[1, 1, 0, 1, 0, 1]
```

The 0 in the fifth position here means the word "laugh" is absent in our review, and the 1 in the sixth position means "cry" is present. This simple encoding of our input review can be used directly to train a model.

14.4.1 Training a bag-of-words model

To do this text processing in code, it would be easy enough to extend our WordTokenizer from earlier in the chapter. An even easier solution is to use the TextVectorization layer built into Keras. The TextVectorization handles word and character tokenization and comes with several additional features, including multi-hot encoding of the layer output.

The TextVectorization layer, like many preprocessing layers in Keras, has an adapt() method to learn a layer state from input data. In the case of TextVectorization, adapt() will learn a vocabulary for a dataset on the fly by iterating over an input dataset. Let's use it to tokenize and encode our input data. We will build a vocabulary of 20,000 words, a good starting place for text classification problems.

Listing 14.12 Applying a bag-of-words encoding to the IMDb reviews

```
from keras import layers

max_tokens = 20_000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    split="whitespace",                    <-- Learns a word-level vocabulary
    output_mode="multi_hot",
)
train_ds_no_labels = train_ds.map(lambda x, y: x)
text_vectorization.adapt(train_ds_no_labels)

bag_of_words_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y), num_parallel_calls=8
)
bag_of_words_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y), num_parallel_calls=8
)
bag_of_words_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y), num_parallel_calls=8
)
```

Let's look a single batch of our preprocessed input data:

```
>>> x, y = next(bag_of_words_train_ds.as_numpy_iterator())
>>> x.shape
(32, 20000)
>>> y.shape
(32, 1)
```

You can see that after preprocessing, each sample in our batch is converted into a vector of 20,000 numbers, each tracking the presence or absence of a vocabulary term.

Next, we can train a very simple linear model. We will save our model-building code as a function so we can use it again later.

Listing 14.13 Building a bag-of-words regression model

```
def build_linear_classifier(max_tokens, name):
    inputs = keras.Input(shape=(max_tokens,))
    outputs = layers.Dense(1, activation="sigmoid")(inputs)
    model = keras.Model(inputs, outputs, name=name)
    model.compile(
        optimizer="adam",
        loss="binary_crossentropy",
        metrics=["accuracy"],
    )
    return model

model = build_linear_classifier(max_tokens, "bag_of_words_classifier")
```

Let's take a look at our model's summary:

```
>>> model.summary()
Model: "bag_of_words_classifier"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 20000)	0
dense (Dense)	(None, 1)	20,001

```
Total params: 20,001 (78.13 KB)
Trainable params: 20,001 (78.13 KB)
Non-trainable params: 0 (0.00 B)
```

This model is dead simple. We have only 20,001 parameters, one for each word in our vocabulary and one for a bias term. Let's train it. We'll add on the `EarlyStopping` callback first covered in chapter 7, which will automatically stop when training when the validation loss stops improving and restore weights from the best epoch.

Listing 14.14 Training the bag-of-words regression model

```
early_stopping = keras.callbacks.EarlyStopping(
    monitor="val_loss",
    restore_best_weights=True,
    patience=2,
)
history = model.fit(
```

```
    bag_of_words_train_ds,
    validation_data=bag_of_words_val_ds,
    epochs=10,
    callbacks=[early_stopping],
)
```

Our model trains in much less than a minute, which is unsurprising given its size. The tokenization and encoding of our input is actually quite a bit more expensive than updating our model parameters. Let's plot the model accuracy (figure 14.2):

```
import matplotlib.pyplot as plt

accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
epochs = range(1, len(accuracy) + 1)

plt.plot(epochs, accuracy, "r--", label="Training accuracy")
plt.plot(epochs, val_accuracy, "b", label="Validation accuracy")
plt.title("Training and validation accuracy")
plt.legend()
plt.show()
```

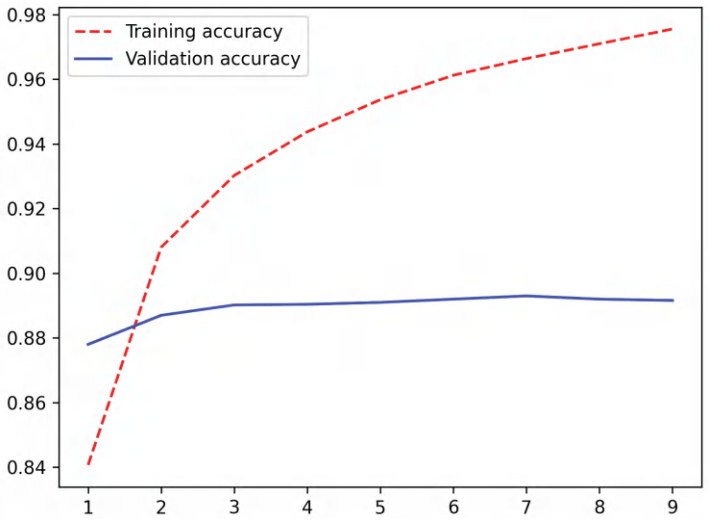


Figure 14 2 Training and validation metrics for our bag of words model

We can see that validation performance levels off rather than significantly declining; our model is so simple it cannot really overfit. Let's try evaluating it on our test set.

Listing 14.15 Evaluating the bag-of-words regression model

```
>>> test_loss, test_acc = model.evaluate(bag_of_words_test_ds)
>>> test_acc
0.88388
```

We can correctly predict the sentiment of a review 88% of the time with a training job light enough that it could run efficiently on a single CPU.

It is worth noting our choice of word tokenization in this example. The reason to avoid character-level tokenization here is pretty obvious—a "bag" of all characters in a movie review will tell you very little about its content. Subword tokenization with a large enough vocabulary would be a good choice, but there is little need for it here. Since the model we are training is so small, it's convenient to use a vocabulary that is quick to train and have our weights correspond to actual English words.

Preprocessing text efficiently

In all applied machine learning, the speed and efficiency of preprocessing are important concerns. A faster program is always desirable, but this becomes more urgent when the cost of accelerators (GPUs and TPUs) is so high. You want to avoid letting expensive GPUs idle while you preprocess your input!

Text preprocessing is unique because it must always run on a CPU. GPUs strictly handle numeric inputs, so all tokenization must happen before your GPU's train step. One option is to precompute your tokenized input—tokenization does not depend on model weights, so you could tokenize all input text files and resave them as integer sequences before starting training. However, this is not always practical. Tokenizing text on the fly allows for more rapid experimentation. If you are running inference on an unseen example, there is no way to precompute the tokenized input; you need to tokenize and run a forward pass in rapid succession.

The name of the game when preprocessing text input on the fly is to be "fast enough." You want to ensure your expensive GPUs always have a new batch of preprocessed data to ingest. If you do that, the GPU is the bottleneck, and there is nothing to gain by improving your tokenization speed.

We've seen `tf.data` in previous chapters, and an important reason we use it is that the library is designed to avoid the CPU becoming a bottleneck for a GPU or TPU. We use it throughout this chapter—`keras.utils.text_dataset_from_directory()` will load a `tf.data.Dataset`, and `map()` will transform our input data, for example, by applying a `TextVectorization` layer. `tf.data` works by running text preprocessing in parallel multiple CPU cores, which is generally sufficient to avoid bottlenecking accelerators during a training run.

It is important to note that the code in this chapter is still multi-backend (in fact, we generated the outputs for this chapter using Jax). You can use `tf.data` with PyTorch, JAX, or TensorFlow itself—Keras will automatically convert input Tensors to the correct format for a given backend.

14.4.2 Training a bigram model

Of course, we can intuitively guess that discarding all word order is very reductive because even atomic concepts can be expressed via multiple words: the term "United States" conveys a concept that is quite distinct from the meaning of the words "states" and "united" taken separately. A movie that is "not bad" and a movie that is "bad" should probably get different sentiment scores.

Therefore, it is usually a good idea to inject some knowledge of local word ordering into a model, even for these simple set-based models we are currently building. One easy way to do that is to consider *bigrams*—a term for two tokens that appear consecutively in the input text. Given our example "this movie made me cry," `{"this", "movie", "made", "me", "cry"}` is the set of all word *unigrams* in the input, and `{"this movie", "movie made", "made me", "me cry"}` is the set of all bigrams. The bag-of-words model we just trained could equivalently be called a unigram model, and the term *n-gram* refers to an ordered sequence of *n* tokens for any *n*.

To add bigrams to our model, we want to consider the frequency of all bigrams while building our vocabulary. We could do this in two ways: by creating a vocabulary of only bigrams or by allowing both bigrams and unigrams to compete for space in the same vocabulary. For the latter case, the term `"United States"` will be included in our vocabulary before `"ventriloquism"` if it occurs more frequently in the input text.

Again, we could build this by extending our `WordTokenizer` from earlier in the chapter, but there is no need. `TextVectorization` provides this out of the box. We will train a slightly larger vocabulary to account for the presence of bigrams, `adapt()` a new vocabulary, and multi-hot encode output vectors including bigrams.

Listing 14.16 Applying a bigram encoding to the IMDb reviews

```
max_tokens = 30_000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    split="whitespace",
    output_mode="multi_hot",
    ngrams=2,
)
text_vectorization.adapt(train_ds_no_labels)

bigram_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y), num_parallel_calls=8
)
bigram_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y), num_parallel_calls=8
)
bigram_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y), num_parallel_calls=8
)
```

Learns a word-level vocabulary

Considers all unigrams and bigrams

Let's examine a batch of our preprocessed input again:

```
>>> x, y = next(bigram_train_ds.as_numpy_iterator())
>>> x.shape
(32, 30000)
```

If we look at a small subsection of our vocabulary, we can see both unigram and bigram terms:

```
>>> text_vectorization.get_vocabulary()[100:108]
["in a", "most", "him", "dont", "it was", "one of", "for the", "made"]
```

With our new encoding for our input data, we can train a linear model unaltered from before.

Listing 14.17 Training the bigram regression model

```
model = build_linear_classifier(max_tokens, "bigram_classifier")
model.fit(
    bigram_train_ds,
    validation_data=bigram_val_ds,
    epochs=10,
    callbacks=[early_stopping],
)
```

This model will be slightly larger than our bag-of words models (30,001 parameters instead of 20,001 parameters), but it still trains in about the same amount of time. How did it do?

Listing 14.18 Evaluating the bigram regression model

```
>>> test_loss, test_acc = model.evaluate(bigram_test_ds)
>>> test_acc
0.90116
```

We're now getting 90% test accuracy, a noticeable improvement!

We could improve this number even further by considering trigrams (triplets of words), although beyond trigrams, the problem quickly becomes intractable. The space of possible 4-grams of words in the English language is immense, and the problem grows exponentially as sequences get longer and longer. You would need an immense vocabulary to provide decent coverage of 4-grams, and your model would lose its ability to generalize, simply memorizing entire snippets of sentences with weights attached. To robustly consider longer-ordered text sequences, we will need more advanced modeling techniques.

14.5 Sequence models

Our last two models indicated that sequence information is important. We improved a basic linear model by adding features with some info on local word order.

However, this was done by manually engineering input features, and we can see how the approach will only scale up to a local ordering of just a few words. As is often the case in deep learning, rather than attempting to build these features ourselves, we should expose the model to the raw word sequence and let it directly learn positional dependencies between tokens.

Models that ingest a complete token sequence are called, simply enough, *sequence models*. We have a few choices for architecture here. We could build an RNN model as we just did for timeseries modeling. We could build a 1D ConvNet, similar to our image processing models, but convolving filters over a single sequence dimension. And as we will dig into in the next chapter, we can build a Transformer.

Before taking on any of these approaches, we must preprocess our inputs into ordered sequences. We want an integer sequence of token IDs, as we saw in the tokenization portion of this chapter, but with one additional wrinkle to handle. When we run computations on a batch of inputs, we want all inputs to be rectangular so all calculations can be effectively parallelized across the batch on a GPU. However, tokenized inputs will almost always have varying lengths. IMDb movie reviews range from just a few sentences to multiple paragraphs, with varying word counts.

To accommodate this fact, we can truncate our input sequences or "pad" them with another special token `"[PAD]"`, similar to the `"[UNK]"` token we used earlier. For example, given two input sentences and a desired length of eight

```
"the quick brown fox jumped over the lazy dog"

"the slow brown badger"
```

we would tokenize to the integer IDs for the following tokens:

```
["the", "quick", "brown", "fox", "jumped", "over", "the", "lazy"]
["the", "slow", "brown", "badger", "[PAD]", "[PAD]", "[PAD]", "[PAD]"]
```

This will allow our batch computation to proceed much faster, although we will need to be careful with our padding tokens to ensure they do not affect the quality of our model predictions.

To keep a manageable input size, we can truncate our IMDb reviews after the first 600 words. This is a reasonable choice, since the average review length is 233 words, and only 5% of reviews are longer than 600 words. Once again, we can use the `Vectorization` layer, which has an option for padding or truncating inputs and includes a `"[PAD]"` at index zero of the learned vocabulary.

Listing 14.19 Padding IMDb reviews to a fixed sequence length

```
max_length = 600
max_tokens = 30_000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    split="whitespace",
    output_mode="int",
    output_sequence_length=max_length,
)
text_vectorization.adapt(train_ds_no_labels)

sequence_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y), num_parallel_calls=8
)
sequence_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y), num_parallel_calls=8
)
sequence_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y), num_parallel_calls=8
)
```

Learns a word-level vocabulary

Outputs a integer sequence of token IDs

Pads and truncates to 600 tokens

Let's take a look at a single input batch:

```
>>> x, y = next(sequence_test_ds.as_numpy_iterator())
>>> x.shape
(32, 600)
>>> x
array([[   11,    29,     7, ...,     0,     0,     0],
       [  132,   115,    35, ...,     0,     0,     0],
       [ 1825,     3, 25819, ...,     0,     0,     0],
       ...,
       [    4,   576,    56, ...,     0,     0,     0],
       [   30,   203,     4, ...,     0,     0,     0],
       [ 5104,     1,    14, ...,     0,     0,     0]])
```

Each batch has the shape `(batch_size, sequence_length)` after preprocessing, and almost all training samples have a number of 0s for padding at the end.

14.5.1 Training a recurrent model

Let's try training an LSTM. As we saw in the previous chapter, LSTMs can work efficiently with sequence data. Before we can apply it, we still need to map our token ID *integers* into floating-point data ingestible by a `Dense` layer.

The most straightforward approach is to *one-hot* our input IDs, similar to the multi-hot encoding we did for an entire sequence. Each token will become a long vector with all 0s and a single 1 at the index of the token in our vocabulary. Let's build a layer to one-hot encode our input sequence.

Listing 14.20 Building a one-hot encoding layer with Keras ops

```
from keras import ops

class OneHotEncoding(keras.Layer):
    def __init__(self, depth, **kwargs):
        super().__init__(**kwargs)
        self.depth = depth

    def call(self, inputs):
        flat_inputs = ops.reshape(ops.cast(inputs, "int"), [-1])
        one_hot_vectors = ops.eye(self.depth)
        outputs = ops.take(one_hot_vectors, flat_inputs, axis=0)
        return ops.reshape(outputs, ops.shape(inputs) + (self.depth,))

one_hot_encoding = OneHotEncoding(max_tokens)
```

Uses our input token IDs to gather the correct vector for each token

Builds an identity matrix with all possible one-hot vectors

Flattens the inputs

Unflattens the output

Let's try this layer out on a single input batch:

```
>>> x, y = next(sequence_train_ds.as_numpy_iterator())
>>> one_hot_encoding(x).shape
(32, 600, 30000)
```

We can build this layer directly into a model and use a bidirectional LSTM to allow information to propagate both forward and backward along the token sequence. Later, when we look at generation, we will see the need for unidirectional sequence models (where a token state only depends on the token state before it). For classification tasks, a bidirectional LSTM is a good fit.

Let's build our model.

Listing 14.21 Building an LSTM sequence model

```
hidden_dim = 64
inputs = keras.Input(shape=(max_length,), dtype="int32")
x = one_hot_encoding(inputs)
x = layers.Bidirectional(layers.LSTM(hidden_dim))(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs, name="lstm_with_one_hot")
model.compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
```

We can take a look at our model summary to get a sense of our parameter count:

```
>>> model.summary()
Model: "lstm_with_one_hot"
```

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 600)	0
one_hot_encoding (OneHotEncoding)	(None, 600, 30000)	0
bidirectional (Bidirectional)	(None, 128)	15,393,280
dropout (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 1)	129

```
Total params: 15,393,409 (58.72 MB)
Trainable params: 15,393,409 (58.72 MB)
Non-trainable params: 0 (0.00 B)
```

This is quite the step up in size from the unigram and bigram models. At about 15 million parameters, this is one of the larger models we have trained in the book so far, with only a single LSTM layer. Let's trying training the model.

Listing 14.22 Training the LSTM sequence model

```
model.fit(
    sequence_train_ds,
    validation_data=sequence_val_ds,
    epochs=10,
    callbacks=[early_stopping],
)
```

How does it perform?

Listing 14.23 Evaluating the LSTM sequence model

```
>>> test_loss, test_acc = model.evaluate(sequence_test_ds)
>>> test_acc
0.84811
```

This model works, but it trains very slowly, especially compared to the lightweight model of the previous section. That's because our inputs are quite large: each input sample is encoded as a matrix of size `(600, 30000)` (600 words per sample, 30,000 possible words). That is 18,000,000 floating-point numbers for a single movie review! Our bidirectional LSTM has a lot of work to do. In addition to being slow, the model only gets to 84% test accuracy—it doesn't perform nearly as well as our very fast set-based models.

Clearly, using one-hot encoding to turn words into vectors, which was the simplest thing we could do, wasn't a great idea. There's a better way—word embeddings.

14.5.2 Understanding word embeddings

When you encode something via one-hot encoding, you're making a feature engineering decision. You're injecting into your model a fundamental assumption about the structure of your feature space. That assumption is that the different tokens you're encoding are all independent from each other: indeed, one-hot vectors are all orthogonal to one another. In the case of words, that assumption is clearly wrong. Words form a structured space: they share information with each other. The words "movie" and "film" are interchangeable in most sentences, so the vector that represents "movie" should not be orthogonal to the vector that represents "film"—they should be the same vector, or close enough.

To get more abstract, the geometric relationship between two-word vectors should reflect the semantic relationship between these words. For instance, in a reasonable word vector space, you would expect synonyms to be embedded into similar word vectors, and in general, you would expect the geometric distance (such as the cosine distance or L2 distance) between any two-word vectors to relate to the "semantic distance" between the associated words. Words that mean different things should lie far away from each other, whereas related words should be closer.

Word embeddings are vector representations of words that achieve precisely this: they map human language into a structured geometric space.

Whereas the vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional (the same dimensionality as the number of words in the vocabulary), word embeddings are low-dimensional floating-point vectors (that is, dense vectors, as opposed to sparse vectors); see figure 14.3. It's common to see word embeddings that are 256-dimensional, 512-dimensional, or 1,024-dimensional when dealing with very large vocabularies. On the other hand, one-hot encoding words generally leads to vectors that are 30,000-dimensional in the case of our current vocabulary. So word embeddings pack more information into far fewer dimensions.

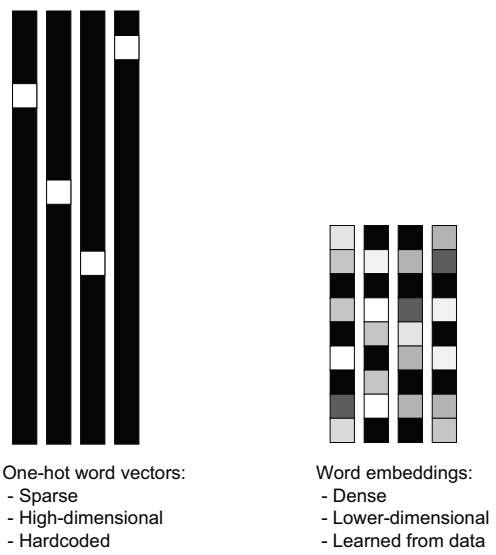


Figure 14.3 Word representations obtained from one-hot encoding or hashing are sparse, high-dimensional, and hardcoded. Word embeddings are dense, relatively low-dimensional, and learned from data.

Besides being dense representations, word embeddings are also structured representations, and their structure is learned from data. Similar words get embedded in close locations, and further, specific directions in the embedding space are meaningful. To make this clearer, let's look at a concrete example. In figure 14.4, four words are embedded on a 2D plane: cat, dog, wolf, and tiger. With the vector representations we chose here, some semantic relationships between these words can be encoded as geometric transformations. For instance, the same vector allows us to go from cat to tiger and from dog to wolf: this vector could be interpreted as the "from pet to wild animal" vector. Similarly, another vector lets us go from dog to cat and from wolf to tiger, which could be interpreted as a "from canine to feline" vector.

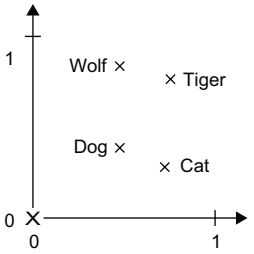


Figure 14.4 A toy example of a word-embedding space

In real-world word-embedding spaces, typical examples of meaningful geometric transformations are "gender" vectors and "plural" vectors. For instance, by adding a "female" vector to the vector "king," we obtain the vector "queen." By adding a "plural" vector, we obtain "kings." Word-embedding spaces typically feature thousands of such interpretable and potentially useful vectors.

Let's look at how to use such an embedding space in practice.

14.5.3 Using a word embedding

Is there an ideal word-embedding space that perfectly maps human language and can be used for any NPL task? Possibly, but we have yet to compute anything of the sort. Also, there is no single human language we could attempt to map—there are many different languages, and they aren't isomorphic to one another because a language is the reflection of a specific culture and a particular context. More pragmatically, what makes a good word-embedding space depends heavily on your task: the perfect word-embedding space for an English-language movie review sentiment-analysis model may look different from the ideal embedding space for an English-language legal document classification model because the importance of certain semantic relationships varies from task to task.

It's thus reasonable to learn a new embedding space with every new task. Fortunately, backpropagation makes this easy, and Keras makes it even easier. It's about learning the weights of the Keras `Embedding` layer.

The `Embedding` layer is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors. It takes integers as input, looks them up in an internal dictionary, and returns the associated vectors. It's effectively a dictionary lookup (see figure 14.5).

The `Embedding` layer takes as input a rank-2 tensor with shape `(batch_size, sequence_length)`, where each entry is a sequence of integers. The layer returns a floating-point tensor of shape `(batch_size, sequence_length, embedding_size)`.

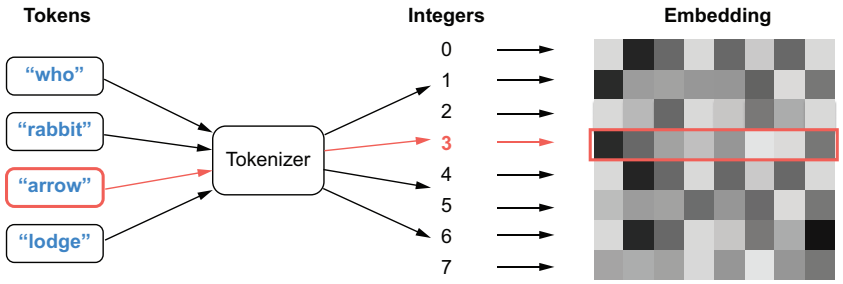


Figure 14.5 An `Embedding` layer acts as a dictionary mapping ints to floating point vectors.

When you instantiate an `Embedding` layer, its weights (its internal dictionary of token vectors) are initially random, just as with any other layer. During training, these word vectors are gradually adjusted via backpropagation, structuring the space into something the downstream model can exploit. Once fully trained, the embedding space will show a lot of structure—a kind of structure specialized for the specific problem for which you're training your model.

Let's build a model with an `Embedding` layer and benchmark it on our task.

Listing 14.24 Building an LSTM sequence model with an `Embedding` layer

```
hidden_dim = 64
inputs = keras.Input(shape=(max_length,), dtype="int32")
x = keras.layers.Embedding(
    input_dim=max_tokens,
    output_dim=hidden_dim,
    mask_zero=True,
)(inputs)
x = keras.layers.Bidirectional(keras.layers.LSTM(hidden_dim))(x)
x = keras.layers.Dropout(0.5)(x)
outputs = keras.layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs, name="lstm_with_embedding")
model.compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
```

The first two arguments to the `Embedding` layer are fairly straightforward. `input_dim` sets the total range of possible values for the integer inputs to the layer—that is, how many possible keys are there in our dictionary lookup. `output_dim` sets the dimensionality of the output vector we look up—that is, the dimensionality of our structured vector space for words.

The third argument, `mask_zero=True`, is a little more subtle. This argument tells Keras which inputs in our sequence are `"[PAD]"` tokens, so we can *mask* these entries later in the model.

Remember that when preprocessing our sequence input, we might add a lot of padding tokens to our original input so that a token sequence might look like:

```
["the", "movie", "was", "awful", "[PAD]", "[PAD]", "[PAD]", "[PAD]"]
```

All of those padding tokens will be first embedded and then fed into the `LSTM` layer. This means the last representation we receive from the `LSTM` cell might contain the results of processing the `"[PAD]"` token representation over and over recurrently. We are not very interested in the learned `LSTM` representation for the last `"[PAD]"` token in the previous sequence. Instead, we are interested in the representation of `"awful"`, the last non-padding token. Or, put equivalently, we want to mask all of the `"[PAD]"` tokens so that they do not affect our final output prediction.

`mask_zero=True` is simply a shorthand to easily do such masking in Keras with the `Embedding` layer. Keras will mark all elements in our sequence that initially contained a zero value, where zero is assumed to be the token ID for the `"[PAD]"` token. This mask will be used internally by the `LSTM` layer. Instead of outputting the last learned representation for the whole sequence, it will output the last non-masked representation.

This form of masking is *implicit* and easy to use, but you can always be explicit about which items in a sequence you would like to mask if the need arises. The `LSTM` layer takes an optional `mask` call argument, for explicit or custom masking.

Before we train this new model, let's take a look at the model summary:

```
>>> model.summary()
Model: "lstm_with_embedding"
```

Layer (type)	Output Shape	Param #	Connected to
input_layer_3 (InputLayer)	(None, 600)	0	-
embedding (Embedding)	(None, 600, 64)	1,920,000	input_layer_6[0][…
not_equal (NotEqual)	(None, 600)	0	input_layer_6[0][…
bidirectional_1 (Bidirectional)	(None, 128)	66,048	embedding[0][0], not_equal[0][0]
dropout_1 (Dropout)	(None, 128)	0	bidirectional_2[0…
dense_3 (Dense)	(None, 1)	129	dropout_2[0][0]

```
Total params: 1,986,177 (7.58 MB)
Trainable params: 1,986,177 (7.58 MB)
Non-trainable params: 0 (0.00 B)
```

We have reduced the number of parameters for our one-hot-encoded LSTM model from 15 million to 2 million. Let's train and evaluate the model.

Listing 14.25 Training and evaluating the LSTM with an `Embedding` layer

```
>>> model.fit(
...     sequence_train_ds,
...     validation_data=sequence_val_ds,
...     epochs=10,
...     callbacks=[early_stopping],
... )
>>> test_loss, test_acc = model.evaluate(sequence_test_ds)
>>> test_acc
0.8443599939346313
```

With the embedding, we have reduced both our training time and model size by an order of magnitude. A learned embedding is clearly far more efficient than one-hot encoding our input.

However, the LSTM's overall performance did not change. Accuracy was stubbornly around 84%, still a far cry from the bag-of-words and bigram models. Does this mean that a "structured embedding space" for input tokens is not that practically useful? Or is it not useful for text classification tasks?

Quite the contrary, a well-trained token embedding space can dramatically improve the practical performance ceiling of a model like this. The issue in this particular case is with our training setup. We lack enough data in our 20,000 review examples to effectively train a good word embedding. By the end of our 10 training epochs, our train set accuracy has cracked 99%. Our model has begun to overfit and memorize our input, and it turns out it is doing so well before we have learned an optimal set of word embeddings for the task at hand.

For cases like this, we can turn to *pretraining*. Rather than training our word embedding jointly with the classification task, we can train it separately, on more data, without the need for positive and negative review labels. Let's take a look.

Augmenting text data

After seeing the importance of data augmentation for computer vision problems, you might wonder if we can do the same for text. The short answer is yes, you can, although it is not nearly as effective in the text domain.

Basic text augmentation techniques look for basic edits we can make to input text that might help make our model more robust. For example, we might randomly delete

(continued)

or swap the position of words in a sentence, so the sentence "The rain in Spain falls mainly on the plain" becomes "The rain Spain falls plain on the mainly." Training your model on such edited inputs can make it robust to typos and grammar mistakes.

However, this example also concisely shows the big pitfall with text augmentation—it is easy to alter the meaning of an input example unintentionally. Unlike image data, where you can crop, rotate, and adjust color levels on a picture of a cat and still have a recognizable cat on the other end, language is order-dependent and highly sensitive to tiny changes. A sentence with two words swapped might mean the opposite of the input sentence. Some augmentation techniques seek to solve this by replacing words from a table of known synonyms, but this, too, can be brittle if we choose the incorrect meaning of a word. These issues have kept text augmentation from being widespread in practice. It is usually a better idea to seek more text examples than to sink time into text augmentation techniques.

Generative models, which we will see in the coming chapters, are beginning to offer a new form of text augmentation that can alleviate these pain points. By generating outputs from a model that has learned how to produce consistent and coherent text, we can create completely unseen inputs that plausibly resemble our input data. This poses its own challenges but opens up a new frontier for text augmentation on problems where data is particularly sparse and challenging to collect.

14.5.4 Pretraining a word embedding

The last decade of rapid advancement in NLP has coincided with the rise of *pretraining* as the dominant approach for text modeling problems. Once we move past simple set-based regression models to sequence models with millions or even billions of parameters, text models become incredibly data-hungry. We are usually limited by our ability to find labeled examples for a particular problem in the text domain.

The idea is to devise an unsupervised task to train model parameters that do not need labeled data. Pretraining data can be text in a similar domain to our final task, or even just arbitrary text in the languages we are interested in working with. Pretraining allows us to learn general patterns in language, effectively priming our model before we specialize it to the final task we are interested in.

Word embeddings were one of the first big successes with text pretraining, and we will show how to pretrain a word embedding in this section. Remember the `unsup/` directory we ignored in our IMDb dataset preparation? It contains another 25,000 reviews—the same size as our training data. We will combine all our training data together and show how to pretrain the parameters of an `Embedding` layer with an unsupervised task.

One of the most straightforward setups for training a word embedding is called the Continuous Bag of Words (CBOW) model.[2] The idea is to slide a window over all the

[2] Mikolov et al., "Efficient Estimation of Word Representations in Vector Space," International Conference on Learning Representations (2013), https://arxiv.org/abs/1301.3781.

text in a dataset, where we continuously attempt to guess a missing word based on the words that appear to its direct right and left (figure 14.6). For example, if our "bag" of surrounding words contained the words "sail," "wave," and "mast," we might guess that the middle word is "boat" or "ocean."

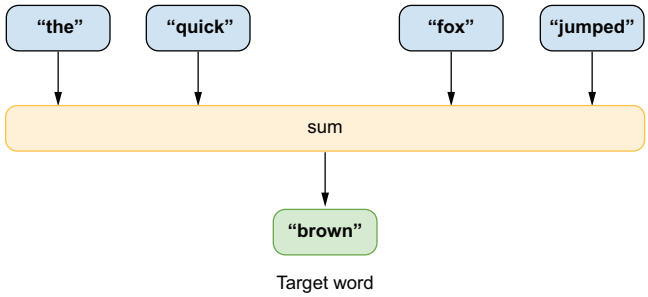


Figure 14.6 The Continuous Bag of Words predicts a word based on its surrounding context with a shallow neural network.

In our particular IMDb classification problem, we are interested in "priming" the word embedding of the LSTM model we just trained. We can reuse the `TextVectorization` vocabulary we computed earlier. All we are trying to do here is to learn a good 64-dimensional vector for each word in this vocabulary.

We can create a new `TextVectorization` layer with the same vocabulary that does not truncate or pad input. We will preprocess the output tokens of this layer by sliding a context window across our text.

Listing 14.26 Removing padding from our `TextVectorization` preprocessing layer

```
imdb_vocabulary = text_vectorization.get_vocabulary()
tokenize_no_padding = keras.layers.TextVectorization(
    vocabulary=imdb_vocabulary,
    split="whitespace",
    output_mode="int",
)
```

To preprocess our data, we will slide a window across our training data, creating "bags" of nine consecutive tokens. Then, we use the middle word as our label and the remaining eight words as an unordered context to predict our label.

To do this, we will again use `tf.data` to preprocess our inputs, although this choice does not limit the backend we use for actual model training.

Listing 14.27 Preprocessing our IMDb data for pretraining a CBOW model

```
import tensorflow as tf

context_size = 4
```

Words to the left or right of label

```
window_size = 9                                    ← Total window size

def window_data(token_ids):
    num_windows = tf.maximum(tf.size(token_ids) - context_size * 2, 0)
    windows = tf.range(window_size)[None, :]
    windows = windows + tf.range(num_windows)[:, None]
    windowed_tokens = tf.gather(token_ids, windows)
    return tf.data.Dataset.from_tensor_slices(windowed_tokens)

def split_label(window):
    left = window[:context_size]
    right = window[context_size + 1 :]
    bag = tf.concat((left, right), axis=0)
    label = window[4]
    return bag, label

dataset = keras.utils.text_dataset_from_directory(
    imdb_extract_dir / "train", batch_size=None
)
dataset = dataset.map(lambda x, y: x, num_parallel_calls=8)
dataset = dataset.map(tokenize_no_padding, num_parallel_calls=8)
dataset = dataset.interleave(window_data, cycle_length=8, num_parallel_calls=8)
dataset = dataset.map(split_label, num_parallel_calls=8)
```

Tokenizes

Drops label

Uses all training data, including the unsup/ directory

Splits middle wonder into a label

Creates context windows

After preprocessing, we can see that we have eight integer token IDs as context paired with a single token ID label.

The model we train with this data is exceedingly simple. We will use an `Embedding` layer to embed all context tokens and a `GlobalAveragePooling1D` to compute the average embedding of our "bag" of context tokens. Then, we use that average embedding to predict the value of our middle label token.

That's it! By repeatedly refining our embedding space so that we are good at predicting a word based on nearby word embeddings, we learn a rich embedding of tokens used in movie reviews.

Listing 14.28 Building a CBOW model

```
hidden_dim = 64
inputs = keras.Input(shape=(2 * context_size,))
cbow_embedding = layers.Embedding(
    max_tokens,
    hidden_dim,
)
x = cbow_embedding(inputs)
x = layers.GlobalAveragePooling1D()(x)
```

```
outputs = layers.Dense(max_tokens, activation="sigmoid")(x)
cbow_model = keras.Model(inputs, outputs)
cbow_model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["sparse_categorical_accuracy"],
)
>>> cbow_model.summary()
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_layer_4 (InputLayer)	(None, 8)	0
embedding_1 (Embedding)	(None, 8, 64)	1,920,000
global_average_pooling1d_2 (GlobalAveragePooling1D)	(None, 64)	0
dense_4 (Dense)	(None, 30000)	1,950,000

```
Total params: 3,870,000 (14.76 MB)
Trainable params: 3,870,000 (14.76 MB)
Non-trainable params: 0 (0.00 B)
```

Because our model is so simple, we can use a large batch size to speed up training without worrying about memory constraints.

We will also call `cache()` on this batched dataset so that we store the entire preprocessed dataset in memory rather than recomputing it each epoch. This is because for this very simple model, we are bottlenecked on preprocessing rather than training. That is, it is slower to tokenize our text and compute sliding windows on the CPU than to update our model parameters on the GPU.

In such cases, saving your preprocessed outputs in memory or on disk is usually a good idea. You will notice how our later epochs are more than three times faster than the first. This is thanks to the cache of preprocessed training data.

Listing 14.29 Training the CBOW model

```
dataset = dataset.batch(1024).cache()
cbow_model.fit(dataset, epochs=4)
```

At the end of training, we are able to guess the middle word around 12% of the time based solely on the neighboring eight words. This may not sound like a great result, but given that we have 30,000 words to guess from each time, this is actually a reasonable accuracy score.

Let's use this word embedding to improve the performance of our LSTM model.

14.5.5 Using the pretrained embedding for classification

Now that we have trained a new word embedding, applying it to our LSTM model is simple. First, we create the model precisely as we did before.

Listing 14.30 Building another LSTM sequence model with an `Embedding` layer

```
inputs = keras.Input(shape=(max_length,))
lstm_embedding = layers.Embedding(
    input_dim=max_tokens,
    output_dim=hidden_dim,
    mask_zero=True,
)
x = lstm_embedding(inputs)
x = layers.Bidirectional(layers.LSTM(hidden_dim))(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs, name="lstm_with_cbow")
```

Then, we apply our embedding weights from the CBOW embedding layer to the LSTM embedding layer. This effectively acts as a new and better initializer for the roughly 2 million embedding parameters in the LSTM model.

Listing 14.31 Reusing the CBOW embedding to prime the LSTM model

```
lstm_embedding.embeddings.assign(cbow_embedding.embeddings)
```

With that, we can compile and train our LSTM model as normal.

Listing 14.32 Training the LSTM model with a pretrained embedding.

```
model.compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
model.fit(
    sequence_train_ds,
    validation_data=sequence_val_ds,
    epochs=10,
    callbacks=[early_stopping],
)
```

Let's evaluate our LSTM model.

Listing 14.33 Evaluating the LSTM model with a pretrained embedding

```
>>> test_loss, test_acc = model.evaluate(sequence_test_ds)
>>> test_acc
0.89139
```

With the pretrained embedding weights, we have boosted our LSTM performance to roughly the same as our set-based models. We do slightly better than the unigram model and slightly worse than the bigram model.

This may seem like a bit of a letdown after all the work we put in. Is learning on the entire sequence, with order information, just a bad idea? The problem is that we are *still data-constrained* for our final LSTM model. The model is expressive and powerful enough that with enough movie reviews, we would easily outperform set-based approaches, but we need a lot more training on *ordered data* before our model's performance ceiling is reached.

This is an easily solvable problem with enough compute resources. In the next chapter, we will cover the transformer model. The model is slightly better at learning dependencies across longer token sequences, but most critically, these models are often trained on large amounts of English text, including all word order information. This allows the model to learn, roughly speaking, a statistical form of the grammatical patterns that govern language. These types of statistical patterns around word order are precisely why our current LSTM model is too data-constrained to learn effectively.

However, as we move on to large, more advanced models that will push the limits of text-classification performance, it is worth pointing out that simple set-based regression approaches like our bigram model give you a lot of "bang for your buck." Set-based models are lightning-fast and can contain just a few thousand parameters, a far cry from the billion-parameter large language models that dominate the news today.

If you are working in an environment where compute is limited and you can sacrifice some accuracy, set-based models can often be the most cost-effective approach.

Summary

- All text modeling problems involve a preprocessing step where text is broken up and transformed to integer data, called *tokenization*.
- Tokenization can be divided into three steps: *standardization*, *splitting*, and *indexing*. Standardization normalizes text, splitting breaks text up into tokens, and indexing assigns each token a unique integer ID.
- There are three main types of tokenization: *character*, *word*, and *subword* tokenization. With an expressive-enough model and sufficient training data, the *subword* tokenization is usually the most effective.
- NLP models differ primarily in handling the order of input tokens:
  - *Set models* discard most order information and learn simple and fast models based solely on the presence or absence of tokens in the input. *Bigram* or *trigram* models consider the presence or absence of two or three consecutive tokens. Set models are incredibly fast to train and deploy.
  - *Sequence models* attempt to learn with the ordered sequence of tokens in the input data. Sequence models need large amounts of data to learn effectively.

- An *embedding* is an efficient way to transform token IDs into a learned latent space. Embeddings can be trained normally with gradient descent.
- *Pretraining* is critical for sequence models as a way to get around the data-hungry nature of these models. During *pretraining*, an unsupervised task allows models to learn from large amounts of unlabeled text data. The learned parameters can then be transferred to a downstream task.

15 Language models and the Transformer

This chapter covers

- How to generate text with a deep learning model
- Training a model to translate from English to Spanish
- The Transformer, a powerful architecture for text modeling problems

With the basics of text preprocessing and modeling covered in the previous chapter, this chapter will tackle some more involved language problems such as machine translation. We will build up a solid intuition for the Transformer model that powers products like ChatGPT and has helped trigger a wave of investment in natural language processing (NLP).

15.1 The language model

In the previous chapter, we learned how to convert text data to numeric inputs, and we used this numeric representation to classify movie reviews. However, text classification is, in many ways, a uniquely simple problem. We only need to output a single floating-point number for binary classification and, at worst, *N* numbers for *N*-way classification.

What about other text-based tasks like question answering or translation? For many real-world problems, we are interested in a model that can generate a text output for a given input. Just like we needed tokenizers and embeddings to help us handle text on the *way in* to a model, we must build up some techniques before we can produce text on the *way out*.

We don't need to start from scratch here; we can continue to use the idea of an integer sequence as a natural numeric representation for text. In the previous chapter, we covered *tokenizing* a string, where we split inputs into tokens and map each token to an int. We can *detokenize* a sequence by proceeding in reverse—map ints back to string tokens and join them together. With this approach, our problem becomes building a model that can predict an integer sequence of tokens.

The simplest option to consider might be to train a direct classifier over the space of all possible output integer sequences, but some back-of-the-envelope math will quickly show this is intractable. With a vocabulary of 20,000 words, there are 20,000 ^ 4, or 160 quadrillion possible 4-word sequences, and fewer atoms in the universe than possible 20-word sequences. Attempting to represent every output sequence as a unique classifier output would overwhelm compute resources no matter how we design our model.

A practical approach for making such a prediction problem feasible is to build a model that only predicts a single token output at a time. A *language model* is a model that, in its simplest form, learns a straightforward but deep probability distribution: `p(token|past tokens)`. Given a sequence of all tokens observed up to a point, a language model will attempt to output a probability distribution over all possible tokens that could come next. A 20,000-word vocabulary means the model needs only predict 20,000 outputs, but by *repeatedly* predicting the next token, we will have built a model that can generate a long sequence of text.

Let's make this more concrete by building a simple language model that predicts the next character in a sequence of characters. We will train a small model that can output Shakespeare-like text.

15.1.1 Training a Shakespeare language model

To begin, we can download a collection of some of Shakespeare's plays and sonnets.

Listing 15.1 Downloading an abbreviated collection of Shakespeare's work

```
import keras

filename = keras.utils.get_file(
    origin=(
        "https://storage.googleapis.com/download.tensorflow.org/"
        "data/shakespeare.txt"
    ),
)
shakespeare = open(filename, "r").read()
```

Let's take a look at some of the data:

```
>>> shakespeare[:250]
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.
```

To build a *language model* from this input, we will need to massage our source text. First, we will split our data into equal-length chunks that we can batch and use for model training, much as we did for weather measurements in the timeseries chapters. Because we will be using a character-level tokenizer here, we can do this chunking directly on the string input. A 100-character string will map to a 100-integer sequence.

We will also split each input into two separate feature and label sequences, with each label sequence simply being the input sequence offset by a single character.

Listing 15.2 Splitting text into chunks for language model training

```
import tensorflow as tf

sequence_length = 100

def split_input(input, sequence_length):
    for i in range(0, len(input), sequence_length):
        yield input[i : i + sequence_length]

features = list(split_input(shakespeare[:-1], sequence_length))
labels = list(split_input(shakespeare[1:], sequence_length))
dataset = tf.data.Dataset.from_tensor_slices((features, labels))
```

The chunk size we will use during training. We only train on sequences of 100 characters at a time.

Let's look at an `(x, y)` input sample. Our label at each position in the sequence is the next character in the sequence:

```
>>> x, y = next(dataset.as_numpy_iterator())
>>> x[:50], y[:50]
(b"First Citizen:\nBefore we proceed any further, hear",
 b"irst Citizen:\nBefore we proceed any further, hear ")
```

To map this input to a sequence of integers, we can again use the `TextVectorization` layer we saw in the last chapter. To learn a character-level vocabulary instead of a word-level vocabulary, we can change our `split` argument. Rather than the default `"whitespace"` splitting, we instead split by `"character"`. We will do no standardization here—to keep things simple, we will preserve case and pass punctuation through unaltered.

Listing 15.3 Learning a character-level vocabulary with the `TextVectorization` layer

```
from keras import layers

tokenizer = layers.TextVectorization(
    standardize=None,
    split="character",
    output_sequence_length=sequence_length,
)
tokenizer.adapt(dataset.map(lambda text, labels: text))
```

Let's inspect the vocabulary:

```
>>> vocabulary_size = tokenizer.vocabulary_size()
>>> vocabulary_size
67
```

We need only 67 characters to handle the full source text.

Next, we can apply our tokenization layer to our input text. And finally, we can shuffle, batch, and cache our dataset so we don't need to recompute it every epoch:

```
dataset = dataset.map(
    lambda features, labels: (tokenizer(features), tokenizer(labels)),
    num_parallel_calls=8,
)
training_data = dataset.shuffle(10_000).batch(64).cache()
```

With that, we are ready to start modeling.

To build our simple language model, we want to predict the probability of a character given all past characters. Of all the modeling possibilities we have seen so far in this book, an RNN is the most natural fit, as the recurrent state of each cell allows the model to propagate information about past characters when predicting the label of the current character. We can also use an `Embedding`, as we saw in the previous chapter, to embed each input character as a unique 256-dimensional vector.

We will use only a single recurrent layer to keep this model small and easy to train. Any recurrent layer would do here, but to keep things simple, we will use a `GRU`, which is fast and has a simpler internal state than an `LSTM`.

Listing 15.4 Building a miniature language model

```
embedding_dim = 256
hidden_dim = 1024

inputs = layers.Input(shape=(sequence_length,), dtype="int", name="token_ids")
x = layers.Embedding(vocabulary_size, embedding_dim)(inputs)
x = layers.GRU(hidden_dim, return_sequences=True)(x)
x = layers.Dropout(0.1)(x)
outputs = layers.Dense(vocabulary_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
```

Outputs a probability distribution over all potential tokens in our vocabulary

Let's take a look at our model summary:

```
>>> model.summary()
Model: "functional"
```

Layer (type)	Output Shape	Param #
token_ids (InputLayer)	(None, 100)	0
embedding (Embedding)	(None, 100, 256)	17,152
gru (GRU)	(None, 100, 1024)	3,938,304
dropout (Dropout)	(None, 100, 1024)	0
dense (Dense)	(None, 100, 67)	68,675

```
Total params: 4,024,131 (15.35 MB)
Trainable params: 4,024,131 (15.35 MB)
Non-trainable params: 0 (0.00 B)
```

This model outputs a softmax probability for every possible character in our vocabulary, and we will `compile()` it with a crossentropy loss. Note that our model is still training on a classification problem, it's just that we will make one classification prediction for every token in our sequence. For our batch of 64 samples with 100 characters each, we will predict 6,400 individual labels. Loss and accuracy metrics reported by Keras during training will be averaged first across each sequence and, second, across each batch.

Let's go ahead and train our language model.

Listing 15.5 Training a miniature language model

```
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
```

```
    metrics=["sparse_categorical_accuracy"],
)
model.fit(training_data, epochs=20)
```

After 20 epochs, our model can eventually predict the next character in our input sequences around 70% of the time.

15.1.2 Generating Shakespeare

Now that we have trained a model that can predict the next *individual* tokens with some accuracy, we would like to use it to extrapolate an entire predicted sequence. We can do this by calling the model in a loop, where the model's predicted output at one time step becomes the model's input at the next time step. A model built for this kind of feedback loop is sometimes called an *autoregressive* model.

To run such a loop, we need to perform a slight surgery on the model we just trained. During training, our model handled only a fixed sequence length of 100 tokens, and the `GRU` cell's state was handled implicitly when calling the layer. During generation, we would like to predict a single output token at a time and explicitly output the state of the `GRU`'s cell. We need to propagate that state, which contains all information the model has encoded about past input characters, the next time we call the model.

Let's make a model that handles a single input character at a time and allows explicitly passing the RNN state. Because this model will have the same computational structure, with slightly modified inputs and outputs, we can assign weights from one model to another.

Listing 15.6 Modifying the language model for autoregressive inference

```
inputs = keras.Input(shape=(1,), dtype="int", name="token_ids")
input_state = keras.Input(shape=(hidden_dim,), name="state")

x = layers.Embedding(vocabulary_size, embedding_dim)(inputs)
x, output_state = layers.GRU(hidden_dim, return_state=True)(
    x, initial_state=input_state
)
outputs = layers.Dense(vocabulary_size, activation="softmax")(x)
generation_model = keras.Model(
    inputs=(inputs, input_state),
    outputs=(outputs, output_state),
)
generation_model.set_weights(model.get_weights())
```

Creates a model that receives and outputs the RNN state

Copies the parameters from the original model

With this, we can call the model to predict an output sequence in a loop. Before we do, we will make explicit lookup tables so we switch from characters to integers and choose a *prompt*—a snippet of text we will feed as input to the model before we begin predicting new tokens:

```
tokens = tokenizer.get_vocabulary()
token_ids = range(vocabulary_size)
char_to_id = dict(zip(tokens, token_ids))
id_to_char = dict(zip(token_ids, tokens))

prompt = """
KING RICHARD III:
"""
```

To begin generation, we first need to "prime" the internal state of the GRU with our prompt. To do this, we will feed the prompt into the model one token at a time. This will compute the exact RNN state the model would see if this prompt had been encountered during training.

When we feed the very last character of the prompt into the model, our state output will capture information about the entire prompt sequence. We can save the final output prediction to later select the first character of our generated response.

Listing 15.7 Using a fixed prompt to compute a language model's starting state

```
input_ids = [char_to_id[c] for c in prompt]
state = keras.ops.zeros(shape=(1, hidden_dim))
for token_id in input_ids:
    inputs = keras.ops.expand_dims([token_id], axis=0)
    predictions, state = generation_model.predict((inputs, state), verbose=0)
```

Feeds the prompt character by character to update state

Now we are ready to let the model predict a new output sequence. In a loop, up to a desired length, we will continually select the most likely next character predicted by the model, feed that to the model, and persist the new RNN state. In this way, we can predict an entire sequence, a token at time.

Listing 15.8 Predicting with the language model a token at a time

```
import numpy as np

generated_ids = []
max_length = 250
for i in range(max_length):
    next_char = int(np.argmax(predictions, axis=-1)[0])
    generated_ids.append(next_char)
    inputs = keras.ops.expand_dims([next_char], axis=0)
    predictions, state = generation_model.predict((inputs, state), verbose=0)
```

The next character is the output index with the highest probability.

Generates characters one by one, computing a new state each iteration

Let's convert our output integer sequence to a string to see what the model predicted. To *detokenize* our input, we simply map all token IDs to strings and join them together:

```
output = "".join([id_to_char[token_id] for token_id in generated_ids])
print(prompt + output)
```

We get the following output:

```
KING RICHARD III:
Stay, men! hear me speak.

FRIAR LAURENCE:
Thou wouldst have done thee here that he hath made for them?

BUCKINGHAM:
What straight shall stop his dismal threatening son,
Thou bear them both. Here comes the king;
Though I be good to put a wife to him,
```

We have yet to produce the next great tragedy, but this is not terrible for two minutes of training on a minimal dataset. The goal of this toy example is to show the power of the language model setup. We trained the model on the narrow problem of guessing a single character at a time but still use it for a much broader problem, generating an open-ended, Shakespearean-like text response.

It's important to notice that this training setup only works because a recurrent neural network only passes information forward in the sequence. If you'd like, try replacing the `GRU` layer with a `Bidirectional(GRU(...))`. The training accuracy will zoom to above 99% immediately, and generation will stop working entirely. During training, our model sees the entire sequence each train step. If we "cheat" by letting information from the next token in the sequence affect the current token's prediction, we've made our problem trivially easy.

This *language modeling* setup is fundamental to countless problems in the text domain. It is also somewhat unique compared to other modeling problems we have seen so far in this book. We cannot simply call `model.predict()` to get the desired output. There is an entire loop, and a nontrivial amount of logic, that exists only at inference time! The looping of state in the RNN cell happens for both training and inference, but at no point during training do we feed a model's predicted labels back into itself as input.

15.2 Sequence-to-sequence learning

Let's take the language model idea and extend it to tackle an important problem—machine translation. Translation belongs to a class of modeling problems often called *sequence-to-sequence* modeling (or *seq2seq* if you are trying to save keystrokes). We seek to build a model that can take in a source text as a fixed input sequence and generate the translated text sequence as a result. Question answering is another classic sequence-to-sequence problem.

The general template behind sequence-to-sequence models is described in figure 15.1. During training the following happen:

- An encoder model turns the source sequence into an intermediate representation.
- A decoder is trained using the language modeling setup we saw previously. It will recursively predict the next token in the target sequence by looking at all previous target tokens *and* our encoder's representation of the source sequence.

During inference, we don't have access to the target sequence—we're trying to predict it from scratch. We will generate it one token at a time, just as we did with our Shakespeare generator:

- We obtain the encoded source sequence from the encoder.
- The decoder starts by looking at the encoded source sequence as well as an initial "seed" token (such as the string `"[start]"`) and uses them to predict the first real token in the sequence.
- The predicted sequence so far is fed back into the decoder, in a loop, until it generates a stop token (such as the string `"[end]"`).

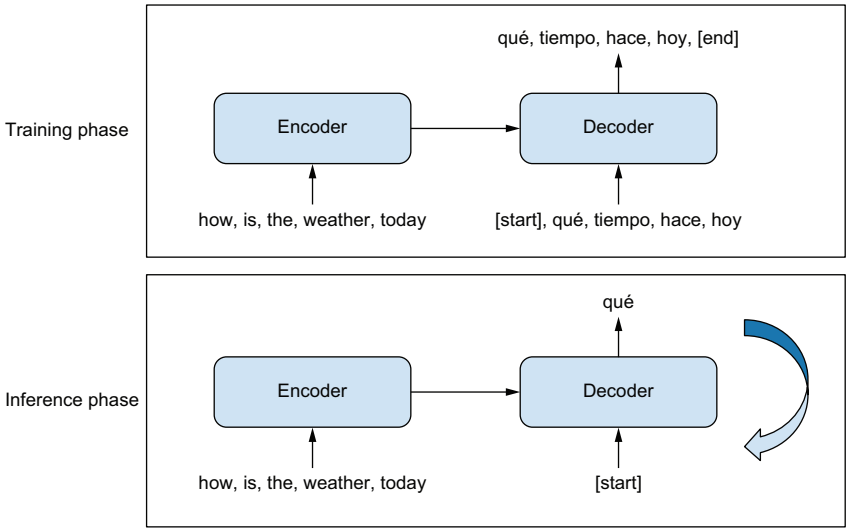


Figure 15.1 Sequence-to-sequence learning: the source sequence is processed by the encoder and is then sent to the decoder. The decoder looks at the target sequence so far and predicts the target sequence offset by one step in the future. During inference, we generate one target token at a time and feed it back into the decoder.

Let's build a sequence-to-sequence translation model.

15.2.1 English-to-Spanish translation

We'll be working with an English-to-Spanish translation dataset. Let's download it:

```
import pathlib

zip_path = keras.utils.get_file(
    origin=(
        "http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip"
    ),
    fname="spa-eng",
    extract=True,
)
text_path = pathlib.Path(zip_path) / "spa-eng" / "spa.txt"
```

The text file contains one example per line: an English sentence, followed by a tab character, followed by the corresponding Spanish sentence. Let's parse this file:

```
with open(text_path) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
for line in lines:
    english, spanish = line.split("\t")
    spanish = "[start] " + spanish + " [end]"
    text_pairs.append((english, spanish))
```

Our `text_pairs` look like this:

```
>>> import random
>>> random.choice(text_pairs)
("Who is in this room?", "[start] ¿Quién está en esta habitación? [end]")
```

Let's shuffle them and split them into the usual training, validation, and test sets:

```
import random

random.shuffle(text_pairs)
val_samples = int(0.15 * len(text_pairs))
train_samples = len(text_pairs) - 2 * val_samples
train_pairs = text_pairs[:train_samples]
val_pairs = text_pairs[train_samples : train_samples + val_samples]
test_pairs = text_pairs[train_samples + val_samples :]
```

Next, let's prepare two separate `TextVectorization` layers: one for English and one for Spanish. We're going to need to customize the way strings are preprocessed:

- We need to preserve the `"[start]"` and `"[end]"` tokens that we've inserted. By default, the characters `[` and `]` would be stripped, but we want to keep them around so we can distinguish the word `"start"` from the start token `"[start]"`.
- Punctuation is different from language to language! In the Spanish `TextVectorization` layer, if we're going to strip punctuation characters, we need to also strip the character `¿`.

Note that for a non-toy translation model, we would treat punctuation characters as separate tokens rather than stripping them since we would want to be able to generate correctly punctuated sentences. In our case, for simplicity, we'll get rid of all punctuation.

Listing 15.9 Learning token vocabularies for English and Spanish text

```
import string
import re

strip_chars = string.punctuation + "¿"
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(
        lowercase, f"[{re.escape(strip_chars)}]", ""
    )

vocab_size = 15000
sequence_length = 20

english_tokenizer = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
spanish_tokenizer = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length + 1,
    standardize=custom_standardization,
)
train_english_texts = [pair[0] for pair in train_pairs]
train_spanish_texts = [pair[1] for pair in train_pairs]
english_tokenizer.adapt(train_english_texts)
spanish_tokenizer.adapt(train_spanish_texts)
```

Finally, we can turn our data into a `tf.data` pipeline. We want it to return a tuple `(inputs, target, sample_weights)` where `inputs` is a dict with two keys, `"english"` (the tokenized English sentence) and `"spanish"` (the tokenized Spanish sentence), and `target` is the Spanish sentence offset by one step ahead. `sample_weights` here will

be used to tell Keras which labels to use when calculating our loss and metrics. Our output translations are not all equal in length, and some of our label sequences will be padded with zeros. We only care about predictions for non-zero labels that represent actual translated text.

This matches the "off by one" label set up in the generation model we just built, with the addition of the fixed encoder inputs, which will be handled separately in our model.

Listing 15.10 Tokenizing and preparing the translation data

```
batch_size = 64

def format_dataset(eng, spa):
    eng = english_tokenizer(eng)
    spa = spanish_tokenizer(spa)
    features = {"english": eng, "spanish": spa[:, :-1]}
    labels = spa[:, 1:]
    sample_weights = labels != 0
    return features, labels, sample_weights

def make_dataset(pairs):
    eng_texts, spa_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    spa_texts = list(spa_texts)
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, spa_texts))
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(format_dataset, num_parallel_calls=4)
    return dataset.shuffle(2048).cache()

train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)
```

Here's what our dataset outputs look like:

```
>>> inputs, targets, sample_weights = next(iter(train_ds))
>>> print(inputs["english"].shape)
(64, 20)
>>> print(inputs["spanish"].shape)
(64, 20)
>>> print(targets.shape)
(64, 20)
>>> print(sample_weights.shape)
(64, 20)
```

The data is now ready—time to build some models.

15.2.2 Sequence-to-sequence learning with RNNs

Before we try the twin encoder/decoder setup we previously mentioned, let's think through simpler options. The easiest, naive way to use RNNs to turn one sequence

into another is to keep the output of the RNN at each time step and predict an output token from it. In Keras, it would look like this:

```
inputs = keras.Input(shape=(sequence_length,), dtype="int32")
x = layers.Embedding(input_dim=vocab_size, output_dim=128)(inputs)
x = layers.LSTM(32, return_sequences=True)(x)
outputs = layers.Dense(vocab_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
```

However, there is a critical issue with this approach. Due to the step-by-step nature of RNNs, the model will only look at tokens `0...N` in the source sequence to predict token `N` in the target sequence. Consider translating the sentence, "I will bring the bag to you." In Spanish, that would be "Te traeré la bolsa," where "Te," the first word of the translation, corresponds to "you" in the English source text. There's simply no way to output the first word of the translation without seeing the last word of the source English text!

If you're a human translator, you'd start by reading the entire source sentence before beginning to translate it. This is especially important if you're dealing with languages with wildly different word ordering. And that's precisely what standard sequence-to-sequence models do. In a proper sequence-to-sequence setup (see figure 15.2), you would first use an encoder RNN to turn the entire source sequence into a single representation of the source text. This could be the last output of the RNN or, alternatively, its final internal state vectors. We can use this representation as the initial state of a decoder RNN in the language model setup instead of an initial state of zeros, which we used in our Shakespeare generator. This decoder learns to predict the next word of the Spanish translation given the current word of the translation, with all information about the English sequence coming from that initial RNN state.

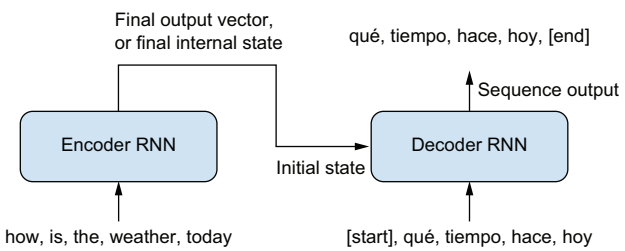


Figure 15.2 A sequence-to-sequence RNN: an RNN encoder is used to produce a vector that encodes the entire source sequence, which is used as the initial state for an RNN decoder.

Let's implement this in Keras, with GRU-based encoders and decoders. We can start with just the encoder. Since we will not actually be predicting tokens in the encoder sequence, we don't have to worry about "cheating" by letting the model pass information from the end of the sequence to positions at the beginning. In fact, this is a good

idea, as we want a rich representation of the source sequence. We can achieve this with a `Bidirectional` layer.

Listing 15.11 Building a sequence-to-sequence encoder

```
embed_dim = 256
hidden_dim = 1024

source = keras.Input(shape=(None,), dtype="int32", name="english")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(source)
rnn_layer = layers.GRU(hidden_dim)
rnn_layer = layers.Bidirectional(rnn_layer, merge_mode="sum")
encoder_output = rnn_layer(x)
```

Next, let's add the decoder—a simple `GRU` layer that takes as its initial state the encoded source sentence. On top of it, we add a `Dense` layer that produces a probability distribution over the Spanish vocabulary for each output step. Here, we want to predict the next tokens based only on what came before, so a `Bidirectional` RNN would break training by making the loss function trivially easy.

Listing 15.12 Building a sequence-to-sequence decoder

```
target = keras.Input(shape=(None,), dtype="int32", name="spanish")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(target)
rnn_layer = layers.GRU(hidden_dim, return_sequences=True)
x = rnn_layer(x, initial_state=encoder_output)
x = layers.Dropout(0.5)(x)
target_predictions = layers.Dense(vocab_size, activation="softmax")(x)
seq2seq_rnn = keras.Model([source, target], target_predictions)
```

Predicts the next word of the translation, given the current word

Let's take a look at the seq2seq model in full:

```
>>> seq2seq_rnn.summary()
Model: "functional_1"
```

Layer (type)	Output Shape	Param #	Connected to
english (InputLayer)	(None, None)	0	-
spanish (InputLayer)	(None, None)	0	-
embedding_1 (Embedding)	(None, None, 256)	3,840,000	english[0][0]
not_equal (NotEqual)	(None, None)	0	english[0][0]

embedding_2 (Embedding)	(None, None, 256)	3,840,000	spanish[0][0]
bidirectional (Bidirectional)	(None, 1024)	7,876,608	embedding_1[0][0], not_equal[0][0]
gru_2 (GRU)	(None, None, 1024)	3,938,304	embedding_2[0][0], bidirectional[0][…
dropout_1 (Dropout)	(None, None, 1024)	0	gru_2[0][0]
dense_1 (Dense)	(None, None, 15000)	15,375,000	dropout_1[0][0]

```
Total params: 34,869,912 (133.02 MB)
Trainable params: 34,869,912 (133.02 MB)
Non-trainable params: 0 (0.00 B)
```

Our model and data are both ready. We can now begin training our translation model:

```
seq2seq_rnn.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    weighted_metrics=["accuracy"],
)
seq2seq_rnn.fit(train_ds, epochs=15, validation_data=val_ds)
```

We picked accuracy as a crude way to monitor validation set performance during training. We get to 65% accuracy: on average, the model correctly predicts the next word in the Spanish sentence 65% of the time. However, in practice, next-token accuracy isn't a great metric for machine translation models, in particular because it makes the assumption that the correct target tokens from `0` to `N` are already known when predicting token `N + 1`. In reality, during inference, you're generating the target sentence from scratch, and you can't rely on previously generated tokens being 100% correct. When working on a real-world machine translation system, metrics must be more carefully designed. There are standard metrics, such as a BLEU score, that measure the similarity of the machine-translated text to a set of high-quality reference translations and can tolerate slightly misaligned sequences.

At last, let's use our model for inference. We'll pick a few sentences in the test set and check how our model translates them. We'll start from the seed token `"[start]"` and feed it into the decoder model, together with the encoded English source sentence. We'll retrieve a next-token prediction, and we'll re-inject it into the decoder repeatedly, sampling one new target token at each iteration, until we get to `"[end]"` or reach the maximum sentence length.

Listing 15.13 Generating translations with a seq2seq RNN

```
import numpy as np

spa_vocab = spanish_tokenizer.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))

def generate_translation(input_sentence):
    tokenized_input_sentence = english_tokenizer([input_sentence])
    decoded_sentence = "[start]"
    for i in range(sequence_length):
        tokenized_target_sentence = spanish_tokenizer([decoded_sentence])
        inputs = [tokenized_input_sentence, tokenized_target_sentence]
        next_token_predictions = seq2seq_rnn.predict(inputs, verbose=0)
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(5):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(generate_translation(input_sentence))
```

The exact translations will vary from run to run, as the final model weights will depend on the random initializations of our weights and the random shuffling of our input data. Here's what we got:

```
-
You know that.
[start] tú lo sabes [end]
-
"Thanks." "You're welcome."
[start] gracias tú [UNK] [end]
-
The prisoner was set free yesterday.
[start] el plan fue ayer a un atasco [end]
-
I will tell you tomorrow.
[start] te lo voy mañana a decir [end]
-
I think they're happy.
[start] yo creo que son felices [end]
```

Our model works decently well for a toy model, although it still makes many basic mistakes.

Note that this inference setup, while very simple, is inefficient, since we reprocess the entire source sentence and the entire generated target sentence every time we sample a new word. In a practical application, you'd want to be careful not to recompute any state that has not changed. All we really need to predict a new token in the decoder is the current token and the previous RNN state, which we could cache before each loop iteration.

There are many ways this toy model could be improved. We could use a deep stack of recurrent layers for both the encoder and the decoder, we could try other RNN layers like `LSTM`, and so on. Beyond such tweaks, however, the RNN approach to sequence-to-sequence learning has a few fundamental limitations:

- The source sequence representation has to be held entirely in the encoder state vector, which significantly limits the size and complexity of the sentences you can translate.
- RNNs have trouble dealing with very long sequences since they tend to progressively forget about the past—by the time you've reached the 100th token in either sequence, little information remains about the start of the sequence.

Recurrent neural networks dominated sequence-to-sequence learning in the mid-2010s. Google Translate circa 2017 was powered by a stack of seven large `LSTM` layers in a setup similar to what we just created. However, these limitations of RNNs eventually led to researchers developing a new style of sequence model, called the Transformer.

15.3 The Transformer architecture

In 2017, Vaswani et al. introduced the Transformer architecture in the seminal paper "Attention Is All You Need."[1] The authors were working on translation systems like the one we just built, and the critical discovery is right in the title. As it turned out, a simple mechanism called *attention* can be used to construct powerful sequence models that don't feature recurrent layers at all. The idea of attention was not new and had been used in NLP systems for a couple of years when they published. But the idea that attention was so useful it could be the *only* mechanism used to pass information across a sequence was quite surprising at the time.

This finding unleashed nothing short of a revolution in natural language processing—and beyond. Attention has fast become one of the most influential ideas in deep learning. In this section, you'll get an in-depth explanation of how it works and why it has proven so effective for sequence modeling. We'll then use attention to rebuild our English-to-Spanish translation model.

So, with all that as build-up, what exactly is attention? And how does it offer a replacement for the recurrent neural networks we have used so far?

Attention was actually developed as a way to augment an RNN model like the one we just built. Researchers noticed that while RNNs excelled at modeling dependencies in

[1] Vaswani et al., "Attention Is All You Need," Proceedings of the 31st International Conference on Neural Information Processing Systems (2017), https://arxiv.org/abs/1706.03762.

a local neighborhood, they struggled with recall as sequences got longer. Say you were building a system to answer questions about a source document. If the document length got too long, RNN results would get plain bad, a far cry from human performance.

As a thought experiment, imagine using this book to build a weather prediction model. If you had enough time, you might read the entire book cover to cover, but when you actually implemented your model, you would pay special attention to just the timeseries chapter. Even within a chapter, you might find specific code samples and explanations you would refer to often. On the other hand, you would not be particularly worried about the details of image convolutions as you worked on your code. The overall word count of this book is well over 100,000, far beyond any sequence length we have tackled, but humans can be *selective* and *contextual* in how we pull information from text.

RNNs, on the other hand, lack any mechanism to refer back to a previous section of a sequence directly. All information must, by design, be passed through an RNN cell's internal state in a loop, through *every* position in a sequence. It's a bit like finishing this book, closing it, and trying to implement that weather prediction model entirely from memory. The idea with attention is to build a mechanism by which a neural network can give more weight to some part of a sequence and less weight to others contextually, depending on the current input being processed (figure 15.3).

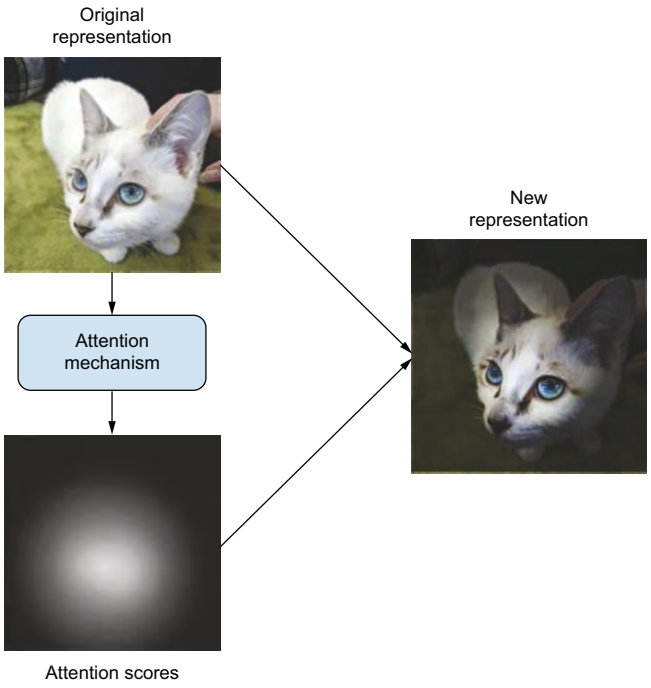


Figure 15.3 The general concept of attention in deep learning: input features get assigned attention scores, which can be used to inform the next representation of the input.

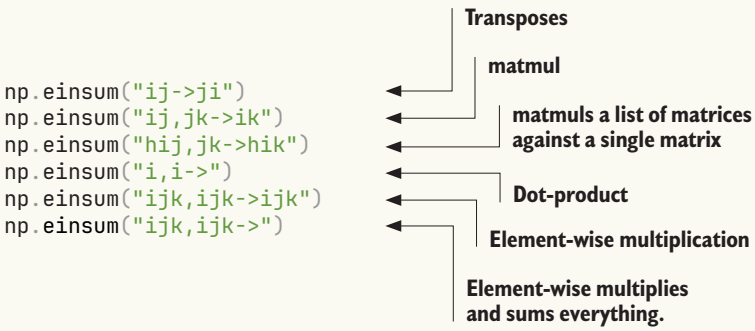
What is Einsum notation?

In machine learning codebases, you will frequently see little snippets that look like this: `np.einsum('ij,jk->ik', a, b)`. This is called *Einsum notation*, short for Einstein summation notation. Once you learn to read them, they can be a clear way to write complex array operations. You will see them frequently in Transformer code for this reason.

The idea with an Einsum equation is to represent each axis of an input with a unique letter. For example, you could represent a rank-3 input as `ijk`. Then, you write an equation with any number of inputs and a single output `input1,input2->output`. The rules of this equation are as follows:

- For any repeated letters across inputs, multiply values along these axes together. The size of these dimensions should match.
- For any letters in the input but not the output, sum over these axes so they do not appear in the returned array.
- Output axes can be returned in any order.

This gets much clearer if we take a look at some examples:



In Keras, you can use `einsum`s in two ways. `keras.ops.einsum` is a drop-in replacement for `np.einsum`, and `keras.layers.EinsumDense` is a `Dense` layer where the `matmul` is replaced by an `einsum` operation.

15.3.1 Dot-product attention

Let's revisit our translation RNN and try to add the notion of selective attention. Consider predicting just a single token. After passing the `source` and `target` sequences through our `GRU` layers, we will have a vector representing the target token we are about to predict and a sequence of vectors representing each word in the source text.

With attention, our goal is to give the model a way to *score* every single vector in our source sequence based on its *relevance* to the current word we are trying to predict (figure 15.4). If the vector representation of a source token has a high score, we consider it particularly important; if not, we care less about it. For now, let's assume we have this function `score(target_vector, source_vector)`.

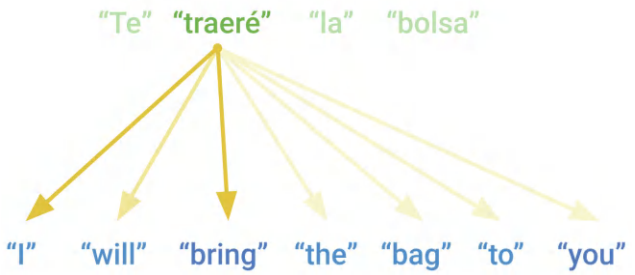


Figure 15.4 Attention assigns a relevance score to each vector in a source for each vector in a target sequence.

For attention to work well, we want to avoid passing information about important tokens through a loop potentially as long as our combined source and target sequence length—this is where RNNs start to fail. A simple way to do this is to take a weighted sum of all the source vectors based on this score we will compute. It would also be convenient if the sum of all attention scores for a given target were 1, as this would give our weighted sum a predictable magnitude. We can achieve this by running the scores through a `softmax` function—something like this, in NumPy pseudocode:

```
scores = [score(target, source) for source in sources]
scores = softmax(scores)
combined = np.sum(scores * sources)
```

But how should we compute this relevance score? When researchers first worked with attention, this question was a big topic of inquiry. It turns out that one of the most straightforward approaches is best. We can use a dot-product as a simple measure of the distance between target and source vectors. If the source and target vectors are close together, we assume that means the source token is relevant to our prediction. At the end of this chapter, we will examine why this assumption makes intuitive sense.

Let's update our pseudocode. We can make our snippet more complete by handling the entire target sequence at once—it will be equivalent to running our previous snippet in a loop for each token in the target sequence. When both `target` and `source` are sequences, the attention scores will be a matrix. Each row represents how much a target word will value a source word in the weighted sum (see figure 15.5). We will use the Einsum notation as a convenient way to write the dot-product and weighted sum:

Takes the dot-product between all target and source vectors, where b = batch size, t = target length, s = source length, and d = vector size

```
def dot_product_attention(target, source):
    scores = np.einsum("btd,bsd->bts", target, source)
    scores = softmax(scores, axis=-1)
```

```
    return np.einsum("bts,bsd->btd", scores, source)

dot_product_attention(target, source)
```

Computes a weighted sum of all source vectors for each target vector

We can make the *hypothesis space* of this attention mechanism much richer if we give the model parameters to control the attention score. If we project both source and target vectors with `Dense` layers, the model can find a good shared space where source vectors are close to target vectors if they help the overall prediction quality. Similarly, we should allow the model to project the source vectors into an entirely separate space before they are combined and once again after the summation.

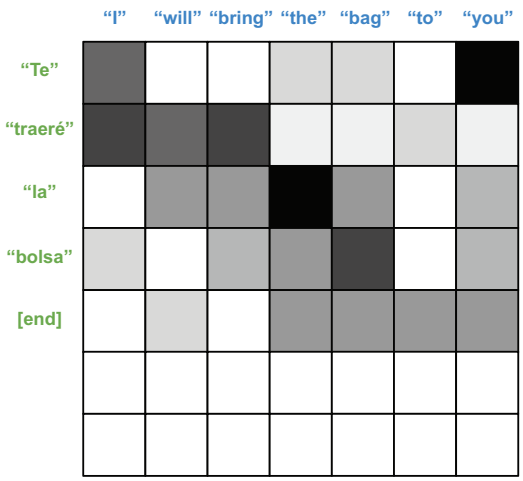


Figure 15.5 When both target and source are sequences, attention scores are a 2D matrix. Each row shows the attention scores for the word we are trying to predict (in green).

We can also adopt a slightly different naming for inputs that has become standard in the field. What we just wrote is roughly summarized as `sum(score(target, source) * source)`. We will write this equivalently with different input names as `sum(score(query, key) * value)`. This three-argument version is more general—in rare cases, you might not want to use the same vector to score your source inputs as you use to sum your source inputs.

The terminology comes from search engines and recommender systems. Imagine a search tool to look up photos in a database—the "query" is your search term, the "keys" are photo tags you use to match with the query, and finally, the "values" are the photos themselves (figure 15.6). The attention mechanism we are building is roughly analogous to this sort of lookup.

Let's update our pseudocode, so we have a parameterized attention using our new terminology:

```
query_dense = layers.Dense(dim)
key_dense = layers.Dense(dim)
value_dense = layers.Dense(dim)
output_dense = layers.Dense(dim)
```

```
def parameterized_attention(query, key, value):
    query = query_dense(query)
    key = key_dense(key)
    value = value_dense(value)
    scores = np.einsum("btd,bsd->bts", query, key)
    scores = softmax(scores, axis=-1)
    outputs = np.einsum("bts,bsd->btd", scores, value)
    return output_dense(outputs)

parameterized_attention(query=target, key=source, value=source)
```

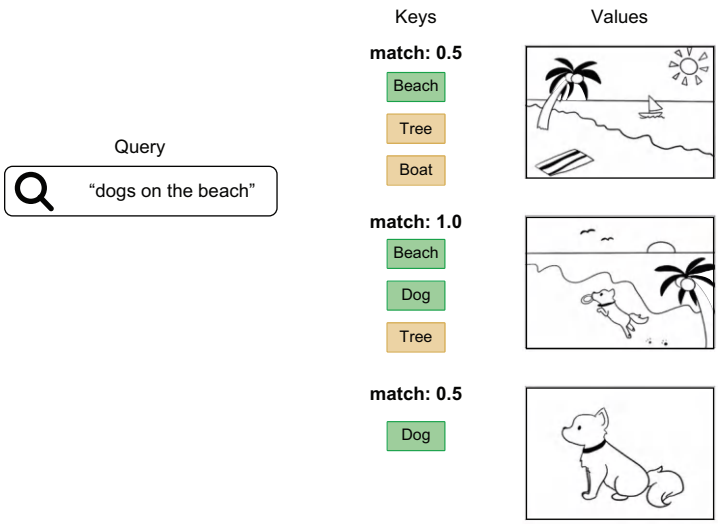


Figure 15.6 Retrieving images from a database: the *query* is compared to a set of *keys*, and the match scores are used to rank *values* (images).

This block is a perfectly functional attention mechanism! We just wrote a function that will allow the model to pull information from anywhere in the source sequence, contextually, depending on the target word we are decoding.

The "Attention is all you need" authors made two more changes to our mechanism through trial and error. The first is a simple scaling factor. When input vectors get long, the dot-product scores can get quite large, which can affect the stability of our softmax gradients. The fix is simple: we can scale down our softmax scores slightly. Scaling by the square root of the vector length works well for any vector size.

The other has to do with the expressivity of the attention mechanism. The softmax sum we are doing is powerful—it allows a direct connection across distant parts of a sequence. But the summation is also blunt: if the model tries to attend to too many

tokens at once, the interesting features of individual source tokens will get "washed out" in the combined representation. A simple trick that works well is to do this attention operation several times for the same sequence, with several different attention *heads* running the same computation with different parameters:

```
query_dense = [layers.Dense(head_dim) for i in range(num_heads)]
key_dense = [layers.Dense(head_dim) for i in range(num_heads)]
value_dense = [layers.Dense(head_dim) for i in range(num_heads)]
output_dense = layers.Dense(head_dim * num_heads)

def multi_head_attention(query, key, value):
    head_outputs = []
    for i in range(num_heads):
        query = query_dense[i](query)
        key = key_dense[i](key)
        value = value_dense[i](value)
        scores = np.einsum("btd,bsd->bts", target, source)
        scores = softmax(scores / math.sqrt(head_dim), axis=-1)
        head_output = np.einsum("bts,bsd->btd", scores, source)
        head_outputs.append(head_output)
    outputs = ops.concatenate(head_outputs, axis=-1)
    return output_dense(outputs)

multi_head_attention(query=target, key=source, value=source)
```

By projecting the query and key differently, one head might learn to match the subject of the source sentence, while another head might attend to punctuation. This multi-headed attention avoids the limitation of needing to combine the entire source sequence with a single softmax sum (figure 15.7).

Figure 15.7 Multi-headed attention allows each target word to attend to different parts of the source sequence in separate partitions of the eventual output vector.

Of course, in practice, you would want to write this code as a reusable layer. Here, Keras has you covered. We can recreate our previous code with the `MultiHeadAttention` layer as follows:

```
multi_head_attention = keras.layers.MultiHeadAttention(
    num_heads=num_heads,
    head_dim=head_dim,
)
multi_head_attention(query=target, key=source, value=source)
```

15.3.2 Transformer encoder block

One way to use the `MultiHeadAttention` layer would be to add it to our existing RNN translation model. We could pass the sequence output from our encoder and decoder into an attention layer and use its output to update our target sequence before prediction. Attention would allow the model to handle long-range dependencies in text that the `GRU` layer will struggle with. This does, in fact, improve an RNN model's capabilities and is how attention was first used in the mid-2010s.

However, the authors of "Attention is all you need" realized you could go further and use attention as a general mechanism for handling all sequence data in a model. Although so far we have only looked at attention as a way to handle information passing between two sequences, you could also use attention as a way to let a sequence attend to itself:

```
multi_head_attention(key=source, value=source, query=source)
```

This is called *self-attention*, and it is quite powerful. With self-attention, each token can attend to every token in its own sequence, including itself, allowing the model to learn a representation of the word in context.

Consider an example sentence: "The train left the station on time." Now, consider one word in the sentence: "station." What kind of station are we talking about? Could it be a radio station? Maybe the International Space Station? With self-attention, the model could learn to give a high attention score to the pair of "station" and "train," summing the vector used to represent "train" into the representation of the word "station."

Self-attention gives the model an effective way to go from representing a word in a vacuum to representing a word conditioned on all other tokens that appear in the sequence. This sounds a lot like what an RNN is supposed to do. Can we just go ahead and replace our RNN layers with `MultiHeadAttention`?

Almost! But not quite; we still need an essential ingredient for any deep neural network—a nonlinear activation function. The `MultiHeadAttention` layer combines linear projections of every element in a source sequence, but that's it. In a sense, it's a very expressive pooling operation. Consider, in the extreme case, a token length of one. In this case, the attention score matrix is always a single one, and the entire layer boils down to a linear projection of the source sequence, with no nonlinearities. You could stack 100 attention layers together and still be able to simplify the entire computation

to a single matrix multiplication! That's a real problem with the expressiveness of our model.

At some point, all recurrent cells pass the input vector for each token through a dense projection and apply an activation function; we need a plan for something similar. The authors of "Attention is all you need" decided to add this back in the simplest way possible—stacking a feedforward network of two dense layers with an activation in the middle. Attention passes information across the sequence, and the feedforward network updates the representation of individual sequence items.

We are ready to start building a Transformer model. Let's start by replacing the encoder of our translation model. We will use self-attention to pass information along the source sequence of English words. We will also add in two things we learned to be particularly important when building ConvNets back in chapter 9, *normalization* and *residual connections*.

Listing 15.14 A Transformer encoder block

```
class TransformerEncoder(keras.Layer):
    def __init__(self, hidden_dim, intermediate_dim, num_heads):
        super().__init__()
        key_dim = hidden_dim // num_heads
        self.self_attention = layers.MultiHeadAttention(num_heads, key_dim)
        self.self_attention_layernorm = layers.LayerNormalization()
        self.feed_forward_1 = layers.Dense(intermediate_dim, activation="relu")
        self.feed_forward_2 = layers.Dense(hidden_dim)
        self.feed_forward_layernorm = layers.LayerNormalization()

    def call(self, source, source_mask):
        residual = x = source
        mask = source_mask[:, None, :]
        x = self.self_attention(query=x, key=x, value=x, attention_mask=mask)
        x = x + residual
        x = self.self_attention_layernorm(x)
        residual = x
        x = self.feed_forward_1(x)
        x = self.feed_forward_2(x)
        x = x + residual
        x = self.feed_forward_layernorm(x)
        return x
```

Feedforward layers

Self-attention layers

Self-attention computation

Feedforward computation

You'll note that the normalization layers we're using here aren't `BatchNormalization` layers like those we've used in image models. That's because `BatchNormalization` doesn't work well for sequence data. Instead, we're using the `LayerNormalization` layer, which normalizes each sequence independently from other sequences in the batch—like this, in NumPy-like pseudocode:

```
def layer_normalization(batch_of_sequences):
    mean = np.mean(batch_of_sequences, keepdims=True, axis=-1)
    variance = np.var(batch_of_sequences, keepdims=True, axis=-1)
    return (batch_of_sequences - mean) / variance
```

Input shape: (batch_size, sequence_length, embedding_dim)

To compute mean and variance, we only pool data over the last axis.

Compare to `BatchNormalization` (during training):

```
def batch_normalization(batch_of_images):
    mean = np.mean(batch_of_images, keepdims=True, axis=(0, 1, 2))
    variance = np.var(batch_of_images, keepdims=True, axis=(0, 1, 2))
    return (batch_of_images - mean) / variance
```

Input shape: (batch_size, height, width, channels)

Pools data over the batch axis (axis 0), which creates interactions between samples in a batch

While `BatchNormalization` collects information from many samples to obtain accurate statistics for the feature means and variances, `LayerNormalization` pools data within each sequence separately, which is more appropriate for sequence data.

We also pass a new input to the `MultiHeadAttention` layer called `attention_mask`. This Boolean tensor input will be broadcast to the same shape as our attention scores `(batch_size, target_length, source_length)`. When set, it will zero the attention score in specific locations, stopping the source tokens at these locations from being used in the attention calculation. We will use this to prevent any token in the sequence from attending to padding tokens, which contain no information. Our encoder layer takes a `source_mask` input that will mark all the non-padding tokens in our inputs and upranks it to shape `(batch_size, 1, source_length)` to use as an `attention_mask`.

Note that the input and outputs of this layer have the same shape, so encoder blocks can be stacked on top of each other, building a progressively more expressive representation of the input English sentence.

15.3.3 Transformer decoder block

Next up is the decoder block. This layer will be almost identical to the encoder block, except we want the decoder to use the encoder output sequence as an input. To do this, we can use attention twice. We first apply a self-attention layer like our encoder, which allows each position in the target sequence to use information from other target positions. We then add another `MultiHeadAttention` layer, which receives both the

source and target sequence as input. We will call this attention layer *cross-attention* as it brings information across the encoder and decoder.

Listing 15.15 A Transformer decoder block

```
class TransformerDecoder(keras.Layer):
    def __init__(self, hidden_dim, intermediate_dim, num_heads):
        super().__init__()
        key_dim = hidden_dim // num_heads
        self.self_attention = layers.MultiHeadAttention(num_heads, key_dim)
        self.self_attention_layernorm = layers.LayerNormalization()
        self.cross_attention = layers.MultiHeadAttention(num_heads, key_dim)
        self.cross_attention_layernorm = layers.LayerNormalization()
        self.feed_forward_1 = layers.Dense(intermediate_dim, activation="relu")
        self.feed_forward_2 = layers.Dense(hidden_dim)
        self.feed_forward_layernorm = layers.LayerNormalization()

    def call(self, target, source, source_mask):
        residual = x = target
        x = self.self_attention(query=x, key=x, value=x, use_causal_mask=True)
        x = x + residual
        x = self.self_attention_layernorm(x)
        residual = x
        mask = source_mask[:, None, :]
        x = self.cross_attention(
            query=x, key=source, value=source, attention_mask=mask
        )
        x = x + residual
        x = self.cross_attention_layernorm(x)
        residual = x
        x = self.feed_forward_1(x)
        x = self.feed_forward_2(x)
        x = x + residual
        x = self.feed_forward_layernorm(x)
        return x
```

Cross-attention layers

Self-attention layers

Feedforward layers

Cross-attention computation

Feedforward computation

Self-attention computation

Our decoder layer takes in both a `target` and `source`. Like with the `TransformerEncoder`, we take in a `source_mask` that marks the location of all padding in the source input (`True` for non-padding, `False` for padding) and use it as an `attention_mask` for the cross-attention layer.

For the decoder's self-attention layer, we need a different type of attention mask. Recall that when we built our RNN decoder, we avoided using a `Bidirectional` RNN. If we had used one, the model would be able to cheat by seeing the label it was trying to predict as a feature! Attention is inherently bidirectional; in self-attention, any token position in the target sequence can attend to any other position. Without special care,

our model will learn to pass the next token in the sequence as the current label and will have no ability to generate novel translations.

We can achieve one-directional information flow with a special "causal" attention mask. Let's say we pass an attention mask with ones in the lower-triangular section like this:

```
[
    [1, 0, 0, 0, 0],
    [1, 1, 0, 0, 0],
    [1, 1, 1, 0, 0],
    [1, 1, 1, 1, 0],
    [1, 1, 1, 1, 1],
]
```

Each row `i` can be read as a mask for attention for the target token at position `i`. In the first row, the first token can only attend to itself. In the second row, the second token can attend to both the first and second tokens, and so forth. This gives us the same effect as our RNN layer, where information can only propagate forward in the sequence, not backward. In Keras, you can specify this lower-triangular mask simply by passing `use_casual_mask` to the `MultiHeadAttention` layer when calling it. Figure 15.8 shows a visual representation of the layers in both the encoder and decoder layers, when stacked into a Transformer model.

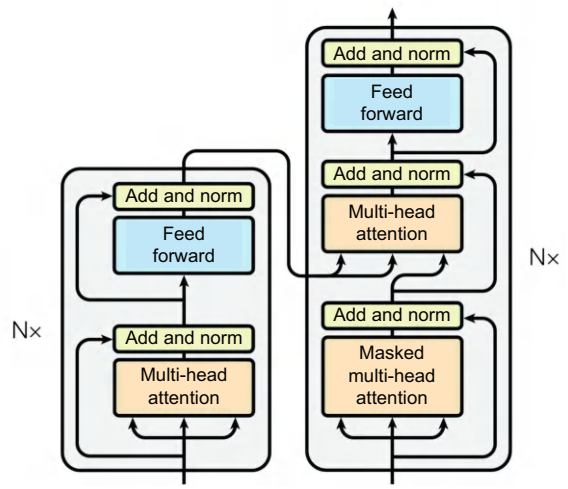


Figure 15.8 A visual representation of the computations for both `TransformerEncoder` and `TransformerDecoder` blocks

15.3.4 Sequence-to-sequence learning with a Transformer

Let's try putting this all together. We will use the same basic setup as our RNN model, replacing the `GRU` layers with our `TransformerEncoder` and `TransformerDecoder`. We will

use 256 as the embedding size throughout the model, except in the feedforward block. In the feedforward block, we scale up the embedding size to 2048 before nonlinearity and scale back to the model's hidden size afterward. This large intermediate dimension works well in practice.

Listing 15.16 Building a Transformer model

```
hidden_dim = 256
intermediate_dim = 2048
num_heads = 8

source = keras.Input(shape=(None,), dtype="int32", name="english")
x = layers.Embedding(vocab_size, hidden_dim)(source)
encoder_output = TransformerEncoder(hidden_dim, intermediate_dim, num_heads)(
    source=x,
    source_mask=source != 0,
)

target = keras.Input(shape=(None,), dtype="int32", name="spanish")
x = layers.Embedding(vocab_size, hidden_dim)(target)
x = TransformerDecoder(hidden_dim, intermediate_dim, num_heads)(
    target=x,
    source=encoder_output,
    source_mask=source != 0,
)
x = layers.Dropout(0.5)(x)
target_predictions = layers.Dense(vocab_size, activation="softmax")(x)
transformer = keras.Model([source, target], target_predictions)
```

Let's take a look at the summary of our Transformer model:

```
>>> transformer.summary()
Model: "functional_3"
```

Layer (type)	Output Shape	Param #	Connected to
english (InputLayer)	(None, None)	0	-
embedding_5 (Embedding)	(None, None, 256)	3,840,000	english[0][0]
not_equal_4 (NotEqual)	(None, None)	0	english[0][0]
spanish (InputLayer)	(None, None)	0	-
transformer_encoder_1 (TransformerEncoder)	(None, None, 256)	1,315,072	embedding_5[0][0], not_equal_4[0][0]
not_equal_5 (NotEqual)	(None, None)	0	english[0][0]

embedding_6 (Embedding)	(None, None, 256)	3,840,000	spanish[0][0]
transformer_decoder_1 (TransformerDecoder)	(None, None, 256)	1,578,752	transformer_encod… not_equal_5[0][0], embedding_6[0][0]
dropout_9 (Dropout)	(None, None, 256)	0	transformer_decod…
dense_11 (Dense)	(None, None, 15000)	3,855,000	dropout_9[0][0]

```
Total params: 14,428,824 (55.04 MB)
Trainable params: 14,428,824 (55.04 MB)
Non-trainable params: 0 (0.00 B)
```

Our model has almost exactly the same structure as the `GRU` translation model we trained earlier, with attention now substituting for recurrent layers as the mechanism to pass information across the sequence. Let's try training the model:

```
transformer.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    weighted_metrics=["accuracy"],
)
transformer.fit(train_ds, epochs=15, validation_data=val_ds)
```

After training, we get to about 58% accuracy: on average, the model correctly predicts the next word in the Spanish sentence 58% of the time. Something is off here. Training is worse than the RNN model by 7 percentage points. Either this Transformer architecture is not what it was hyped up to be, or we missed something in our implementation. Can you spot what it is?

This section is ostensibly about sequence models. In the previous chapter, we saw how vital word order could be to meaning. And yet, the Transformer we just built wasn't a sequence model at all. Did you notice? It's composed of dense layers that process sequence tokens independently of each other and an attention layer that looks at the tokens as a set. You could change the order of the tokens in a sequence, and you'd get identical pairwise attention scores and the same context-aware representations. If you were to rearrange every word in every English source sentence completely, the model wouldn't notice, and you'd still get the same accuracy. Attention is a set-processing mechanism, focused on the relationships between pairs of sequence elements—it's blind to whether these elements occur at the beginning, at the end, or in the middle of a sequence. So why do we say that Transformer is a sequence model? And how could it possibly be suitable for machine translation if it doesn't look at word order?

For RNNs, we relied on the layer's *computation* to be order aware. In the case of the Transformer, we instead inject positional information directly into our embedded sequence itself. This is called a *positional embedding*. Let's take a look.

15.3.5 Embedding positional information

The idea behind a positional embedding is very simple: to give the model access to word order information, we will add the word's position in the sentence to each word embedding. Our input word embeddings will have two components: the usual word vector, which represents the word independently of any specific context, and a position vector, which represents the position of the word in the current sentence. Hopefully, the model will then figure out how to best use this additional information.

The most straightforward scheme to add position information would be concatenating each word's position to its embedding vector. You'd add a "position" axis to the vector and fill it with `0` for the first word in the sequence, `1` for the second, and so on.

However, that may not be ideal because the positions can potentially be very large integers, which will disrupt the range of values in the embedding vector. As you know, neural networks don't like very large input values or discrete input distributions.

The "Attention is all you need" authors used an interesting trick to encode word positions: they added to the word embeddings a vector containing values in the range `[-1, 1]` that varied cyclically depending on the position (they used cosine functions to achieve this). This trick offers a way to uniquely characterize any integer in a large range via a vector of small values. It's clever, but it turns out we can do something simpler and more effective: we'll learn positional embedding vectors the same way we learn to embed word indices. We'll then add our positional embeddings to the corresponding word embeddings to obtain a position-aware word embedding. This is called a *positional embedding*. Let's implement it.

Listing 15.17 A learned position embedding layer

```
from keras import ops

class PositionalEmbedding(keras.Layer):
    def __init__(self, sequence_length, input_dim, output_dim):
        super().__init__()
        self.token_embeddings = layers.Embedding(input_dim, output_dim)
        self.position_embeddings = layers.Embedding(sequence_length, output_dim)

    def call(self, inputs):
        positions = ops.cumsum(ops.ones_like(inputs), axis=-1) - 1
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions
```

Computes incrementing positions [0, 1, 2...] for each sequence in the batch

We would use this `PositionalEmbedding` layer just like a regular `Embedding` layer. Let's see it in action as we try training our Transformer for a second time.

Listing 15.18 Building a Transformer model with positional embeddings

```
hidden_dim = 256
intermediate_dim = 2056
num_heads = 8

source = keras.Input(shape=(None,), dtype="int32", name="english")
x = PositionalEmbedding(sequence_length, vocab_size, hidden_dim)(source)
encoder_output = TransformerEncoder(hidden_dim, intermediate_dim, num_heads)(
    source=x,
    source_mask=source != 0,
)

target = keras.Input(shape=(None,), dtype="int32", name="spanish")
x = PositionalEmbedding(sequence_length, vocab_size, hidden_dim)(target)
x = TransformerDecoder(hidden_dim, intermediate_dim, num_heads)(
    target=x,
    source=encoder_output,
    source_mask=source != 0,
)
x = layers.Dropout(0.5)(x)
target_predictions = layers.Dense(vocab_size, activation="softmax")(x)
transformer = keras.Model([source, target], target_predictions)
```

With the positional embedding now added to our model, let's try training again:

```
transformer.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    weighted_metrics=["accuracy"],
)
transformer.fit(train_ds, epochs=30, validation_data=val_ds)
```

With positional information back in the model, things went much better. We achieved a 67% accuracy when guessing the next word. It's a noticeable improvement from the `GRU` model, and that's all the more impressive when you consider that this model has half the parameters of the GRU counterpart.

There's one other important thing to notice about this training run. Training is noticeably faster than the RNN—each epoch takes about a third of the time. This would be true even if we matched parameter count with the RNN model, and it is a side effect of getting rid of the looped state passing of our `GRU` layers. With attention, there is no looping computation to handle during training, meaning that on a GPU or TPU, we can handle the entire attention computation in one go. This makes the `Transformer` quicker to train on accelerators.

Let's rerun generation with our newly trained `Transformer`. We can use the same code as we did for our RNN sampling.

Listing 15.19 Generating translations with a Transformer

```
import numpy as np

spa_vocab = spanish_tokenizer.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))

def generate_translation(input_sentence):
    tokenized_input_sentence = english_tokenizer([input_sentence])
    decoded_sentence = "[start]"
    for i in range(sequence_length):
        tokenized_target_sentence = spanish_tokenizer([decoded_sentence])
        tokenized_target_sentence = tokenized_target_sentence[:, :-1]
        inputs = [tokenized_input_sentence, tokenized_target_sentence]
        next_token_predictions = transformer.predict(inputs, verbose=0)
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(5):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(generate_translation(input_sentence))
```

Running the generation code, we get the following output:

```
-
The resemblance between these two men is uncanny.
[start] el parecido entre estos cantantes de dos hombres son asombrosa [end]
-
I'll see you at the library tomorrow.
[start] te veré en la biblioteca mañana [end]
-
Do you know how to ride a bicycle?
[start] sabes montar en bici [end]
-
Tom didn't want to do their dirty work.
[start] tom no quería hacer su trabajo [end]
-
Is he back already?
[start] ya ha vuelto [end]
```

Subjectively, the Transformer performs significantly better than the GRU-based translation model. It's still a toy model, but it's a better toy model.

The Transformer is a powerful architecture that has laid the basis for an explosion of interest in text-processing models. It's also fairly complex, as deep learning models go. After seeing all of these implementation details, one might reasonably protest that this all seems quite arbitrary. There are so many small details to take on faith. How could we possibly know this choice and configuration of layers is optimal?

The answer is simple—it's not. Over the years, a number of improvements have been proposed to the Transformer architecture by making changes to attention, normalization, and positional embeddings. Many new models in research today are replacing attention altogether with something less computationally complex as sequence lengths get very long. Eventually, perhaps by the time you are reading this book, something will have supplanted the Transformer as the dominant architecture used for language modeling.

There's a lot we can learn from the Transformer that will stand the test of time. At the end of this chapter, we will discuss what makes the Transformer so effective. But it's worth remembering that, as a whole, the field of machine learning moves empirically. Attention grew out of an attempt to augment RNNs, and after years of guessing and checking by a ton of people, it gave rise to the Transformer. There is little reason to think this process is done playing out.

15.4 Classification with a pretrained Transformer

After "Attention is all you need," people started to notice how far Transformer training could scale, especially compared to models that had come before. As we just mentioned, one big plus was that the model is faster to train than RNNs. No more loops during training, which is always good when working with a GPU or TPU.

It is also a very data hungry model architecture. We actually got a little taste of this in the last section. While our RNN translation model plateaued in validation performance after 5 or so epochs, the Transformer model was still improving its validation score after 30 epochs of training.

These observations prompted many to try scaling up the Transformer with more data, layers, and parameters—with great results. This caused a distinctive shift in the field toward large pretrained models that can cost millions to train but perform noticeably better on a wide range of problems in the text domain.

For our last code example in the text section, we will revisit our IMDb text-classification problem, this time with a pretrained Transformer model.

15.4.1 Pretraining a Transformer encoder

One of the first pretrained Transformers to become popular in NLP was called BERT, short for Bidirectional Encoder Representations from Transformers.[2] The paper and

[2] Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (2019), https://arxiv.org/abs/1810.04805.

model were released a year after "Attention Is All You Need." The model structure was exactly the same as the encoder part of the translation Transformer we just built. This encoder model is *bidirectional* in that every position in the sequence can attend to positions in front of and behind it. This means it's a good model for computing a rich representation of input text, but not a model meant to run generation in a loop.

BERT was trained in sizes between 100 million and 300 million parameters, much bigger than the 14 million parameter Transformer we just trained. This meant the model needed a lot of training data to perform well. To achieve this, the authors used a riff on the classic language modeling setup called *masked language modeling*. To pretrain the model, we take a sequence of text and replace about 15% of the tokens with a special `[MASK]` token. The model will attempt to predict the original masked token values during training. While the classic language model, sometimes called a *causal language model*, attempts to predict `p(token|past tokens)`, the masked language model attempts to predict `p(token|surrounding tokens)`.

This training setup is unsupervised. You don't need any labels about the text you feed in; for any text sequence, you can easily choose some random tokens and mask them out. That made it easy for the authors to find a large amount of text data needed to train models of this size. For the most part, they pulled from Wikipedia as a source.

Using pretrained word embeddings was already common practice when BERT was released—we saw this ourselves in the last chapter. But pretraining an entire Transformer gave something much more powerful—the ability to compute a word embedding for a word in the *context* of the words around it. And the Transformer allowed doing this at a scale and quality that were unheard of at the time.

The authors of BERT took this model, pretrained on a huge amount of text, and specialized it to achieve state-of-the-art results on several NLP benchmarks at the time. This marked a distinctive shift in the field toward using very large, pretrained models, often with only a small amount of fine-tuning. Let's try this out.

15.4.2 Loading a pretrained Transformer

Instead of using BERT here, let's use a follow-up model called RoBERTa,[3] short for Robustly Optimized BERT. RoBERTa made some minor simplifications to BERT's architecture, but most notably used more training data to improve performance. BERT used 16 GB of English language text, mainly from Wikipedia. The RoBERTa authors used 160 GB of text from all over the web. It's estimated that RoBERTa cost a few hundred thousand dollars to train at the time. Because of this extra training data, the model performs noticeably better for an equivalent overall parameter count.

To use a pretrained model we will need a few things:

- *A matching tokenizer*—Used with the pretrained model itself. Any text must be tokenized in the same way as during pretraining. If the words of our IMDb

[3] Liu et al., "RoBERTa: A Robustly Optimized BERT Pretraining Approach" (2019), https://arxiv.org/abs/1907.11692.

reviews map to different token indices than they would have during pretraining, we cannot use the learned representations of each token in the model.

- *A matching model architecture*—To use the pretrained model, we need to recreate the math used internally by the model for pretraining exactly.
- *The pretrained weights*—These weights were created by training the model for about a day on 1,024 GPUs and billions of input words.

Recreating the tokenizer and architecture code ourselves would not be too hard. The model internals almost exactly match the `TransformerEncoder` we built previously. However, matching a model implementation is a time-consuming process, and as we have done earlier in this book, we can instead use the KerasHub library to access pretrained model implementations for Keras.

Let's use KerasHub to load a RoBERTa tokenizer and model. We can use the special constructor `from_preset()` to load a pretrained model's weights, configuration, and tokenizer assets from disk. We will load RoBERTa's base model, which is the smallest of the few pretrained checkpoints released with the RoBERTa paper.

Listing 15.20 Loading the RoBERTa pretrained model with KerasHub

```
import keras_hub

tokenizer = keras_hub.models.Tokenizer.from_preset("roberta_base_en")
backbone = keras_hub.models.Backbone.from_preset("roberta_base_en")
```

The `Tokenizer` maps from text to sequences of integers, as we would expect. Remember the `SubWordTokenizer` we built in the last chapter? RoBERTa's tokenizer is almost the same as that tokenizer, with minor tweaks to handle Unicode characters from any language.

Given the size of RoBERTa's pretraining dataset, subword tokenization is a must. Using a character-level tokenizer would make input sequences way too long, making the model much more expensive to train. Using a word-level tokenizer would require a massive vocabulary to attempt to cover all the distinct words in the millions of documents of text used from across the web. Getting good coverage of words would blow up our vocabulary size and make the `Embedding` layer at the front of the Transformer unworkably large. Using a subword tokenizer allows the model to handle any word with only a 50,000-term vocabulary:

```
>>> tokenizer("The quick brown fox")
Array([  133,  2119,  6219, 23602], dtype=int32)
```

What is this `Backbone` we just loaded? We saw in chapter 8 that a *backbone* is a term often used in computer vision for a network that maps from input images to a latent space—basically a vision model without a head for making predictions. In KerasHub,

a backbone refers to any pretrained model that is not yet specialized for a task. The model we just loaded takes in an input sequence and embeds it to an output sequence with shape `(batch_size, sequence_length, 768)`, but it's not set up for a particular loss function. You could use it for any number of downstream tasks—classifying sentences, identifying text spans with certain information, identifying parts of speech, etc.

Next, we will attach a classification head to this backbone that specializes it for our IMDb review classification fine-tuning. You can think of this as attaching different heads to a screwdriver: a Phillips head for one task, a flat head for another.

Let's take a look at our backbone. We loaded the *smallest* variant of RoBERTa here, but it still has 124 million parameters, which is the biggest model we have used in this book:

```
>>> backbone.summary()
Model: "roberta_backbone"
```

Layer (type)	Output Shape	Param #	Connected to
token_ids (InputLayer)	(None, None)	0	-
embeddings (TokenAndPositionEmb…	(None, None, 768)	38,996,736	token_ids[0][0]
embeddings_layer_norm (LayerNormalization)	(None, None, 768)	1,536	embeddings[0][0]
embeddings_dropout (Dropout)	(None, None, 768)	0	embeddings_layer_…
padding_mask (InputLayer)	(None, None)	0	-
transformer_layer_0 (TransformerEncoder)	(None, None, 768)	7,087,872	embeddings_dropou… padding_mask[0][0]
transformer_layer_1 (TransformerEncoder)	(None, None, 768)	7,087,872	transformer_layer… padding_mask[0][0]
...	...	...	...
transformer_layer_11 (TransformerEncoder)	(None, None, 768)	7,087,872	transformer_layer… padding_mask[0][0]

```
Total params: 124,052,736 (473.22 MB)
Trainable params: 124,052,736 (473.22 MB)
Non-trainable params: 0 (0.00 B)
```

RoBERTa uses 12 Transformer encoder layers stacked on top of each other. That's a big step up from our translation model!

15.4.3 Preprocessing IMDb movie reviews

We can reuse the IMDb loading code we used in chapter 14 unchanged. This will download the movie review data to a `train_dir` and `test_dir` and split a validation dataset into a `val_dir`:

```
zip_path = keras.utils.get_file(
    origin="https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz",
    fname="imdb",
    extract=True,
)

imdb_extract_dir = pathlib.Path(zip_path) / "aclImdb"
train_dir = pathlib.Path("imdb_train")
test_dir = pathlib.Path("imdb_test")
val_dir = pathlib.Path("imdb_val")

shutil.copytree(imdb_extract_dir / "test", test_dir, dirs_exist_ok=True)

val_percentage = 0.2
for category in ("neg", "pos"):
    src_dir = imdb_extract_dir / "train" / category
    src_files = os.listdir(src_dir)
    random.Random(1337).shuffle(src_files)
    num_val_samples = int(len(src_files) * val_percentage)

    os.makedirs(train_dir / category, exist_ok=True)
    os.makedirs(val_dir / category, exist_ok=True)
    for index, file in enumerate(src_files):
        if index < num_val_samples:
            shutil.copy(src_dir / file, val_dir / category / file)
        else:
            shutil.copy(src_dir / file, train_dir / category / file)

from keras.utils import text_dataset_from_directory

batch_size = 16
train_ds = text_dataset_from_directory(train_dir, batch_size=batch_size)
val_ds = text_dataset_from_directory(val_dir, batch_size=batch_size)
test_ds = text_dataset_from_directory(test_dir, batch_size=batch_size)
```

After loading, we once again have a training set of 20,000 movie reviews and a validation set of 5,000 movie reviews.

Before fine-tuning our classification model, we must tokenize our movie reviews with the RoBERTa tokenizer we loaded. During pretraining, RoBERTa used a specific form of "packing" tokens into a sequence, similar to what we did for our translation model. Each sequence would begin with an `<s>` token, end with an `</s>` token, and be followed by any number of `<pad>` tokens like this:

```
[
    ["<s>", "the", "quick", "brown", "fox", "jumped", ".", "</s>"],
    ["<s>", "the", "panda", "slept", ".", "</s>", "<pad>", "<pad>"],
]
```

It's important to match the token ordering used for pretraining as closely as possible; the model will train more quickly and accurately if we do. KerasHub provides a layer for this type of token packing called the `StartEndPacker`. The layer appends start, end, and padding tokens, trimming long sequences to a given sequence length if necessary. Let's use it along with our tokenizer.

Listing 15.21 Preprocessing IMDb movie reviews with RoBERTa's tokenizer

```
def preprocess(text, label):
    packer = keras_hub.layers.StartEndPacker(
        sequence_length=512,
        start_value=tokenizer.start_token_id,
        end_value=tokenizer.end_token_id,
        pad_value=tokenizer.pad_token_id,
        return_padding_mask=True,
    )
    token_ids, padding_mask = packer(tokenizer(text))
    return {"token_ids": token_ids, "padding_mask": padding_mask}, label

preprocessed_train_ds = train_ds.map(preprocess)
preprocessed_val_ds = val_ds.map(preprocess)
preprocessed_test_ds = test_ds.map(preprocess)
```

Let's take a look at a single preprocessed batch:

```
>>> next(iter(preprocessed_train_ds))
({"token_ids": <tf.Tensor: shape=(16, 512), dtype=int32, numpy=
  array([[   0,  713,   56, ...,    1,    1,    1],
         [   0, 1121,    5, ...,  101,   24,    2],
         [   0,  713, 1569, ...,    1,    1,    1],
         ...,
         [   0,  100, 3996, ...,    1,    1,    1],
         [   0,  100,   64, ..., 4655,  101,    2],
         [   0,  734,    8, ...,    1,    1,    1]], dtype=int32)>,
  "padding_mask": <tf.Tensor: shape=(16, 512), dtype=bool, numpy=
  array([[ True,  True,  True, ..., False, False, False],
         [ True,  True,  True, ...,  True,  True,  True],
         [ True,  True,  True, ..., False, False, False],
         ...,
         [ True,  True,  True, ..., False, False, False],
         [ True,  True,  True, ...,  True,  True,  True],
         [ True,  True,  True, ..., False, False, False]])>},
 <tf.Tensor: shape=(16,), dtype=int32, numpy=array([0, 1, ...], dtype=int32)>)
```

With our inputs preprocessed, we are ready to start fine-tuning our model.

Where does pretraining data come from?

Transformers are data-hungry models. They perform better the more input you throw at them at a scale previously unseen in deep learning. The original Transformer was trained on translation data with many millions of tokens. Not long after, Transformers were trained with billions, and today, trillions of tokens. That's a lot of written words.

So where does all this data come from? The answer has changed over time, but the short answer is the internet. Another answer is that this has increasingly become a secret. Companies will often not release the exact data they have used to train a model or describe the precise mixture of data sources used during training.

We can take a look at some pretraining datasets over time:

- The very first Transformer trained on a well-known English-German translation dataset with 4 million sentence pairs.
- BERT used a dump of English Wikipedia plus a dataset containing 7,000 self-published books.
- GPT2, a precursor to ChatGPT, scraped a dataset by following outgoing links from Reddit.
- The latest version of Llama, a pretrained Transformer released by Meta, is trained on "15 trillion tokens of data from publicly available sources." It is becoming increasingly common to leave the precise composition of data vague.

In the next chapter, we will see how important the exact mixture of pretraining data sources can be. When possible, it is always good to pay close attention to where a model's data came from, as it will shape the biases and performance of the model.

15.4.4 Fine-tuning a pretrained Transformer

Before we fine-tune our backbone to predict movie reviews, we need to update it so it outputs a binary classification label. The backbone outputs an entire sequence with shape `(batch_size, sequence_length, 768)`, where each 768-dimensional vector represents an input word in the context of its surrounding words. Before predicting a label, we must condense this sequence to a single vector per sample.

One option would be to do mean pooling or max pooling across the whole sequence, computing an average of all token vectors. What works slightly better is simply using the first token's representation as the pooled value. This is due to the nature of the attention in our model—the first position in the final encoder layer will be able to attend to all other positions in the sequence and pull information from them. So rather than pooling information with something coarse, like taking an average, attention allows us to pool information *contextually* across the sequence.

Let's now add a classification head to our backbone. We will also add one final `Dense` projection with a nonlinearity before generating an output prediction.

Listing 15.22 Extending the base RoBERTa model for classification

```
inputs = backbone.input
x = backbone(inputs)
x = x[:, 0, :]                                          ← Uses the hidden representation of the first token
x = layers.Dropout(0.1)(x)
x = layers.Dense(768, activation="relu")(x)
x = layers.Dropout(0.1)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
classifier = keras.Model(inputs, outputs)
```

With that, we are ready to fine-tune and evaluate the model on the IMDb dataset.

Listing 15.23 Training the RoBERTa classification model

```
classifier.compile(
    optimizer=keras.optimizers.Adam(5e-5),
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
classifier.fit(
    preprocessed_train_ds,
    validation_data=preprocessed_val_ds,
)
```

Finally, let's evaluate the trained model:

```
>>> classifier.evaluate(preprocessed_test_ds)
[0.168127179145813, 0.9366399645805359]
```

In just a single epoch of training, our model reached 93%, a noticeable improvement from the 90% ceiling we hit in our last chapter. Of course, this is a far more expensive model to use than the simple bigram classifier we built previously, but there are clear benefits to using such a large model. And this is all with the smaller size of the RoBERTa model. Using the larger 300 million parameter model, we could achieve an accuracy of over 95%.

15.5 What makes the Transformer effective?

In 2013, at Google, Tomas Mikolov and his colleagues noticed something remarkable. They were building a pretrained embedding called "Word2Vec," similar to the Continuous Bag of Words (CBOW) embedding we built in the last chapter. Much like our CBOW model, their training objective sought to turn correlation relationships between words into distance relationships in the embedding space: a vector was associated with each word in a vocabulary, and the vectors were optimized so that the dot-product (cosine proximity) between vectors representing frequently co-occurring words would

be closer to 1, while the dot-product between vectors representing rarely co-occurring words would be closer to 0.

They found that the resulting embedding space did much more than capture semantic similarity. It featured some form of emergent learning—a sort of "word arithmetic." A vector existed in the space that you could add to many male nouns to obtain a point that would land close to its female equivalent, as in `V(king) - V(man) + V(woman) = V(queen)`, a gender vector. This was quite surprising; the model had not been trained for this in any explicit way. There seemed to be dozens of such magic vectors—a plural vector, a vector to go from wild animals' names to their closest pet equivalent, etc.

Fast-forward about 10 years—we are now in the age of large, pretrained Transformer models. On the surface, these models couldn't seem any further from the primitive Word2Vec model. A Transformer can generate perfectly fluent language—a feat Word2Vec was entirely incapable of. As we will see in the next chapter, such models can seem knowledgeable about virtually any topic. And yet, they actually have a lot in common with good old Word2Vec.

Both models seek to embed tokens (words or subwords) in a vector space. Both rely on the same fundamental principle to learn this space: tokens that appear together end up close in the embedding space. The distance function used to compare tokens is cosine distance in both cases. Even the dimensionality of the embedding space is similar: a vector with somewhere between 1,000 and 10,000 dimensions to represent each word.

At this point, you might interject: A Transformer is trained to predict missing words in a sequence, not to group tokens in an embedding space. How does the language model loss function relate to Word2Vec's objective of maximizing the dot-product between co-occurring tokens? The answer is the attention mechanism.

Attention is, by far, the most critical component in the Transformer architecture. It's a mechanism for learning a new token embedding space by linearly recombining token embeddings from some prior space, in weighted combinations that give greater importance to tokens that are already "closer" to each other (i.e., that have a higher dot-product). It will tend to pull together the vectors of already close tokens, resulting over time in a space where token correlation relationships turn into embedding proximity relationships (in terms of cosine distance). Transformers work by learning a series of incrementally refined embedding spaces, each based on recombining elements from the previous one.

Attention provides Transformers with two crucial properties:

- The embedding spaces they learn are semantically continuous—that is, moving a bit in an embedding space only changes the human-facing meaning of the corresponding tokens by a bit. The Word2Vec space also exhibited this property.
- The embedding spaces they learn are semantically interpolative—that is, taking the intermediate point between two points in an embedding space produces a point representing the "intermediate meaning" between the corresponding

tokens. This comes from the fact that each new embedding space is built by interpolating between vectors from the previous space.

This is not entirely unlike the way the brain learns. The key learning principle in the brain is Hebbian learning—in short, "neurons that fire together, wire together." Correlation relationships between neural firing events (which may represent actions or perceptual inputs) are turned into proximity relationships in the brain network, just like the Transformer and Word2Vec turn correlation relationships into vector proximity relationships. Both are maps of a space of information.

Of course, there are significant differences between Word2Vec and the Transformer. Word2Vec was not designed for generative text sampling. A Transformer can get far bigger and can encode vastly more complex transformations. The thing is, Word2Vec is very much a toy model: it is to today's language models as a logistic regression on MNIST pixels is to state-of-the-art computer vision models. The fundamental principles are mostly the same, but the toy model lacks any meaningful representation power. Word2Vec wasn't even a deep neural network—it had a shallow, single-layer architecture. Meanwhile, today's Transformer models have the highest representation power of any model anyone has ever trained—they feature dozens of stacked attention and feedforward layers, and their parameter count ranges in the billions.

Like with Word2Vec, the Transformer learns useful semantic functions as a byproduct of organizing tokens into a vector space. But thanks to this increased representation power and a much more refined autoregressive optimization objective, we're no longer confined to linear transformations like a gender vector or a plural vector. Transformers can store arbitrarily complex vector functions—so complex, in fact, that it would be more accurate to refer to them as vector programs rather than functions.

Word2Vec enabled you to do basic things like `plural(cat) → cats` or `male_to_female(king) → queen`. Meanwhile, a large Transformer model can do pure magic—things like `write_this_in_style_of_shakespeare("...your poem...") → "...new poem..."`. And a single model can contain millions of such programs.

You can see a Transformer as analogous to a database: it stores information you can retrieve via the tokens you pass in. But there are two important differences between a Transformer and a database.

The first difference is that a Transformer is a continuous, interpolative kind of database. Instead of being stored as a set of discrete entries, your data is stored as a vector space—a curve. You can move around on the curve (it's semantically continuous, as we discussed) to explore nearby, related points. And you can interpolate on the curve between different data points to find their in-between. This means that you can retrieve a lot more from your database than you put into it—although not all of it will be accurate or meaningful. Interpolation can lead to generalization, but it can also lead to hallucinations—a significant problem facing the generative language models trained today.

The second difference is that a Transformer doesn't just contain data. For models like RoBERTa, trained on hundreds of thousands of documents scraped from the

internet, there is a lot of data: facts, places, people, dates, things, and relationships. But it's also—perhaps primarily—a database of programs.

They're different from the kind of programs you're used to dealing with, mind you. These are not like Python programs—series of symbolic statements processing data step by step. Instead, these vector programs are highly nonlinear functions that map the latent embedding space unto itself, analogous to Word2Vec's magic vectors, but far more complex.

In the next chapter, we will push Transformer models to an entirely new scale. Models will use billions of parameters and train on trillions of words. Output from these models can often feel like magic—like an intelligent operator sitting inside our model and pulling the strings. But it's important to remember that these models are fundamentally interpolative—thanks to attention, they learn an interpolative embedding space for a significant chunk of all text written in the English language. Wandering this embedding space can lead to interesting, unexpected generalizations, but it cannot synthesize something fundamentally new with anything close to genuine, human-level intelligence.

Summary

- A *language model* is a model that learns a specific probability distribution—`p(token|past tokens)`:
  - Language models have broad applications, but the most important is that you can generate text by calling them in a loop, where the output token at one time step becomes the input token in the next.
  - A *masked language model* learns a related probability distribution `p(tokens|surrounding tokens)` and can be helpful for classifying text and individual tokens.
  - A *sequence-to-sequence language model* learns to predict the next token given both past tokens in a target sequence and an entirely separate, fixed source sequence. Sequence-to-sequence models are useful for problems like translation and question answering.
  - A sequence-to-sequence model usually has two separate components. An *encoder* computes a representation of the source sequence, and a *decoder* takes this representation as input and predicts the next token in a target sequence based on past tokens.
- *Attention* is a mechanism that allows a model to pull information from anywhere in a sequence selectively based on the context of the token currently being processed:
  - Attention avoids the problems RNNs have with long-range dependencies in text.
  - Attention works by taking the dot-product of two vectors to compute an attention score. Vectors near each other in an embedding space will be summed together in the attention mechanism.

- The *Transformer* is a sequence modeling architecture that uses attention as the only mechanism to pass information across a sequence:
  - The Transformer works by stacking blocks of alternating attention and two-layer feedforward networks.
  - The Transformer can scale to many parameters and lots of training data while still improving accuracy in the language modeling problem.
  - Unlike RNNs, the Transformer involves no sequence-length loops at training time, making the model much easier to train in parallel across many machines.
  - A Transformer encoder uses bidirectional attention to build a rich representation of a sequence.
  - A Transformer decoder uses causal attention to predict the next word in a language model setup.

16 Text generation

This chapter covers

- A brief history of generative modeling
- Training a miniature GPT model from scratch
- Using a pretrained Transformer model to build a chatbot
- Building a multimodal model that can describe images in natural language

When I first claimed that in a not-so-distant future, most of the cultural content we consume would be created with substantial help from AIs, I was met with utter disbelief, even from long-time machine learning practitioners. That was in 2014. Fast-forward a decade, and that disbelief had receded at an incredible speed. Generative AI tools are now common additions to word processors, image editors, and development environments. Prestigious awards are going out to literature and art created with generative models—to considerable controversy and debate.[1] It no

[1] In 2022, Jason Allen used the image generation software Midjourney to win an award for digital artists, and in 2024, Rie Kudan won one of Japan's most prestigious literary awards for a novel written with substantial help from generative software.

longer feels like science fiction to consider a world where AI and artistic endeavors are often intertwined.

In any practical sense, AI is nowhere close to rivaling human screenwriters, painters, or composers. But replacing humans need not, and should not, be the point. In many fields, but especially in creative ones, people will use AI to augment their capabilities—more augmented intelligence than artificial intelligence.

Much of artistic creation consists of pattern recognition and technical skill. Our perceptual modalities, language, and artwork all have statistical structure, and deep learning models excel at learning this structure. Machine learning models can learn the statistical latent spaces of images, music, and stories, and they can then sample from these spaces, creating new artworks with characteristics similar to those the model has seen in its training data. Such sampling is hardly an act of artistic creation in itself—it's a mere mathematical operation. Only our interpretation, as human spectators, gives meaning to what the model generates. But in the hands of a skilled artist, algorithmic generation can be steered to become meaningful—and beautiful. Latent space sampling can become a brush that empowers the artist, augments our creative affordances, and expands the space of what we can imagine. It can even make artistic creation more accessible by eliminating the need for technical skill and practice—setting up a new medium of pure expression, factoring art apart from craft.

Figure 16.1 An image generated with the generative image software Midjourney. The prompt was "A hand-drawn, sci-fi landscape of residents living in a building shaped like a red letter K."

Iannis Xenakis, a visionary pioneer of electronic and algorithmic music, beautifully expressed this same idea in the 1960s, in the context of the application of automation technology to music composition:[2]

> *Freed from tedious calculations, the composer is able to devote himself to the general problems that the new musical form poses and to explore the nooks and crannies of this form while modifying the values of the input data. For example, he may test all instrumental combinations from soloists to chamber orchestras, to large orchestras. With the aid of electronic computers the composer becomes a sort of pilot: he presses the buttons, introduces coordinates, and supervises the controls of a cosmic vessel sailing in the space of sound, across sonic constellations and galaxies that he could formerly glimpse only as a distant dream.*

The potential for generative AI extends well beyond artistic endeavors. In many professions, people create content where pattern recognition is even more apparent: think of summarizing large documents, transcribing speech, editing for typos, or flagging common mistakes in code. These rote tasks play directly to the strengths of deep learning approaches. There is a lot to consider regarding how we choose to deploy AI in the workplace—with real societal implications.

In the following two chapters, we will explore the potential of deep learning to assist with creation. We will learn to curate latent spaces in text and image domains and pull new content from these spaces. We will start with text, scaling up the idea of a language model we first worked with in the last chapter. These *large language models*, or *LLMs* for short, are behind digital assistants like ChatGPT and a quickly growing list of real-world applications.

16.1 A brief history of sequence generation

Until quite recently, the idea of generating sequences from a model was a niche subtopic within machine learning—generative recurrent networks only began to hit the mainstream in 2016. However, these techniques have a fairly long history, starting with the development of the LSTM algorithm in 1997.

In 2002, Douglas Eck applied LSTM to music generation for the first time, with promising results. Eck became a researcher at Google Brain, and in 2016, he started a new research group called Magenta, which focused on applying modern deep learning techniques to produce engaging music. Sometimes, good ideas take 15 years to get started.

In the late 2000s and early 2010s, Alex Graves pioneered the use of recurrent networks for new types of sequence data generation. In particular, some see his 2013 work on applying recurrent mixture density networks to generate human-like handwriting using timeseries of pen positions as a turning point. Graves left a commented-out remark hidden in a 2013 LaTeX file uploaded to the preprint server arXiv: "Generating sequential data is the closest computers get to dreaming." This work and the notion of machines that dream were significant inspirations when I started developing Keras.

---

[2] Iannis Xenakis, "Musiques formelles: nouveaux principes formels de composition musicale," special issue of *La Revue musicale*, nos. 253-254 (1963).

In 2018, a year after the "Attention Is All You Need" paper we discussed in the last chapter, a group of researchers at an organization called OpenAI put out a new paper "Improving Language Understanding by Generative Pre-Training."[3] They combined a few ingredients:

- Unsupervised pretraining of a language model—essentially training a model to "guess the next token" in a sequence, as we did with our Shakespeare generator in chapter 15
- The Transformer architecture
- Textual data on various topics via thousands of self-published books

The authors showed that such a pretrained model could be fine-tuned to achieve state-of-the-art performance on a wide array of text classification tasks—from gauging the similarity of two sentences to answering a multiple-choice question. They called the pretrained model *GPT*, short for Generative Pretrained Transformer.

GPT didn't come with any modeling or training advancements. What was interesting about the results was that such a general training setup could beat out more involved techniques across a number of tasks. There was no complex text normalization, no need to customize the model architecture or training data per benchmark, just a lot of pretraining data and compute.

In the following years, OpenAI set about scaling this idea with a single-minded focus. The model architecture changed only slightly. Over four years, OpenAI released three versions of GPT, scaling up as follows:

- Released in 2018, GPT-1 had 117 million parameters and was trained on 1 billion tokens.
- Released in 2019, GPT-2 had 1.5 billion parameters and was trained on more than 10 billion tokens.
- Released in 2020, GPT-3 had 175 billion parameters and was trained on somewhere around half a trillion tokens.

The language modeling setup enabled each of these models to generate text, and the developers at OpenAI noticed that with each leap in scale, the quality of this generative output shot up substantially.

With GPT-1, the model's generative capabilities were mostly a by-product of its pretraining and not the primary focus. They evaluated the model by fine-tuning it with an extra dense layer for classification, as we did with RoBERTa in the last chapter.

With GPT-2, the authors noticed that you could prompt the model with a few examples of a task and generate quality output without any fine-tuning. For instance, you could prompt the model with the following to receive a French translation of the word cheese:

[3] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever, "Improving Language Understanding by Generative Pre-Training," (2018), https://mng.bz/GweD.

```
Translate English to French:

sea otter => loutre de mer
peppermint => menthe poivrée
plush giraffe => peluche girafe
cheese =>
```

This type of setup is called *few-shot learning*, where you attempt to teach a model a new problem with only a handful of supervised examples—too few for standard gradient descent.

With GPT-3, examples weren't always necessary. You could prompt the model with a simple text description of the problem and the input and often get quality results:

```
Translate English to French:

cheese =>
```

GPT-3 was still plagued by fundamental issues that have yet to be solved. LLMs "hallucinate" often—their output can veer from accurate to completely false with zero indication. They're extremely sensitive to prompt phrasing, with seemingly minor prompt rewording triggering large jumps up or down in performance. And they cannot adapt to problems that weren't extensively featured in their training data.

However, the generative output from GPT-3 was good enough that the model became the basis for ChatGPT—the first widespread, consumer-facing generative model. In the months and years since, ChatGPT has sparked a deluge of investment and interest in building LLMs and finding new use cases for them. In the next section, we will make a miniature GPT model of our own to better understand how such a model works, what it can do, and where it fails.

16.2 Training a mini-GPT

To begin pretraining our mini-GPT, we will need a lot of text data. GPT-1 used a dataset called BooksCorpus, which contained a number of free, self-published books added to the dataset without the explicit permission of the authors. The dataset has since been taken down by its publishers.

We will use a more recent pretraining dataset called the "Colossal Clean Crawled Corpus" (C4), released by Google in 2020. At 750 GB, it's far bigger than we could reasonably train on for a book example, so we will use less than 1% of the overall corpus.

Let's start by downloading and extracting our data.

Listing 16.1 Downloading a portion of the C4 dataset

```
import keras
import pathlib
```

```
extract_dir = keras.utils.get_file(
    fname="mini-c4",
    origin=(
        "https://hf.co/datasets/mattdangerw/mini-c4/resolve/main/mini-c4.zip"
    ),
    extract=True,
)
extract_dir = pathlib.Path(extract_dir) / "mini-c4"
```

Running the code in this chapter

Generative language models are big and take a lot of compute to run. While we've taken pains to make the code in this chapter accessible, this is still the most compute-intensive chapter in this book.

If you'd like, you can run everything on the free Colab GPU runtime (a T4 GPU as of this writing), but be prepared to wait! This mini-GPT example will take about 6 hours to train, and you will need to restart your Colab runtime in the middle of the notebook to free up GPU memory before loading a larger pretrained model. A larger GPU will make quicker work of these examples; we developed this example on an A100, which can run this chapter's code end-to-end in a little over an hour.

You can always read through the expensive `fit()` calls and edit down the number of training steps for quick experimentation. And if you are running this a few years down the line, there is a good chance these examples are mere child's play to modern hardware!

We have 50 shards of text data, each with about 75 MB of raw text. Each line contains a document in the crawl with newlines escaped. Let's look at a document in our first shard:

```
>>> with open(extract_dir / "shard0.txt", "r") as f:
>>>     print(f.readline().replace("\\n", "\n")[:100])
Beginners BBQ Class Taking Place in Missoula!
Do you want to get better at making delicious BBQ? You
```

We will need to preprocess a lot of data to run pretraining for an LLM, even a miniature one like the one we are training. Using a fast tokenization routine to preprocess our source documents into integer tokens can simplify our lives.

We will use SentencePiece, a library for subword tokenization of text data. The actual tokenization technique is the same as the byte-pair encoding tokenization we built ourselves in chapter 14, but the library is written in C++ for speed and adds a `detokenize()` function that will reverse integers to strings and join them together. We will use a premade vocabulary with 32,000 vocabulary terms stored in a particular format needed by the SentencePiece library.

As in the last chapter, we can use the KerasHub library to access some extra functions for working with large language models. KerasHub wraps the SentencePiece library as a Keras layer. Let's try it out.

Listing 16.2 Downloading a SentencePiece vocabulary and instantiating a tokenizer

```
import keras_hub
import numpy as np

vocabulary_file = keras.utils.get_file(
    origin="https://hf.co/mattdangerw/spiece/resolve/main/vocabulary.proto",
)
tokenizer = keras_hub.tokenizers.SentencePieceTokenizer(vocabulary_file)
```

We can use this tokenizer to map from text to int sequences bidirectionally:

```
>>> tokenizer.tokenize("The quick brown fox.")
array([  450,  4996, 17354,  1701, 29916, 29889], dtype=int32)
>>> tokenizer.detokenize([450, 4996, 17354, 1701, 29916, 29889])
"The quick brown fox."
```

Let's use this layer to tokenize our input text and then use `tf.data` to window our input into sequences of length 256.

When training GPT, the developers chose to keep things simple and make no attempt to keep document boundaries from occurring in the middle of a sample. Instead, they marked a document boundary with a special `<|endoftext|>` token. We will do the same here. Once again, we will use `tf.data` for the input data pipeline and train with any backend.

We will load each file shard individually and interleave the output data into a single dataset. This keeps our data loading fast, and we don't need to worry about text lining up across sample boundaries—each is independent. With interleaving, each processor on our CPU can read and tokenize a separate file simultaneously.

Listing 16.3 Preprocessing text input for Transformer pretraining

```
import tensorflow as tf

batch_size = 128
sequence_length = 256
suffix = np.array([tokenizer.token_to_id("<|endoftext|>")])

def read_file(filename):
    ds = tf.data.TextLineDataset(filename)
    ds = ds.map(lambda x: tf.strings.regex_replace(x, r"\\n", "\n"))
    ds = ds.map(tokenizer, num_parallel_calls=8)
    return ds.map(lambda x: tf.concat([x, suffix], -1))
```

Adds the <|endoftext|> token

Tokenizes data

Restores newlines

```
files = [str(file) for file in extract_dir.glob("*.txt")]
ds = tf.data.Dataset.from_tensor_slices(files)
ds = ds.interleave(read_file, cycle_length=32, num_parallel_calls=32)
ds = ds.rebatch(sequence_length + 1, drop_remainder=True)
ds = ds.map(lambda x: (x[:-1], x[1:]))
ds = ds.batch(batch_size).prefetch(8)
```

Splits labels, offset by one

Windows tokens into even samples of 256 tokens

Combines our file shards into a single dataset

As we first did in chapter 8, we will end our `tf.data` pipeline with a call to `prefetch()`. This will make sure we always have some batches loaded onto our GPU and ready for the model.

We have 29,373 batches. You could count this yourself if you would like—the line `ds.reduce(0, lambda c, _: c + 1)` will iterate over the entire dataset and increment a counter. But simply tokenizing a dataset of this size will take a few minutes on a decently fast CPU.

At 128 samples per batch and 256 tokens per sample, this is just under a billion tokens of data. Let's split off 500 batches as a quick validation set, and we are ready to start pretraining:

```
num_batches = 29373
num_val_batches = 500
num_train_batches = num_batches - num_val_batches
val_ds = ds.take(num_val_batches).repeat()
train_ds = ds.skip(num_val_batches).repeat()
```

16.2.1 Building the model

The original GPT model simplifies the sequence-to-sequence Transformer we saw in the last chapter. Rather than take in a source and target sequence with an encoder and decoder, as we did for our translation model, the GPT approach does away with the encoder entirely and only uses the decoder. This means that information can only travel from left to right in a sequence.

This was an interesting bet on the part of the GPT developers. A decoder-only model can still handle sequence-to-sequence problems like question-answering. However, rather than feeding in the question and answer as separate inputs, we must combine both into a single sequence to feed it to our model. So, unlike the original Transformer, the question tokens would not be handled any differently than answer tokens. All tokens are embedded into the same latent space with the same set of parameters.

The other consequence of this approach is that the information flow is no longer bidirectional, even for input sequences. Given an input, such as "Where is the capital of

France?", the learned representation of the word "Where" cannot attend to the words "capital" and "France" in the attention layer. This limits the expressivity of the model but has a massive advantage in terms of simplicity of pretraining. We don't need to curate datasets with pairs of inputs and outputs; everything can be a single sequence. We can train on any text we can find on the internet at a massive scale.

Let's copy the `TransformerDecoder` from chapter 15 but remove the cross-attention layer, which allowed the decoder to attend to the encoder sequence. We will also make one minor change, adding dropout after the attention and feedforward blocks. In chapter 15, we only used a single Transformer layer in our encoder and decoder, so we could get away with only using a single dropout layer at the end of our entire model. For our GPT model, we will stack quite a few layers, so adding dropout within each decoder layer is important to prevent overfitting.

Listing 16.4 A Transformer decoder block without cross-attention

```
from keras import layers

class TransformerDecoder(keras.Layer):
    def __init__(self, hidden_dim, intermediate_dim, num_heads):
        super().__init__()
        key_dim = hidden_dim // num_heads
        self.self_attention = layers.MultiHeadAttention(
            num_heads, key_dim, dropout=0.1
        )
        self.self_attention_layernorm = layers.LayerNormalization()
        self.feed_forward_1 = layers.Dense(intermediate_dim, activation="relu")
        self.feed_forward_2 = layers.Dense(hidden_dim)
        self.feed_forward_layernorm = layers.LayerNormalization()
        self.dropout = layers.Dropout(0.1)

    def call(self, inputs):
        residual = x = inputs
        x = self.self_attention(query=x, key=x, value=x, use_causal_mask=True)
        x = self.dropout(x)
        x = x + residual
        x = self.self_attention_layernorm(x)
        residual = x
        x = self.feed_forward_1(x)
        x = self.feed_forward_2(x)
        x = self.dropout(x)
        x = x + residual
        x = self.feed_forward_layernorm(x)
        return x
```

Self-attention layers

Feedforward layers

Self-attention computation

Feedforward computation

Next, we can copy the `PositionalEmbedding` layer from chapter 15. Recall that this layer gives us a simple way to learn an embedding for each position in a sequence and combine that with our token embeddings.

There's a neat trick we can employ here to save some GPU memory. The biggest weights in a Transformer model are the input token embeddings and output dense

prediction layer because they deal with our vocabulary space. The token embedding weight has shape `(vocab_size, hidden_dim)` to embed every possible token. Our output projection has shape `(hidden_dim, vocab_size)` to make a floating-point prediction for every possible token.

We can actually tie these two weight matrices together. To compute our model's final predictions, we will multiply our hidden states by the transpose of our token embedding matrix. You can very much think of our final projection as a "reverse embedding." It maps from hidden space to token space, whereas an embedding maps from token space to hidden space. It turns out that using the same weights for this input and output projection is a good idea.

Adding this to our `PositionalEmbedding` is simple; we will just add a `reverse` argument to the `call` method, which computes the projection by the transpose of the token embedding.

Listing 16.5 A positional embedding layer that can reverse a text embedding

```
from keras import ops

class PositionalEmbedding(keras.Layer):
    def __init__(self, sequence_length, input_dim, output_dim):
        super().__init__()
        self.token_embeddings = layers.Embedding(input_dim, output_dim)
        self.position_embeddings = layers.Embedding(sequence_length, output_dim)

    def call(self, inputs, reverse=False):
        if reverse:
            token_embeddings = self.token_embeddings.embeddings
            return ops.matmul(inputs, ops.transpose(token_embeddings))
        positions = ops.cumsum(ops.ones_like(inputs), axis=-1) - 1
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions
```

Let's build our model. We will stack eight decoder layers into a single "mini" GPT model.

We will also turn on a Keras setting called *mixed precision* to speed up training. This will allow Keras to run some of the model's computations much faster by sacrificing some numerical fidelity. For now, this will remain a little mysterious, but a full explanation is waiting in chapter 18.

Listing 16.6 Creating a mini-GPT functional model

```
keras.config.set_dtype_policy("mixed_float16")

vocab_size = tokenizer.vocabulary_size()
hidden_dim = 512
intermediate_dim = 2056
```

Enables mixed precision (see chapter 18)

```
num_heads = 8
num_layers = 8

inputs = keras.Input(shape=(None,), dtype="int32", name="inputs")
embedding = PositionalEmbedding(sequence_length, vocab_size, hidden_dim)
x = embedding(inputs)
x = layers.LayerNormalization()(x)
for i in range(num_layers):
    x = TransformerDecoder(hidden_dim, intermediate_dim, num_heads)(x)
outputs = embedding(x, reverse=True)
mini_gpt = keras.Model(inputs, outputs)
```

This model has 41 million parameters, which is large for models in this book but quite small compared to most LLMs today, which range from a couple of billion to trillions of parameters.

16.2.2 Pretraining the model

Training a large Transformer is famously finicky—the model is sensitive to initializations of parameters and choice of optimizer. When many Transformer layers are stacked, it is easy to suffer from exploding gradients, where parameters update too quickly and our loss function does not converge. A trick that works well is to linearly ease into a full learning rate over a number of warmup steps, so our initial updates to our model parameters are small. This is easy to implement in Keras with a `LearningRateSchedule`.

Listing 16.7 Defining a custom learning rate schedule

```
class WarmupSchedule(keras.optimizers.schedules.LearningRateSchedule):
    def __init__(self):
        self.rate = 2e-4                          # Peak learning rate
        self.warmup_steps = 1_000.0

    def __call__(self, step):
        step = ops.cast(step, dtype="float32")
        scale = ops.minimum(step / self.warmup_steps, 1.0)
        return self.rate * scale
```

We can plot our learning rate over time to make sure it is what we expect (figure 16.2):

```
import matplotlib.pyplot as plt

schedule = WarmupSchedule()
x = range(0, 5_000, 100)
y = [ops.convert_to_numpy(schedule(step)) for step in x]
plt.plot(x, y)
plt.xlabel("Train Step")
plt.ylabel("Learning Rate")
plt.show()
```

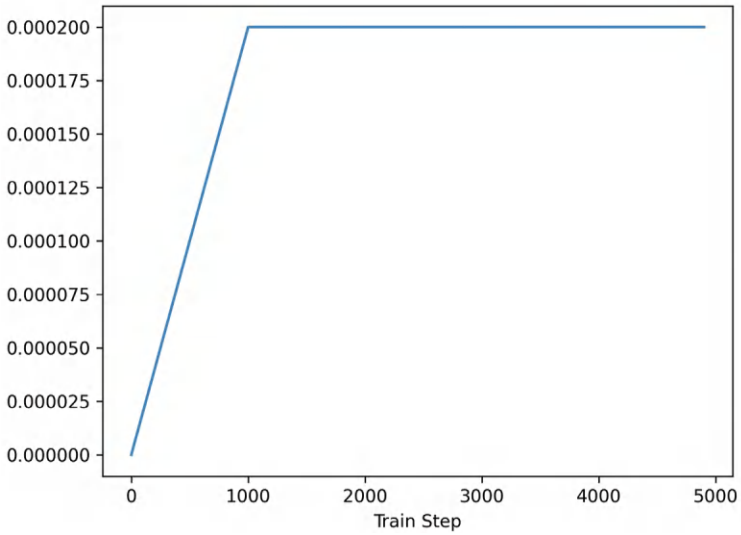


Figure 16.2 Warmup makes our updates to model parameters smaller at the beginning of training and can help with stability.

We will train our model using one pass over our 1 billion tokens, split across eight epochs so we can occasionally check our validation set loss and accuracy.

We are training a miniature version of GPT, using 3× fewer parameters than GPT-1 and 100× fewer overall training steps. But despite this being two orders of magnitude cheaper to train than the smallest GPT model, this call to `fit()` will be the most computationally expensive training run in the entire book. If you are running the code as you read, set things off and take a breather!

Listing 16.8 Training the mini-GPT model

```
num_epochs = 8
steps_per_epoch = num_train_batches // num_epochs
validation_steps = num_val_batches

mini_gpt.compile(
    optimizer=keras.optimizers.Adam(schedule),
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[„accuracy"],
)
mini_gpt.fit(
    train_ds,
    validation_data=val_ds,
    epochs=num_epochs,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
)
```

Set these to a lower value if you don't want to wait for training.

What is a logit?

As we compile our model, you will notice a new value for the loss `SparseCategoricalCrossentropy(from_logits=True)`. What is a logit?

The output projection at the end of our transformer model does not contain the usual softmax activation. You can think of this output as a bunch of "unnormalized log probabilities" for each token. If you exponentiate each output value and normalize all values to sum to 1 (this is all the `softmax` function does), you will get a probability value. A common term for an "unnormalized log probability" is a *logit*, and logits can be easier to work with when generating text, as we will see in the next section.

Keras gives you a choice of where to apply the `softmax` function. For classification problems, you can either use a `softmax` as the last activation of the model and output probabilities or move the `softmax` into the loss function and output logits. To do the latter, you should pass `SparseCategoricalCrossentropy(from_logits=True)` as a classification loss.

After training, our model can predict the next token in a sequence about 36% of the time on our validation set, though such a metric is just a crude heuristic.

Note that our model is undertrained. Our validation loss will continue to tick down after each epoch, which is unsurprising given that we used a hundred times fewer training steps than GPT-1. Training for longer would be a great idea, but we would need both time and money to pay for compute.

Let's play around with our mini-GPT model.

16.2.3 Generative decoding

To sample some output from our model, we can follow the approach we used to generate Shakespeare or Spanish translations in chapter 15. We feed a prompt of fixed tokens into the model. For each position in the input sequence, the model outputs a probability distribution over the entire vocabulary for the next token. By selecting the most likely next token at the last location, adding it to our sequence, and then repeating this process, we are able to generate a new sequence, one token at a time.

Listing 16.9 A simple generation function for the mini-GPT model

```
def generate(prompt, max_length=64):
    tokens = list(ops.convert_to_numpy(tokenizer(prompt)))
    prompt_length = len(tokens)
    for _ in range(max_length - prompt_length):
        prediction = mini_gpt(ops.convert_to_numpy([tokens]))
        prediction = ops.convert_to_numpy(prediction[0, -1])
        tokens.append(np.argmax(prediction).item())
    return tokenizer.detokenize(tokens)
```

Let's try this out with a text prompt:

```
>>> prompt = "A piece of advice"
>>> generate(prompt)
A piece of advice, and the best way to get a feel for yourself is to get a sense
of what you are doing.
If you are a business owner, you can get a sense of what you are doing. You can
get a sense of what you are doing, and you can get a sense of what
```

The first thing you will notice when running this is that it takes minutes to complete. That's a bit puzzling. We predicted about 200,000 tokens a second on our reference hardware during training. The generative loop may add time, but a minute delay is much too slow. What happened? The biggest reason for our slowness, at least on the Jax and TensorFlow backends, is that we are running an uncompiled computation.

Every time you run `fit()` or `predict()`, Keras compiles the computation that runs on each batch of data. All the `keras.ops` used will be lifted out of Python and heavily optimized by the backend framework. It's slow for one batch but massively faster for each subsequent call. However, when we directly call the model as we did previously, the backend framework will need to run the forward pass live and unoptimized at each step.

The easy solution here is to lean on `predict()`. With `predict()`, Keras will handle compilation for us, but there is one important gotcha to watch out for. When TensorFlow or Jax compiles a function, it will do so for a specific input shape. With a known shape, the backend can optimize for particular hardware, knowing exactly how many individual processor instructions make up a tensor operation. But in our generation function, we call our model with a sequence that changes shape after each prediction. This would trigger recompilation each time we call `predict()`.

Instead, we can avoid recompiling the `predict()` function if we pad our input so that our sequence is always the same length. Let's try that out.

Listing 16.10 A compiled generation function for the mini-GPT model

```
def compiled_generate(prompt, max_length=64):
    tokens = list(ops.convert_to_numpy(tokenizer(prompt)))
    prompt_length = len(tokens)
    tokens = tokens + [0] * (max_length - prompt_length)
    for i in range(prompt_length, max_length):
        prediction = mini_gpt.predict(np.array([tokens]), verbose=0)
        prediction = prediction[0, i - 1]
        tokens[i] = np.argmax(prediction).item()
    return tokenizer.detokenize(tokens)
```

Pads tokens to the full sequence length

Let's see how fast this new function is:

```
>>> import timeit
>>> tries = 10
>>> timeit.timeit(lambda: compiled_generate(prompt), number=tries) / tries
0.4866470648999893
```

Our generation call went from minutes to less than a second with compilation. That is quite an improvement.

Cached generation

There's still one more major inefficiency in the generation function we just built. Can you spot it?

Each time we call our model, we call it for an *entire sequence* and then throw away everything but the predictions for a single position. This is wasteful—our sequence only changes by a single token between generation steps. When we did generation with an RNN in chapter 15, we could keep around our RNN state and only compute outputs for a single token at each step. This state vector contained all the information the model needed about the past sequence. Transformers that use a causal attention, like GPT, actually have a similar notion of state.

If we think through the entire model we just built, you'll note that attention is the *only* place where the model passes information from position to position. The feedforward blocks of a transformer only modify the hidden representation of each token position in isolation.

Inside attention, we incorporate information about past tokens through the `key` and `value` vectors. For a given `query` at a position, we compute attention scores by dotting the `query` with all previous `key` vectors and combining all previous `value` vectors. These `key` and `value` vectors never change for past tokens in the sequence—past input is fixed, and the causal mask prevents the Transformer from "looking ahead" to future tokens. So if we cache all `key` and `value` vectors, at each layer of the Transformer, we have the equivalent of an RNN's state. We can use it to compute Transformer outputs for a single position at a time.

Implementing this is a bit clunky, as it involves saving and reusing intermediate arrays from every attention layer in the Transformer, but it's important. Your model inputs can go from being as long as the maximum length of your output to being a single token in length. If you are generating a sequence that is thousands of tokens long, caching can amount to a thousandfold speedup! Any efficient implementation of generative sampling will include `key` and `value` caching.

16.2.4 Sampling strategies

Another obvious problem with our generative output is that our model often repeats itself. On our particular training run, the model repeats the group of words "get a sense of what you are doing" over and over.

This isn't so much a bug as it's a direct consequence of our training objective. Our model is trying to predict the most likely next token in a sequence across about a billion words on many, many topics. If there's no obvious choice for where a sequence of text should head next, an effective strategy is to guess common words or repeated patterns of words. Unsurprisingly, our model learns to do this during training almost immediately. If you were to stop training our model very early on, it would likely generate the word `"the"` incessantly, as `"the"` is the most common word in the English language.

During our generative loop, we have always chosen the most likely predicted token in our model's output. But our output is not just a single predicted token; it is a probability distribution across all 32,000 tokens in our vocabulary.

Using the most likely output at each generation step is called *greedy search*. It's the most straightforward approach to using model predictions, but it is hardly the only one. If we instead add some randomness to the process, we can explore the probability distribution learned by the model more broadly. This can keep us from getting stuck in loops of high-probability token sequences.

Let's try this out. We can start by refactoring our generation function so that we can pass a function that maps from a model's predictions to a choice for the next token. We will call this our *sampling strategy*:

```
def compiled_generate(prompt, sample_fn, max_length=64):
    tokens = list(ops.convert_to_numpy(tokenizer(prompt)))
    prompt_length = len(tokens)
    tokens = tokens + [0] * (max_length - prompt_length)
    for i in range(prompt_length, max_length):
        prediction = mini_gpt.predict(np.array([tokens]), verbose=0)
        prediction = prediction[0, i - 1]
        next_token = ops.convert_to_numpy(sample_fn(prediction))
        tokens[i] = np.array(next_token).item()
    return tokenizer.detokenize(tokens)
```

Now we can write our greedy search as a simple function we pass to `compiled_generate()`:

```
def greedy_search(preds):
    return ops.argmax(preds)

compiled_generate(prompt, greedy_search)
```

The Transformer outputs define a categorical distribution where each token has a certain probability of being output at each time step. Instead of just choosing the most likely token, we could sample this distribution directly. `keras.random.categorical()` will pass our predictions through a `softmax` function to get a probability distribution and then randomly sample it. Let's try it out:

```
def random_sample(preds, temperature=1.0):
    preds = preds / temperature
    return keras.random.categorical(preds[None, :], num_samples=1)[0]
>>> compiled_generate(prompt, random_sample)
A piece of advice, just read my knees and stick with getables and a hello to me.
However, the bar napkin doesn't last as long. I happen to be waking up close and
pull it up as I wanted too and I still get it, really, shouldn't be a reaction
```

Our outputs are more diverse, and the model no longer gets stuck in loops. But our sampling is now exploring too much; the output jumps around wildly without any continuity.

You'll notice we added a parameter called `temperature`. We can use this to sharpen or widen our probability distribution so our sampling explores our distribution less or more.

If we pass a low temperature, we will make all logits larger before the `softmax` function, which makes our most likely output even more likely. If we pass a high temperature, our logits will be smaller before the softmax, and our probability distribution will be more spread out. Let's try this out a few times to see how this affects our sampling:

```
>>> from functools import partial
>>> compiled_generate(prompt, partial(random_sample, temperature=2.0))
A piece of advice tran writes using ignore unnecessary pivot - come without
introdu accounts indicugelâ per\u3000divuren sendSolisżsilen om transparent
Gill Guide pover integer song arrays coding\u3000LIST**…Allow index criteria
Draw Reference Ex artifactincluding lib tak Br basunker increases entirelytembre
AnyкaTextView cardinal spiritual heavenToen
>>> compiled_generate(prompt, partial(random_sample, temperature=0.8))
A piece of advice I wrote about the same thing today. I have been a writer for
two years now. I am writing this blog and I just wrote about it. I am writing
this blog and it was really interesting. I have been writing about the book and
I have read many things about my life.
The
>>> compiled_generate(prompt, partial(random_sample, temperature=0.2))
A piece of advice, and a lot of people are saying that they have to be careful
about the way they think about it.
I think it's a good idea to have a good understanding of the way you think about
it.
I think it's a good idea to have a good understanding of the
```

At a high temperature, our outputs no longer resemble English, settling on seemingly random tokens. At a low temperature, our model behavior starts to resemble greedy search, repeating certain patterns of text over and over.

Another popular technique for shaping our distribution is restricting our sampling to a set of the most likely tokens. This is called *top-K sampling*, where K is the number of candidates you should explore. Figure 16.3 shows how top-K sampling strikes a middle ground between greedy and random approaches.

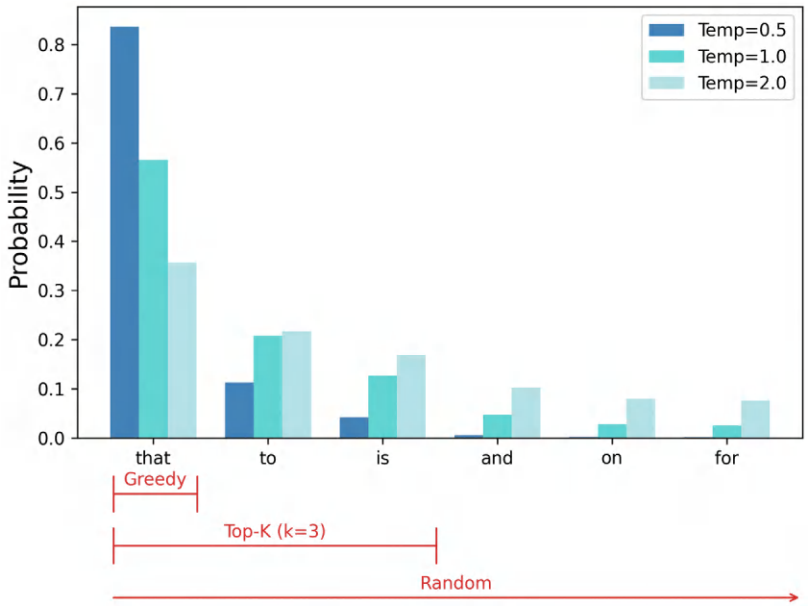


Figure 16.3 Greedy, top-K, and random sampling strategies shown on the same probability distribution

Let's try this out in code. We can use `keras.ops.top_k` to find the top K elements of an array:

```
def top_k(preds, k=5, temperature=1.0):
    preds = preds / temperature
    top_preds, top_indices = ops.top_k(preds, k=k, sorted=False)
    choice = keras.random.categorical(top_preds[None, :], num_samples=1)[0]
    return ops.take_along_axis(top_indices, choice, axis=-1)
```

We can try a few different variations of top-K to see how it affects sampling:

```
>>> compiled_generate(prompt, partial(top_k, k=5))
A piece of advice that I can't help it. I'm not going to be able to do anything
for a few months, but I'm trying to get a little better. It's a little too much.
I have a few other questions on this site, but I'm sure I
>>> compiled_generate(prompt, partial(top_k, k=20))
A piece of advice and guidance from the Audi Bank in 2015. With all the above,
it's not just a bad idea, but it's very good to see that is going to be a great
year for you in 2017.
That's really going to
```

Passing a top-K cutoff is different than temperature sampling. Passing a low temperature makes likely tokens more likely, but it does not rule any token out. Top-K sampling

zeros out the probability of anything outside the K candidates. You can combine the two, for example, sampling the top five candidates with a temperature of 0.5:

```
>>> compiled_generate(prompt, partial(top_k, k=5, temperature=0.5))
A piece of advice that you can use to get rid of the problem.
The first thing you need to do is to get the job done. It is important that you
have a plan that will help you get rid of it.
The first thing you need to do is to get rid of the problem yourself.
```

A sampling strategy is an important control when generating text, and there are many more approaches. For example, beam search is a technique that heuristically explores multiple chains of predicted tokens by keeping a fixed number of "beams" (different chains of predicted tokens) to explore at each timestep.

With top-K sampling, our model generates something closer to plausible English text, but there is little apparent utility to such output. This fits with the results of GPT-1. For the initial GPT paper, the generated output was more of a curiosity, and state-of-the-art results were only achieved by fine-tuning classification models. Our mini-GPT is far less trained than GPT-1.

To reach the scale of generative LLMs today, we'd need to increase our parameter count by at least 100× and our train step count by at least 1,000×. If we did, we would see the same leaps in quality observed by OpenAI with GPT. And we could do it! The training recipe we used previously is the exact blueprint used by everyone training LLMs today. The only missing pieces are a very large compute budget and some tricks for training across multiple machines that we will cover in chapter 18.

For a more practical approach, we will transition to using a pretrained model. This will allow us to explore the behavior of an LLM at today's scale.

16.3 Using a pretrained LLM

Now that we've trained a mini-language model from scratch, let's try using a billion-parameter pretrained model and see what it can do. Given how prohibitively expensive pretraining a Transformer can be, most of the industry has centered around using pretrained models developed by a relatively short list of companies. This is not purely a cost concern but also an environmental one—generative model training is now making up a large percentage of the total data center power consumption of large tech companies.

Meta published some environmental data on Llama 2, an LLM it published in 2023. It's a good bit smaller than GPT-3, but it needed an estimated 1.3 million kilowatt hours of electricity to train—the daily power usage of about 45,000 American households. If every organization using an LLM ran pretraining themselves, the scale of energy use would be a noticeable percentage of global energy consumption.

Let's play around with a pretrained generative model from Google called Gemma. We will use the third version of the Gemma model, which was released to the public in 2025. To keep the examples in this book accessible, we will use the smallest variation of

Gemma available, which clocks in at almost exactly 1 billion parameters. This "small" model was trained on roughly 2 trillion tokens of pretraining data—2,000 times more tokens than the mini-GPT we just trained!

16.3.1 Text generation with the Gemma model

To load this pretrained model, we can use KerasHub, as we have done in previous chapters.

Accessing Gemma weights

If you are running the code for this chapter yourself, you will need to accept a Terms of Use for the Gemma models before you can download the weights. The model weights are stored on Kaggle, and we can use the `kagglehub` API to log in as we did in chapter 8. Before we do, you will need to do two things:

1. Go to https://www.kaggle.com/models/keras/gemma3, and accept the Gemma Terms of Use at the top of the page.
2. Go to https://www.kaggle.com/settings, and generate a Kaggle API Key (if you did not already do this in chapter 8).

With that, we can use our API key to authenticate with Kaggle from our notebook:

```
import kagglehub

kagglehub.login()
```

As LLMs become increasingly powerful, terms of service like this are becoming more and more common. The Gemma terms of use prohibit using the model for things like generating spam or hate speech.

Listing 16.11 Instantiating a pretrained LLM with KerasHub

```
gemma_lm = keras_hub.models.CausalLM.from_preset(
    "gemma3_1b",
    dtype="float32",
)
```

`CausalLM` is another example of the high-level task API, much like the `ImageClassifier` and `ImageSegmenter` tasks we used earlier in the book. The `CausalLM` task will combine a tokenizer and correctly initialized architecture into a single Keras model. KerasHub will load the Gemma weights into a correctly initialized architecture and load a matching tokenizer for the pretrained weights.

Let's take a look at the Gemma model summary:

```
>>> gemma_lm.summary()
Preprocessor: "gemma3_causal_lm_preprocessor"
```

Layer (type)	Config
gemma3_tokenizer (Gemma3Tokenizer)	Vocab size: 262,144

Model: "gemma3_causal_lm"

Layer (type)	Output Shape	Param #	Connected to
padding_mask (InputLayer)	(None, None)	0	-
token_ids (InputLayer)	(None, None)	0	-
gemma3_backbone (Gemma3Backbone)	(None, None, 1152)	999,885,952	padding_mask[0][0… token_ids[0][0]
token_embedding (ReversibleEmbedding)	(None, None, 262144)	301,989,888	gemma3_backbone[0…

Total params: 999,885,952 (3.72 GB)
Trainable params: 999,885,952 (3.72 GB)
Non-trainable params: 0 (0.00 B)

Rather than implementing a generation routine ourselves, we can simplify our lives by using the `generate()` function that comes as part of the `CausalLM` class. This `generate()` function can be compiled with different sampling strategies, as we explored in section 16.2:

```
>>> gemma_lm.compile(sampler="greedy")
>>> gemma_lm.generate("A piece of advice", max_length=40)
A piece of advice from a former student of mine:

<blockquote>"I'm not sure if you've heard of it, but I've been told that the
best way to learn
>>> gemma_lm.generate("How can I make brownies?", max_length=40)
How can I make brownies?

[User 0001]

I'm trying to make brownies for my son's birthday party. I've never made
brownies before.
```

We can notice a few things right off the bat. First, the output is much more coherent than our mini-GPT model. It would be hard to distinguish this text from much of the training data in the C4 dataset. Second, the output is still not that useful. The model will generate vaguely plausible text, but what you could do with it is unclear.

As we saw with the mini-GPT example, this is not so much a bug as a consequence of our pretraining objective. The Gemma model was trained with the same "guess the next word" objective we used for mini-GPT, which means it's effectively a fancy autocomplete for the internet. It will just keep rattling off the most probable word in its single sequence as if your prompt was a snippet of text found in a random document on the web.

One way to change our output is to prompt the model with a longer input that makes it obvious which type of output we are looking for. For example, if we prompt the Gemma model with the beginning two sentences of a brownie recipe, we get more helpful output:

```
>>> gemma_lm.generate(
>>>     "The following brownie recipe is easy to make in just a few "
>>>     "steps.\n\nYou can start by",
>>>     max_length=40,
>>> )
The following brownie recipe is easy to make in just a few steps.

You can start by melting the butter and sugar in a saucepan over medium heat.

Then add the eggs and vanilla extract
```

Though it's tempting when working with a model that can "talk" to imagine it interpreting our prompt in some sort of human, conversational way, nothing of the sort is going on here. We have just constructed a prompt for which an actual brownie recipe is a more likely continuation than mimicking someone posting on a forum asking for baking help.

You can go much further in constructing prompts. You might prompt a model with some natural language instructions of the role it is supposed to fill, for example, `"You are a large language model that gives short, helpful answers to people's questions."` Or you might feed the model a prompt containing a long list of harmful topics that should not be included in any generated responses.

If this all sounds a bit hand-wavy and hard to control, that's a good assessment. Attempting to visit different parts of a model's distribution through prompting is often useful, but predicting how a model will respond to a given prompt is very difficult.

Another well-documented problem faced by LLMs is hallucinations. A model will always say something—there is always a most-likely next token to a given sequence. Finding locations in our LLM distribution that have no grounding in actual fact is easy:

```
>>> gemma_lm.generate(
>>>     "Tell me about the 542nd president of the United States.",
>>>     max_length=40,
>>> )
Tell me about the 542nd president of the United States.

The 542nd president of the United States was James A. Garfield.
```

Of course, this is utter nonsense, but the model could not find a more likely way to complete this prompt.

Hallucinations and uncontrollable output are fundamental problems with language models. If there is a silver bullet, we have yet to find it. However, one approach that helps immensely is to further fine-tune a model with examples of the specific types of generative outputs you would like.

In the specific case of wanting to build a chatbot that can follow instructions, this type of training is called *instruction fine-tuning*. Let's try some instruction fine-tuning with Gemma to make it a lot more useful as a conversation partner.

16.3.2 Instruction fine-tuning

Instruction fine-tuning involves feeding the model input/output pairs—a user instruction followed by a model response. We combine these into a single sequence that becomes new training data for the model. To make it clear during training when an instruction or response ends, we can add special markers like `"[instruction]"` and `"[response]"` directly to the combined sequence. The precise markup will not matter much as long as it is consistent.

We can use the combined sequence as regular training data, with the same "guess the next word" loss we used to pretrain an LLM. By doing further training with examples containing desired responses, we are essentially bending the model's output in the direction we want. We won't be learning a latent space for language here; that's already been done over trillions of tokens of pretraining. We are simply nudging the learned representation a bit to control the tone and content of the output.

To begin, we will need a dataset of instruction-response pairs. Training chatbots is a hot topic, so there are many datasets made specifically for this purpose. We will use a dataset made public by the company Databricks. Employees contributed to a dataset of 15,000 instructions and handwritten responses. Let's download it and join the data into a single sequence.

Listing 16.12 Loading an instruction fine-tuning dataset

```
import json

PROMPT_TEMPLATE = """[instruction]\n{}[end]\n[response]\n"""
RESPONSE_TEMPLATE = """{}[end]"""

dataset_path = keras.utils.get_file(
    origin=(
        "https://hf.co/datasets/databricks/databricks-dolly-15k/"
        "resolve/main/databricks-dolly-15k.jsonl"
    ),
)
data = {"prompts": [], "responses": []}
with open(dataset_path) as file:
    for line in file:
```

```
        features = json.loads(line)
        if features["context"]:
            continue
        data["prompts"].append(PROMPT_TEMPLATE.format(features["instruction"]))
        data["responses"].append(RESPONSE_TEMPLATE.format(features["response"]))
```

Note that some examples have additional context—textual information related to the instruction. To keep things simple for now, we will discard those examples.

Let's take a look at a single element in our dataset:

```
>>> data["prompts"][0]
[instruction]
Which is a species of fish? Tope or Rope[end]
[response]

>>> data["responses"][0]
Tope[end]
```

Our prompt template gives our examples a predictable structure. Although Gemma is not a sequence-to-sequence model like our English-to-Spanish translator, we can still use it in a sequence-to-sequence setting by training on prompts like this and only generating the output after the `"[response]"` marker.

Let's make a `tf.data.Dataset` and split some validation data:

```
ds = tf.data.Dataset.from_tensor_slices(data).shuffle(2000).batch(2)
val_ds = ds.take(100)
train_ds = ds.skip(100)
```

The `CausalLM` we loaded from the KerasHub library is a high-level object for end-to-end causal language modeling. It wraps two objects: a `preprocessor` layer, which preprocesses text input, and a `backbone` model, which contains the numerics of the model forward pass.

Preprocessing is included by default in high-level Keras functions like `fit()` and `predict()`. But let's run our preprocessing on a single batch so we can better see what it is doing:

```
>>> preprocessor = gemma_lm.preprocessor
>>> preprocessor.sequence_length = 512
>>> batch = next(iter(train_ds))
>>> x, y, sample_weight = preprocessor(batch)
>>> x["token_ids"].shape
(2, 512)
>>> x["padding_mask"].shape
```

```
(2, 512)
>>> y.shape
(2, 512)
>>> sample_weight.shape
(2, 512)
```

The preprocessor layer will pad all inputs to a fixed length and compute a padding mask to track which token ID inputs are just padded zeros. The `sample_weight` tensor allows us to only compute a loss value for our response tokens. We don't really care about the loss for the user prompt; it is fixed, and we definitely don't want to compute the loss for the zero padding we just added.

If we print a snippet of our token IDs and labels, we can see that this is the regular language model setup, where each label is the next token value:

```
>>> x["token_ids"][0, :5], y[0, :5]
(Array([     2,  77074,  22768, 236842,    107], dtype=int32),
 Array([ 77074,  22768, 236842,    107,  24249], dtype=int32))
```

16.3.3 Low-Rank Adaptation (LoRA)

If we ran `fit()` right now on a Colab GPU with 16 GB of device memory, we would quickly trigger an out of memory error. But we've already loaded the model and run generation, so why would we run out of memory now?

Our 1-billion-parameter model takes up about 3.7 GB of memory. You can see it in our previous model summary. The `Adam` optimizer we have been using will need to track three extra floating-point numbers for *each* parameter—the actual gradients, a velocity value, and a momentum value. All told, it comes out to 15 GB just for the weights and optimizer state. We also need a few gigabytes of memory to keep track of intermediate values in the forward pass of the model, but we have none left to spare. Running `fit()` would crash on the first train step. This is a common problem when training LLMs. Because these models have large parameter counts, the throughput of your GPUs and CPUs is a secondary concern to fitting the model on accelerator memory.

We've seen earlier in this book how we can freeze certain parts of a model during fine-tuning. What we did not mention is that this will save a lot of memory! We do not need to track any optimizer variables for frozen parameters—they will never update. This allows us to save a lot of space on an accelerator.

Researchers have experimented extensively with freezing different parameters in a Transformer model during fine-tuning, and it turns out, perhaps intuitively, that the most important weights to leave unfrozen are in the attention mechanism. But our attention layers still have hundreds of millions of parameters. Can we do even better?

In 2021, researchers at Microsoft proposed a technique called LoRA, short for *Low-Rank Adaptation of Large Language Models*, specifically to solve this memory issue.[4] To explain it, let's imagine a simple linear projection layer:

```
class Linear(keras.Layer):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.kernel = self.add_weight(shape=(input_dim, output_dim))

    def call(self, inputs):
        return ops.matmul(inputs, self.kernel)
```

The LoRA paper proposes freezing the `kernel` matrix and adding a new "low rank" decomposition of the kernel projection. This decomposition has two new projection matrices, `alpha` and `beta`, which project to and from an inner `rank`. Let's take a look:

```
class LoraLinear(keras.Layer):
    def __init__(self, input_dim, output_dim, rank):
        super().__init__()
        self.kernel = self.add_weight(
            shape=(input_dim, output_dim), trainable=False
        )
        self.alpha = self.add_weight(shape=(input_dim, rank))
        self.beta = self.add_weight(shape=(rank, output_dim))

    def call(self, inputs):
        frozen = ops.matmul(inputs, self.kernel)
        update = ops.matmul(ops.matmul(inputs, self.alpha), self.beta)
        return frozen + update
```

If our `kernel` is shape 2048 × 2048, that is 4,194,304 frozen parameters. But if we keep the `rank` low, say, 8, we will have only 32,768 parameters for the low-rank decomposition. This update will not have the same expressive power as the original kernel; at the narrow middle point, the entire update must be represented as eight floats. But during LLM fine-tuning, you no longer need the expressive power you needed during pretraining (figure 16.4).

The LoRA authors suggest freezing the entire Transformer and adding LoRA weights to only the query and key projections in the attention layer. Let's try that out. KerasHub models have a built-in method for LoRA training.

Listing 16.13 Enabling LoRA training for a KerasHub model

```
gemma_lm.backbone.enable_lora(rank=8)
```

[4] J. Edward Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," arXiv (2021), https://arxiv.org/abs/2106.09685.

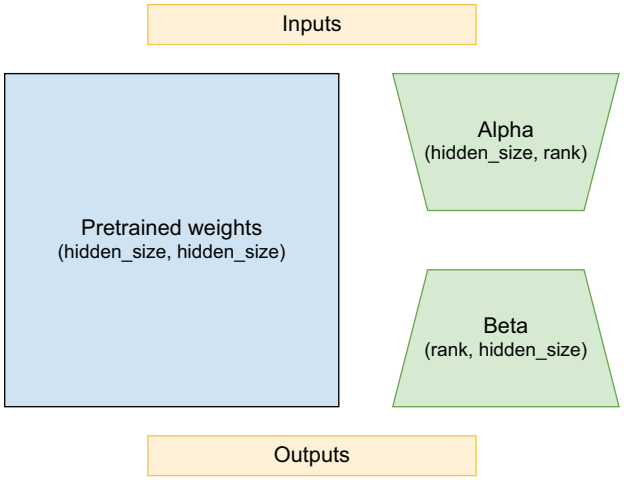


Figure 16.4 The low-rank kernel decomposition contains far fewer parameters than the kernel itself.

Customizing LoRA training

The `enable_lora()` method is also available on individual `Dense` layers. We could equivalently write the previous call a little more verbosely by iterating through the layers of the Transformer:

```
gemma_lm.backbone.trainable = False
for i in range(gemma_lm.backbone.num_layers):
    layer = gemma_lm.backbone.get_layer(f"decoder_block_{i}")
    layer.attention.key_dense.trainable = True
    layer.attention.key_dense.enable_lora(rank=8)
    layer.attention.query_dense.trainable = True
    layer.attention.query_dense.enable_lora(rank=8)
```

Sets all layers to not be trainable

Makes key and query projections trainable and enables LoRA

With this approach, we could add more trainable parameters earlier or later in the model and also add LoRA to the value projection.

Let's look at our model summary again:

```
>>> gemma_lm.summary()
Preprocessor: "gemma3_causal_lm_preprocessor"
```

Layer (type)	Config
gemma3_tokenizer (Gemma3Tokenizer)	Vocab size: 262,144

```
Model: "gemma3_causal_lm"
```

Layer (type)	Output Shape	Param #	Connected to
padding_mask (InputLayer)	(None, None)	0	-
token_ids (InputLayer)	(None, None)	0	-
gemma3_backbone (Gemma3Backbone)	(None, None, 1152)	1,001,190,…	padding_mask[0][0… token_ids[0][0]
token_embedding (ReversibleEmbedding)	(None, None, 262144)	301,989,888	gemma3_backbone[0…

```
Total params: 1,001,190,528 (3.73 GB)
Trainable params: 1,304,576 (4.98 MB)
Non-trainable params: 999,885,952 (3.72 GB)
```

Although our model parameters still occupy 3.7 GB of space, our trainable parameters now use only 5 MB of data—a thousandfold decrease! This can take our optimizer state from many gigabytes to just megabytes on the GPU (figure 16.5).

Figure 16.5 LoRA greatly reduces the memory we need for gradients and optimizer states.

With this optimization in place, we are at last ready to instruction-tune our Gemma model. Let's give it a go.

Listing 16.14 Fine-tuning a pretrained LLM

```
gemma_lm.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=keras.optimizers.Adam(5e-5),
    weighted_metrics=[keras.metrics.SparseCategoricalAccuracy()],
)
gemma_lm.fit(train_ds, validation_data=val_ds, epochs=1)
```

After training, we get to 55% accuracy when guessing the next word in our model's response. That's a huge jump from the 35% accuracy of our mini-GPT model. This shows the power of a larger model and more extensive pretraining.

Did our fine-tuning make our model better at following directions? Let's give it a try:

```
>>> gemma_lm.generate(
...     "[instruction]\nHow can I make brownies?[end]\n"
...     "[response]\n",
...     max_length=512,
... )
[instruction]
How can I make brownies?[end]
[response]
You can make brownies by mixing together 1 cup of flour, 1 cup of sugar, 1/2
cup of butter, 1/2 cup of milk, 1/2 cup of chocolate chips, and 1/2 cup of
chocolate chips. Then, you can bake it in a 9x13 pan for 30 minutes at 350
degrees Fahrenheit. You can also add a little bit of vanilla extract to the
batter to make it taste better.[end]
>>> gemma_lm.generate(
...     "[instruction]\nWhat is a proper noun?[end]\n"
...     "[response]\n",
...     max_length=512,
... )
[instruction]
What is a proper noun?[end]
[response]
A proper noun is a word that refers to a specific person, place, or thing.
Proper nouns are usually capitalized and are used to identify specific
individuals, places, or things. Proper nouns are often used in formal writing
and are often used in titles, such as "The White House" or "The Eiffel Tower."
Proper nouns are also used in titles of books, movies, and other works of
literature.[end]
```

Much better. Our model will now respond to questions, instead of trying to simply carry on the thought of the prompt text.

Have we solved the hallucination problem?

```
>>> gemma_lm.generate(
...     "[instruction]\nWho is the 542nd president of the United States?[end]\n"
...     "[response]\n",
...     max_length=512,
... )
[instruction]
Who is the 542nd president of the United States?[end]
[response]
The 542nd president of the United States was James A. Garfield.[end]
```

Not at all. However, we could still use instruction tuning to make some inroads here. A common technique is to train the model on a lot of instruction/response pairs where the desired response is `"I don't know"` or `"As a language model, I cannot help you with that"`. This can train the model to avoid attempting to answer specific topics where it would often give poor-quality results.

16.4 Going further with LLMs

We have now trained a GPT model from scratch and fine-tuned a language model into our very own chatbot. However, we are just scratching the surface of LLM research today. In this section, we will cover a non-exhaustive list of extensions and improvements to the basic "autocomplete the internet" language modeling setup.

16.4.1 Reinforcement Learning with Human Feedback (RLHF)

The type of instruction fine-tuning we just did is often called *supervised fine-tuning*. It is *supervised* because we are curating, by hand, a list of example prompts and responses we want from the model.

Any need to manually write text examples will almost always become a bottleneck—such data is slow and expensive to come by. Moreover, this approach will be limited by the human performance ceiling on the instruction-following task. If we want to do better than human performance in a chatbot-like experience, we cannot rely on manually written output to supervise LLM training.

The real problem we are trying to optimize is our preference for certain responses over others. With a large enough sample of people, this preference problem is perfectly defined, but figuring out how to translate from "our preferences" to a loss function we could use to compute gradients is quite tricky. This is what *Reinforcement Learning with Human Feedback*, or *RLHF*, attempts to solve.

The first step in RLHF fine-tuning is exactly what we did in the last section—supervised fine-tuning with handwritten prompts and responses. This gets us to a good baseline performance; we now need to improve on this baseline. To this end, we will build a *reward model* that can act as a proxy for human preference. We can gather a large number of prompts and responses to these prompts. Some of these responses can be handwritten; the model can write others. Responses could even be written by other chatbot LLMs. We then need to get human evaluators to rank these responses by preference. Given a prompt and several potential responses, an evaluator's task is to rank them from most helpful to least helpful. Such data collection is expensive and slow, but still faster than writing all the desired responses by hand.

We can use this ranked preference dataset to build the reward model, which takes in a prompt-response pair and outputs a single floating-point value. The higher the value, the better the response. This reward model is usually another, smaller Transformer. Instead of predicting the next token, it reads a whole sequence and outputs a single float—a rating for a given response.

We can then use this reward model to tune our model further, using a reinforcement learning setup. We won't get too deep into the details of reinforcement learning in this book, but don't be too intimidated by the term—it refers to any training setup where a deep learning model learns by making predictions (called *actions*) and getting feedback on that output (called *rewards*). In short, a model's own predictions become its training data.

In our case, the action is simply generating a response to an input prompt, like we have been doing above with the `generate()` function. The reward is simply applying a separate regression model to that string output. Here's a simple example in pseudocode.

Listing 16.15 Pseudocode for the simplest possible RLHF algorithm

```
for prompts in dataset:
    responses = model.generate(prompts)
    rewards = reward_model.predict(responses)
    good_responses = []
    for response, score in zip(responses, rewards):
        if score > cutoff:
            good_responses.append(response)
    model.fit(good_responses)
```

Takes an action

Receives a reward

Updates the model parameters. We do not update the reward model.

In this simple example, we filter our generated responses with a reward cutoff, and simply treat the "good" output as new training data for more supervised fine-tuning like we just did in the last section. In practice, you will usually not discard your bad responses but rather use specialized gradient update algorithms to steer your model's parameters using all responses and rewards. After all, a bad response gives a good signal on what not to do. OpenAI originally described RLHF in a 2022 paper[5] and used this training setup to go from GPT-3's initial pretrained parameters to the first version of ChatGPT.

An advantage of this setup is that it can be iterative. You can take this newly trained model, generate new and improved responses to prompts, rank these responses by human preference, and train a new and improved reward model.

4.1.1. Using a chatbot trained with RLHF

We can make this more concrete by trying a model trained with this form of iterative preference tuning. Since building chatbots is the "killer app" for large Transformer models, it is common practice for companies that release pretrained models like Gemma to release specialized "instruction-tuned" versions, built just for chat. Let's try loading one now. This will be a 4-billion-parameter model, quadruple the size of the model we just loaded and the largest model we will use in this book.

Listing 16.16 Loading an instruction-tuned Gemma variant

```
gemma_lm = keras_hub.models.CausalLM.from_preset(
    "gemma3_instruct_4b",
    dtype="bfloat16",
)
```

[5] Ouyang et al., "Training Language Models to Follow Instructions with Human Feedback," Proceedings of the 36th International Conference on Neural Information Processing Systems (2022), https://arxiv.org/abs/2203.02155.

Choosing a dtype for large models

You might have noticed we passed `dtype="float32"` when creating our Gemma model the first time and `dtype="bfloat16"` now. What is going on?

For models like Gemma with more than a billion parameters, the number of bytes used for each floating-point number is an important consideration. When you are training a model, it is often a good idea to use 32 bits (4 bytes) per parameter. 32-bit floats can represent very small values, which can help keep training gradients stable. Here we aren't doing any training, so we pass `bfloat16`, which uses only 2 bytes per parameter. We don't need to worry about gradient stability, and we will save many gigabytes of memory by using a lower precision.

There's a detailed discussion of floating-point precision coming up in chapter 18.

Like the earlier Gemma model we fine-tuned ourselves, this instruction-tuned checkpoint comes with a specific template for formatting its input. Again, the exact text does not matter, what is important is that our prompt template matches what was used to tune the model:

```
PROMPT_TEMPLATE = """<start_of_turn>user
{}<end_of_turn>
<start_of_turn>model
"""
```

Let's try asking it a question:

```
>>> prompt = "Why can't you assign values in Jax tensors? Be brief!"
>>> gemma_lm.generate(PROMPT_TEMPLATE.format(prompt), max_length=512)
<start_of_turn>user
Why can't you assign values in Jax tensors? Be brief!<end_of_turn>
<start_of_turn>model
Jax tensors are designed for efficient automatic differentiation. Directly
assigning values disrupts this process, making it difficult to track gradients
correctly. Instead, Jax uses operations to modify tensor values, preserving the
differentiation pipeline.<end_of_turn>
```

This 4-billion-parameter model was first pretrained on 14 trillion tokens of text and then extensively fine-tuned to make it more helpful when answering questions. Some of this tuning was done with supervised fine-tuning like we did in the previous section, some with RLHF as we covered in this section, and some with still other techniques—like using an even larger model as a "teacher" to guide training. The increase in ability to do question-answering is easily noticeable.

Let's try this model on the prompt that has been giving us trouble with hallucinations:

```
>>> prompt = "Who is the 542nd president of the United States?"
>>> gemma_lm.generate(PROMPT_TEMPLATE.format(prompt), max_length=512)
<start_of_turn>user
Who is the 542nd president of the United States?<end_of_turn>
<start_of_turn>model
This is a trick question! As of today, November 2, 2023, the United States has
only had 46 presidents. There hasn't been a 542nd president yet. ☺

You're playing with a very large number!<end_of_turn>
```

This more capable model refuses to take the bait. This is not the result of a new modeling technique, but rather the result of extensive training on trick questions like this one with responses like the one we just received. In fact, you can see clearly here why removing hallucinations can be a bit like playing whack-a-mole—even though it refused to hallucinate a US president, the model now manages to make up today's date.

16.4.2 Multimodal LLMs

One obvious chatbot extension is the ability to handle new modalities of input. An assistant that can respond to audio input and process images would be far more useful than one that can only operate on text.

Extending a Transformer to different modalities can be done in a conceptually simple way. The Transformer is not a text-specific model; it's a highly effective model for *learning patterns in sequence data*. If we can figure out how to coerce other data types into a sequence representation, we can feed this sequence into a Transformer and train with it.

In fact, the Gemma model we just loaded does just that. The model comes with a separate 420-million-parameter image encoder that cuts an input image into 256 patches and encodes each patch as a vector with the same dimensionality as Gemma's hidden transformer dimension. Each image will be embedded as a `(256, 2560)` sequence. Because 2560 is the hidden dimensionality of the Gemma Transformer model, this image representation can simply be spliced into our text sequence after the token embedding layer. You can think of it like 256 special tokens representing the image, where each `(1, 2560)` vector is sometimes called a "soft token" (figure 16.6). Unlike our normal "hard tokens," where each token ID can only take on a fixed number of possible vectors in our token embedding matrix, these image soft tokens can take on any vector value output by the vision encoder.

Let's load an image to see how this works in a little more detail (figure 16.7):

```
import matplotlib.pyplot as plt

image_url = (
    "https://github.com/mattdangerw/keras-nlp-scripts/"
    "blob/main/learned-python.png?raw=true"
)
image_path = keras.utils.get_file(origin=image_url)
```

```
image = np.array(keras.utils.load_img(image_path))
plt.axis("off")
plt.imshow(image)
plt.show()
```

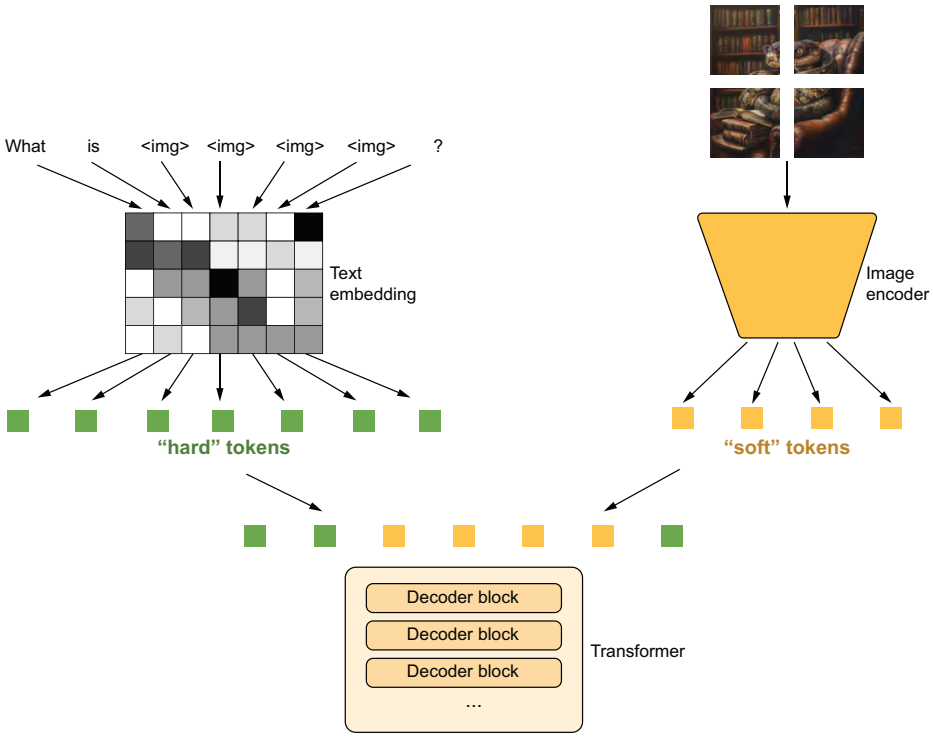


Figure 16.6 Handling image input by splicing text tokens and soft image tokens together

Figure 16.7 A test image for the Gemma model

We can use Gemma to ask some questions about this image:

```
>>> gemma_lm.preprocessor.max_images_per_prompt = 1
>>> gemma_lm.preprocessor.sequence_length = 512
>>> prompt = "What is going on in this image? Be concise!<start_of_image>"
>>> gemma_lm.generate({
...     "prompts": PROMPT_TEMPLATE.format(prompt),
...     "images": [image],
... })
<start_of_turn>user
What is going on in this image? Be concise!

<start_of_image>

<end_of_turn>
<start_of_turn>model
A snake wearing glasses is sitting in a leather armchair, surrounded by a large
bookshelf, and reading a book. It's a whimsical, slightly surreal image.
<end_of_turn>
>>> prompt = "What is the snake wearing?<start_of_image>"
>>> gemma_lm.generate({
...     "prompts": PROMPT_TEMPLATE.format(prompt),
...     "images": [image],
... })
<start_of_turn>user
What is the snake wearing?

<start_of_image>

<end_of_turn>
<start_of_turn>model
The snake is wearing a pair of glasses! They are red-framed and perched on its
head.<end_of_turn>
```

Limits the maximum input size of the model

Each of our input prompts contains the special token `<start_of_image>`. This is turned into 256 placeholder values in our input sequence, which, in turn, is replaced with the soft tokens representing our image.

Training for a multimodal model like this is quite similar to regular LLM pretraining and fine-tuning. Usually, you would want to first pretrain your image encoder separately, like we first did in Chapter 8 of this book. Then you can simply do the same basic "guess the next word" pretraining and also feed in mixed image and text content combined into a single sequence. Our transformer would not be trained to output image soft tokens; we would simply zero the loss at these image token locations.

It might seem almost magical that we can simply add image data to an LLM, but when we consider the power of the sequence model we're working with, it's really quite an expected result. We've taken a Transformer, recast our image input as sequence data, and done a lot of extra training. The model can preserve the original language model's ability to ingest and produce text while learning to also embed images in the Transformer's latent space.

FOUNDATION MODELS

As LLMs venture into different modalities, the "large language model" moniker can become a bit misleading. They *do* model language, but also images, audio, maybe even structured data. In the next chapter, we will see a distinct architecture, called *diffusion models*, that works quite differently in terms of underlying structure but has a similar feel—they too are trained on massive amounts of data at "internet scale" with a self-supervised loss.

An umbrella term for models like this is *foundation models*. More specifically, a foundation model is any model that is trained on broad data (generally using self-supervision at scale) that can be fine-tuned to a wide range of downstream tasks.

In general, you can think of a foundation model as learning to *reconstruct* data pulled from large swaths of the internet, given a partial representation of it. While LLMs are the first and best-known of these models, there are many others. The hallmarks of a foundation model are the self-supervised learning objective (a reconstruction loss) and the fact that these models are not specialized to a single task and can be used for a number of downstream purposes.

This is an important and striking shift that has happened quite recently in the long history of machine learning. Rather than training a model from scratch on your individual dataset, you will often be better off using a foundation model to get a rich representation of your input (whether it's images, text, or something else) and then specialize that model for your final downstream task. Of course, this comes with the downside of needing to run large models with billions of parameters, so it's hardly a fit for all real-world applications of machine learning.

16.4.3 Retrieval Augmented Generation (RAG)

Sticking extra information in the prompt is not just helpful in handling image data; it can be a general way to extend the capabilities of an LLM. One notable example is when using an LLM for search. If we naively compare an LLM to a search engine, it has a couple of fatal flaws:

- An LLM will occasionally make things up. It will output false "facts" that were not present in the training data but could be interpolated from the training data. This information can range from misleading to dangerous.
- An LLM's knowledge of the world has a cutoff date—at best, the date the model was pretrained. Training an LLM is quite expensive, and it is not feasible to train continuously on new data. So at some arbitrary point in time, an LLM's knowledge of the world will just stop.

No one wants to use a search engine that can only tell you about things that happened six months ago. But if we think of an LLM as more like "conversational software" that can handle any sequence data in a prompt, what if we instead used the model as the interface to information retrieved by more traditional search? That's the idea behind *retrieval-augmented generation* or *RAG*.

RAG works by taking an initial user question and doing some form of a query to pull in additional text context. This query can be to a database, a search engine, or anything that can give further information on the question asked by a user. This extra information is then added straight into the prompt. For example, you might construct a prompt like this:

FC
```
Use the following pieces of context to answer the question.
Question: What are some good ways to improve sleep?
Context: {text from a medical journal on improving sleep}
Answer:
```

A common approach for looking up relevant information is to use a *vector database*. To build a vector database, you can use an LLM, or any model, to embed a series of source documents as vectors. The document text will be stored in the database, with the embedding vector used as a key. During retrieval, an LLM can again be used to embed the user query as a vector. The vector database is responsible for searching for key vectors close to the query vector and for surfacing the corresponding text. This might sound a lot like the attention mechanism itself—recall that the terms "query," "key," and "value" actually came from database systems.

Surfacing information to assist with generation does a few things:

- It gives you an obvious way to work around the cutoff date of the model.
- It allows the model to access private data. Companies might want to use an LLM trained on public data to serve as an interface to information stored privately.
- It can help factually ground the model. There is no silver bullet that stops hallucinations entirely, but an LLM is much less likely to make up facts on a topic if presented with correct context about the subject in a prompt.

16.4.4 "Reasoning" models

For years since the first LLMs, researchers have struggled with the well-known fact that these models were abysmal at math problems and logic puzzles. A model might give a perfect response to a problem directly in its training data, but substitute a few names or numbers in the prompt, and it would become evident that the model had no grasp on what it was trying to solve. For many problems in natural language processing, LLMs gave an easy recipe for progress: increase the amount of training data, increase some benchmark score. Grade school math problems, however, defied progress.

In 2023, researchers from Google noticed that if you prompted the model with a few examples of "showing your work" on a math problem—as in literally writing out the steps like you would on a homework assignment—the model would start to do the same. As the model mimicked writing out intermediate steps, it would actually do far better at reaching the correct solution by attending to its own output. They called this "chain-of-thought" prompting, and the name stuck. Another group of researchers noticed that

you didn't even need examples; you could simply prompt the model with the phrase "Let's think step by step" and get better output.

Since these discoveries, there has been heavy interest in directly training LLMs to get better at chain-of-thought reasoning. Models like OpenAI's o1 and DeepSeek's r1 have made headlines by showing significant strides in math and coding problems by training a model to "think out loud" on difficult questions.

The approach for this chain-of-thought fine-tuning is very similar to RLHF. We will first train the model on a few supervised examples of "showing your work" on a math problem and arriving at a correct answer. Next, we will prompt the model with a new math question and check whether the model got the final answer correct. Finally, we use these new generated outputs to further tune the model's weights.

Let's try this out with the Gemma model. We can write out our own word problem and turn on random sampling so we get a somewhat random response each time:

```
prompt = """Judy wrote a 2-page letter to 3 friends twice a week for 3 months.
How many letters did she write?
Be brief, and add "ANSWER:" before your final answer."""

gemma_lm.compile(sampler="random")
```

Turns on random sampling to get a diverse range of outputs

Let's try generating a couple of responses:

```
>>> gemma_lm.generate(PROMPT_TEMPLATE.format(prompt))
<start_of_turn>user
Judy wrote a 2-page letter to 3 friends twice a week for 3 months.
How many letters did she write?
Be brief, and add "ANSWER:" before your final answer.<end_of_turn>
<start_of_turn>model
Here's how to solve the problem:

* **Letters per week:** 3 friends * 2 letters/week = 6 letters/week
* **Letters per month:** 6 letters/week * 4 weeks/month = 24 letters/month
* **Letters in 3 months:** 24 letters/month * 3 months = 72 letters
* **Total letters:** 72 letters * 2 = 144 letters

ANSWER: 144<end_of_turn>
>>> gemma_lm.generate(PROMPT_TEMPLATE.format(prompt))
<start_of_turn>user
Judy wrote a 2-page letter to 3 friends twice a week for 3 months.
How many letters did she write?
Be brief, and add "ANSWER:" before your final answer.<end_of_turn>
<start_of_turn>model
Here's how to solve the problem:

* **Letters per week:** 3 friends * 2 letters/week = 6 letters/week
```

```
* **Letters per month:** 6 letters/week * 4 weeks/month = 24 letters/month
* **Total letters:** 24 letters/month * 3 months = 72 letters

ANSWER: 72<end_of_turn>
```

In the first attempt, our model was hung up on the superfluous detail that each letter has two pages. In the second attempt, the model gets the problem right. This instruction-tuned Gemma model we are working with has already been tuned on math problems like this; you would not get nearly as good results from the "untuned" Gemma model from the last section.

We could extend this idea to a very simple form of chain-of-thought training:

1. Collect a bunch of basic math and reasoning problems and desired answers.
2. Generate (with some randomness) a number of responses.
3. Find all the responses with a correct answer via string parsing. You can prompt the model to use a specific text marker for the final answer as we did previously.
4. Run supervised fine-tuning on correct responses, including all the intermediate output.
5. Repeat!

The previously described process is a reinforcement learning algorithm. Our answer checking acts as the *environment*, and the generated outputs are the *actions* the model uses to learn. As with RLHF, in practice you would use a more complex gradient update step to use information from all responses (even the incorrect ones), but the basic principle is the same.

The same idea is being used to improve LLM performance in other domains that have obvious, verifiable answers to text prompts. Coding is an important one—you can prompt the LLM to output code and then actually run the code to test the quality of the response.

In all these domains, one trend is clear—as a model learns to solve more difficult questions, the model will spend more and more time "showing its work" before reaching a final answer. You can think of this as the model learning to *search* over its own output of potential solutions. We will discuss this idea further in the final chapter of the book.

16.5 Where are LLMs heading next?

Given the trajectory of LLMs discussed at the beginning of this chapter, it may seem obvious where LLMs will be heading. More parameters! Even better performance! In a general sense, that's probably correct, but our trajectory might not be quite so linear.

If you have a fixed budget for pretraining, say, a million dollars, you can roughly think of it as buying you a fixed amount of compute or floating-point operations (flops). You can either spend those flops on training with more data or training a bigger model. Recent research has pointed out that GPT-3, at 175 billion parameters, was way too big for its computing budget. Training a smaller model on more data would have led to

better model performance. So recently, model sizes have trended flatter while data sizes have trended up.

This doesn't mean that scaling will stop—more computing power *does* generally lead to better LLM performance, and we have yet to see signs of an asymptote where next token prediction performance levels off. Companies are continuing to invest billions of dollars in scaling LLMs and seeing what new capabilities emerge.

Figure 16.8 shows details for some of the major LLMs released from 2018 to 2025.

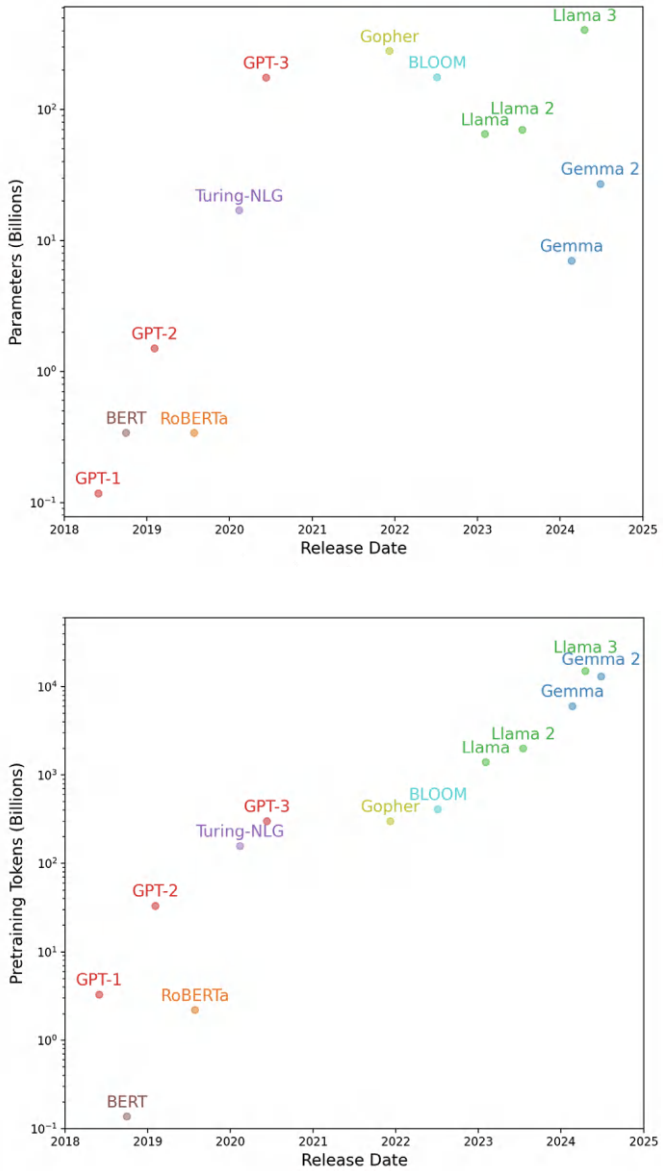


Figure 16.8 LLM parameter counts (top) and pretraining dataset sizes (bottom) over time. Many recent proprietary LLMs (e.g., GPT-4 and Gemini) are not included because model details have not been disclosed.

We can note that while the total number of tokens used for pretraining has climbed steadily and massively, model parameter counts have varied substantially since GPT-3. In part, this is because we now know GPT-3 was undertrained, but it is also for a more practical reason. When deploying a model, it's often worth it to sacrifice performance for a smaller model that fits on cheaper hardware. A really good model won't help very much if it's prohibitively expensive to run.

There's another reason we might not be able to just scale up these models thoughtlessly: we are starting to run out of pretraining data! Tech companies are starting to have trouble finding more high-quality, public, human-written content to throw at pretraining. Models are even starting to "eat their own tail" by training on a significant portion of content created by other LLMs, which runs into a whole other host of concerns. This is one of the reasons reinforcement learning is getting a lot of attention recently. If you can create a difficult, self-contained *environment* that generates new problems for an LLM to attempt, you will have found a way to continue training using the model's own output—no need to scrounge the web for more morsels of quality text.

None of the solutions we touched on will be a silver bullet for the issues facing LLMs. At the end of the day, the fundamental problem remains that LLMs are wildly inefficient at learning compared to humans. Model capabilities only come from training on many orders of magnitude more text than people will read in their lifetimes. As scaling LLMs continues, so too will more fundamental research in how to make models that can learn quickly with limited data.

Still, LLMs represent the ability to build fluent natural language interfaces, and that alone will bring about a massive shift in what we can accomplish with computing devices. In this chapter, we have laid out the basic recipe that many LLMs use to achieve these capabilities.

Summary

- Large language models, or LLMs, are the combination of a few key ingredients:
  - The Transformer architecture
  - A language modeling task (predicting the next token based on past tokens)
  - A large amount of unlabeled text data
- An LLM learns a probability distribution for predicting individual tokens. This can be combined with a sampling strategy to generate a long string of text. There are many popular ways to sample text:
  - *Greedy search* takes the most likely predicted token at each generation step.
  - *Random sampling* directly samples the predicted categorical distribution over all tokens.
  - *Top-K sampling* restricts the categorical distribution to the top set of K candidates.
- LLMs use billions of parameters and are trained on trillions of words of text.

- LLM output is unreliable, and all LLMs will occasionally hallucinate factually incorrect information.
- LLMs can be fine-tuned to follow instructions in a chat dialog. This type of fine-tuning is called *instruction fine-tuning*:
  - The simplest form of instruction fine-tuning involves directly training the model on instruction and response pairs.
  - More advanced forms of instruction fine-tuning involve reinforcement learning.
- The most common resource bottleneck when working with LLMs is accelerator memory.
- LoRA is a technique to reduce memory usage by freezing most Transformer parameters and only updating a low-rank decomposition of attention projection weights.
- LLMs can input or output data from different modalities if you can figure out how to frame these inputs or outputs as sequences in a sequence prediction problem.
- A *foundation model* is a general term for models of any modality trained using self-supervision for a wide range of downstream tasks.

17 Image generation

This chapter covers

- Variational autoencoders
- Diffusion models
- Using a pretrained text-to-image model
- Exploring the latent image spaces learned by text-to-image models

The most popular and successful application of creative AI today is image generation: learning latent visual spaces and sampling from them to create entirely new pictures, interpolated from real ones—pictures of imaginary people, imaginary places, imaginary cats and dogs, and so on.

17.1 Deep learning for image generation

In this section and the next, we'll review some high-level concepts pertaining to image generation, alongside implementation details relative to two of the main techniques in this domain: *variational autoencoders* (VAEs) and *diffusion models*. Do note that the techniques we present here aren't specific to images—you could develop

latent spaces of sound or music using similar models—but in practice, the most interesting results so far have been obtained with pictures, and that's what we focus on here.

17.1.1 Sampling from latent spaces of images

The key idea of image generation is to develop a low-dimensional *latent space* of representations (which, like everything else in deep learning, is a vector space) where any point can be mapped to a "valid" image: an image that looks like the real thing. The module capable of realizing this mapping, taking as input a latent point and outputting an image (a grid of pixels), is usually called a *generator*, or sometimes a *decoder*. Once such a latent space has been learned, you can sample points from it, and, by mapping them back to image space, generate images that have never been seen before (see figure 17.1)—the in-betweens of the training images.

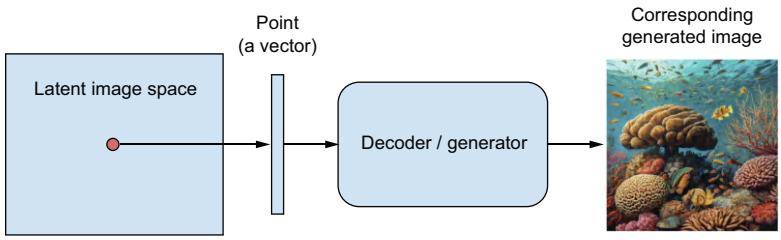


Figure 17.1 Using a latent vector space to sample new images

Further, *text-conditioning* makes it possible to map a space of prompts in natural language to the latent space (see figure 17.2), making it possible to do *language-guided image generation*—generating pictures that correspond to a text description. This category of models is called *text-to-image* models.

Interpolating between many training images in the latent space enables such models to generate infinite combinations of visual concepts, including many that no one had explicitly come up with before. A horse riding a bike on the moon? You got it. This makes image generation a powerful brush for creative-minded people to play with.

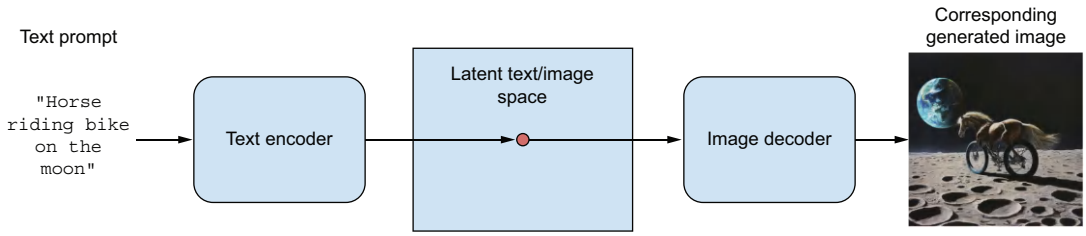


Figure 17.2 Language-guided image generation

Of course, there are still challenges to overcome. Like with all deep learning models, the latent space doesn't encode a consistent model of the physical world, so you might occasionally see hands with extra fingers, incoherent lighting, or garbled objects. The coherence of generated images is still an area of active research. In the case of figure 17.2, despite having seen tens of thousands of images of people riding bikes, the model doesn't understand in a human sense what it means to ride a bike—concepts like pedaling, steering, or maintaining upright balance. That's why your bike-riding horse is unlikely to get depicted pedaling with its hind legs in a believable manner, the way a human artist would draw it.

There's a range of different strategies for learning such latent spaces of image representations, each with its own characteristics. The most common types of image generation models are

- Diffusion models
- Variational autoencoders (VAEs)
- Generative adversarial networks (GANs)

While previous editions of this book covered GANs, they have gradually fallen out of fashion in recent years and have been all but replaced by diffusion models. In this edition, we'll cover both VAEs and diffusion models and we will skip GANs. In the models we'll build ourselves, we'll focus on unconditioned image generation—sampling images from a latent space without text conditioning. However, you will also learn how to use a pretrained text-to-image model and how to explore its latent space.

17.1.2 Variational autoencoders

VAEs, simultaneously discovered by Kingma and Welling in December 2013[1] and Rezende, Mohamed, and Wierstra in January 2014,[2] are a kind of generative model that's especially appropriate for the task of image editing via concept vectors. They're a kind of *autoencoder*—a type of network that aims to encode an input to a low-dimensional latent space and then decode it back—that mixes ideas from deep learning with Bayesian inference.

VAEs have been around for over a decade, but they remain relevant to this day and continue to be used in recent research. While VAEs will never be the first choice for generating high-fidelity images—where diffusion models excel—they remain an important tool in the deep learning toolbox, particularly when interpretability, control over the latent space, and data reconstruction capabilities are crucial. It's also your first contact with the concept of the autoencoder, which is useful to know about. VAEs beautifully illustrate the core idea behind this class of models.

A classical image autoencoder takes an image, maps it to a latent vector space via an encoder module, and then decodes it back to an output with the same dimensions as

[1] Diederik P. Kingma and Max Welling, "Auto-Encoding Variational Bayes," arXiv (2013), https://arxiv.org/abs/1312.6114.

[2] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra, "Stochastic Backpropagation and Approximate Inference in Deep Generative Models," arXiv (2014), https://arxiv.org/abs/1401.4082.

the original image, via a decoder module (see figure 17.3). It's then trained by using as target data the *same images* as the input images, meaning the autoencoder learns to reconstruct the original inputs. By imposing various constraints on the code (the output of the encoder), you can get the autoencoder to learn more or less interesting latent representations of the data. Most commonly, you'll constrain the code to be low dimensional and sparse (mostly zeros), in which case the encoder acts as a way to compress the input data into fewer bits of information.

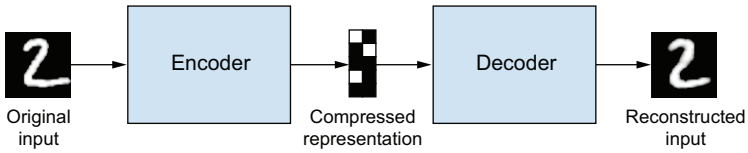


Figure 17.3 An autoencoder: mapping an input `x` to a compressed representation and then decoding it back as `x'`

In practice, such classical autoencoders don't lead to particularly useful or nicely structured latent spaces. They're not much good at compression either. For these reasons, they have largely fallen out of fashion. VAEs, however, augment autoencoders with a little bit of statistical magic that forces them to learn continuous, highly structured latent spaces. They have turned out to be a powerful tool for image generation.

A VAE, instead of compressing its input image into a fixed code in the latent space, turns the image into the parameters of a statistical distribution: a mean and a variance. Essentially, this means we're assuming the input image has been generated by a statistical process, and that the randomness of this process should be taken into account during encoding and decoding. The VAE then uses the mean and variance parameters to randomly sample one element of the distribution, and decodes that element back to the original input (see figure 17.4). The stochasticity of this process improves robustness and forces the latent space to encode meaningful representations everywhere: every point sampled in the latent space is decoded to a valid output.

In technical terms, here's how a VAE works:

1. An encoder module turns the input sample `input_img` into two parameters in a latent space of representations, `z_mean` and `z_log_variance`.
2. You randomly sample a point `z` from the latent normal distribution that's assumed to generate the input image, via `z = z_mean + exp(z_log_variance) * epsilon`, where `epsilon` is a random tensor of small values.
3. A decoder module maps this point in the latent space back to the original input image.

Because `epsilon` is random, the process ensures that every point that's close to the latent location where you encoded `input_img` (`z-mean`) can be decoded to something

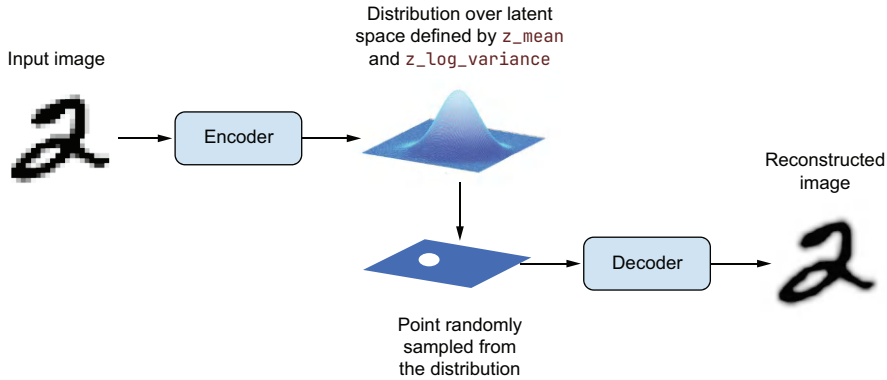


Figure 17.4 A VAE maps an image to two vectors, `z_mean` and `z_log_variance`, which define a probability distribution over the latent space, used to sample a latent point to decode.

similar to `input_img`, thus forcing the latent space to be continuously meaningful. Any two close points in the latent space will decode to highly similar images. Continuity, combined with the low dimensionality of the latent space, forces every direction in the latent space to encode a meaningful axis of variation of the data, making the latent space very structured and thus highly suitable to manipulation via concept vectors.

The parameters of a VAE are trained via two loss functions: a *reconstruction loss* that forces the decoded samples to match the initial inputs, and a *regularization loss* that helps learn well-rounded latent distributions and reduces overfitting to the training data. Schematically, the process looks like this:

```
z_mean, z_log_variance = encoder(input_img)
z = z_mean + exp(z_log_variance) * epsilon
reconstructed_img = decoder(z)
model = Model(input_img, reconstructed_img)
```

Encodes the input into a mean and variance parameter

Draws a latent point using a small random epsilon

Decodes z back to an image

Instantiates the autoencoder model, which maps an input image to its reconstruction

You can then train the model using the reconstruction loss and the regularization loss. For the regularization loss, we typically use an expression (the Kullback–Leibler divergence) meant to nudge the distribution of the encoder output toward a well-rounded normal distribution centered around 0. This provides the encoder with a sensible assumption about the structure of the latent space it's modeling.

Now let's see what implementing a VAE looks like in practice!

17.1.3 Implementing a VAE with Keras

We're going to be implementing a VAE that can generate MNIST digits. It's going to have three parts:

- An encoder network that turns a real image into a mean and a variance in the latent space
- A sampling layer that takes such a mean and variance and uses them to sample a random point from the latent space
- A decoder network that turns points from the latent space back into images

The following listing shows the encoder network you'll use, mapping images to the parameters of a probability distribution over the latent space. It's a simple ConvNet that maps the input image `x` to two vectors, `z_mean` and `z_log_var`. One important detail is that we use strides for downsampling feature maps, instead of max pooling. The last time we did this was in the image segmentation example of chapter 11. Recall that, in general, strides are preferable to max pooling for any model that cares about *information location*—that is, *where* stuff is in the image—and this one does, since it will have to produce an image encoding that can be used to reconstruct a valid image.

Listing 17.1 VAE encoder network

```
import keras
from keras import layers

latent_dim = 2

image_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="same")(
    image_inputs
)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(image_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.

Its summary looks like this:

```
>>> encoder.summary()
Model: "encoder"
```

Layer (type)	Output Shape	Param #	Connected to

input_layer (InputLayer)	(None, 28, 28, 1)	0	-
conv2d (Conv2D)	(None, 14, 14, 32)	320	input_layer[0][0]
conv2d_1 (Conv2D)	(None, 7, 7, 64)	18,496	conv2d[0][0]
flatten (Flatten)	(None, 3136)	0	conv2d_1[0][0]
dense (Dense)	(None, 16)	50,192	flatten[0][0]
z_mean (Dense)	(None, 2)	34	dense[0][0]
z_log_var (Dense)	(None, 2)	34	dense[0][0]

```
Total params: 69,076 (269.83 KB)
Trainable params: 69,076 (269.83 KB)
Non-trainable params: 0 (0.00 B)
```

Next is the code for using `z_mean` and `z_log_var`, the parameters of the statistical distribution assumed to have produced `input_img`, to generate a latent space point `z`.

Listing 17.2 Latent space sampling layer

```
from keras import ops

class Sampler(keras.Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.seed_generator = keras.random.SeedGenerator()
        self.built = True

    def call(self, z_mean, z_log_var):
        batch_size = ops.shape(z_mean)[0]
        z_size = ops.shape(z_mean)[1]
        epsilon = keras.random.normal(
            (batch_size, z_size), seed=self.seed_generator
        )
        return z_mean + ops.exp(0.5 * z_log_var) * epsilon
```

We need a seed generator to use functions from keras.random in call().

Draws a batch of random normal vectors

Applies the VAE sampling formula

The following listing shows the decoder implementation. We reshape the vector `z` to the dimensions of an image and then use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

Listing 17.3 VAE decoder network, mapping latent space points to images

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(
    x
)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(
    x
)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

Produces the same number of coefficients we had at the level of the Flatten layer in the encoder

Input where we'll feed z

Reverts the Flatten layer of the encoder

The output ends up with shape (28, 28, 1).

Reverts the Conv2D layers of the encoder

Its summary looks like this:

```
>>> decoder.summary()
Model: "decoder"
```

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 2)	0
dense_1 (Dense)	(None, 3136)	9,408
reshape (Reshape)	(None, 7, 7, 64)	0
conv2d_transpose (Conv2DTranspose)	(None, 14, 14, 64)	36,928
conv2d_transpose_1 (Conv2DTranspose)	(None, 28, 28, 32)	18,464
conv2d_2 (Conv2D)	(None, 28, 28, 1)	289

```
Total params: 65,089 (254.25 KB)
Trainable params: 65,089 (254.25 KB)
Non-trainable params: 0 (0.00 B)
```

Now, let's create the VAE model itself. This is your first example of a model that isn't doing supervised learning (an autencoder is an example of *self-supervised* learning because it uses its inputs as targets). Whenever you depart from classic supervised learning, it's common to subclass the `Model` class and implement a custom `train_step()` to specify the new training logic, a workflow you've learned about in chapter 7. We could easily do that here, but a downside of this technique is that the `train_step()` contents must be backend specific—you'd use `GradientTape` with TensorFlow, you'd use `loss.backward()` with PyTorch, and so on. A simpler way to customize your training logic is to just implement the `compute_loss()` method instead and keep the default `train_step()`. `compute_loss()` is the key bit of differentiable logic called by the built-in `train_step()`. Since it doesn't involve direct manipulation of gradients, it's easy to keep it backend agnostic.

Its signature is as follows:

```
compute_loss(x, y, y_pred, sample_weight=None, training=True)
```

where `x` is the model's input; `y` is the model's target (in our case, it is `None` since the dataset we use only has inputs, no targets); and `y_pred` is the output of `call()`—the model's predictions. In any supervised training workflow, you'd compute the loss based on `y` and `y_pred`. In our case, since `y` is `None` and `y_pred` contains the latent parameters, we'll compute the loss using `x` (the original input) and the `reconstruction` derived from `y_pred`.

The method must return a scalar, the loss value to be minimized. You can also use `compute_loss()` to update the state of your metrics, which is something we'll want to do in our case.

Now, let's write our VAE with a custom `compute_loss()` method. It works with all backends with no code changes!

Listing 17.4 VAE model with custom `compute_loss()` method

```
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.sampler = Sampler()
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    def call(self, inputs):
        return self.encoder(inputs)

    def compute_loss(self, x, y, y_pred, sample_weight=None, training=True):
```

We'll use these metrics to keep track of the loss averages over each epoch.

```
        original = x
        z_mean, z_log_var = y_pred
        reconstruction = self.decoder(self.sampler(z_mean, z_log_var))

        reconstruction_loss = ops.mean(
            ops.sum(
                keras.losses.binary_crossentropy(x, reconstruction), axis=(1, 2)
            )
        )
        kl_loss = -0.5 * (
            1 + z_log_var - ops.square(z_mean) - ops.exp(z_log_var)
        )
        total_loss = reconstruction_loss + ops.mean(kl_loss)

        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return total_loss
```

Argument x is the model's input.

Argument y_pred is the output of call().

This is our reconstructed image.

We sum the reconstruction loss over the spatial dimensions (axes 1 and 2) and take its mean over the batch dimension.

Updates the state of our loss-tracking metrics

Adds the regularization term (Kullback–Leibler divergence)

Finally, you're ready to instantiate and train the model on MNIST digits. Because `compute_loss()` already takes care of the loss, you don't specify an external loss at compile time (`loss=None`), which, in turn, means you won't pass target data during training (as you can see, you only pass `x_train` to the model in `fit`).

Listing 17.5 Training the VAE

```
import numpy as np

(x_train, _), (x_test, _) = keras.datasets.mnist.load_data()
mnist_digits = np.concatenate([x_train, x_test], axis=0)
mnist_digits = np.expand_dims(mnist_digits, -1).astype("float32") / 255

vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam())
vae.fit(mnist_digits, epochs=30, batch_size=128)
```

We train on all MNIST digits, so we concatenate the training and test samples.

We don't pass a loss argument in compile(), since the loss is already part of the train_step().

We don't pass targets in fit(), since train_step() doesn't expect any.

Once the model is trained, you can use the `decoder` network to turn arbitrary latent space vectors into images.

Listing 17.6 Sampling a grid of points from the 2D latent space and decoding them to images

```
import matplotlib.pyplot as plt

n = 30                                  # We'll display a grid of 30 × 30 digits (900 digits total).
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

grid_x = np.linspace(-1, 1, n)          # Samples points linearly on a 2D grid
grid_y = np.linspace(-1, 1, n)[::-1]

for i, yi in enumerate(grid_y):         # Iterates over grid locations
    for j, xi in enumerate(grid_x):
        z_sample = np.array([[xi, yi]])
        x_decoded = vae.decoder.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[                         # For each location, samples a digit and adds it to our figure
            i * digit_size : (i + 1) * digit_size,
            j * digit_size : (j + 1) * digit_size,
        ] = digit

plt.figure(figsize=(15, 15))
start_range = digit_size // 2
end_range = n * digit_size + start_range
pixel_range = np.arange(start_range, end_range, digit_size)
sample_range_x = np.round(grid_x, 1)
sample_range_y = np.round(grid_y, 1)
plt.xticks(pixel_range, sample_range_x)
plt.yticks(pixel_range, sample_range_y)
plt.xlabel("z[0]")
plt.ylabel("z[1]")
plt.axis("off")
plt.imshow(figure, cmap="Greys_r")
```

The grid of sampled digits (see figure 17.5) shows a completely continuous distribution of the different digit classes, with one digit morphing into another as you follow a path through latent space. Specific directions in this space have a meaning: for example, there's a direction for "four-ness," "one-ness," and so on.

In the next section, we'll cover in detail another major tool for generating images: diffusion models, the architecture behind nearly all commercial image generation services today.

17.2 Diffusion models

A long-standing application of autoencoders has been *denoising*: feeding into a model an input that features a small amount of noise—for instance, a low-quality JPEG image—and getting back a cleaned-up version of the same input. This is the one task that autoencoders excel at. In the late 2010s, this idea gave rise to very successful *image super-resolution* models, capable of taking in low-resolution, potentially noisy images and outputting high-quality, high-resolution versions of them (see figure 17.6). Such models have been shipped as part of every major smartphone camera app for the past few years.

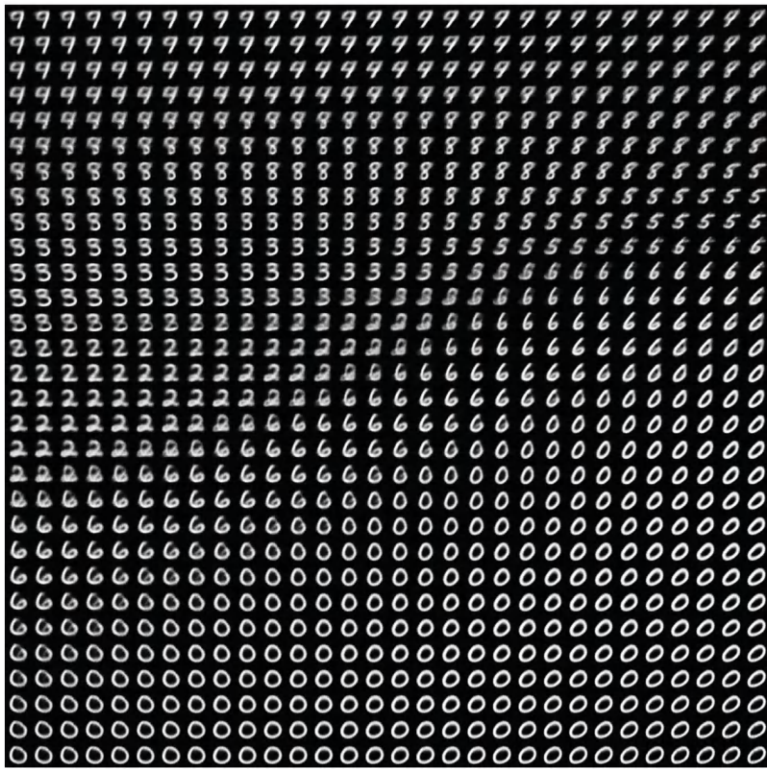


Figure 17.5 Grid of digits decoded from the latent space

Figure 17.6 Image super-resolution

Of course, these models aren't magically recovering lost details hidden in the input, like in the "enhance" scene from *Blade Runner* (1982). Rather, they're making educated guesses about what the image should look like—they're *hallucinating* a cleaned-up,

higher-resolution version of what you give them. This can potentially lead to funny mishaps. For instance, with some AI-enhanced cameras, you can take a picture of something that looks vaguely moon-like (such as a printout of a severely blurred moon image), and you will get in your camera roll a crisp picture of the moon's craters. A lot of detail that simply wasn't present in the printout gets straight-up hallucinated by the camera, because the super-resolution model it uses is overfitted to moon photography images. So, unlike Rick Deckard, definitely don't use this technique for forensics!

Early successes in image denoising led researchers to an arresting idea: since you can use an autoencoder to remove a small amount of noise from an image, surely it would be possible to repeat the process multiple times in a loop to remove a large amount of noise. Ultimately, could you denoise an image made out of *pure noise*?

As it turns out, yes, you can. By doing this, you can effectively hallucinate brand new images out of nothing, like in figure 17.7. This is the key insight behind diffusion models, which should more accurately be called *reverse diffusion* models, since "diffusion" refers to the process of gradually adding noise to an image until it disperses into nothing.

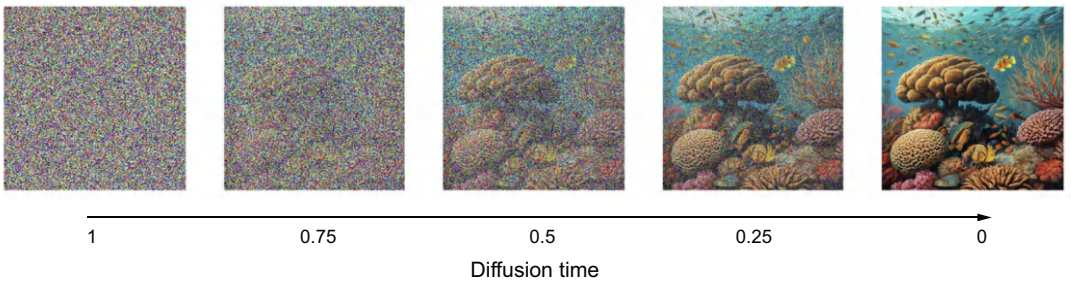


Figure 17.7 Reverse diffusion: turning pure noise into an image via repeated denoising

A diffusion model is essentially a denoising autoencoder in a loop, capable of turning pure noise into sharp, realistic imagery. You may know this poetic quote from Michelangelo, "Every block of stone has a statue inside it and it is the task of the sculptor to discover it"—well, every square of white noise has an image inside it, and it is the task of the diffusion model to discover it.

Now, let's build one with Keras.

17.2.1 The Oxford Flowers dataset

The dataset we're going to use is the Oxford Flowers dataset (https://www.robots.ox.ac.uk/~vgg/data/flowers/102/), a collection of 8,189 images of flowers that belong to 102 different species.

Let's get the dataset archive and extract it:

```
import os

fpath = keras.utils.get_file(
    origin="https://www.robots.ox.ac.uk/~vgg/data/flowers/102/102flowers.tgz",
    extract=True,
)
```

`fpath` is now the local path to the extracted directory. The images are contained in the `jpg` subdirectory there. Let's turn them into an iterable dataset using `image_dataset_from_directory()`.

We need to resize our images to a fixed size, but we don't want to distort their aspect ratio since this would negatively affect the quality of our generated images, so we use the `crop_to_aspect_ratio` option to extract maximally large undistorted crops of the right size (128×128):

```
batch_size = 32
image_size = 128
images_dir = os.path.join(fpath, "jpg")
dataset = keras.utils.image_dataset_from_directory(
    images_dir,
    labels=None,
    image_size=(image_size, image_size),
    crop_to_aspect_ratio=True,
)
dataset = dataset.rebatch(
    batch_size,
    drop_remainder=True,
)
```

We won't need the labels, just the images.

Crops images when resizing them to preserve their aspect ratio

We'd like all batches to have the same size, so we drop the last (irregular) batch.

Here's an example image (figure 17.8):

```
from matplotlib import pyplot as plt

for batch in dataset:
    img = batch.numpy()[0]
    break
plt.imshow(img.astype("uint8"))
```

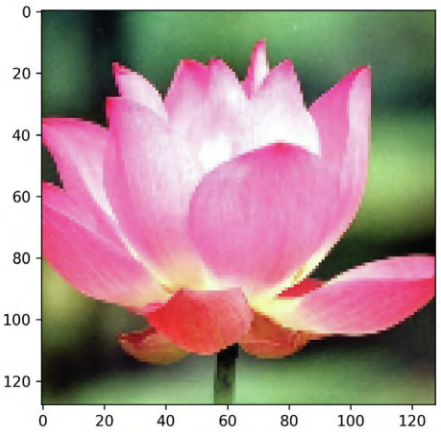


Figure 17.8 An example image from the Oxford Flowers dataset

17.2.2 A U-Net denoising autoencoder

The same denoising model gets reused across each iteration of the diffusion denoising process, erasing a little bit of noise each time. To make the job of the model easier, we tell it how much noise it is supposed to

extract for a given input image—that's the `noise_rates` input. Rather than outputting a denoised image, we make our model output a predicted noise mask, which we can subtract from the input to denoise it.

For our denoising model, we're going to use a U-Net—a kind of ConvNet originally developed for image segmentation. It looks like figure 17.9.

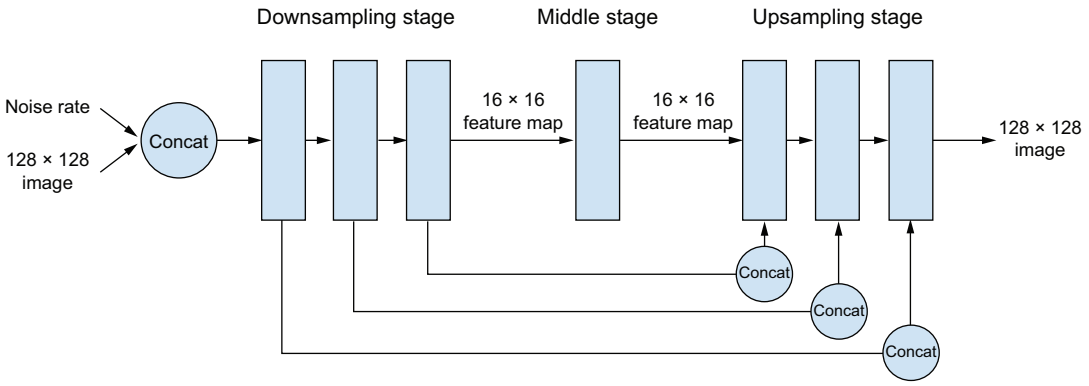


Figure 17.9 Our U-Net-style denoising autoencoder architecture

This architecture features three stages:

1. A *downsampling stage*, made of several blocks of convolution layers, where the inputs get downsampled from their original 128 × 128 size down to a much smaller size (in our case, 16 × 16).
2. A *middle stage*, where the feature map has a constant size.
3. An *upsampling stage*, where the feature map get upsampled back to 128 × 128.

There is a 1:1 mapping between the blocks of the downsampling and upsampling stages: each upsampling block is the inverse of a downsampling block. Importantly, the model features concatenative residual connections going from each downsampling block to the corresponding upsampling block. These connections help avoid loss of image detail information across the successive downsampling and upsampling operations.

Let's assemble the model using the Functional API:

```
def residual_block(x, width):                    ← Utility function to apply a block of
    input_width = x.shape[3]                       layers with a residual connection
    if input_width == width:
        residual = x
    else:
        residual = layers.Conv2D(width, 1)(x)
    x = layers.BatchNormalization(center=False, scale=False)(x)
```

```
    x = layers.Conv2D(width, 3, padding="same", activation="swish")(x)
    x = layers.Conv2D(width, 3, padding="same")(x)
    x = x + residual
    return x

def get_model(image_size, widths, block_depth):
    noisy_images = keras.Input(shape=(image_size, image_size, 3))
    noise_rates = keras.Input(shape=(1, 1, 1))

    x = layers.Conv2D(widths[0], 1)(noisy_images)
    n = layers.UpSampling2D(image_size, interpolation="nearest")(noise_rates)
    x = layers.Concatenate()([x, n])

    skips = []
    for width in widths[:-1]:
        for _ in range(block_depth):
            x = residual_block(x, width)
            skips.append(x)
        x = layers.AveragePooling2D(pool_size=2)(x)

    for _ in range(block_depth):
        x = residual_block(x, widths[-1])

    for width in reversed(widths[:-1]):
        x = layers.UpSampling2D(size=2, interpolation="bilinear")(x)
        for _ in range(block_depth):
            x = layers.Concatenate()([x, skips.pop()])
            x = residual_block(x, width)

    pred_noise_masks = layers.Conv2D(3, 1, kernel_initializer="zeros")(x)

    return keras.Model([noisy_images, noise_rates], pred_noise_masks)
```

Dowsampling stage

Upsampling stage

Middle stage

Creates the functional model

We set the kernel initializer for the last layer to "zeros," making the model predict only zeros after initialization (that is, our default assumption before training is "no noise").

You would instantiate the model with something like `get_model(image_size=128, widths=[32, 64, 96, 128], block_depth=2)`. The `widths` argument is a list containing the `Conv2D` layer sizes for each successive downsampling or upsampling stage. We typically want the layers to get bigger as we downsample the inputs (going from 32 to 128 units here) and then get smaller as as upsample (from 128 back to 32 here).

17.2.3 The concepts of diffusion time and diffusion schedule

The diffusion process is a series of steps in which we apply our denoising autoencoder to erase a small amount of noise from an image, starting with a pure-noise image, and ending with a pure-signal image. The index of the current step in the loop is called the *diffusion time* (see figure 17.7). In our case, we'll use a continuous value between 1

and 0 for this index—a value of 1 indicates the start of the process, where the amount of noise is maximal and the amount of signal is minimal, and a value of 0 indicates the end of the process, where the image is almost all signal and no noise.

The relationship between the current diffusion time and the amount of noise and signal present in the image is called the *diffusion schedule*. In our experiment, we're going to use a cosine schedule to smoothly transition from a high signal rate (low noise) at the beginning to a low signal rate (high noise) at the end of the diffusion process.

Listing 17.7 The diffusion schedule

```
def diffusion_schedule(
    diffusion_times,
    min_signal_rate=0.02,
    max_signal_rate=0.95,
):
    start_angle = ops.cast(ops.arccos(max_signal_rate), "float32")
    end_angle = ops.cast(ops.arccos(min_signal_rate), "float32")
    diffusion_angles = start_angle + diffusion_times * (end_angle - start_angle)
    signal_rates = ops.cos(diffusion_angles)
    noise_rates = ops.sin(diffusion_angles)
    return noise_rates, signal_rates
```

This `diffusion_schedule()` function takes as input a `diffusion_times` tensor, which represents the progression of the diffusion process and returns the corresponding `noise_rates` and `signal_rates` tensors. These rates will be used to guide the denoising process. The logic behind using a cosine schedule is to maintain the relationship `noise_rates ** 2 + signal_rates ** 2 == 1` (see figure 17.10).

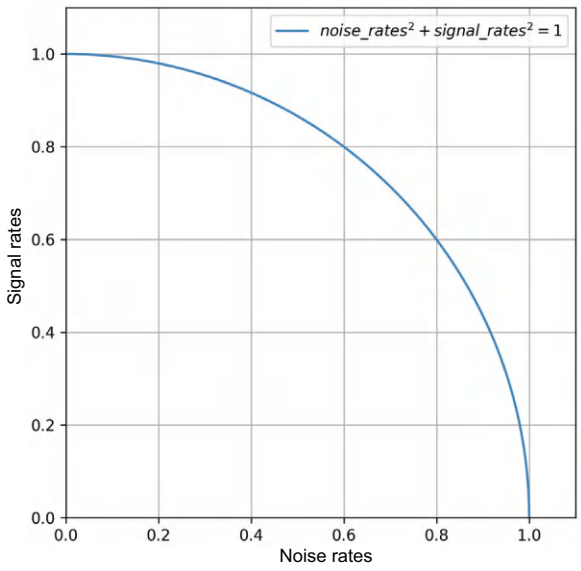


Figure 17.10 Cosine relationship between noise rates and signal rates

Let's plot how this function maps diffusion times (between 0 and 1) to specific noise rates and signal rates (see figure 17.11):

```
diffusion_times = ops.arange(0.0, 1.0, 0.01)
noise_rates, signal_rates = diffusion_schedule(diffusion_times)

diffusion_times = ops.convert_to_numpy(diffusion_times)
noise_rates = ops.convert_to_numpy(noise_rates)
signal_rates = ops.convert_to_numpy(signal_rates)

plt.plot(diffusion_times, noise_rates, label="Noise rate")
plt.plot(diffusion_times, signal_rates, label="Signal rate")

plt.xlabel("Diffusion time")
plt.legend()
```

These lines are only necessary if you're using PyTorch, in which case tensor conversion to NumPy is no longer trivial.

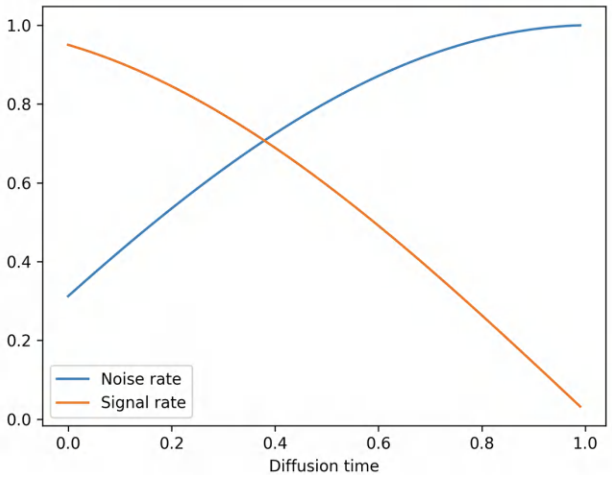


Figure 17.11 Our cosine diffusion schedule

17.2.4 The training process

Let's create a `DiffusionModel` class to implement the training procedure. It's going to have our denoising autoencoder as one of its attributes. We're also going to need a couple more things:

- *A loss function*—We'll use mean absolute error as our loss, that is to say `mean(abs(real_noise_mask - predicted_noise_mask))`.
- *An image normalization layer*—The noise we'll add to the images will have unit variance and zero mean, so we'd like our images to be normalized as such too, for the value range of the noise to match the value range of the images.

Let's start by writing the model constructor:

```
class DiffusionModel(keras.Model):
    def __init__(self, image_size, widths, block_depth, **kwargs):
        super().__init__(**kwargs)
        self.image_size = image_size
        self.denoising_model = get_model(image_size, widths, block_depth)
        self.seed_generator = keras.random.SeedGenerator()
        self.loss = keras.losses.MeanAbsoluteError()
        self.normalizer = keras.layers.Normalization()
```

Our loss function

We'll use this to normalize input images.

The first method we're going to need is the denoising method. It simply calls the denoising model to retrieve a predicted noise mask, and it uses it to reconstruct a denoised image:

```
    def denoise(self, noisy_images, noise_rates, signal_rates):
        pred_noise_masks = self.denoising_model([noisy_images, noise_rates])
        pred_images = (
            noisy_images - noise_rates * pred_noise_masks
        ) / signal_rates
        return pred_images, pred_noise_masks
```

Calls the denoising model

Reconstructs the predicted clean image

Next comes the training logic. This is the most important part! Like in the VAE example, we're going to implement a custom `compute_loss()` method to keep our model backend agnostic. Of course, if you are set on using one specific backend, you could also write a custom `train_step()` with the exact same logic in it, plus the backend-specific logic for gradient computation and weight updates.

Since `compute_loss()` receives as input the output of `call()`, we're going to put the denoising forward pass in `call()`. Our `call()` takes a batch of clean input images and applies the following steps:

1. Normalizes the images
2. Samples random diffusion times (the denoising model needs to be trained on the full spectrum of diffusion times)
3. Computes corresponding noise rates and signal rates (using the diffusion schedule)
4. Adds random noise to the clean images (based on the computed noise rates and signal rates)
5. Denoises the images

It returns

- The predicted denoised images
- The predicted noise masks
- The actual noise masks it applied

These last two quantities are then used in `compute_loss()` to compute the loss of the model on the noise mask prediction task:

```
    def call(self, images):
        images = self.normalizer(images)
        noise_masks = keras.random.normal(
            (batch_size, self.image_size, self.image_size, 3),
            seed=self.seed_generator,
        )
        diffusion_times = keras.random.uniform(
            (batch_size, 1, 1, 1),
            minval=0.0,
            maxval=1.0,
            seed=self.seed_generator,
        )
        noise_rates, signal_rates = diffusion_schedule(diffusion_times)
        noisy_images = signal_rates * images + noise_rates * noise_masks
        pred_images, pred_noise_masks = self.denoise(
            noisy_images, noise_rates, signal_rates
        )
        return pred_images, pred_noise_masks, noise_masks

    def compute_loss(self, x, y, y_pred, sample_weight=None, training=True):
        _, pred_noise_masks, noise_masks = y_pred
        return self.loss(noise_masks, pred_noise_masks)
```

Samples random noise masks

Samples random diffusion times

Adds noise to the images

Denoises them

17.2.5 The generation process

Finally, let's implement the image generation process. We start from pure random noise, and we repeatedly apply the `denoise()` method until we get high-signal, low-noise images.

```
    def generate(self, num_images, diffusion_steps):
        noisy_images = keras.random.normal(
            (num_images, self.image_size, self.image_size, 3),
            seed=self.seed_generator,
        )
        step_size = 1.0 / diffusion_steps
        for step in range(diffusion_steps):
            diffusion_times = ops.ones((num_images, 1, 1, 1)) - step * step_size
            noise_rates, signal_rates = diffusion_schedule(diffusion_times)
```

Computes appropriate noise rates and signal rates

Starts from pure noise

```
            pred_images, pred_noises = self.denoise(
                noisy_images, noise_rates, signal_rates
            )
            next_diffusion_times = diffusion_times - step_size
            next_noise_rates, next_signal_rates = diffusion_schedule(
                next_diffusion_times
            )
            noisy_images = (
               next_signal_rates * pred_images + next_noise_rates * pred_noises
            )
        images = (
            self.normalizer.mean + pred_images * self.normalizer.variance**0.5
        )
        return ops.clip(images, 0.0, 255.0)
```

Calls denoising model

Denormalizes images so their values fit between 0 and 255

Prepares noisy images for the next iteration

17.2.6 Visualizing results with a custom callback

We don't have a proper metric to judge the quality of our generated images, so you're going to want to visualize the generated images yourself over the course of training to judge if your model is getting somewhere. An easy way to do this is with a custom callback. The following callback uses the `generate()` method at the end of each epoch to display a 3×6 grid of generated images:

```
class VisualizationCallback(keras.callbacks.Callback):
    def __init__(self, diffusion_steps=20, num_rows=3, num_cols=6):
        self.diffusion_steps = diffusion_steps
        self.num_rows = num_rows
        self.num_cols = num_cols

    def on_epoch_end(self, epoch=None, logs=None):
        generated_images = self.model.generate(
            num_images=self.num_rows * self.num_cols,
            diffusion_steps=self.diffusion_steps,
        )

        plt.figure(figsize=(self.num_cols * 2.0, self.num_rows * 2.0))
        for row in range(self.num_rows):
            for col in range(self.num_cols):
                i = row * self.num_cols + col
                plt.subplot(self.num_rows, self.num_cols, i + 1)
                img = ops.convert_to_numpy(generated_images[i]).astype("uint8")
                plt.imshow(img)
                plt.axis("off")
        plt.tight_layout()
        plt.show()
        plt.close()
```

17.2.7 It's go time!

It's finally time to train our diffusion model on the Oxford Flowers dataset. Let's instantiate the model:

```
model = DiffusionModel(image_size, widths=[32, 64, 96, 128], block_depth=2)
model.normalizer.adapt(dataset)
```

Computes the mean and variance necessary to perform normalization—don't forget it!

We're going to use `AdamW` as our optimizer, with a few neat options enabled to help stabilize training and improve the quality of the generated images:

- *Learning rate decay*—We gradually reduce the learning rate during training, via an `InverseTimeDecay` schedule.
- *Exponential moving average of model weights*—Also known as Polyak averaging. This technique maintains a running average of the model's weights during training. Every 100 batches, we overwrite the model's weights with this averaged set of weights. This helps stabilize the model's representations in scenarios where the loss landscape is noisy.

The code is

```
model.compile(
    optimizer=keras.optimizers.AdamW(
        learning_rate=keras.optimizers.schedules.InverseTimeDecay(
            initial_learning_rate=1e-3,
            decay_steps=1000,
            decay_rate=0.1,
        ),
        use_ema=True,
        ema_overwrite_frequency=100,
    ),
)
```

Configures the learning rate decay schedule

Turns on Polyak averaging

Configures how often to overwrite the model's weights with their exponential moving average

Let's fit the model. We'll use our `VisualizationCallback` callback to plot examples of generated images after each epoch, and we'll save the model's weights with the `ModelCheckpoint` callback:

```
model.fit(
    dataset,
    epochs=100,
```

```
    callbacks=[
        VisualizationCallback(),
        keras.callbacks.ModelCheckpoint(
            filepath="diffusion_model.weights.h5",
            save_weights_only=True,
            save_best_only=True,
        ),
    ],
)
```

If you're running on Colab, you might run into the error, "Buffered data was truncated after reaching the output size limit." This happens because the logs of `fit()` include images, which take up a lot of space, whereas the allowed output for a single notebook cell is limited. To get around the problem, you can simply chain five `model.fit(..., epochs=20)` calls, in five successive cells. This is equivalent to a single `fit(..., epochs=100)` call.

After 100 epochs (which takes about 90 minutes on a T4, the free Colab GPU), we get pretty generative flowers like these (see figure 17.12).

Figure 17.12 Examples of generated flowers

You can keep training for even longer and get increasingly realistic results.

So that's how image generation with diffusion works! Now, the next step to unlock their potential is to add *text conditioning*, which would result in a text-to-image model, capable of producing images that match a given text caption.

17.3 Text-to-image models

We can use the same basic diffusion process to create a model that maps text input to image output. To do this we need to take a pretrained text encoder (think a transformer encoder like RoBERTa from chapter 15) that can map text to vectors in a continuous embedding space. Then we can train a diffusion model on `(prompt, image)` pairs, where each prompt is a short, textual description of the input image.

We can handle the image input in the same way as we did previously, mapping noisy input to a denoised output that progressively approaches our input image. Critically, we can extend this setup by also passing the embedded text prompt to the denoising model. So rather than our denoising model simply taking in a `noisy_images` input, our model will take two inputs: `noisy_images` and `text_embeddings`. This gives a leg up on the flower denoiser we trained previously. Instead of learning to remove noise from an image without any additional information, the model gets to use a textual representation of the final image to help guide the denoising process.

After training is when things get a bit more fun. Because we have trained a model that can map pure noise to images *conditioned on* a vector representation of some text, we can now pass in pure noise and a never-before-seen prompt and denoise it into an image for our prompt.

Let's try this out. We won't actually train one of these models from scratch in this book—you have all the ingredients you need, but it's quite expensive and time consuming to train a text-to-image diffusion model that works well. Instead, we will play with a popular pretrained model in KerasHub called Stable Diffusion (figure 17.13). Stable Diffusion is made by a company named Stability AI that specializes in making open models for image and video generation. We can use the third version of their image generation model in KerasHub with just a couple of lines of code.

Figure 17.13 An example output from our Stable Diffusion model

Listing 17.8 Creating a Stable Diffusion text-to-image model

```
import keras_hub

height, width = 512, 512
task = keras_hub.models.TextToImage.from_preset(
    "stable_diffusion_3_medium",
    image_shape=(height, width, 3),
```

```
    dtype="float16",                          ← A trick to keep memory usage down. More details in chapter 18.
)
prompt = "A NASA astraunaut riding an origami elephant in New York City"
task.generate(prompt)
```

Like the `CausalLM` task we covered last chapter, the `TextToImage` task is a high-level class for performing image generation conditioned on text input. It wraps tokenization and the diffusion process into a high-level generate call.

The Stable Diffusion model actually adds a second "negative prompt" to its model, which can be used to steer the diffusion process away from certain text inputs. There's nothing magic here. To add a negative prompt, you could simply train a model on triplets: `(image, positive_prompt, negative_prompt)`, where the positive prompt is a description of the image, and the negative prompt is a series of words that do not describe the image. By feeding the positive and negative text embedding to the denoiser, the denoiser will learn to steer the noise toward images that match the positive prompt and away from images that match the negative prompt (figure 17.14). Let's try removing the color blue from our input:

Figure 17.14 Using a negative prompt to steer the model away from the color blue

```
task.generate(
    {
        "prompts": prompt,
        "negative_prompts": "blue color",
    }
)
```

Visual artifacts in the Stable Diffusion output

You will notice plenty of visual artifacts in our Stable Diffusion output if you look closely. Notably, our second elephant has duplicated tusks!

Some of this is unavoidable when using diffusion models. Figuring out how to truly draw a human in a space suit sitting on an elephant made of paper would require some understanding of anatomy and physics that our model lacks. The model will

always try its best to interpolate an output based on its training data, but it doesn't have any real understanding of the objects it is attempting to represent.

However, there is another factor that is easily fixable: we are using the less powerful version of Stable Diffusion 3. The "medium" model we are using is the smallest one released by Stability AI and uses about 3 billion parameters in total. There is a larger 9-billion parameter model available that would produce substantially higher-quality images with fewer visual artifacts. We do not use it simply to keep the code example in this book accessible—9 billion parameters need a lot of RAM!

Like the `generate()` method for text models we used in the last chapter, we have a few additional parameters we can pass to control the generation process. Let's try passing a variable number of diffusion steps to our model to see the denoising process in action (figure 17.15):

```
import numpy as np
from PIL import Image

def display(images):
    return Image.fromarray(np.concatenate(images, axis=1))

display([task.generate(prompt, num_steps=x) for x in [5, 10, 15, 20, 25]])
```

Figure 17.15 Controlling the number of diffusion steps

17.3.1 Exploring the latent space of a text-to-image model

There is probably no better way to see the interpolative nature of deep neural networks than text diffusion models. The text encoder used by our model will learn a smooth, low-dimensional manifold to represent our input prompts. It's continuous, meaning we have learned a space where we can walk from the text representation of one prompt to another, and each intermediate point will have semantic meaning. We can couple that with our diffusion process to morph between two images by simply describing each end state with a text prompt.

Before we can do this, we will need to break up our high-level `generate()` function into its constituent parts. Let's try that out.

Listing 17.9 Breaking down the `generate()` function

```
from keras import random

def get_text_embeddings(prompt):
    token_ids = task.preprocessor.generate_preprocess([prompt])
    negative_token_ids = task.preprocessor.generate_preprocess([""])
    return task.backbone.encode_text_step(token_ids, negative_token_ids)

def denoise_with_text_embeddings(embeddings, num_steps=28, guidance_scale=7.0):
    latents = random.normal((1, height // 8, width // 8, 16))
    for step in range(num_steps):
        latents = task.backbone.denoise_step(
            latents,
            embeddings,
            step,
            num_steps,
            guidance_scale,
        )
    return task.backbone.decode_step(latents)[0]

def scale_output(x):
    x = ops.convert_to_numpy(x)
    x = np.clip((x + 1.0) / 2.0, 0.0, 1.0)
    return np.round(x * 255.0).astype("uint8")

embeddings = get_text_embeddings(prompt)
image = denoise_with_text_embeddings(embeddings)
scale_output(image)
```

Creates pure noise to denoise into an image

We don't care about negative prompts here, but the model expects them.

Rescales our images back to [0, 255]

Our generation process has three distinct steps:

1. First, we take our prompts, tokenize them, and embed them with our text encoder.
2. Second, we take our text embeddings and pure noise and progressively "denoise" the noise into an image. This is the same as the flower model we just built.
3. Lastly, we map our model outputs, which are from `[-1, 1]` back to `[0, 255]`, so we can render the image.

One thing to note here is that our text embeddings actually contain four separate tensors:

```
>>> [x.shape for x in embeddings]
[(1, 154, 4096), (1, 154, 4096), (1, 2048), (1, 2048)]
```

Rather than only passing the final, embedded text vector to the denoising model, the Stable Diffusion authors chose to pass both the final output vector and the last

representation of the entire token sequence learned by the text encoder. This effectively gives our denoising model more information to work with. The authors do this for both the positive and negative prompts, so we have a total of four tensors here:

- The positive prompt's encoder sequence
- The negative prompt's encoder sequence
- The positive prompt's encoder vector
- The negative prompt's encoder vector

With our `generate()` function decomposed, we can now try walking the latent space between two text prompts. To do so, let's build a function to interpolate between the text embeddings outputted by the model.

Listing 17.10 A function to interpolate text embeddings

```
from keras import ops

def slerp(t, v1, v2):
    v1, v2 = ops.cast(v1, "float32"), ops.cast(v2, "float32")
    v1_norm = ops.linalg.norm(ops.ravel(v1))
    v2_norm = ops.linalg.norm(ops.ravel(v2))
    dot = ops.sum(v1 * v2 / (v1_norm * v2_norm))
    theta_0 = ops.arccos(dot)
    sin_theta_0 = ops.sin(theta_0)
    theta_t = theta_0 * t
    sin_theta_t = ops.sin(theta_t)
    s0 = ops.sin(theta_0 - theta_t) / sin_theta_0
    s1 = sin_theta_t / sin_theta_0
    return s0 * v1 + s1 * v2

def interpolate_text_embeddings(e1, e2, start=0, stop=1, num=10):
    embeddings = []
    for t in np.linspace(start, stop, num):
        embeddings.append(
            (
                slerp(t, e1[0], e2[0]),
                e1[1],
                slerp(t, e1[2], e2[2]),
                e1[3],
            )
        )
    return embeddings
```

The second and fourth text embeddings are for the negative prompt, which we do not use.

You'll notice we use a special interpolation function called `slerp` to walk between our text embeddings. This is short for *spherical linear interpolation*—it's a function that has been used in computer graphics for decades to interpolate points on a sphere.

Don't worry too much about the math; it's not important for our example, but it is important to understand the motivation. If we imagine our text manifold as a sphere and our two prompts as random points on that sphere, directly linearly interpolating

between these two points would land us inside the sphere. We would no longer be on its surface. We would like to stay on the surface of the smooth manifold learned by our text embedding—that's where embedding points have meaning for our denoising model. See figure 17.16.

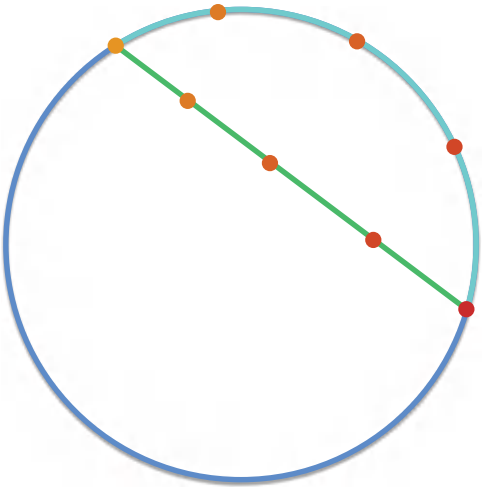


Figure 17.16 Spherical interpolation keeps us close to the surface of our manifold.

Of course, the manifold learned by our text embedding model is not actually spherical. But it's a smooth surface of numbers all with the same rough magnitude—it is *sphere-like*, and interpolating as if we were on a sphere is a better approximation than interpolating as if we were on a line.

With our interpolation defined, let's try walking between the text embeddings for two prompts and generating an image at each interpolated output. We will run our `slerp` function from 0.5 to 0.6 (out of 0 to 1) to zoom in on the middle of the interpolation right when the "morph" becomes visually obvious (figure 17.17):

```
prompt1 = "A friendly dog looking up in a field of flowers"
prompt2 = "A horrifying, tentacled creature hovering over a field of
flowers"
e1 = get_text_embeddings(prompt1)
e2 = get_text_embeddings(prompt2)

images = []
for et in interpolate_text_embeddings(e1, e2, start=0.5, stop=0.6, num=9):
    image = denoise_with_text_embeddings(et)
    images.append(scale_output(image))
display(images)
```

Zooms in to the middle of the overall interpolation from [0, 1]

Figure 17.17 Interpolating between two prompts and generating outputs

This might feel like magic the first time you try it, but there's nothing magic about it—interpolation is fundamental to the way deep neural networks learn. This will be the last substantive model we work with in the book, and it's a great visual metaphor to end with. Deep neural networks are interpolation machines; they map complex, real-world probability distributions to low-dimensional manifolds. We can exploit this fact even for input as complex as human language and output as complex as natural images.

Summary

- Image generation with deep learning is done by learning latent spaces that capture statistical information about a dataset of images. By sampling and decoding points from the latent space, you can generate never-before-seen images. There are three major tools to do this: VAEs, diffusion models, and GANs.
- VAEs result in highly structured, continuous latent representations. For this reason, they work well for doing all sorts of image editing in latent space: face swapping, turning a frowning face into a smiling face, and so on. They also work nicely for doing latent space–based animations, such as animating a walk along a cross section of the latent space, showing a starting image slowly morphing into different images in a continuous way.
- Diffusion models result in very realistic outputs and are the dominant method of image generation today. They work by repeatedly denoising an image, starting from pure noise. They can easily be conditioned on text captions to create text-to-image models.
- Stable Diffusion 3 is a state-of-the-art pretrained text-to-image model that you can use to create highly realistic images of your own.
- The visual latent space learned by such text-to-image diffusion models is fundamentally interpolative. You can see this by interpolating between the text embeddings used as inputs to the diffusion process and achieving a smooth interpolation between images as output.

18 Best practices for the real world

This chapter covers

- Hyperparameter tuning
- Model ensembling
- Training Keras models on multiple GPUs or on TPU
- Mixed-precision training
- Quantization

You've come quite far since the beginning of this book. You can now train image classification models, image segmentation models, models for classification or regression on vector data, timeseries forecasting models, text classification models, sequence-to-sequence models, and even generative models for text and images. You've got all the bases covered.

However, your models so far have all been trained at a small scale—on small datasets, with a single GPU—and they generally haven't reached the best achievable performance on each dataset we've looked at. This book is, after all, an introductory book. If you are to go out into the real world and achieve state-of-the-art results on brand new problems, there's still a bit of a chasm that you'll need to cross.

This chapter is about bridging that gap and giving you the best practices you'll need as you go from machine learning student to a fully fledged machine learning engineer. We'll review essential techniques for systematically improving model performance: hyperparameter tuning and model ensembling. Then we'll look at how you can speed up and scale up model training, with multi-GPU and TPU training, mixed precision, and quantization.

18.1 Getting the most out of your models

Blindly trying out different architecture configurations works well enough if you just need something that works okay. In this section, we'll go beyond "works okay" to "works great and wins machine learning competitions" via a quick guide to a set of must-know techniques for building state-of-the-art deep learning models.

18.1.1 Hyperparameter optimization

When building a deep learning model, you have to make many seemingly arbitrary decisions: How many layers should you stack? How many units or filters should go in each layer? Should you use `relu` as an activation, or a different function? Should you use `BatchNormalization` after a given layer? How much dropout should you use? And so on. These architecture-level parameters are called *hyperparameters* to distinguish them from the *parameters* of a model, which are trained via backpropagation.

In practice, experienced machine learning engineers and researchers build intuition over time as to what works and what doesn't when it comes to these choices—they develop hyperparameter-tuning skills. But there are no formal rules. If you want to get to the very limit of what can be achieved on a given task, you can't be content with such arbitrary choices. Your initial decisions are almost always suboptimal, even if you have very good intuition. You can refine your choices by tweaking them by hand and retraining the model repeatedly—that's what machine learning engineers and researchers spend most of their time doing. But it shouldn't be your job as a human to fiddle with hyperparameters all day—that is better left to a machine.

Thus, you need to explore the space of possible decisions automatically and systematically in a principled way. You need to search the architecture space and find the best-performing ones empirically. That's what the field of automatic hyperparameter optimization is about: it's an entire field of research, and an important one.

The process of optimizing hyperparameters typically looks like this:

1. Choose a set of hyperparameters (automatically).
2. Build the corresponding model.
3. Fit it to your training data, and measure performance on the validation data.
4. Choose the next set of hyperparameters to try (automatically).
5. Repeat.
6. Eventually, measure performance on your test data.

The key to this process is the algorithm that analyzes the relationship between validation performance and various hyperparameter values to choose the next set of hyperparameters to evaluate. Many different techniques are possible: Bayesian optimization, genetic algorithms, simple random search, and so on.

Training the weights of a model is relatively easy: you compute a loss function on a mini-batch of data and then use backpropagation to move the weights in the right direction. Updating hyperparameters, on the other hand, presents unique challenges. Consider that

- The hyperparameter space is typically made of discrete decisions and thus isn't continuous or differentiable. Hence, you typically can't do gradient descent in hyperparameter space. Instead, you must rely on gradient-free optimization techniques, which, naturally, are far less efficient than gradient descent.
- Computing the feedback signal of this optimization process (does this set of hyperparameters lead to a high-performing model on this task?) can be extremely expensive: it requires creating and training a new model from scratch on your dataset.
- The feedback signal may be noisy: if a training run performs 0.2% better, is that because of a better model configuration or because you got lucky with the initial weight values?

Thankfully, there's a tool that makes hyperparameter tuning simpler: KerasTuner. Let's check it out.

USING KERASTUNER

Let's start by installing KerasTuner:

```
!pip install keras-tuner -q
```

The key idea that KerasTuner is built upon is to let you replace hardcoded hyperparameter values, such as `units=32`, with a range of possible choices, such as `Int(name="units", min_value=16, max_value=64, step=16)`. The set of such choices in a given model is called the *search space* of the hyperparameter tuning process.

To specify a search space, define a model-building function (see the next listing). It takes an `hp` argument, from which you can sample hyperparameter ranges, and it returns a compiled Keras model.

Listing 18.1 A KerasTuner model-building function

```
import keras
from keras import layers

def build_model(hp):
    units = hp.Int(name="units", min_value=16, max_value=64, step=16)
    model = keras.Sequential(
```

Sample hyperparameter values from the hp object. After sampling, these values (such as the "units" variable here) are just regular Python constants.

```
        [
            layers.Dense(units, activation="relu"),
            layers.Dense(10, activation="softmax"),
        ]
    )
    optimizer = hp.Choice(name="optimizer", values=["rmsprop", "adam"])
    model.compile(
        optimizer=optimizer,
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

Different kinds of hyperparameters are available: Int, Float, Boolean, Choice.

The function returns a compiled model.

If you want to adopt a more modular and configurable approach to model-building, you can also subclass the `HyperModel` class and define a `build` method.

Listing 18.2 A KerasTuner `HyperModel`

Thanks to the object-oriented approach, we can configure model constants as constructor arguments (instead of hardcoding them in the model-building function).

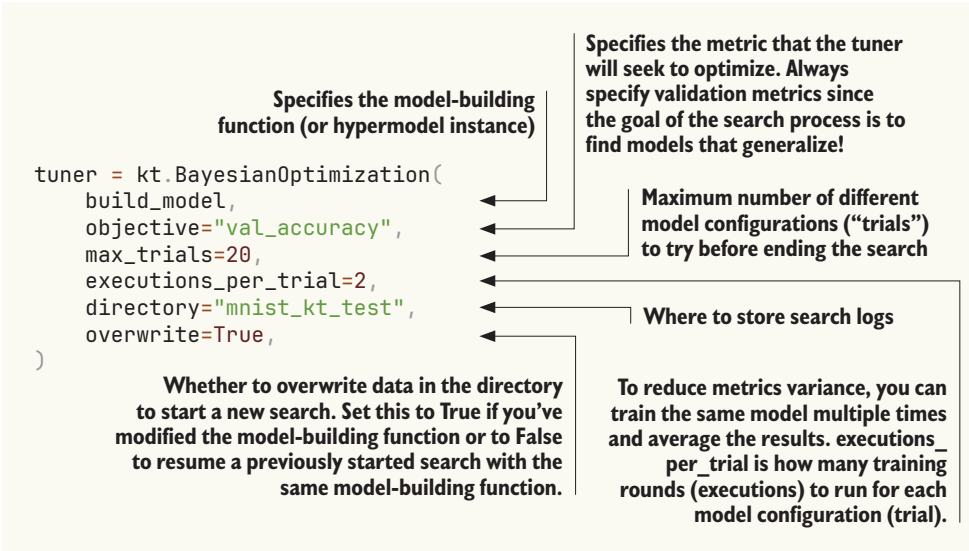
The build method is identical to our prior build_model standalone function.

```
import keras_tuner as kt

class SimpleMLP(kt.HyperModel):
    def __init__(self, num_classes):
        self.num_classes = num_classes

    def build(self, hp):
        units = hp.Int(name="units", min_value=16, max_value=64, step=16)
        model = keras.Sequential(
            [
                layers.Dense(units, activation="relu"),
                layers.Dense(self.num_classes, activation="softmax"),
            ]
        )
        optimizer = hp.Choice(name="optimizer", values=["rmsprop", "adam"])
        model.compile(
            optimizer=optimizer,
            loss="sparse_categorical_crossentropy",
            metrics=["accuracy"],
        )
        return model

hypermodel = SimpleMLP(num_classes=10)
```

The next step is to define a "tuner." Schematically, you can think of a tuner as a `for` loop, which will repeatedly

- Pick a set of hyperparameter values
- Call the model-building function with these values to create a model
- Train the model and record its metrics

KerasTuner has several built-in tuners available—`RandomSearch`, `BayesianOptimization`, and `Hyperband`. Let's try `BayesianOptimization`, a tuner that attempts to make smart predictions for which new hyperparameter values are likely to perform best given the outcome of previous choices:




```
tuner = kt.BayesianOptimization(
    build_model,
    objective="val_accuracy",
    max_trials=20,
    executions_per_trial=2,
    directory="mnist_kt_test",
    overwrite=True,
)
```

You can display an overview of the search space via `search_space_summary()`:

```
>>> tuner.search_space_summary()
Search space summary
Default search space size: 2
units (Int)
{"default": None,
 "conditions": [],
 "min_value": 128,
 "max_value": 1024,
 "step": 128,
 "sampling": None}
optimizer (Choice)
{"default": "rmsprop",
 "conditions": [],
 "values": ["rmsprop", "adam"],
 "ordered": False}
```

Objective maximization and minimization

For built-in metrics (like accuracy, in our case), the *direction* of the metric (accuracy should be maximized, but a loss should be minimized) is inferred by KerasTuner. However, for a custom metric, you should specify it yourself, like this:

```
objective = kt.Objective(
    name="val_accuracy",
    direction="max",
)
tuner = kt.BayesianOptimization(
    build_model,
    objective=objective,
    ...
)
```

The metric's name, as found in epoch logs

The metric's desired direction: "min" or "max"

Finally, let's launch the search. Don't forget to pass validation data and make sure not to use your test set as validation data—otherwise, you'd quickly start overfitting to your test data, and you wouldn't be able to trust your test metrics anymore:

Sets aside a validation set

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape((-1, 28 * 28)).astype("float32") / 255
x_test = x_test.reshape((-1, 28 * 28)).astype("float32") / 255
x_train_full = x_train[:]
y_train_full = y_train[:]
num_val_samples = 10000
x_train, x_val = x_train[:-num_val_samples], x_train[-num_val_samples:]
y_train, y_val = y_train[:-num_val_samples], y_train[-num_val_samples:]
callbacks = [
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=5),
]
tuner.search(
    x_train,
    y_train,
    batch_size=128,
    epochs=100,
    validation_data=(x_val, y_val),
    callbacks=callbacks,
    verbose=2,
)
```

Reserves these for later

This takes the same arguments as fit() (it simply passes them down to fit() for each new model).

Uses a large number of epochs (you don't know in advance how many epochs each model will need) and uses an EarlyStopping callback to stop training when you start overfitting

The previous example will run in just a few minutes since we're only looking at a few possible choices and we're training on MNIST. However, with a typical search space and dataset, you'll often find yourself letting the hyperparameter search run overnight or even over several days. If your search process crashes, you can always restart it—just specify `overwrite=False` in the tuner so that it can resume from the trial logs stored on disk.

Once the search is complete, you can query the best hyperparameter configurations, which you can use to create high-performing models that you can then retrain.

Listing 18.3 Querying the best hyperparameter configurations

```
top_n = 4
best_hps = tuner.get_best_hyperparameters(top_n)
```

Returns a list of HyperParameters objects, which you can pass to the model-building function

Usually, when retraining these models, you may want to include the validation data as part of the training data since you won't be making any further hyperparameter changes, and thus you will no longer be evaluating performance on the validation data. In our example, we'd train these final models on the totality of the original MNIST training data, without reserving a validation set.

Before we can train on the full training data, though, there's one last parameter we need to settle: the optimal number of epochs to train for. Typically, you'll want to train the new models for longer than you did during the search: using an aggressive `patience` value in the `EarlyStopping` callback saves time during the search, but may lead to underfitted models. Just use the validation set to find the best epoch:

```
def get_best_epoch(hp):
    model = build_model(hp)
    callbacks = [
        keras.callbacks.EarlyStopping(
            monitor="val_loss", mode="min", patience=10
        )
    ]
    history = model.fit(
        x_train,
        y_train,
        validation_data=(x_val, y_val),
        epochs=100,
        batch_size=128,
        callbacks=callbacks,
    )
    val_loss_per_epoch = history.history["val_loss"]
    best_epoch = val_loss_per_epoch.index(min(val_loss_per_epoch)) + 1
    print(f"Best epoch: {best_epoch}")
    return best_epoch
```

Note the very high patience value.

And finally, train on the full dataset for just a bit longer than this epoch count, since you're training on more data—20% more, in this case:

```
def get_best_trained_model(hp):
    best_epoch = get_best_epoch(hp)
    model = build_model(hp)
    model.fit(
```

```
        x_train_full, y_train_full, batch_size=128, epochs=int(best_epoch * 1.2)
    )
    return model

best_models = []
for hp in best_hps:
    model = get_best_trained_model(hp)
    model.evaluate(x_test, y_test)
    best_models.append(model)
```

If you're not worried about slightly underperforming, there's a shortcut you can take: just use the tuner to reload the top-performing models with the best weights saved during the hyperparameter search, without retraining new models from scratch:

```
best_models = tuner.get_best_models(top_n)
```

NOTE One important issue to keep in mind when doing automatic hyperparameter optimization at scale is validation-set overfitting. Because you're updating hyperparameters based on a signal that is computed using your validation data, you're effectively training them on the validation data, and thus, they will quickly overfit to the validation data. Always keep this in mind.

THE ART OF CRAFTING THE RIGHT SEARCH SPACE

Overall, hyperparameter optimization is a powerful technique that is an absolute requirement to get to state-of-the-art models on any task or to win machine learning competitions. Think about it: once upon a time, people handcrafted the features that went into shallow machine learning models. That was very suboptimal. Now deep learning automates the task of hierarchical feature engineering—features are learned using a feedback signal, not hand-tuned, and that's the way it should be. In the same way, you shouldn't handcraft your model architectures; you should optimize them in a principled way.

However, doing hyperparameter tuning is not a replacement for being familiar with model architecture best practices: search spaces grow combinatorially with the number of choices, so it would be far too expensive to turn everything into a hyperparameter and let the tuner sort it out. You need to be smart about designing the right search space. Hyperparameter tuning is automation, not magic: you use it to automate experiments that you would otherwise have run by hand, but you still need to handpick experiment configurations that have the potential to yield good metrics.

The good news: by using hyperparameter tuning, the configuration decisions you have to make graduate from micro-decisions (What number of units do I pick for this layer?) to higher-level architecture decisions (Should I use residual connections throughout this model?). And while micro-decisions are specific to a certain model

and a certain dataset, higher-level decisions generalize better across different tasks and datasets: for instance, pretty much every image classification problem can be solved via the same sort of search space template.

Following this logic, KerasTuner attempts to provide *premade search spaces* that are relevant to broad categories of problems—such as image classification. Just add data, run the search, and get a pretty good model. You can try the hypermodels `kt.applications.HyperXception` and `kt.applications.HyperResNet`, which are effectively tunable versions of Keras Applications models.

18.1.2 Model ensembling

Another powerful technique for obtaining the best possible results on a task is *model ensembling*. Ensembling consists of pooling together the predictions of a set of different models to produce better predictions. If you look at machine learning competitions—in particular, on Kaggle—you'll see that the winners use very large ensembles of models that inevitably beat any single model, no matter how good.

Ensembling relies on the assumption that different well-performing models trained independently are likely to be good for different reasons: each model looks at slightly different aspects of the data to make its predictions, getting part of the "truth" but not all of it. You may be familiar with the ancient parable of the blind men and the elephant: a group of blind men come across an elephant for the first time and try to understand what the elephant is by touching it. Each man touches a different part of the elephant's body—just one part, such as the trunk or a leg. Then the men describe to each other what an elephant is: "It's like a snake," "Like a pillar or a tree," and so on. The blind men are essentially machine learning models trying to understand the manifold of the training data, each from its own perspective, using its own assumptions (provided by the unique architecture of the model and the unique random weight initialization). Each of them gets part of the truth of the data, but not the whole truth. By pooling their perspectives together, you can get a far more accurate description of the data. The elephant is a combination of parts: no single blind man gets it quite right, but interviewed together, they can tell a fairly accurate story.

Let's use classification as an example. The easiest way to pool the predictions of a set of classifiers (to *ensemble the classifiers*) is to average their predictions at inference time:

```
preds_a = model_a.predict(x_val)
preds_b = model_b.predict(x_val)
preds_c = model_c.predict(x_val)
preds_d = model_d.predict(x_val)
final_preds = 0.25 * (preds_a + preds_b + preds_c + preds_d)
```

Uses four different models to compute initial predictions

This new prediction array should be more accurate than any of the initial ones.

However, this will work only if the classifiers are more or less equally good. If one of them is significantly worse than the others, the final predictions may not be as good as the best classifier of the group.

A smarter way to ensemble classifiers is to do a weighted average, where the weights are learned on the validation data—typically, the better classifiers are given a higher weight, and the worse classifiers are given a lower weight. To search for a good set of ensembling weights, you can use random search or a simple optimization algorithm, such as the Nelder-Mead algorithm:

```
preds_a = model_a.predict(x_val)
preds_b = model_b.predict(x_val)
preds_c = model_c.predict(x_val)
preds_d = model_d.predict(x_val)
final_preds = 0.5 * preds_a + 0.25 * preds_b + 0.1 * preds_c + 0.15 * preds_d
```

These weights (0.5, 0.25, 0.1, 0.15) are assumed to be learned empirically.

There are many possible variants: you can do an average of an exponential of the predictions, for instance. In general, a simple weighted average with weights optimized on the validation data provides a very strong baseline.

The key to making ensembling work is the *diversity* of the set of classifiers. Diversity is strength. If all the blind men only touched the elephant's trunk, they would agree that elephants are like snakes, and they would forever stay ignorant of the truth of the elephant. Diversity is what makes ensembling work. In machine learning terms, if all of your models are biased in the same way, then your ensemble will retain this same bias. If your models are *biased in different ways*, the biases will cancel each other out, and the ensemble will be more robust and more accurate.

For this reason, you should ensemble models that are *as good as possible* while being *as different as possible*. This typically means using very different architectures or even different brands of machine learning approaches. One thing that is largely not worth doing is ensembling the same network trained several times independently, from different random initializations. If the only difference between your models is their random initialization and the order in which they were exposed to the training data, then your ensemble will be low in diversity and will provide only a tiny improvement over any single model.

One thing I have found to work well in practice—but that doesn't generalize to every problem domain—is the use of an ensemble of tree-based methods (such as random forests or gradient-boosted trees) and deep neural networks. In 2014, Andrei Kolev and I took fourth place in the Higgs Boson decay detection challenge on Kaggle (www.kaggle.com/c/higgs-boson) using an ensemble of various tree models and deep neural networks. Remarkably, one of the models in the ensemble originated from a different method than the others (it was a regularized greedy forest) and had a significantly worse score than the others. Unsurprisingly, it was assigned a small weight in the ensemble. But to our surprise, it turned out to improve the overall ensemble by a large factor because it

was so different from every other model: it provided information that the other models didn't have access to. That's precisely the point of ensembling. It's not so much about how good your best model is; it's about the diversity of your set of candidate models.

18.2 Scaling up model training with multiple devices

Recall the "loop of progress" concept we introduced in chapter 7: the quality of your ideas is a function of how many refinement cycles they've been through (figure 18.1). And the speed at which you can iterate on an idea is a function of how fast you can set up an experiment, how fast you can run that experiment, and, finally, how well you can analyze the resulting data.

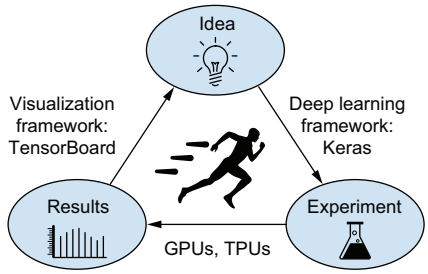


Figure 18.1 The loop of progress

As you develop your expertise in the Keras API, how fast you can code up your deep learning experiments will cease to be the bottleneck of this progress cycle. The next bottleneck will become the speed at which you can train your models. Fast training infrastructure means that you can get your results back in 10 or 15 minutes and, hence, that you can go through dozens of iterations every day. Faster training directly improves the quality of your deep learning solutions.

In this section, you'll learn about how to scale up your training runs by using multiple GPUs or TPUs.

18.2.1 Multi-GPU training

While GPUs are getting more powerful every year, deep learning models are also getting increasingly larger, requiring ever more computational resources. Training on a single GPU puts a hard bound on how fast you can move. The solution? You could simply add more GPUs and start doing *multi-GPU distributed training*.

There are two ways to distribute computation across multiple devices: *data parallelism* and *model parallelism*.

With data parallelism, a single model gets replicated on multiple devices or multiple machines. Each of the model replicas processes different batches of data, and then they merge their results.

With model parallelism, different parts of a single model run on different devices, processing a single batch of data together at the same time. This works best with models that have a naturally parallel architecture, such as models that feature multiple branches. In practice, model parallelism is only used in the case of models that are too large to fit on any single device: it isn't used as a way to speed up training of regular models but as a way to train larger models.

Then, of course, you can also mix both data parallelism and model parallelism: a single model can be split across multiple devices (e.g., 4), and that split model can be

replicated across multiple groups of devices (e.g., twice, for a total of 2 * 4 = 8 devices used).

Let's see how that works in detail.

DATA PARALLELISM: REPLICATING YOUR MODEL ON EACH GPU

Data parallelism is the most common form of distributed training. It operates on a simple principle: divide and conquer. Each GPU receives a copy of the entire model, called a *replica*. Incoming batches of data are split into *N* sub-batches, which are processed by one model replica each, in parallel. This is why it's called *data parallelism*: different samples (data points) are processed in parallel. For instance, with two GPUs, a batch of size 128 would be split into two sub-batches of size 64, which would be processed by two model replicas. Then

- *In inference*—We would retrieve the predictions for each sub-batch and concatenate them to obtain the predictions for the full batch.
- *In training*—We would retrieve the gradients for each sub-batch, average them, and update all model replicas based on the gradient average. The state of the model would then be the same as if you had trained it on the full batch of 128 samples. This is called *synchronous* training, since all replicas are kept in sync—their weights have the same value at all times. Nonsynchronous alternatives exist, but they are less efficient and aren't used anymore in practice.

Data parallelism is a simple and highly scalable way to train your models faster. If you get more devices, just increase your batch size, and your training throughput increases accordingly. It has one limitation, though: it requires your model to be able to fit into one of your devices. However, it is now common to train foundation models that have tens of billions of parameters, which wouldn't fit on any single GPU.

MODEL PARALLELISM: SPLITTING YOUR MODEL ACROSS MULTIPLE GPUS

That's where *model parallelism* comes in. While data parallelism works by splitting your batches of data into sub-batches and processing the sub-batches in parallel, model parallelism works by splitting your model into submodels and running each one on a different device—in parallel. For instance, consider the following model.

Listing 18.4 A large densely connected model

```
model = keras.Sequential(
    [
        keras.layers.Input(shape=(16000,)),
        keras.layers.Dense(64000, activation="relu"),
        keras.layers.Dense(8000, activation="sigmoid"),
    ]
)
```

Each sample has 16,000 features and gets classified into 8,000 potentially overlapping categories by two `Dense` layers. Those are large layers—the first one has about 1 billion

parameters, and the last one has about 512 million parameters. If you're working with two small devices, you won't be able to use data parallelism, since you can't fit the model on a single device. What you can do is *split* a single instance of the model across multiple devices. This is often called *sharding* or *partitioning* a model. There are two main ways to split a model across devices: horizontal partitioning and vertical partitioning.

In horizontal partitioning, each device processes different layers of the model. For example, in the previous model, one GPU would handle the first `Dense` layer, and the other one would handle the second `Dense` layer. The main drawback of this approach is that it can introduce communication overhead. For example, the output of the first layer needs to be copied to the second device before it can be processed by the second layer. This can become a bottleneck, especially if the output of the first layer is large—you'd risk keeping your GPUs idle.

In vertical partitioning, each layer is split across all available devices. Since layers are usually implemented in terms of `matmul` or `convolution` operations, which are highly parallelizable, this strategy is easy to implement in practice and is almost always the best fit for large models. For example, in the previous model, you could split the kernel and bias of the first `Dense` layer into two halves so that each device only receives a kernel of shape `(16000, 32000)` (split along its last axis) and a bias of shape `(32000,)`. You'd compute `matmul(inputs, kernel) + bias` with this half-kernel and half-bias for each device, and you'd merge the two outputs by concatenating them like this:

```
half_kernel_0 = kernel[:, :32000]
half_bias_0 = bias[:32000]

half_kernel_1 = kernel[:, 32000:]
half_bias_1 = bias[32000:]

with keras.device("gpu:0"):
    half_output_0 = keras.ops.matmul(inputs, half_kernel_0) + half_bias_0

with keras.device("gpu:1"):
    half_output_1 = keras.ops.matmul(inputs, half_kernel_1) + half_bias_1
```

In reality, you will want to mix data parallelism and model parallelism. You will split your model across, say, four devices, and you will replicate that split model across multiple groups of two devices—let's say two—each processing one sub-batch of data in parallel. You will then have two replicas, each running on four devices, for a total of eight devices used (figure 18.2).

18.2.2 Distributed training in practice

Now let's see how to implement these concepts in practice. We will only cover the JAX backend, as it is the most performant and most scalable of the various Keras backends, by a mile. If you're doing any kind of large-scale distributed training and you aren't

using JAX, you're making a mistake—and wasting your dollars burning way more compute than you actually need.

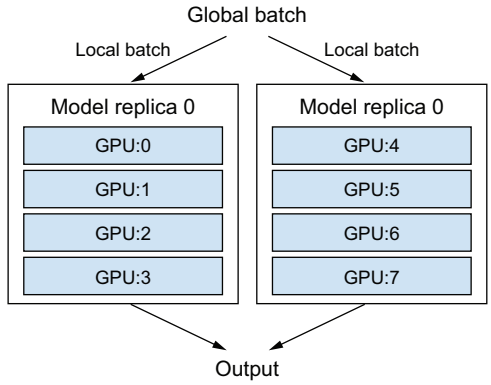


Figure 18.2 Distributing a model across eight devices: two model replicas, each handled by a group of four devices

GETTING YOUR HANDS ON TWO OR MORE GPUS

First, you need to get access to several GPUs. As of now, Google Colab only lets you use a single GPU, so you will need to do one of two things:

- Acquire two to eight GPUs, mount them on a single machine (it will require a beefy power supply), and install CUDA drivers, cuDNN, etc. For most people, this isn't the best option.
- Rent a multi-GPU virtual machine (VM) on Google Cloud, Azure, or AWS. You'll be able to use VM images with pre-installed drivers and software, and you'll have very little setup overhead. This is likely the best option for anyone who isn't training models 24/7.

We won't cover the details of how to spin up multi-GPU cloud VMs because such instructions would be relatively short-lived, and this information is readily available online.

USING DATA PARALLELISM WITH JAX

Using data parallelism with Keras and JAX is very simple: before building your model, just add the following line of code:

```
keras.distribution.set_distribution(keras.distribution.DataParallel())
```

That's it.

If you want more granular control, you can specify which devices you want to use. You can list available devices via

```
keras.distribution.list_devices()
```

It will return a list of strings—the names of your devices, such as `"gpu:0"`, `"gpu:1"`, and so on. You can then pass these to the `DataParallel` constructor:

```
keras.distribution.set_distribution(
    keras.distribution.DataParallel(["gpu:0", "gpu:1"])
)
```

In an ideal world, training on *N* GPUs would result in a speedup of factor *N*. In practice, however, distribution introduces some overhead—in particular, merging the weight deltas originating from different devices takes some time. The effective speedup you get is a function of the number of GPUs used:

- With two GPUs, the speedup stays close to 2×.
- With four, the speedup is around 3.8×.
- With eight, it's around 7.3×.

This assumes that you're using a large-enough global batch size to keep each GPU utilized at full capacity. If your batch size is too small, the local batch size won't be enough to keep your GPUs busy.

USING MODEL PARALLELISM WITH JAX

Keras also provides powerful tools for fully customizing how you want to do distributed training, including model parallel training and any mixture of data parallel and model parallel training you can imagine. Let's dive in.

THE DEVICEMESH API

First, you need to understand the concept of a *device mesh*. A device mesh is simply a grid of devices. Consider this example, with eight GPUs:

```
gpu:0   |   gpu:4
--------|---------
gpu:1   |   gpu:5
--------|---------
gpu:2   |   gpu:6
--------|---------
gpu:3   |   gpu:7
```

The big idea is to separate devices into groups, organized along axes. Typically, one axis will be responsible for data parallelism, and one axis will be responsible for model parallelism (like in figure 18.2, your devices form a grid, where the horizontal axis handles data parallelism and the vertical axis handles model parallelism).

A device mesh doesn't have to be 2D—it could be any shape you want. In practice, however, you will only ever see 1D and 2D meshes.

Let's make a 2 × 4 device mesh in Keras:

```
device_mesh = keras.distribution.DeviceMesh(
    shape=(2, 4),
    axis_names=["data", "model"],
)
```

We assume eight devices, organized as a 2 × 4 grid.

It's convenient to give your axes meaningful names!

Mind you, you can also explicitly specify the devices you want to use:

```
devices = [f"gpu:{i}" for i in range(8)]
device_mesh = keras.distribution.DeviceMesh(
    shape=(2, 4),
    axis_names=["data", "model"],
    devices=devices,
)
```

As you may have guessed from the `axis_names` argument, we intend to use the devices along axis 0 for data parallelism and the devices along axis 1 for model parallelism. Since there are two devices along axis 0 and four along axis 1, we'll split our model's computation across four GPUs, and we'll make two copies of our split model, running each copy on a different sub-batch of data in parallel.

Now that we have our mesh, we need to tell Keras how to split different pieces of computation across our devices. For that, we'll use the `LayoutMap` API.

The LayoutMap API

To specify where different bits of computation should take place, we use *variables* as our frame of reference. We will split or replicate variables across our devices, and we will let the compiler move all computation associated with that part of the variable to the corresponding device.

Consider a variable. Its shape is, let's say, `(32, 64)`. There are two things you could do with this variable:

- You could *replicate it* (copy it) across an axis of your mesh so each device along that axis sees the same value.
- You could *shard it* (split it) across an axis of your mesh—for instance, you could shard it into four chunks of shape `(32, 16)`—so that each device along that axis sees one different chunk.

Now, do note that our variable has two dimensions. Importantly, "sharding" or "replicating" are decisions that you can make independently for each dimension of the variable.

The API you will use to tell Keras about such decisions is the `LayoutMap` class. A `LayoutMap` is similar to a dictionary. It maps model variables (for instance, the kernel variable of the first dense layer in your model) to a bit of information about how that variable should be replicated or sharded over a device mesh. Specifically, it maps a *variable path* to a tuple that has as many entries as your variable has dimensions, where each entry specifies what to do with that variable dimension. It looks like this:

```
{
    "sequential/dense_1/kernel": (None, "model"),
    "sequential/dense_1/bias": ("model",),
    ...
}
```

None means "replicate the variable along this dimension."

"model" means "shard the variable along this dimension across the devices of the model axis of the device mesh."

This is the first time you encountered the concept of a *variable path*—it is simply a string identifier that looks like `"sequential/dense_1/kernel"`. It's a useful way to refer to a variable without keeping a handle on the actual variable instance.

Here's how you can print the paths for all variables in a model:

```
for v in model.variables:
    print(v.path)
```

On the example model from listing 18.4, here's what we get:

```
sequential/dense/kernel
sequential/dense/bias
sequential/dense_1/kernel
sequential/dense_1/bias
```

Now let's shard and replicate these variables. In the case of a simple model like this one, your go-to rule of thumb for variable sharding should be as follows:

- Shard the last dimension of the variable along the `"model"` mesh axis.
- Leave all other dimensions as replicated.

Simple enough, right? Like this:

```
layout_map = keras.distribution.LayoutMap(device_mesh)
layout_map["sequential/dense/kernel"] = (None, "model")
layout_map["sequential/dense/bias"] = ("model",)
layout_map["sequential/dense_1/kernel"] = (None, "model")
layout_map["sequential/dense_1/bias"] = ("model",)
```

Finally, we tell Keras to refer to this sharding layout when instantiating the variables by setting the distribution configuration like this:

```
model_parallel = keras.distribution.ModelParallel(
    layout_map=layout_map,
    batch_dim_name="data",
)
keras.distribution.set_distribution(model_parallel)
```

This argument tells Keras to use the mesh axis named "data" for data parallelism.

Once the distribution configuration is set, you can create your model and `fit()` it. No other part of your code changes—your model definition code is the same, and your training code is the same. That's true whether you're using built-in APIs like `fit()` and `evaluate()` or your own training logic. Assuming that you have the right `LayoutMap` for your variables, the little code snippets you just saw are enough to distribute

computation for any large language model training run—it scales to as many devices as you have available and arbitrary model sizes.

To check how your variables were sharded, you can inspect the `variable.value.sharding` property, like this:

```
>>> model.layers[0].kernel.value.sharding
NamedSharding(
    mesh=Mesh("data": 2, "model": 4),
    spec=PartitionSpec(None, "model")
)
```

You can even visualize it via the JAX utility `jax.debug.visualize_sharding`:

```
import jax

value = model.layers[0].kernel.value
jax.debug.visualize_sharding(value.shape, value.sharding)
```

tf.data performance tips

When doing distributed training, always provide your data as a `tf.data.Dataset` object to guarantee best performance (passing your data as NumPy arrays also works since those get converted to `Dataset` objects by `fit()`). You should also make sure to use data prefetching: before passing the dataset to `fit()`, call `dataset.prefetch(buffer_size)`. If you aren't sure what buffer size to pick, try the `dataset.prefetch(tf.data.AUTOTUNE)` option, which will pick a buffer size for you.

18.2.3 TPU training

Beyond just GPUs, there is generally a trend in the deep learning world toward moving workflows to increasingly specialized hardware designed specifically for deep learning workflows; such single-purpose chips are known as ASICs (application-specific integrated circuits). Various companies big and small are working on new chips, but today the most prominent effort along these lines is Google's Tensor Processing Unit (TPU), which is available on Google Cloud and via Google Colab.

Training on TPU does involve jumping through some hoops. But it's worth the extra work: TPUs are really, really fast. Training on a TPU v2 (available on Colab) will typically be 15× faster than training a NVIDIA P100 GPU. For most models, TPU training ends up being 3× more cost-effective than GPU training on average.

You can actually use TPU v2 for free in Colab. In the Colab menu, under the Runtime tab, in the Change Runtime Type option, you'll notice that you have access to a

TPU runtime in addition to the GPU runtime. For more serious training runs, Google Cloud also makes available TPU v3 through v5, which are even faster.

When running Keras code with the JAX backend on a TPU-enabled notebook, you don't need anything more than calling `keras.distribution.set_distribution(distribution)` with a `DataParallel` or `ModelParallel` distribution instance to start using your TPU cores. Make sure to call it before creating your model!

> **NOTE** Because TPUs can process batches of data extremely quickly, the speed at which you can read data from Google Cloud Storage (GCS) can easily become a bottleneck. If your dataset is small enough, you should keep it in the memory of the virtual machine. You can do so by calling `dataset.cache()` on your `tf.data.Dataset` instance. That way, the data will only be read from GCS once.

USING STEP FUSING TO IMPROVE TPU UTILIZATION

Because a TPU has a lot of compute power available, you need to train with very large batches to keep the TPU cores busy. For small models, the batch size required can get extraordinarily large—upward of 10,000 samples per batch. When working with enormous batches, you should make sure to increase your optimizer learning rate accordingly: you're going to be making fewer updates to your weights, but each update will be more accurate (since the gradients are computed using more data points); hence, you should move the weights by a greater magnitude with each update.

There is, however, a simple trick you can use to keep reasonably sized batches while maintaining full TPU utilization: *step fusing*. The idea is to run multiple steps of training during each TPU execution step. Basically, do more work in between two roundtrips from the virtual machine memory to the TPU. To do this, simply specify the `steps_per_execution` argument in `compile()`—for instance, `steps_per_execution=8` to run eight steps of training during each TPU execution. For small models that are underutilizing the TPU, this can result in a dramatic speedup:

```
model.compile(..., steps_per_execution=8)
```

18.3 Speeding up training and inference with lower-precision computation

What if I told you there's a simple technique you could use to speed up training and inference of almost any model by up to 2×, basically for free? It seems too good to be true, and yet, such a trick does exist. To understand how it works, first, we need to take a look at the notion of "precision" in computer science.

18.3.1 Understanding floating-point precision

Precision is to numbers what resolution is to images. Because computers can only process 1s and 0s, any number seen by a computer has to be encoded as a binary string.

For instance, you may be familiar with `uint8` integers, which are integers encoded on eight bits: `00000000` represents `0` in `uint8`, and `11111111` represents 255. To represent integers beyond 255, you'd need to add more bits—eight isn't enough. Most integers are stored on 32 bits, with which we can represent signed integers ranging from –2147483648 to 2147483647.

Floating-point numbers are the same. In mathematics, real numbers form a continuous axis: there's an infinite number of points in between any two numbers. You can always zoom in on the axis of reals. In computer science, this isn't true: there's only a finite number of intermediate points between 3 and 4, for instance. How many? Well, it depends on the *precision* you're working with: the number of bits you're using to store a number. You can only zoom up to a certain resolution.

There are three levels of precision you'd typically use:

- Half precision, or `float16`, where numbers are stored on 16 bits
- Single precision, or `float32`, where numbers are stored on 32 bits
- Double precision, or `float64`, where numbers are stored on 64 bits

You could even go up to `float8`, as you'll see in a bit.

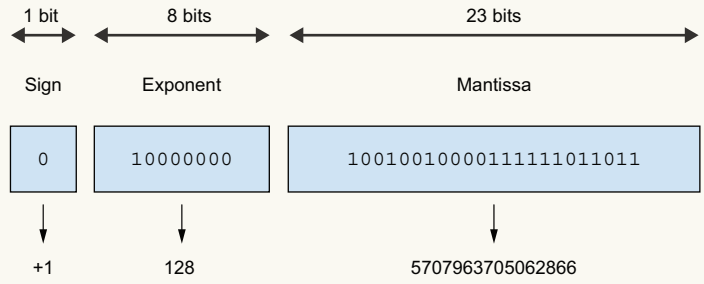
On floating-point encoding

A counterintuitive fact about floating-point numbers is that representable numbers are not uniformly distributed. Larger numbers have lower precision: there's the same number of representable values between `2 ** N` and `2 ** (N + 1)` as there is between 1 and 2, for any `N`.

That's because floating-point numbers are encoded in three parts—the sign, the significant value (called the *mantissa*), and the exponent in the form

```
{sign} * (2 ** ({exponent} - 127)) * 1.{mantissa}
```

For example, figure 18.3 demonstrates how you would encode the closest `float32` value approximating Pi:




```
value = +1 * (2 ** (128 - 127)) * 1.5707963705062866
value = 3.1415927410125732
```

Figure 18.3 The number Pi encoded in single precision via a sign bit, an integer exponent, and an integer mantissa

(continued)

For this reason, the numerical error incurred when converting a number to its floating-point representation can vary wildly depending on the exact value considered, and the error tends to get larger for numbers with a large absolute value.

The way to think about the resolution of floating-point numbers is in terms of the smallest distance between two arbitrary numbers that you'll be able to safely process. In single precision, that's around 1e-7. In double precision, that's around 1e-16. And in half precision, it's only 1e-3.

18.3.2 Float16 inference

Every model you've seen in this book so far has used single-precision numbers: it stored its state as `float32` weight variables and ran its computations on `float32` inputs. That's enough precision to run the forward and backwards pass of a model without losing any information—in particular when it comes to small gradient updates (recall that the typically learning rate is 1e-3, and it's pretty common to see weight updates on the order of 1e-6).

Modern GPUs and TPUs feature specialized hardware that can run 16-bit operations much faster and using less memory than equivalent 32-bit operations. By using these lower-precision operations whenever possible, you can speed up training on those devices by a significant factor. You can set the default floating point precision to `float16` in Keras via

```
import keras

keras.config.set_dtype_policy("float16")
```

Note that this should be done before you define your model. Doing this will net you a nice speedup for model inference, for instance, via `model.predict()`. You should expect a nearly 2× speed boost on GPU and TPU.

There's also an alternative to `float16` that works better on some devices, in particular TPUs: `bfloat16`. `bfloat16` is also a 16-bit precision floating-point type, but it differs from `float16` in its structure: it uses 8 exponent bits instead of 5, and 7 mantissa bits instead of 10 (see table 18.1). This means it can cover a much wider range of values, but it has a lower "resolution" over this range. Some devices are better optimized for `bfloat16` compared to `float16`, so it can be a good idea to try both before settling for the option that turns out to be the fastest.

Table 18.1 Difference between `float16` and `bfloat16`

dtype	float16	bfloat16
Exponent bits	5	8
Mantissa bits	10	7
Sign bits	1	1

18.3.3 Mixed-precision training

Setting your default float precision to 16 bits is a great way to speed up inference. Now, when it comes to training, there's a significant complication. The gradient descent process wouldn't run smoothly in `float16` or `bfloa16`, since we couldn't represent small gradient updates of around 1e-5 or 1e-6, which are quite common.

You can, however, use a hybrid approach: that's what *mixed-precision training* is about. The idea is to use 16-bit computation in places where precision isn't an issue, while working with 32-bit values in other places to maintain numerical stability—in particular, when handling gradients and variable updates. By maintaining the precision-sensitive parts of the model in full precision, you can get most of the speed benefits of 16-bit computation without meaningfully impacting model quality.

You can turn on mixed precision like this:

```
import keras

keras.config.set_dtype_policy("mixed_float16")
```

Typically, most of the forward pass of the model will be done in `float16` (with the exception of numerically unstable operations like softmax), while the weights of the model will be stored and updated in `float32`. Your `float16` gradients will be cast to `float32` before updating the `float32` variables.

Keras layers have a `variable_dtype` and a `compute_dtype` attribute. By default, both of these are set to `float32`. When you turn on mixed precision, the `compute_dtype` of most layers switches to `float16`. As a result, those layer will cast their inputs to `float16` and will perform their computation in `float16` (using half-precision copies of the weights). However, since their `variable_dtype` is still `float32`, their weights will be able to receive accurate `float32` updates from the optimizer, as opposed to half-precision updates.

Some operations may be numerically unstable in `float16` (in particular, softmax and crossentropy). If you need to opt out of mixed precision for a specific layer, just pass the argument `dtype="float32"` to the constructor of this layer.

18.3.4 Using loss scaling with mixed precision

During training, gradients can become very small. When using mixed precision, your gradients remain in `float16` (same as the forward pass). As a result, the limited range

of representable numbers can cause small gradients to be rounded down to zero. This prevents the model from learning effectively.

Gradient values are proportional to the loss value, so to encourage gradients to be larger, a simple trick is to multiply the loss by a large scalar factor. Your gradients will then be much less likely to get rounded to zero.

Keras makes this easy. If you want to use a fixed loss scaling factor, you can simply pass a `loss_scale_factor` argument to your optimizer like this:

```
optimizer = keras.optimizers.Adam(learning_rate=1e-3, loss_scale_factor=10)
```

If you would like for the optimizer to automatically figure out the right scaling factor, you can also use the `LossScaleOptimizer` wrapper:

```
optimizer = keras.optimizers.LossScaleOptimizer(
    keras.optimizers.Adam(learning_rate=1e-3)
)
```

Using `LossScaleOptimizer` is usually your best option: the right scaling value can change over the course of training!

18.3.5 Beyond mixed precision: float8 training

If running your forward pass in 16-bit precision yields such neat performance benefits, you might want to ask: Could we go even lower? What about 8-bit precision? Four bits, maybe? Two bits? The answer is, it's complicated.

Mixed precision training using `float16` in the forward pass is that last level of precision that "just works"—`float16` precision has enough bits to represent all intermediate tensors (except for gradient updates, which is why we use `float32` for those). This is no longer true if you go down to `float8` precision: you are simply losing too much information. It is still possible to use `float8` in some computations, but this requires you to make considerable modifications to your forward pass. You will *not* be able to simply set your `compute_dtype` to `float8` and run.

The Keras framework provides a built-in implementation for `float8` training. Because it specifically targets Transformer use cases, it only covers a restricted set of layers: `Dense`, `EinsumDense` (the version of `Dense` that is used by the `MultiHeadAttention` layer), and `Embedding` layers. The way it works is not simple—it keeps track of past activation values to rescale activations at each step so as to utilize the full range of values representable in `float8`. It also needs to override part of the backward pass to do the same with gradient values.

Importantly, this added overhead has a computational cost. If your model is too small or if your GPU isn't powerful enough, that cost will exceed the benefits of doing certain operations in `float8`, and you will see a slowdown instead of a speedup. `float8` training

is only viable for very large models (typically over 5B parameters) and large, recent GPUs such as the NVDIA H100. `float8` is rarely used in practice, except in foundation model training runs.

18.3.6 Faster inference with quantization

Running inference in `float16`—or even `float8`—will result in a nice speedup for your models. But there's also another trick you can use: *`int8` quantization*. The big idea is to take an already trained model with weights in `float32` and convert these weights to a lower-precision dtype (typically `int8`) while preserving the numerical correctness of the forward pass as much as possible.

If you want to implement quantization from scratch, the math is simple: the general idea is to scale all `matmul` input tensors by a certain factor so that their coefficients fit in the range representable with `int8`, which is `[-127, 127]`—a total of 256 possible values. After scaling the inputs, you cast them to `int8` and perform the `matmul` operation in `int8` precision, which should be quite a bit faster than `float16`. Finally, you cast the output back to `float32`, and you divide it by the product of the input scaling factors. Since `matmul` is a linear operation, this final unscaling cancels out the initial scaling, and you should get the same output as if you used the original values—any loss of accuracy only comes from the value rounding that happens when you cast the inputs to `int8`.

Let's make this concrete with an example. Let's say you want to perform `matmul(x, kernel)`, with the following values:

```
from keras import ops

x = ops.array([[0.1, 0.9], [1.2, -0.8]])
kernel = ops.array([[-0.1, -2.2], [1.1, 0.7]])
```

If you were to naively cast these values to `int8` without scaling first, that would be very destructive—for instance, your `x` would become `[[0, 0], [1, 0]]`. So let's apply the "abs-max" scaling scheme, which spreads out the values of each tensor across the `[-127, 127]` range:

```
def abs_max_quantize(value):
    abs_max = ops.max(ops.abs(value), keepdims=True)
    scale = ops.divide(127, abs_max + 1e-7)
    scaled_value = value * scale
    scaled_value = ops.clip(ops.round(scaled_value), -127, 127)
```

Scales the value

Scale is max of int range divided by max of tensor (1e-7 is to avoid dividing by 0).

Max of absolute value of the tensor

Rounding and clipping first is more accurate than directly casting.

```
    scaled_value = ops.cast(scaled_value, dtype="int8")
    return scaled_value, scale

int_x, x_scale = abs_max_quantize(x)
int_kernel, kernel_scale = abs_max_quantize(kernel)
```

<- Casts to int8

Now we can perform a faster `matmul` and unscale the output:

```
int_y = ops.matmul(int_x, int_kernel)
y = ops.cast(int_y, dtype="float32") / (x_scale * kernel_scale)
```

How accurate is it? Let's compare our `y` with the output of the `float32` `matmul`:

```
>>> y
array([[ 0.9843736,  0.3933239],
       [-1.0151455, -3.1965137]])
>>> ops.matmul(x, kernel)
array([[ 0.98      ,  0.40999997],
       [-1.        , -3.2       ]])
```

Pretty accurate! For a large `matmul`, doing this will save you a lot of compute, since `int8` computation can be considerably faster than even `float16` computation, and you only had to add fairly fast elementwise ops to the computation graphs—`abs`, `max`, `clip`, `cast`, `divide`, `multiply`.

Now, of course, I don't expect you to ever implement quantization by hand—that would be tremendously impractical. Similarly to `float8`, `int8` quantization is built directly into specific Keras layers: `Dense`, `EinsumDense`, and `Embedding`. This unlocks `int8` inference support for any Transformer-based model. Here's how to use it with any Keras model that includes such layers:

```
model = ...
model.quantize("int8")
predictions = model.predict(...)
```

<- Instantiates a model (or any quantizable layer)

<- Boom!

<- Now predict() and call() will run (partially) in int8!

Summary

- You can use hyperparameter tuning and KerasTuner to automate the tedium out of finding the best model configuration. But be mindful of validation-set overfitting!
- An ensemble of diverse models can often significantly improve the quality of your predictions.
- To further scale your workflows, you can use *data parallelism* to train a model on multiple devices, as long as the model is small enough to fit on a single device.
- For larger models, you can also use *model parallelism* to split your model's variables and computation across several devices.
- You can speed up model training on GPUs or TPUs by turning on mixed precision—you'll generally get a nice speed boost at virtually no cost.
- You can also speed up inference by using `float16` precision or even `int8` quantization.

19 The future of AI

This chapter covers

- The limitations of deep learning
- The nature of intelligence
- What's missing from current approaches
- What the future might look like

To use a tool appropriately, you should not only understand what it *can* do but also be aware of what it *can't* do. I'm going to present an overview of some key limitations of deep learning. Then, I'll offer some speculative thoughts about the future evolution of AI and what it would take to get to human-level general intelligence. This should be especially interesting to you if you'd like to get into fundamental research.

19.1 The limitations of deep learning

There are infinitely many things you can do with deep learning. But deep learning can't do *everything*. To use a tool well, you should be aware of its limitations, not just its strengths. So where does deep learning fall short?

19.1.1 Deep learning models struggle to adapt to novelty

Deep learning models are big, parametric curves fitted to large datasets. That's the source of their power—they're easy to train, and they scale really well, both in terms of model size and dataset size. But that's also a source of significant weaknesses. Curve fitting has inherent limitations.

First and foremost, a parametric curve is only capable of information storage—it's a kind of *database*. Recall our discussion of Transformers as an "interpolative database" from chapter 15? Second, crucially, this database is *static*. The model's parameters are determined during a distinct "training time" phase. Afterward, these parameters are frozen, and this fixed version is used during "inference time" for making predictions on new data.

The only thing you can do with a static database is information retrieval. And that's exactly what deep learning models excel at: recognizing or generating patterns highly similar to those encountered during training. The flip side is that they are inherently poor at *adaptation*. The database is backward-looking—it fits past data but can't handle a changing future. At inference time, you'd better hope that the situations the model faces are part of the training data distribution, because otherwise, the model will break down. A model trained on ImageNet will classify a leopard-print sofa as an actual leopard, for instance—sofas were not part of its training data.

This also applies to the largest of generative models. In recent years, the rise of large language models (LLMs) and their application to programming assistance and reasoning-like problems has provided extensive empirical proof of this. Despite frequent claims that LLMs can perform *in-context learning* to pick up new skills from just a few examples, there is overwhelming evidence that what they're actually doing is fetching vector functions they've memorized during training and reapplying them to the task at hand. By learning to do next-token prediction across a web-sized text dataset, an LLM has collected millions of potentially useful mini text-processing programs, and it can easily be prompted into reusing them on a new problem. But show it something that has no direct equivalent in its training data, and it's helpless.

Take a look at the puzzle in figure 19.1. Did you figure out the solution? Good. It's not very hard, is it? But today, no state-of-the-art LLM or vision-language model can do this because this particular problem doesn't directly map to anything they've seen at training time—even after having been trained on the entire internet and then some. An LLM's ability to solve a given problem has nothing to do with problem complexity, and everything to do with *familiarity*—they will break their teeth on any sufficiently novel problem, no matter how simple.

This failure mode even applies to tiny variations of a pattern that an LLM encountered many times in its training data. For instance, for a few months after the release of ChatGPT, if you asked it, "What's heavier, 10 kilos of steel or one kilo of feathers?," it would answer that they weigh the same. That's because the question "What's heavier, one kilo of steel or one kilo of feathers?" is found many times on the internet—as a trick question. The right answer, of course, is that they both weigh the same, so the GPT

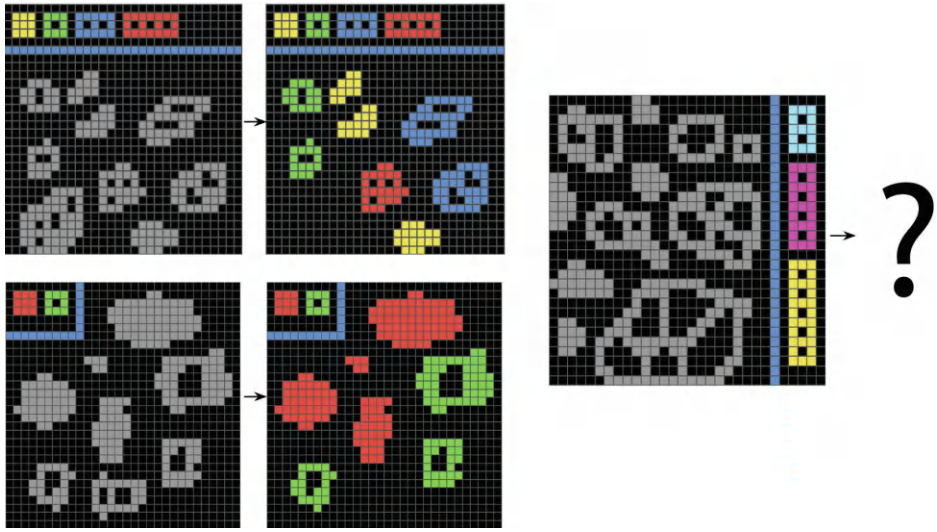


Figure 19.1 An easy yet novel puzzle

model would just repeat the answer it had memorized without paying any attention to the actual numbers in the query, or what the query really *meant*. Similarly, LLMs struggle to adapt to variations of the Monty Hall problem (see figure 19.2) and will tend to always output the canonical answer to the puzzle, which they've seen many times during training, regardless of whether it makes sense in context.

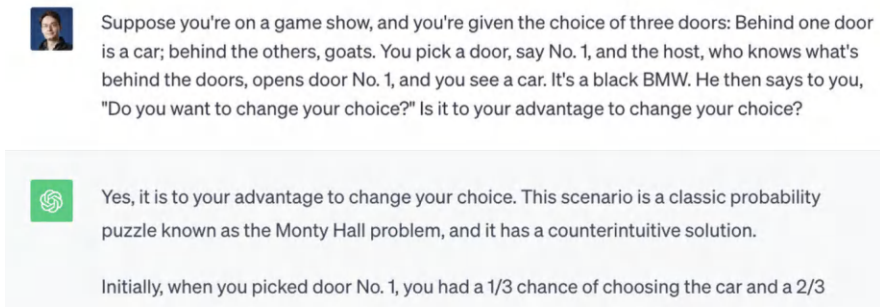


Figure 19.2 A variation of the Monty Hall problem

To note, these specific prompts were patched later on by special-casing them. Today, there are over 25,000 people who are employed full time to provide training data for LLMs by reviewing failure cases and suggesting better answers. LLM maintenance is a constant game of whack-a-mole where failing prompts are patched one at a time,

without addressing the more general underlying issue. Even already patched prompts will still fail if you make small changes to them!

19.1.2 Deep learning models are highly sensitive to phrasing and other distractors

A closely related problem is the extreme sensitivity of deep learning models to how their input is presented. For instance, image models are affected by *adversarial examples*, which are samples fed to a deep learning network that are designed to trick the model into misclassifying them. You're already aware that it's possible to do gradient ascent in input space to generate inputs that maximize the activation of some ConvNet filter—this is the basis of the filter visualization technique introduced in chapter 10.

Similarly, through gradient ascent, you can slightly modify an image to maximize the class prediction for a given class. By taking a picture of a panda and adding to it a gibbon gradient, we can get a neural network to classify the panda as a gibbon (see figure 19.3). This evidences both the brittleness of these models and the deep difference between their input-to-output mapping and our human perception.

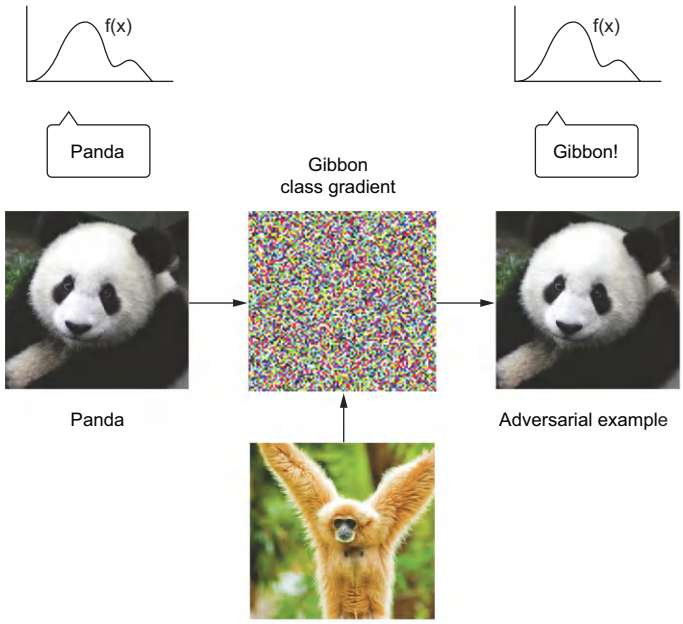


Figure 19.3 An adversarial example: imperceptible changes in an image can upend a model's classification of the image.

Similarly, LLMs suffer from an extremely high sensitivity to minor details in their prompts. Innocuous prompt modifications, such as changing place and people's

names in a text paragraph or variable names in a block of code, can significantly degrade LLM performance. Consider the well-known *Alice in Wonderland* riddle:[1]

> *"Alice has N brothers and she also has M sisters. How many sisters does Alice's brother have?"*

The answer, of course, is $M + 1$ (Alice's sisters plus Alice herself). For an LLM, asking the question with values commonly found in online instances of the riddle (like $N = 3$ and $M = 2$) will generally result in the correct answer, but try tweaking the values of M and N, and you will quickly get incorrect answers.

This oversensitivity to phrasing has given rise to the concept of *prompt engineering*. Prompt engineering is the art of formulating LLM prompts in a way that maximizes performance on a task. For instance, it turns out that adding the instruction "Please think step by step" to a prompt that involves reasoning can significantly boost performance. The term *prompt engineering* is a very optimistic framing of the underlying issue: "Your models are better than you know! You just need to use them right!" A more negative framing would be to point out that for any query that seems to work, there's a range of minor changes that have the potential to tank performance. To what extent do LLMs understand something if you can break their understanding with simple rewordings?

What's behind this phenomenon is that an LLM is a big parametric curve—a medium for storing knowledge and programs where you can interpolate between any two objects to produce infinitely many intermediate objects. Your prompt is a way to address a particular location of the database: if you ask, "How do you sort a list in Python? Answer like a pirate," that's a kind of database lookup, where you first retrieve a piece of knowledge (how to sort a list in Python) and then retrieve and execute a style transfer program ("Answer like a pirate").

Since the knowledge and programs indexed by the LLM are interpolative, you can *move around in latent space* to explore nearby locations. A slightly different prompt, like "Explain Python list sorting, but answer like a buccaneer" would still have pointed to a very similar location in the database, resulting in an answer that would be pretty close but not quite identical. There are thousands of variations you could have used, each resulting in a similar yet slightly different answer. And that's why prompt engineering is needed. There is no a priori reason for your first, naive prompt to be optimal for your task. The LLM is not going to understand what you meant and then perform it in the best possible way—it's merely going to fetch the program that your prompt points to, among many possible locations you could have landed on.

Prompt engineering is the process of searching through latent space to find the lookup query that seems to perform best on your target task by trial and error. It's no different from trying different keywords when doing a Google search. If LLMs actually understood what you asked them, there would be no need for this search process, since the amount of information conveyed about your target task does not change whether

[1] See also Marianna Nezhurina, Lucia Cipolina-Kun, Mehdi Cherti, and Jenia Jitsev, "Alice in Wonderland: Simple Tasks Showing Complete Reasoning Breakdown in State-Of-the-Art Large Language Models," arXiv, https://arxiv.org/abs/2406.02061.

your prompt uses the word "rewrite" instead of "rephrase" or whether you prefix your prompt with "Think step by step." Never assume that the LLM "gets it" the first time—keep in mind that your prompt is but an address in an infinite ocean of programs, all memorized as a by-product of learning to complete an enormous amount of token sequences.

19.1.3 Deep learning models struggle to learn generalizable programs

The problem with deep learning models isn't just that they're limited to blindly reapplying patterns they've memorized at training time or that they're highly sensitive to how their input is presented. Even if you just need to query and apply a well-known program, and you know exactly how to address this program in latent space, you still face a major issue: the programs memorized by deep learning models often don't generalize well. They will work for some input values and fail for some other input values. This is especially true for programs that encode any kind of discrete logic.

Consider the problem of adding two numbers, represented as character sequences—like "4 3 5 7 + 8 9 3 6." Try training a Transformer on hundreds of thousands of such digit pairs: you will reach a very high accuracy. Very high, but not 100%—you will keep regularly seeing incorrect answers, because the Transformer doesn't manage to encode the exact addition algorithm (you know, the one you learned in primary school). It is instead guessing the output by interpolating between the data points it has seen at training time.

This applies to state-of-the-art LLMs, too—at least those that weren't explicitly hardcoded to execute snippets like "4357 + 8936" in Python to provide the right answer. They've seen enough examples of digit addition that they can add numbers, but they only have about 70% accuracy—quite underwhelming. Further, their accuracy is strongly dependent on *which* digits are being added, with more common digits leading to higher accuracy.

The reason why a deep learning model does not end up learning an exact addition algorithm even after seeing millions of examples is that it is just *a static chain of simple, continuous geometric transformations* mapping one vector space into another. That is a good fit for perceptual pattern recognition, but it's a very poor fit for encoding any sort of step-by-step discrete logic, such as concepts like place value or carrying over. All it can do is map one data manifold X into another manifold Y, assuming the existence of a learnable continuous transform from X to Y. A deep learning model can be interpreted as a kind of program, but inversely, *most programs can't be expressed as deep-learning models*. For most tasks, either there exists no corresponding neural network of reasonable size that solves the task or, even if one exists, it may not be *learnable*: the corresponding geometric transform may be far too complex, or there may not be appropriate data available to learn it.

19.1.4 The risk of anthropomorphizing machine-learning models

Our own understanding of images, sounds, and language is grounded in our sensorimotor experience as humans. Machine learning models have no access to such

experiences and thus can't understand their inputs in a human-relatable way. By feeding a large number of training examples into our models, we get them to learn a geometric transform that maps data to human concepts on a specific set of examples, but this mapping is a simplistic sketch of the original model in our minds—the one developed from our experience as embodied agents. It's like a dim image in a mirror (see figure 19.4). The models you create will take any shortcut available to fit their training data.

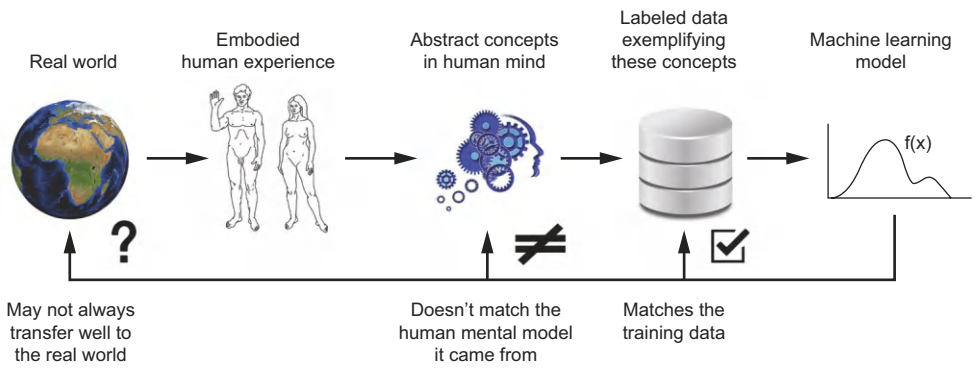


Figure 19.4 Current machine-learning models: like a dim image in a mirror

One real risk with contemporary AI is misinterpreting what deep learning models do and overestimating their abilities. A fundamental feature of humans is our *theory of mind*: our tendency to project intentions, beliefs, and knowledge on the things around us. Drawing a smiley face on a rock suddenly makes it "happy"—in our minds. Applied to deep learning, this means that when we train models capable of using language, we're led to believe that the model "understands" the contents of the word sequences they generate just the way we do. Then we're surprised when any slight departure from the patterns present in the training data causes the model to generate completely absurd answers.

As a machine learning practitioner, always be mindful of this and never fall into the trap of believing that neural networks understand the task they perform—they don't, at least not in a way that would make sense to us. They were trained on a different, far narrower task than the one we wanted to teach them: that of mapping training inputs to training targets, point by point. Show them anything that deviates from their training data, and they will break in absurd ways.

19.2 Scale isn't all you need

Could we just keep scaling our models to overcome the limitations of deep learning? Is *scale* all we need? This has long been the prevailing narrative in the field, one that was especially prominent in early 2023, during peak LLM hype. Back then, GPT-4 had just

been released, and it was essentially a scaled-up version of GPT-3: more parameters, more training data. Its significantly improved performance seemed to suggest that you could just keep going—that there could be a GPT-5 that would simply be more of the same and from which artificial general intelligence (AGI) would spontaneously emerge.

Proponents of this view would point to "scaling laws" as evidence. Scaling laws are an empirical relationship observed between the size of a deep learning model (as well as the size of its training dataset) and its performance on specific tasks. They suggest that increasing the size of a model reliably leads to better performance in a predictable manner. But the key thing that scaling law enthusiasts are missing is that the benchmarks they're using to measure "performance" are effectively memorization tests, the kind we like to give university students. LLMs perform well on these tests by memorizing the answers, and naturally, cramming more questions and more answers into the models improves their performance accordingly.

The reality is that scaling up our models hasn't led to any progress on the issues I've listed so far in these pages—inability to adapt to novelty, oversensitivity to phrasing, and the inability to infer generalizable programs for reasoning problems—because these issues are inherent to curve fitting, the paradigm of deep learning. I started pointing out these problems in 2017, and we're still struggling with them today—with models that are now four or five orders of magnitude larger and more knowledgeable. We have not made any progress on these problems because *the models we're using are still the same.* They've been the same for over seven years—they're still parametric curves fitted to a dataset via gradient descent, and they're still using the Transformer architecture.

Scaling up current deep learning techniques by stacking more layers and using more training data won't solve the fundamental problems of deep learning:

- Deep-learning models are limited to using interpolative programs they memorize at training time. They are not able, on their own, to synthesize brand-new programs at inference time to adapt to substantially novel situations.
- Even within known situations, these interpolative programs suffer from generalization issues, which lead to oversensitivity to phrasing and confounder features.
- Deep learning models are limited in what they can represent, and most of the programs you may wish to learn can't be expressed as a continuous geometric morphing of a data manifold. This is true in particular of algorithmic reasoning tasks.

Let's take a closer look at what separates biological intelligence from the deep learning approach.

19.2.1 Automatons vs. intelligent agents

There are fundamental differences between the straightforward geometric morphing from input to output that deep learning models do and the way humans think and learn. It isn't just the fact that humans learn by themselves from embodied experience instead of being presented with explicit training examples. The human brain is an entirely different beast compared to a differentiable parametric function.

Let's zoom out a little bit and ask, What's the purpose of intelligence? Why did it arise in the first place? We can only speculate, but we can make fairly informed speculations. We can start by looking at brains—the organ that produces intelligence. Brains are an evolutionary adaptation—a mechanism developed incrementally over hundreds of millions of years, via random trial and error guided by natural selection, that dramatically expanded the ability of organisms to adapt to their environment. Brains originally appeared more than half a billion years ago as a way to *store and execute behavioral programs*. Behavioral programs are just sets of instructions that make an organism reactive to its environment: "If this happens, then do that." They link the organism's sensory inputs to its motor controls. In the beginning, brains would have served to hardcode behavioral programs (as neural connectivity patterns), which would allow an organism to react appropriately to its sensory input. This is the way insect brains still work—flies, ants, *C. elegans* (see figure 19.5), etc. Because the original "source code" of these programs was DNA, which would get decoded as neural connectivity patterns, evolution was suddenly able to *search over behavior space* in a largely unbounded way—a major evolutionary shift.

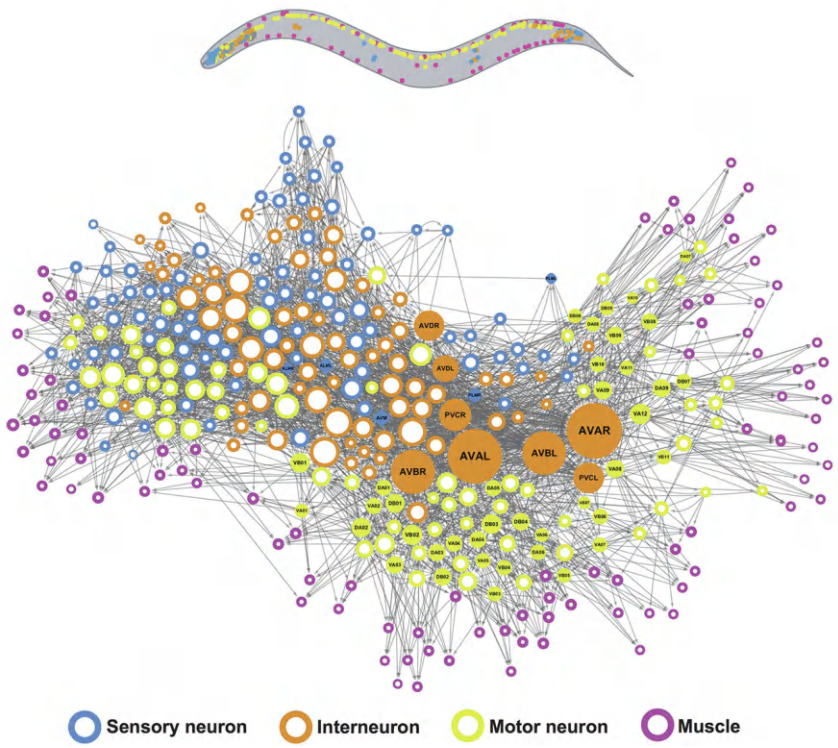


Figure 19.5 The brain network of the *C. elegans* worm: a behavioral automaton "programmed" by natural evolution. Figure created by Emma Towlson (from "Network control principles predict neuron function in the Caenorhabditis elegans connectome," Yan et al., *Nature*, Oct. 2017).

Evolution was the programmer, and brains were computers carefully executing the code evolution gave them. Because neural connectivity is a very general computing substrate, the sensorimotor space of all brain-enabled species could suddenly start undergoing a dramatic expansion. Eyes, ears, mandibles, 4 legs, 24 legs—as long as you have a brain, evolution will kindly figure out for you behavioral programs that make good use of these. Brains can handle any modality, or combination of modalities, you throw at them.

Now, mind you, these early brains weren't exactly intelligent per se. They were very much *automatons*: they would merely execute behavioral programs hardcoded in the organism's DNA. They could only be described as intelligent in the same sense that a thermostat is "intelligent." Or a list-sorting program. Or a trained deep neural network (of the artificial kind). This is an important distinction, so let's look at it carefully: What's the difference between automatons and actual intelligent agents?

19.2.2 Local generalization vs. extreme generalization

The field of AI has long suffered from conflating the notions of *intelligence* and *automation*. An automation system (or automaton) is static, crafted to accomplish specific things in a specific context—"If this, then that"—while an intelligent agent can adapt on the fly to novel, unexpected situations. When an automaton is exposed to something that doesn't match what it was "programmed" to do (whether we're talking about human-written programs, evolution-generated programs, or the implicit programming process of fitting a model on a training dataset), it will fail.

Meanwhile, intelligent agents, like us humans, will use their fluid intelligence to find a way forward. How do you tell the difference between a student who has memorized the past three years of exam questions but has no understanding of the subject and one who actually understands the material? You give them a brand-new problem.

Humans are capable of far more than mapping immediate stimuli to immediate responses as a deep network or an insect would. We can assemble on-the-fly complex, abstract models of our current situation, of ourselves, and of other people, and can use these models to anticipate different possible futures and perform long-term planning. We can quickly adapt to unexpected situations and pick up new skills after just a little bit of practice.

This ability to use *abstraction* and *reasoning* to handle experiences we weren't prepared for is the defining characteristic of human cognition. I call it *extreme generalization*: an ability to adapt to novel, never-before-experienced situations using little data or even no new data at all. This capability is key to the intelligence displayed by humans and advanced animals.

This stands in sharp contrast with what automaton-like systems do. A very rigid automaton wouldn't feature any generalization at all; it would be incapable of handling anything that it wasn't precisely told about in advance. A Python dict, or a basic question-answering program implemented as hardcoded if-then-else statements would fall into this category. Deep nets do slightly better: they can successfully process

inputs that deviate a bit from what they're familiar with, which is precisely what makes them useful. Our dogs-versus-cats model from chapter 8 could classify cat or dog pictures it had not seen before, as long as they were close enough to what it was trained on. However, deep nets are limited to what I call *local generalization* (see figure 19.6): the mapping from inputs to outputs performed by a deep net quickly stops making sense as inputs start deviating from what the net saw at training time. Deep nets can only generalize to *known unknowns,* to factors of variation that were anticipated during model development and that are extensively featured in the training data, such as different camera angles or lighting conditions for pet pictures. That's because deep nets generalize via interpolation on a manifold (remember chapter 5): any factor of variation in their input space needs to be captured by the manifold they learn. That's why basic data augmentation is so helpful in improving deep net generalization. Unlike humans, these models have no ability to improvise in the face of situations for which little or no data is available.

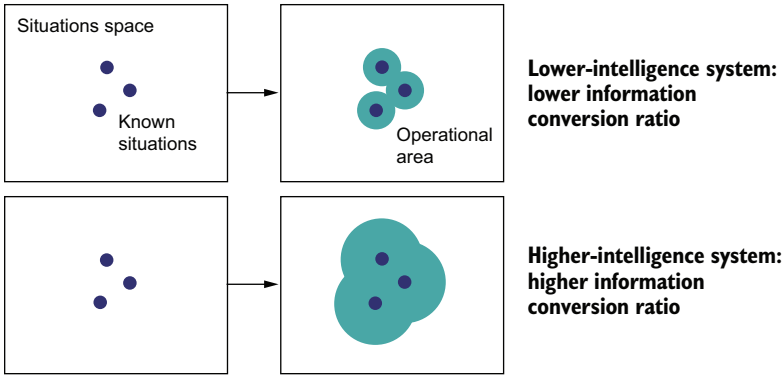


Figure 19.6 Local generalization vs. extreme generalization

Consider, for instance, the problem of learning the appropriate launch parameters to get a rocket to land on the moon. If you used a deep net for this task and trained it using supervised learning or reinforcement learning, you'd have to feed it tens of thousands or even millions of launch trials: you'd need to expose it to a *dense sampling* of the input space for it to learn a reliable mapping from input space to output space. In contrast, as humans, we can use our power of abstraction to come up with physical models—rocket science—and derive an exact solution that will land the rocket on the moon in one or a few trials. Similarly, if you developed a deep net controlling a human body and you wanted it to learn to safely navigate a city without getting hit by cars, the net would have to die many thousands of times in various situations until it could infer that cars are dangerous and develop appropriate avoidance behaviors. Dropped into a new city, the net would have to relearn most of what it knows. On the other

hand, humans are able to learn safe behaviors without having to die even once—again, thanks to our power of abstract modeling of novel situations.

19.2.3 The purpose of intelligence

This distinction between highly adaptable intelligent agents and rigid automatons leads us back to brain evolution. Why did brains—originally a mere medium for natural evolution to develop behavioral automatons—eventually turn intelligent? Like every significant evolutionary milestone, it happened because natural selection constraints encouraged it to happen.

Brains are responsible for behavior generation. If the set of situations an organism had to face was mostly static and known in advance, behavior generation would be an easy problem: evolution would just figure out the correct behaviors via random trial and error and hardcode them into the organism's DNA. This first stage of brain evolution—brains as automatons—would already be optimal. However, crucially, as organism complexity—and alongside it, environmental complexity—kept increasing, the situations animals had to deal with became much more dynamic and more unpredictable. A day in your life, if you look closely, is unlike any day you've ever experienced and unlike any day ever experienced by any of your evolutionary ancestors. You need to be able to face unknown and surprising situations constantly. There is no way for evolution to find and hardcode as DNA the sequence of behaviors you've been executing to successfully navigate your day since you woke up a few hours ago. It has to be generated on the fly every day.

The brain, as a good behavior-generation engine, simply adapted to fit this need. It optimized for adaptability and generality themselves, rather than merely optimizing for fitness to a fixed set of situations. This shift likely occurred multiple times throughout evolutionary history, resulting in highly intelligent animals in very distant evolutionary branches—apes, octopuses, ravens, and more. Intelligence is an answer to challenges presented by complex, dynamic ecosystems.

That's the nature of intelligence: it is the ability to efficiently use the information at your disposal to produce successful behavior in the face of an uncertain, ever-changing future. What Descartes calls "understanding" is the key to this remarkable capability: the power to mine your past experience to develop modular, reusable abstractions that can be quickly repurposed to handle novel situations and achieve extreme generalization.

19.2.4 Climbing the spectrum of generalization

As a crude caricature, you could summarize the evolutionary history of biological intelligence as a slow climb up the *spectrum of generalization*. It started with automaton-like brains that could only perform local generalization. Over time, evolution started producing organisms capable of increasingly broader generalization that could thrive in ever-more complex and variable environments. Eventually, in the past few million years—an instant in evolutionary terms—certain hominin species started trending toward an implementation of biological intelligence capable of extreme

generalization, precipitating the start of the Anthropocene and forever changing the history of life on Earth.

The progress of AI over the past 70 years bears striking similarities to this evolution. Early AI systems were pure automatons, like the ELIZA chat program from the 1960s, or SHRDLU,[2] a 1970 AI capable of manipulating simple objects from natural language commands. In the 1990s and 2000s, we saw the rise of machine learning systems capable of local generalization that could deal with some level of uncertainty and novelty. In the 2010s, deep learning further expanded the local generalization power of these systems by enabling engineers to use much larger datasets and much more expressive models.

Today, we may be on the cusp of the next evolutionary step. We are moving toward systems that achieve *broad generalization*, which I define as the ability to deal with *unknown unknowns* within a single broad domain of tasks (including situations the system was not trained to handle and that its creators could not have anticipated). Examples are a self-driving car capable of safely dealing with any situation you throw at it or a domestic robot that could pass the "Woz test of intelligence"—entering a random kitchen and making a cup of coffee.[3] By combining deep learning and painstakingly handcrafted abstract models of the world, we're already making visible progress toward these goals.

However, the deep learning paradigm has remained limited to cognitive automation: The "intelligence" label in "artificial intelligence" has been a category error. It would be more accurate to call our field "artificial cognition," with "cognitive automation" and "artificial intelligence" being two nearly independent subfields within it. In this subdivision, AI would be a greenfield where almost everything remains to be discovered.

Now, I don't mean to diminish the achievements of deep learning. Cognitive automation is incredibly useful, and the way deep learning models are capable of automating tasks from exposure to data alone represents an especially powerful form of cognitive automation, far more practical and versatile than explicit programming. Doing this well is a game changer for essentially every industry. But it's still a long way from human (or animal) intelligence. Our models, so far, can only perform local generalization: they map space X to space Y via a smooth geometric transform learned from a dense sampling of X-to-Y data points, and any disruption within spaces X or Y invalidates this mapping. They can only generalize to new situations that stay similar to past data, whereas human cognition is capable of extreme generalization, quickly adapting to radically novel situations and planning for long-term future situations.

19.3 How to build intelligence

So far, you've learned that there's a lot more to intelligence than the sort of latent manifold interpolation that deep learning does. But what, then, do we need to start building real intelligence? What are the core pieces that are currently eluding us?

---

[2] Terry Winograd, "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," 1971.

[3] Michael Shick, "Wozniak: Could a Computer Make a Cup of Coffee?" Fast Company, March 2010.

19.3.1 The kaleidoscope hypothesis

Intelligence is the ability to use your past experience (and innate prior knowledge) to face novel, unexpected future situations. Now, if the future you had to face was *truly novel*—sharing no common ground with anything you've seen before—you'd be unable to react to it, no matter how intelligent you are.

Intelligence works because nothing is ever truly without precedent. When we encounter something new, we're able to make sense of it by drawing analogies to our past experience and articulating it in terms of the abstract concepts we've collected over time. A person from the 17th century seeing a jet plane for the first time might describe it as a large, loud metal bird that doesn't flap its wings. A car? That's a horseless carriage. If you're trying to teach physics to a grade schooler, you can explain how electricity is like water in a pipe or how spacetime is like a rubber sheet getting distorted by heavy objects.

Besides such clear-cut, explicit analogies, we're constantly making smaller, implicit analogies—every second, with every thought. Analogies are how we navigate life. Shopping at a new supermarket? You'll find your way by relating it to similar stores you've been to. Talking to someone new? They'll remind you of a few people you've met before. Even seemingly random patterns, like the shape of clouds, instantly evoke in us vivid images—an elephant, a ship, a fish.

These analogies aren't just in our minds, either: physical reality itself is full of isomorphisms. Electromagnetism is analogous to gravity. Animals are all structurally similar to each other, due to shared origins. Silica crystals are similar to ice crystals. And so on.

I call this the *kaleidoscope hypothesis*: our experience of the world seems to feature incredible complexity and never-ending novelty, but everything in this sea of complexity is similar to everything else. The number of *unique atoms of meaning* that you need to describe the universe you live in is relatively small, and everything around you is a recombination of these atoms: a few seeds, endless variation, much like what goes on inside a kaleidoscope, where a few glass beads are reflected by a system of mirrors to produce rich, seemingly endless patterns (see figure 19.7).

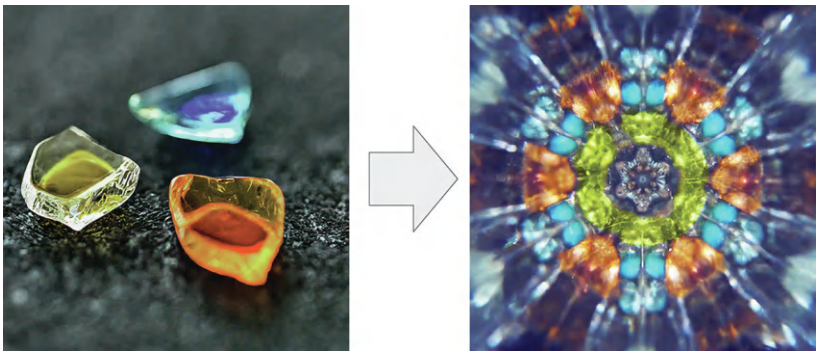


Figure 19.7 A kaleidoscope produces rich (yet repetitive) patterns from just a few beads of colored glass.

19.3.2 The essence of intelligence: Abstraction acquisition and recombination

Intelligence is the ability to mine your experience to identify these atoms of meaning that can seemingly be reused across many different situations—the core beads of the kaleidoscope. Once extracted, they're called *abstractions*. Whenever you encounter a new situation, you make sense of it by recombining on the fly abstractions from your collections, to weave a brand new "model" adapted to the situation.

This process consists of two key parts:

- *Abstraction acquisition*—Efficiently extracting compact, reusable abstractions from a stream of experience or data. This involves identifying underlying structures, principles, or invariants.
- *On-the-fly recombination*—Efficiently selecting and recombining these abstractions in novel ways to model new problems and situations, even ones far removed from past experience.

The emphasis on *efficiency* is crucial. How intelligent you are is determined by how efficiently you can acquire good abstractions from limited experience and how efficiently you can recombine them to navigate uncertainty and novelty. If you need hundreds of thousands of hours of practice to acquire a skill, you are not very intelligent. If you need to enumerate every possible move on the chess board to find the best one, you are not very intelligent.

And that's the source of the two main issues with the classic deep learning paradigm:

- These models are completely missing on-the-fly recombination. They do a decent job at acquiring abstractions at training time, via gradient descent, but by design they have zero ability to recombine what they know at test time. They behave like a static abstract database, limited purely to retrieval. They're missing half of the picture—the most important half.
- They're terribly inefficient. Gradient descent requires vast amounts of data to distill neat abstractions—many orders of magnitude more data than humans.

So how can we move beyond these limitations?

19.3.3 The importance of setting the right target

Biological intelligence was the answer to a question asked by nature. Likewise, if we want to develop true AI, first, we need to be asking the right questions. Ultimately, the capabilities of AI systems reflect the objectives they were designed and optimized for.

An effect you see constantly in systems design is the *shortcut rule*: if you focus on optimizing one success metric, you will achieve your goal, but at the expense of everything in the system that wasn't covered by your success metric. You end up taking every available shortcut toward the goal. Your creations are shaped by the incentives you give yourself.

You see this often in machine learning competitions. In 2009, Netflix ran a challenge that promised a $1 million prize to the team that would achieve the highest score on a movie recommendation task. It ended up never using the system created by the winning team because it was way too complex and compute intensive. The winners had

optimized for prediction accuracy alone—what they were incentivized to achieve—at the expense of every other desirable characteristic of the system: inference cost, maintainability, explainability. The shortcut rule holds true in most Kaggle competitions as well—the models produced by Kaggle winners can rarely, if ever, be used in production.

The shortcut rule has been everywhere in AI over the past few decades. In the 1970s, psychologist and computer science pioneer Allen Newell, concerned that his field wasn't making any meaningful progress toward a proper theory of cognition, proposed a new grand goal for AI: chess playing. The rationale was that playing chess, in humans, seemed to involve—perhaps even require—capabilities such as perception, reasoning and analysis, memory and study from books, and so on. Surely, if we could build a chess-playing machine, it would have to feature these attributes as well. Right?

Over two decades later, the dream came true: in 1997, IBM's Deep Blue beat Gary Kasparov, the best chess player in the world. Researchers had then to contend with the fact that creating a chess-champion AI had taught them little about human intelligence. The A-star algorithm at the heart of Deep Blue wasn't a model of the human brain and couldn't generalize to tasks other than similar board games. It turned out it was easier to build an AI that could only play chess than to build an artificial mind—so that's the shortcut researchers took.

So far, *the driving success metric of the field of AI has been to solve specific tasks*, from chess to Go, from MNIST classification to ImageNet, from high school math tests to the bar exam. Consequently, the history of the field has been defined by a series of "successes" where *we figured out how to solve these tasks without featuring any intelligence*.

If that sounds like a surprising statement, keep in mind that human-like intelligence isn't characterized by skill at any particular task—rather, it is the ability to adapt to novelty to efficiently acquire new skills and master never-before-seen tasks. By fixing the task, you make it possible to provide an arbitrarily precise description of what needs to be done—either via hardcoding human-provided knowledge or by supplying humongous amounts of data. You make it possible for engineers to "buy" more skill for their AI by just adding data or adding hardcoded knowledge, without increasing the generalization power of the AI (see figure 19.8). If you have near-infinite training data, even a very crude algorithm like nearest-neighbor search can play video games with superhuman skill. Likewise, if you have a near-infinite amount of human-written if-then-else statements—that is, until you make a small change to the rules of the game, the kind a human could adapt to instantly—that will require the unintelligent system to be retrained or rebuilt from scratch.

In short, by fixing the task, you remove the need to handle uncertainty and novelty, and since the nature of intelligence is the ability to handle uncertainty and novelty, you're effectively removing the need for intelligence. And because it's always easier to find an unintelligent solution to a specific task than to solve the general problem of intelligence, that's the shortcut you will take 100% of the time. Humans can use their general intelligence to acquire skills at any new task, but in reverse, there is no path from a collection of task-specific skills to general intelligence.

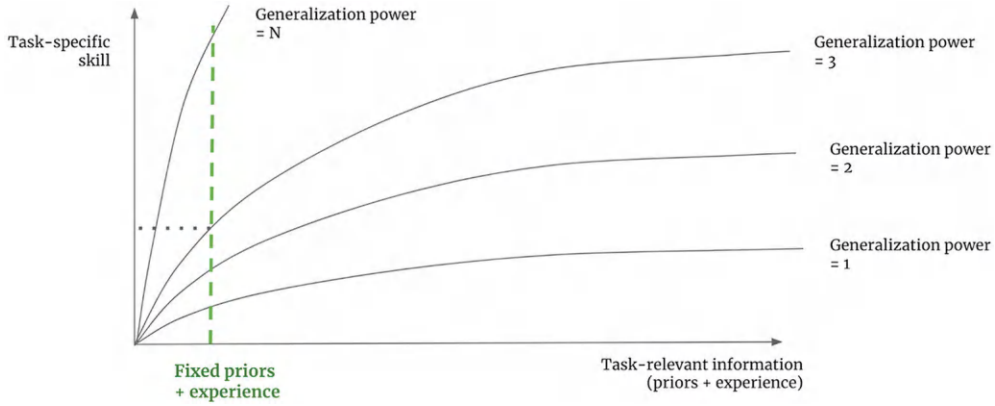


Figure 19.8 A low-generalization system can achieve arbitrary skill at a fixed task given unlimited task-specific information.

19.3.4 A new target: On-the-fly adaptation

To make AI actually intelligent and give it the ability to deal with the incredible variability and ever-changing nature of the real world, first, we need to move away from seeking to achieve *task-specific skill* and, instead, start targeting generalization power itself. We need new metrics of progress that will help us develop increasingly intelligent systems: metrics that will point in the right direction and that will give us an actionable feedback signal. As long as we set our goal to be "create a model that solves task X," the shortcut rule will apply, and we'll end up with a model that does X, period.

In my view, intelligence can be precisely quantified as an *efficiency ratio*: the conversion ratio between the *amount of relevant information* you have available about the world (which could be either past experience or innate prior knowledge) and your *future operating area*, the set of novel situations where you will be able to produce appropriate behavior (you can view this as your *skill set*). A more intelligent agent will be able to handle a broader set of future tasks and situations using a smaller amount of past experience. To measure such a ratio, you just need to fix the information available to your system—its experience and its prior knowledge—and measure its performance on a set of reference situations or tasks that are known to be sufficiently different from what the system has had access to. Trying to maximize this ratio should lead you toward intelligence. Crucially, to avoid cheating, you're going to need to make sure to test the system only on tasks it wasn't programmed or trained to handle—in fact, you need tasks that the *creators of the system could not have anticipated.*

In 2018 and 2019, I developed a benchmark dataset called the *Abstraction & Reasoning Corpus for Artificial General Intelligence (ARC-AGI)*[4] that seeks to capture this definition

[4] François Chollet, "On the Measure of Intelligence," 2019.

of intelligence. ARC-AGI is meant to be approachable by both machines and humans, and it looks very similar to human IQ tests, such as Raven's progressive matrices. At test time, you'll see a series of "tasks." Each task is explained via three or four "examples" that take the form of an input grid and a corresponding output grid (see figure 19.9). You'll then be given a brand-new input grid, and you'll have three tries to produce the correct output grid, before moving on to the next task.

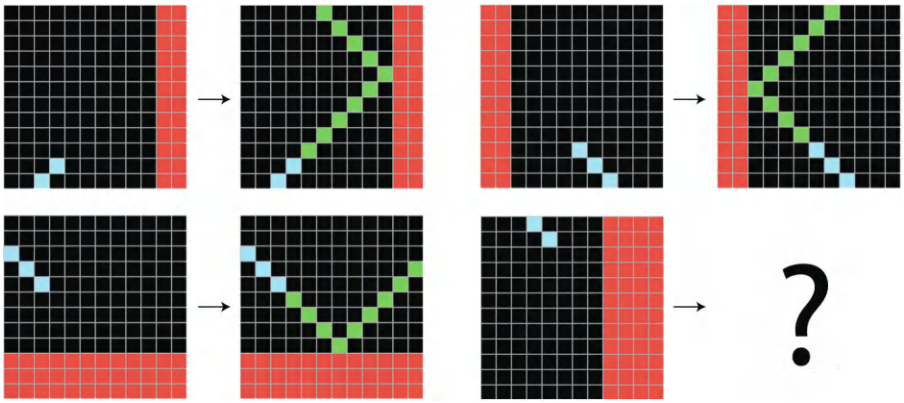


Figure 19.9 An ARC-AGI task: the nature of the task is demonstrated by a couple of input-output pair examples. Provided with a new input, you must construct the corresponding output.

Compared to IQ tests, two things are unique about ARC-AGI. First, ARC seeks to measure generalization power by only testing you on tasks you've never seen before. That means that ARC-AGI is *a game you can't practice for*, at least in theory: the tasks you will get tested on will have their own unique logic that you will have to understand on the fly. You can't just memorize specific strategies from past tasks.

In addition, ARC-AGI tries to control for the *prior knowledge* that you bring to the test. You never approach a new problem entirely from scratch—you bring to it preexisting skills and information. ARC-AGI makes the assumption that all test takers should start from the set of knowledge priors, called *Core Knowledge priors*, which represent the knowledge systems humans are born with. Unlike an IQ test, ARC-AGI tasks will never involve acquired knowledge, like English sentences, for instance.

19.3.5 ARC Prize

In 2024, to accelerate progress toward AI systems capable of the kind of fluid abstraction and reasoning measured by ARC-AGI, I partnered with Mike Knoop to establish the nonprofit ARC Prize Foundation. The foundation runs a yearly competition, with a substantial prize pool (over $1 million in its 2024 iteration) to incentivize researchers to develop AI that can solve ARC-AGI and thus display genuine fluid intelligence.

The ARC-AGI benchmark has proven remarkably resistant to the prevailing deep learning scaling paradigm. Most other benchmarks have saturated quickly in the age of LLMs. That's because they can be hacked via memorization, whereas ARC-AGI is designed to be resistant to it. From 2019, when ARC-AGI was first released, to 2025, base LLMs underwent a roughly 50,000× scale-up—from, say, GPT-2 (2019) to GPT-4.5 (2025), but their performance on the 2019 version of ARC-AGI only went from 0% to around 10%. Given that you, reader, would easily score above 95%, that isn't very good.

If you scale up your system by 50,000× and you're still not making meaningful progress, that's like a big warning sign telling you that you need to try new ideas. Simply making models bigger or training them on more data has not unlocked the kind of fluid intelligence that ARC-AGI requires. ARC-AGI was clearly showing that on-the-fly recombination capabilities are necessary to tackle reasoning.

19.3.6 The test-time adaptation era

In 2024, everything changed. That year saw a major narrative shift—one that was partly catalyzed by ARC Prize. The prevailing "Scale is all you need" story that was a bedrock dogma of 2023 started giving way to "Actually, we need on-the-fly recombination." The results of the competition, announced in December 2024, were illuminating: the leading solutions did not emerge from simply scaling existing deep learning architectures. They all used some form of test-time adaptation (TTA)—either test-time search or test-time training.

TTA refers to methods where the AI system performs active reasoning or learning during the test itself, using the specific problem information provided—the key component that was missing from the classic deep learning paradigm.

There are several ways to implement test-time adaptation:

- *Test-time training*—The model adjusts some of its parameters based on the examples given in the test task, using gradient descent.
- *Search methods*—The system searches through many possible reasoning steps or potential solutions at test time to find the best one. This could be done in natural language (chain-of-thought synthesis) or in a space of symbolic, verifiable programs (program synthesis).

These TTA approaches allow AI systems to be more flexible and handle novelty better than static models. Every single top entry in ARC Prize 2024 used them.

Shortly following the competition's conclusion, in late December 2024, OpenAI previewed its o3 test-time reasoning model and used ARC-AGI to showcase its unprecedented capabilities. Using considerable test-time compute resources, this model achieved scores of 76% at a cost of about $200 per task, and 88% at a cost of over $20,000 per task, surpassing the nominal human baseline. For the very first time, we were seeing an AI model that showed signs of genuine fluid intelligence. This breakthrough opened the floodgates of a new wave of interest and investment in similar techniques—the test-time adaptation era had begun. Importantly, ARC-AGI was one of the only benchmarks at the time that provided a clear signal that a major paradigm shift was underway.

19.3.7 ARC-AGI 2

Does that mean AGI is solved? Was o3 as intelligent as a human?

Not quite. First, while o3's performance was a landmark achievement, it came at a tremendous cost—tens of thousands of dollars of compute per ARC-AGI puzzle. Intelligence isn't just about capability; it's fundamentally about efficiency. Brute-forcing the solution space given enormous compute is a shortcut that makes all kinds of tasks possible without requiring intelligence. In principle, you could even solve ARC-AGI by simply walking down the tree of every possible solution program and testing each one until you find one that works on the demonstration pairs. The o3 results, impressive as they were, felt more like cracking a code with a supercomputer than a display of nimble, human-like fluid reasoning. The entire point of intelligence is to achieve results with the least amount of resources possible.

Second, we found that o3 was still stumped by many tasks that humans found very easy (like the one in figure 19.10). This strongly suggests that o3 wasn't quite human-level yet. Here's the thing—the 2019 version of ARC-AGI was intended to be easy. It was essentially a binary test of fluid intelligence: either you have no fluid intelligence, like all base LLMs, in which case you score near zero, or you do display some genuine fluid intelligence, in which case you immediately score extremely high, like any human—or o3. There wasn't much room in between. It was clear that the benchmark needed to evolve alongside the AI capabilities it was designed to measure. There was a need for a new ARC-AGI version that was less brute-forcible and that could better differentiate between systems possessing varying levels of fluid reasoning ability, up to human-level fluid intelligence. Good news: we had been working on one since 2022.

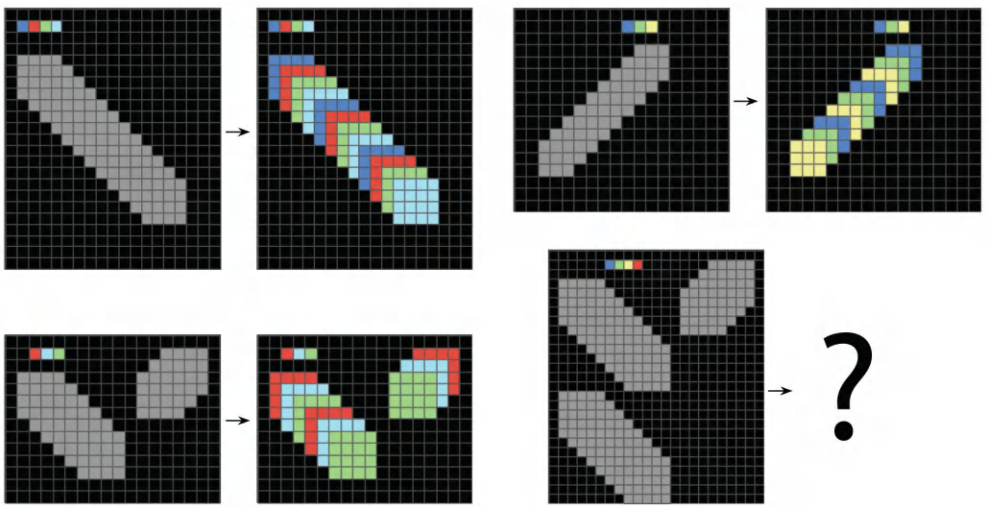


Figure 19.10 Example of a task that couldn't be solved by o3 on the highest compute settings (over $20,000 per task)

And so, in March 2025, the ARC Prize Foundation introduced ARC-AGI-2. It kept the exact same format as the first version but significantly improved the task content. The new iteration was designed to raise the bar, incorporating tasks that demand more complex reasoning chains and are inherently more resistant to exhaustive search methods. The goal was to create a benchmark where computational efficiency becomes a more critical factor for success, pushing systems toward more genuinely intelligent, efficient strategies rather than simply exploring billions of possibilities. While most ARC-AGI-1 tasks could be solved almost instantaneously by a human without requiring much cognitive effort, all tasks in ARC-AGI-2 require some amount of deliberate thinking (see figure 19.11)—for instance, the average time for task completion among human test takers in our experiments was 5 minutes.

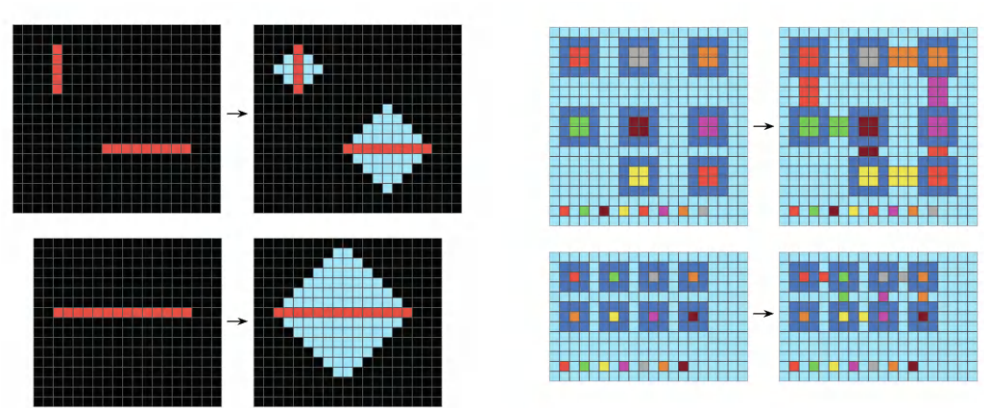


Figure 19.11 Typical ARC-AGI-1 task (left) vs. typical ARC-AGI-2 task (right)

The initial AI testing results on ARC-AGI 2 were sobering: even o3 struggled significantly with this new set of challenges, its scores plummeting back into the low double digits when constrained to reasonable computational budgets. As for base LLMs? Their performance on ARC-AGI-2 was effectively back at 0%—fitting, as base LLMs don't possess fluid intelligence. The challenge of building AI with truly efficient, human-like fluid intelligence is still far from solved. We're going to need something beyond current TTA techniques.

19.4 The missing ingredients: Search and symbols

What would it take to fully solve ARC-AGI, in particular version 2? Hopefully, this challenge will get you thinking. That's the entire point of ARC-AGI: to give you a goal of a different kind, that will nudge you in a new direction—hopefully, a productive direction. Now, let's take a quick look at the key ingredients you're going to need if you want to answer the call.

I've said that intelligence consists of two components: *abstraction acquisition* and *abstraction recombination*. They are tightly coupled—what *kind* of abstractions you manipulate determines how, and how well, you can recombine them. Deep learning models only manipulate abstractions stored via parametric curves, fitted via gradient descent. Could there be a better way?

19.4.1 The two poles of abstraction

Abstraction acquisition starts with *comparing things to one another*. Crucially, there are two distinct ways to compare things, from which arise two different kinds of abstraction and two modes of thinking, each better suited to a different kind of problem. Together, these two poles of abstraction form the basis for all of our thoughts.

The first way to relate things to each other is *similarity comparison*, which gives rise to *value-centric analogies*. The second way is *exact structural match*, which gives rise to *program-centric analogies* (or structure-centric analogies). In both cases, you start from *instances* of a thing, and you merge together related instances to produce an *abstraction* that captures the common elements of the underlying instances. What varies is how you tell that two instances are related and how you merge instances into abstractions. Let's take a close look at each type.

VALUE-CENTRIC ANALOGY

Let's say you come across a number of different beetles in your backyard, belonging to multiple species. You'll notice similarities between them. Some will be more similar to one another, and some will be less similar: the notion of similarity is implicitly a smooth, continuous *distance function* that defines a latent manifold where your instances live. Once you've seen enough beetles, you can start clustering more similar instances together and merging them into a set of *prototypes* that captures the shared visual features of each cluster (figure 19.12). These prototypes are abstract: they don't look like any specific instance you've seen, although they encode properties that are common across all of them. When you encounter a new beetle, you won't need to compare it to every single beetle you've seen before to know what to do with it. You can simply compare it to your handful of prototypes to find the closest prototype—the beetle's *category*—and use it to make useful predictions: Is the beetle likely to bite you? Will it eat your apples?

Does this sound familiar? It's pretty much a description of what unsupervised machine learning (such as the K-means clustering algorithm) does. In general, all of modern machine learning, unsupervised or not, works by learning latent manifolds that describe a space of instances, encoded via prototypes. (Remember the ConvNet features you visualized in chapter 10? They were visual prototypes.) Value-centric analogy is the kind of analogy-making that enables deep learning models to perform local generalization.

It's also what many of your own cognitive abilities run on. As a human, you perform value-centric analogies all the time. It's the type of abstraction that underlies *pattern*

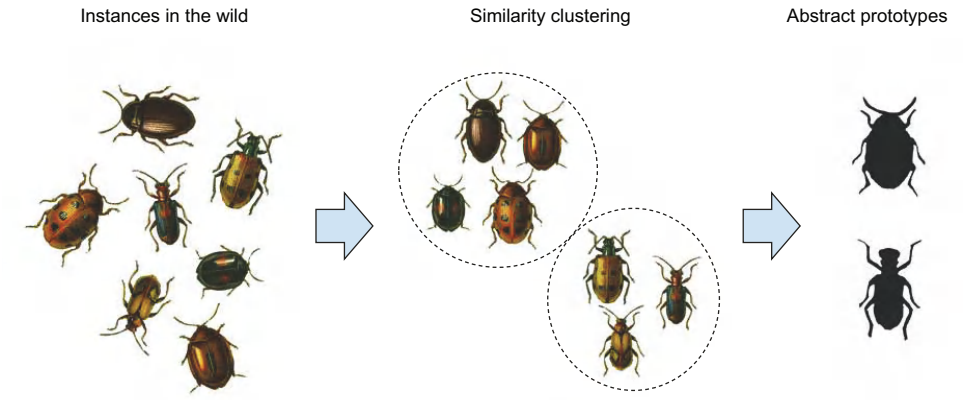


Figure 19.12 Value-centric analogy relates instances via a continuous notion of similarity to obtain abstract prototypes.

recognition, *perception*, and *intuition*. If you can do a task without thinking about it, you're heavily relying on value-centric analogies. If you're watching a movie and you start subconsciously categorizing the different characters into "types," that's value-centric abstraction.

PROGRAM-CENTRIC ANALOGY

Crucially, there's more to cognition than the kind of immediate, approximate, intuitive categorization that value-centric analogy enables. There's another type of abstraction-generation mechanism, slower, exact, deliberate: program-centric (or structure-centric) analogy.

In software engineering, you often write different functions or classes that seem to have a lot in common. When you notice these redundancies, you start asking, Could there be a more abstract function that performs the same job that could be reused twice? Could there be an abstract base class that both of your classes could inherit from? The definition of abstraction you're using here corresponds to program-centric analogy. You're not trying to compare your classes and functions by *how similar* they look, the way you'd compare two human faces, via an implicit distance function. Rather, you're interested in whether there are *parts* of them that have *exactly the same structure*. You're looking for what is called a *subgraph isomorphism* (see figure 19.13): programs can be represented as graphs of operators, and you're trying to find subgraphs (program subsets) that are exactly shared across your different programs.

This kind of analogy-making via exact structural match within different discrete structures isn't at all exclusive to specialized fields like computer science, or mathematics—you're constantly using it without noticing. It underlies *reasoning*, *planning*, and the general concept of *rigor* (as opposed to intuition). Any time you're thinking about objects connected to each other by a discrete network of relationships (rather than a continuous similarity function), you're using program-centric analogies.

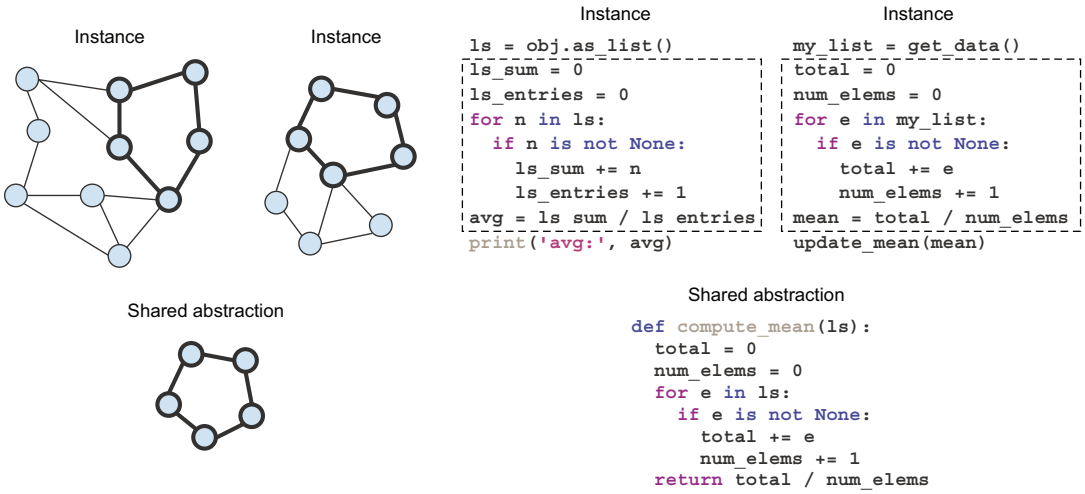


Figure 19.13 Program-centric analogy identifies and isolates isomorphic substructures across different instances.

19.4.2 Cognition as a combination of both kinds of abstraction

Table 19.1 compares these two poles of abstraction side by side.

Table 19.1 The two poles of abstraction

Value-centric abstraction	Program-centric abstraction
Relates things by distance	Relates things by exact structural match
Continuous, grounded in geometry	Discrete, grounded in topology
Produces abstractions by "averaging" instances into "prototypes"	Produces abstractions by isolating isomorphic substructures across instances
Underlies perception and intuition	Underlies reasoning and planning
Immediate, fuzzy, approximative	Slow, exact, rigorous
Requires a lot of experience to produce reliable results	Experience efficient: can operate on as few as two instances

Everything we do, everything we think, is a combination of these two types of abstraction. You'd be hard pressed to find tasks that only involve one of the two. Even a seemingly "pure perception" task, like recognizing objects in a scene, involves a fair amount of implicit reasoning about the relationships between the objects you're looking at. And even a seemingly "pure reasoning" task, like finding the proof of a mathematical theorem, involves a good amount of intuition. When a mathematician puts their pen to the paper, they've already got a fuzzy vision of the direction in which they're going. The discrete reasoning steps they take to get to the destination are guided by high-level intuition.

These two poles are complementary, and it's their interleaving that enables extreme generalization. No mind could be complete without both of them.

19.4.3 Why deep learning isn't a complete answer to abstraction generation

Deep learning is very good at encoding value-centric abstraction, but it has basically no ability to generate program-centric abstraction. Human-like intelligence is a tight interleaving of both types, so we're literally missing half of what we need—arguably the most important half.

Now, here's a caveat. So far, I've presented each type of abstraction as entirely separate from the other—opposite, even. In practice, however, they're more of a spectrum: to an extent, you could do reasoning by embedding discrete programs in continuous manifolds—just like you may fit a polynomial function through any set of discrete points, as long as you have enough coefficients. And inversely, you could use discrete programs to emulate continuous distance functions—after all, when you're doing linear algebra on a computer, you're working with continuous spaces, entirely via discrete programs that operate on 1s and 0s.

However, there are clearly types of problems that are better suited to one or the other. Try to train a deep learning model to sort a list of five numbers, for instance. With the right architecture, it's not impossible, but it's an exercise in frustration. You'll need a massive amount of training data to make it happen—and even then, the model will still make occasional mistakes when presented with new numbers. And if you want to start sorting lists of 10 numbers instead, you'll need to completely retrain the model—on even more data. Meanwhile, writing a sorting algorithm in Python takes just a few lines, and the resulting program, once validated on a couple more examples, will work every time on lists of any size. That's pretty strong generalization: going from a couple of demonstration examples and test examples to a program that can successfully process literally any list of numbers.

In reverse, perception problems are a terrible fit for discrete reasoning processes. Try to write a pure-Python program to classify MNIST digits, without using any machine learning technique: you're in for a ride. You'll find yourself painstakingly coding functions that can detect the number of closed loops in a digit, the coordinates of the center of mass of a digit, and so on. After thousands of lines of code, you might achieve 90% test accuracy. In this case, fitting a parametric model is much simpler; it can better utilize the large amount of data that's available, and it achieves much more robust results. If you have lots of data and you're faced with a problem where the manifold hypothesis applies, go with deep learning.

For this reason, it's unlikely that we'll see the rise of an approach that would reduce reasoning problems to manifold interpolation or that would reduce perception problems to discrete reasoning. The way forward in AI is to develop a unified framework that incorporates *both* types of abstraction generation.

19.4.4 An alternative approach to AI: Program synthesis

Until 2024, AI systems capable of genuine discrete reasoning were all hardcoded by human programmers—for instance, software that relies on search algorithms, graph manipulation, and formal logic. In the test-time adaptation (TTA) era, this is finally starting to change. A branch of TTA that is especially promising is *program synthesis*—a field that is still very niche today, but that I expect to take off in a big way over the next few decades.

Program synthesis consists of automatically generating simple programs by using a search algorithm (possibly genetic search, as in *genetic programming*) to explore a large space of possible programs (see figure 19.14). The search stops when a program is found that matches the required specifications, often provided as a set of input-output pairs. This is highly reminiscent of machine learning: given training data provided as input-output pairs, we find a program that matches inputs to outputs and can generalize to new inputs. The difference is that instead of learning parameter values in a hardcoded program (a neural network), we generate source code via a discrete search process (see table 19.2).

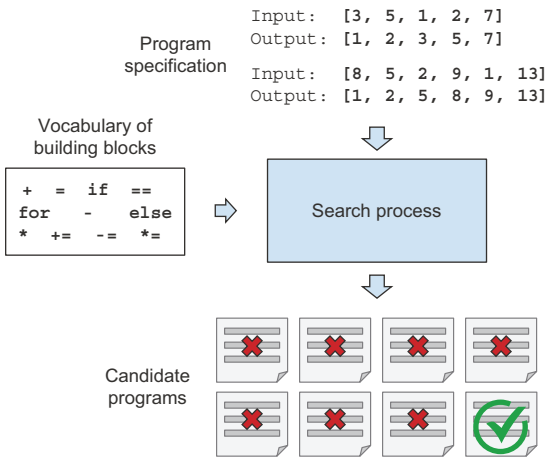


Figure 19.14 A schematic view of program synthesis: given a program specification and a set of building blocks, a search process assembles the building blocks into candidate programs, which are then tested against the specification. The search continues until a valid program is found.

Table 19.2 Machine learning vs. program synthesis

Machine learning	Program synthesis
Model: differentiable parametric function	Model: graph of operators from a programming language
Engine: gradient descent	Engine: discrete search (such as genetic search)
Requires a lot of data to produce reliable results	Data efficient: can work with a couple of training examples

Program synthesis is how we're going to add program-centric abstraction capabilities to our AI systems. It's the missing piece of the puzzle.

19.4.5 Blending deep learning and program synthesis

Of course, deep learning isn't going anywhere. Program synthesis isn't its replacement; it is its complement. It's the hemisphere that has been so far missing from our artificial brains. We're going to be using both, in combination. There are two major ways this will take place:

- Developing systems that integrate both deep learning modules and discrete algorithmic modules
- Using deep learning to make the program search process itself more efficient

Let's review each of these possible avenues.

INTEGRATING DEEP LEARNING MODULES AND ALGORITHMIC MODULES INTO HYBRID SYSTEMS

Today, many of the most powerful AI systems are hybrid: they use both deep learning models and handcrafted symbol-manipulation programs. In DeepMind's AlphaGo, for example, most of the intelligence on display is designed and hardcoded by human programmers (such as Monte Carlo Tree Search). Learning from data happens only in specialized submodules (value networks and policy networks). Or consider the Waymo self-driving car: it's able to handle a large variety of situations because it maintains a model of the world around it—a literal 3D model—full of assumptions hardcoded by human engineers. This model is constantly updated via deep learning perception modules (powered by Keras) that interface it with the surroundings of the car.

For both of these systems—AlphaGo and self-driving vehicles—the combination of human-created discrete programs and learned continuous models is what unlocks a level of performance that would be impossible with either approach in isolation, such as an end-to-end deep net or a piece of software without machine learning elements. So far, the discrete algorithmic elements of such hybrid systems are painstakingly hardcoded by human engineers. But in the future, such systems may be fully learned, with no human involvement.

What will this look like? Consider a well-known type of network: recurrent neural networks (RNNs). It's important to note that RNNs have slightly fewer limitations than feedforward networks. That's because RNNs are a bit more than mere geometric transformations: they're geometric transformations *repeatedly applied inside a* `for` *loop*. The temporal `for` loop is itself hardcoded by human developers: it's a built-in assumption of the network. Naturally, RNNs are still extremely limited in what they can represent, primarily because each step they perform is a differentiable geometric transformation, and they carry information from step to step via points in a continuous geometric space (state vectors). Now imagine a neural network that's augmented in a similar way with programming primitives but instead of a single hardcoded `for` loop with hardcoded continuous-space memory, the network includes a large set of programming primitives that the model is free to manipulate to expand its processing function, such as `if` branches, `while` statements, variable creation, disk storage for long-term memory, sorting operators, advanced data structures (such as lists, graphs, and hash tables), and many more. The space of programs that such a network could represent would be far

broader than what can be represented with current deep learning models, and some of these programs could achieve superior generalization power. Importantly, such programs will not be differentiable end to end, although specific modules will remain differentiable, and thus will need to be generated via a combination of discrete program search and gradient descent.

We'll move away from having, on one hand, hardcoded algorithmic intelligence (handcrafted software) and, on the other hand, learned geometric intelligence (deep learning). Instead, we'll have a blend of formal algorithmic modules that provide reasoning and abstraction capabilities and geometric modules that provide informal intuition and pattern-recognition capabilities (figure 19.15). The entire system will be learned with little or no human involvement. This should dramatically expand the scope of problems that can be solved with machine learning—the space of programs that we can generate automatically, given appropriate training data. Systems like AlphaGo—or even RNNs—can be seen as a prehistoric ancestor of such hybrid algorithmic-geometric models.

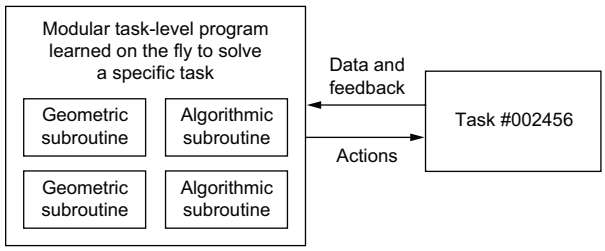


Figure 19.15 A learned program relying on both geometric primitives (pattern recognition, intuition) and algorithmic primitives (reasoning, search, memory)

USING DEEP LEARNING TO GUIDE PROGRAM SEARCH

Today, program synthesis faces a major obstacle: it's tremendously inefficient. To caricature, typical program synthesis techniques work by trying every possible program in a search space until it finds one that matches the specification provided. As the complexity of a program specification increases, or as the vocabulary of primitives used to write programs expands, the program search process runs into what's known as *combinatorial explosion*: the set of possible programs to consider grows very fast, in fact, much faster than merely exponentially fast. As a result, today, program synthesis can only be used to generate very short programs. You're not going to be generating a new OS for your computer anytime soon.

To move forward, we're going to need to make program synthesis efficient by bringing it closer to the way humans write software. When you open your editor to code up a script, you're not thinking about every possible program you could potentially write. You only have in mind a handful of possible approaches: you can use your understanding of the problem and your past experience to drastically cut through the space of possible options to consider.

Deep learning can help program synthesis do the same: although each specific program we'd like to generate might be a fundamentally discrete object that performs non-interpolative data manipulation, evidence so far indicates that *the space of all useful programs* may look a lot like a continuous manifold. That means that a deep learning model that has been trained on millions of successful program-generation episodes might start to develop solid *intuition* about the *path through program space* that the search process should take to go from a specification to the corresponding program—just like a software engineer might have immediate intuition about the overall architecture of the script they're about to write and about the intermediate functions and classes they should use as stepping stones on the way to the goal.

Remember that human reasoning is heavily guided by value-centric abstraction—that is, by pattern recognition and intuition. The same should be true of program synthesis. I expect the general approach of guiding program search via learned heuristics to see increasing research interest over the next 10 to 20 years.

19.4.6 Modular component recombination and lifelong learning

If models become more complex and are built on top of richer algorithmic primitives, then this increased complexity will require higher reuse between tasks, rather than training a new model from scratch every time we have a new task or a new dataset. Many datasets don't contain enough information for us to develop a new, complex model from scratch, and it will be necessary to use information from previously encountered datasets (much as you don't learn English from scratch every time you open a new book—that would be impossible). Training models from scratch on every new task is also inefficient due to the large overlap between the current tasks and previously encountered tasks.

With modern foundation models, we're starting to move closer to a world where AI systems possess enormous amounts of acquired knowledge and skills and can bring them to bear on whatever comes their way. But LLMs are missing a key ingredient: recombination. LLMs are very good at fetching and reapplying memorized functions, but they're not yet able to recombine those functions on the fly into brand-new programs adapted to the situation at hand. They are, in fact, entirely incapable of performing function composition, as investigated in a recent paper by Dziri et al.[5] What's more, the kind of functions they learn aren't sufficiently abstract or modular, making them a poor fit for recombination in the first place. Remember how we pointed out that LLMs have low accuracy when adding large integers? You probably wouldn't want to build your next codebase on top of such brittle functions.

To solve *compositional generalization*, we're going to need to reuse robust *program components* like the functions and classes found in human programming languages. These components will be evolved specifically for modular reuse in a new context—unlike the patterns that LLMs memorize. And our AIs will recombine them on the fly to synthesize

[5] Nouha Dziri et. al., "Faith and Fate: Limits of Transformers on Compositionality," Proceedings of the 37th International Conference on Neural Information Processing Systems (2023), https://arxiv.org/abs/2305.18654.

new programs adapted to the current task. Crucially, libraries of such reusable components will be built through the cumulative experience of all instances of our AIs and will then be accessible by all in perpetuity. Any single problem encountered by our AIs would only need to be solved once—making them constantly self-improving.

Think of the process of software development today: once an engineer solves a specific problem (HTTP queries in Python, for instance), they package it as an abstract, reusable library, accessible by anyone on the planet. Engineers who face a similar problem in the future will be able to search for existing libraries, download one, and use it in their own project. In a similar way, in the future, meta-learning systems will be able to assemble new programs by sifting through a global library of high-level reusable blocks. When the system finds itself developing similar program subroutines for several different tasks, it can come up with an *abstract*, reusable version of the subroutine and store it in the global library (see figure 19.16). These subroutines can be either geometric (deep learning modules with pretrained representations) or algorithmic (closer to the libraries that contemporary software engineers manipulate).

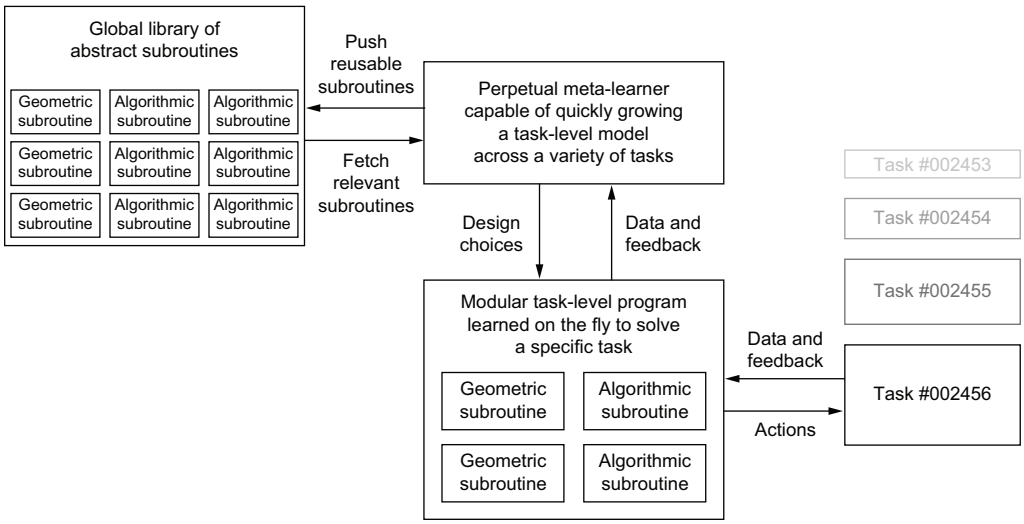


Figure 19.16 A meta-learner capable of quickly developing task-specific models using reusable primitives (both algorithmic and geometric), thus achieving extreme generalization

19.4.7 The long-term vision

In short, here's my long-term vision for AI:

- Models will be more like programs and will have capabilities that go far beyond the continuous geometric transformations of the input data we currently work with. These programs will arguably be much closer to the abstract mental models

that humans maintain about their surroundings and themselves, and they will be capable of stronger generalization due to their rich algorithmic nature.

- In particular, models will blend *algorithmic modules* providing formal reasoning, search, and abstraction capabilities with *geometric modules* providing informal intuition and pattern-recognition capabilities. This will achieve a blend of value-centric and program-centric abstraction. AlphaGo or self-driving cars (systems that required a lot of manual software engineering and human-made design decisions) provide an early example of what such a blend of symbolic and geometric AI could look like.
- Such models will be *grown* automatically rather than hardcoded by human engineers, using modular parts stored in a global library of reusable subroutines—a library evolved by learning high-performing models on thousands of previous tasks and datasets. As frequent problem-solving patterns are identified by the meta-learning system, they will be turned into reusable subroutines—much like functions and classes in software engineering—and added to the global library.
- The process that searches over possible combinations of subroutines to grow new models will be a discrete search process (program synthesis), but it will be heavily guided by a form of *program-space intuition* provided by deep learning.
- This global subroutine library and associated model-growing system will be able to achieve some form of human-like *extreme generalization*: given a new task or situation, the system will be able to assemble a new working model appropriate for the task using very little data, thanks to rich program-like primitives that generalize well, and extensive experience with similar tasks. In the same way, humans can quickly learn to play a complex new video game if they have experience with many previous games because the models derived from this previous experience are abstract and program-like, rather than a basic mapping between stimuli and action.

This perpetually learning model-growing system can be interpreted as an *artificial general intelligence* (AGI). But don't expect any singularitarian robot apocalypse to ensue: that's pure fantasy, coming from a long series of profound misunderstandings of both intelligence and technology. Such a critique, however, doesn't belong in this book.

20 Conclusions

This chapter covers

- Important takeaways from this book
- Resources for learning further and applying your skills in practice

We'll start with a bird's-eye view of what you should take away from this book. This should refresh your memory regarding some of the concepts you've learned. Then, we'll give you a short list of resources and strategies for learning further about machine learning and staying up to date with new advances.

Becoming an effective AI practitioner is a journey, and finishing this book is merely your first step on it. I want to make sure you realize this and are properly equipped to take the next steps of this journey on your own.

20.1 Key concepts in review

This section briefly synthesizes key takeaways from this book. If you ever need a quick refresher to help you recall what you've learned, you can read these few pages.

20.1.1 Various approaches to artificial intelligence

First, deep learning isn't synonymous with artificial intelligence (AI), or even with machine learning:

- *Artificial intelligence* (AI) is an ancient, broad field that can generally be understood as "all attempts to automate human cognitive processes." This can range from the very basic, such as an Excel spreadsheet, to the very advanced, like a humanoid robot that can walk and talk.
- *Machine learning* is a specific subfield of AI that aims at automatically developing programs (called *models*) purely from exposure to training data. This process of turning data into a program is called *learning*. Although machine learning has been around for a long time, it only started to take off in the 1990s, before becoming the dominant form of AI in the 2000s.
- *Deep learning* is one of many branches of machine learning, where the models are long chains of geometric transformations, applied one after the other. These operations are structured into modules called *layers*: deep learning models are typically stacks of layers—or, more generally, graphs of layers. These layers are parameterized by *weights*, which are the parameters learned during training. The *knowledge* of a model is stored in its weights, and the process of learning consists of finding "good values" for these weights—values that minimize a *loss function*. Because the chain of geometric transformations considered is differentiable, updating the weights to minimize the loss function is done efficiently via *gradient descent*.
- *Generative AI* is a specific subset of deep learning, where models are capable of generating text, images, videos, or sound. These models tend to be very large—billions of parameters. They're trained in a self-supervised manner; that is, they're trained to reconstruct artificially missing or corrupted parts of an input—for instance, denoising images, predicting the next word in a sentence, and so on. This learning process enables the models to learn sophisticated "maps" (embedding manifolds) of their input space, which can be used for sampling new inputs. These models have launched AI into its "consumer" era with the rise of products like ChatGPT or Midjourney.

Even though deep learning is just one among many approaches to machine learning, it isn't on an equal footing with the others. Deep learning is a breakout success. Here's why.

20.1.2 What makes deep learning special within the field of machine learning

In the span of only a few years, deep learning has achieved tremendous breakthroughs across a wide range of tasks that have been historically perceived as extremely difficult for computers, especially in the area of machine perception: extracting useful information from images, videos, sound, and more. Given sufficient training data (in particular, training data appropriately labeled by humans), deep learning makes it possible

to extract from perceptual data almost anything a human could. Hence, it's sometimes said that deep learning has "solved perception"—although that's true only for a fairly narrow definition of perception.

Due to its unprecedented technical successes, deep learning has singlehandedly brought about the third and by far the largest *AI summer*: a period of intense interest, investment, and hype in the field of AI. As this book is being written, we're in the middle of it. Whether this period will end in the near future and what happens after it ends are topics of debate. One thing is certain: in stark contrast with previous AI summers, deep learning has provided enormous business value to both large and small technology companies and has become a huge consumer success, enabling human-level speech recognition, chatbot assistants, photorealistic image generation, human-level machine translation, and more. The hype may (and likely will) recede, but the sustained economic and technological impact of deep learning will remain. In that sense, deep learning could be analogous to the internet: it may be overly hyped up for a few years, but in the longer term, it will still be a major revolution that will transform our economy and our lives.

One reason I'm particularly optimistic about deep learning is that even if we were to make no further technological progress in the next decade, deploying existing algorithms to every applicable problem would be a game changer for most industries. Deep learning is nothing short of a revolution, and progress is currently happening at an incredibly fast rate due to an exponential investment in resources and headcount. From where we stand, the future looks bright, although short-term expectations are somewhat overoptimistic; deploying deep learning to the full extent of its potential will likely take multiple decades.

20.1.3 How to think about deep learning

The most surprising thing about deep learning is how simple it is. Fifteen years ago, no one expected that we would achieve such amazing results on machine-perception and natural language processing problems by using simple parametric models trained with gradient descent. Now it turns out that all you need is sufficiently large parametric models trained with gradient descent on sufficiently many examples. As Feynman once said about the universe, "It's not complicated, it's just a lot of it."[1]

In deep learning, everything is a vector; that is, everything is a *point* in a *geometric space*. Model inputs (text, images, and so on) and targets are first *vectorized*—turned into an initial input vector space and target vector space. Each layer in a deep learning model operates one simple geometric transformation on the data that goes through it. Together, the chain of layers in the model forms one complex geometric transformation, broken down into a series of simple ones. This complex transformation attempts to map the input space to the target space, one point at a time. This transformation is parameterized by the weights of the layers, which are iteratively updated based on how well the model is currently performing. A key characteristic of this geometric

[1] Richard Feynman, interview, *The World from Another Point of View*, Yorkshire Television, 1972.

transformation is that it must be *differentiable*, which is required for us to be able to learn its parameters via gradient descent. Intuitively, this means the geometric morphing from inputs to outputs must be smooth and continuous—a significant constraint.

The entire process of applying this complex geometric transformation to the input data can be visualized in 3D by imagining a person trying to uncrumple a paper ball: the crumpled paper ball is the manifold of the input data that the model starts with. Each movement operated by the person on the paper ball is similar to a simple geometric transformation operated by one layer. The full uncrumpling gesture sequence is the complex transformation of the entire model. Deep learning models are mathematical machines for uncrumpling complicated manifolds of high-dimensional data.

That's the magic of deep learning—turning meaning into vectors, into geometric spaces, and then incrementally learning complex geometric transformations that map one space to another. All you need are spaces of sufficiently high dimensionality to capture the full scope of the relationships found in the original data.

The whole thing hinges on two core ideas: that *meaning is derived from the pairwise relationship between things* (between words in a language, between pixels in an image, and so on) and that *these relationships can be captured by a distance function*. But note that whether the brain implements meaning via geometric spaces is an entirely separate question. Vector spaces are efficient to work with from a computational standpoint, but different data structures for intelligence can easily be envisioned—in particular, graphs. Neural networks initially emerged from the idea of using graphs as a way to encode meaning, which is why they're named *neural networks*; the surrounding field of research used to be called *connectionism*. Nowadays, the name *neural network* exists purely for historical reasons—it's an extremely misleading name because they're neither neural nor networks. In particular, neural networks have hardly anything to do with the brain. A more appropriate name would have been *layered representations learning* or *hierarchical representations learning*, or maybe even deep *differentiable models* or *chained geometric transforms*, to emphasize the fact that continuous geometric space manipulation is at their core.

20.1.4 Key enabling technologies

The technological revolution that's currently unfolding didn't start with any single breakthrough invention. Rather, like any other revolution, it's the product of a vast accumulation of enabling factors—slowly at first, and then suddenly. In the case of deep learning, we can point out the following key factors:

- Incremental algorithmic innovations, first spread over two decades (starting with backpropagation) and then happening increasingly faster as more research effort was poured into deep learning after 2012. One major such breakthrough was the Transformer architecture in 2017.
- The availability of large amounts of image, video, and text data, which is a requirement to realize that sufficiently large models trained on sufficiently large data are all we need. This is, in turn, a by-product of the rise of the consumer internet and

Moore's law applied to storage media. Today, state-of-the-art language models are trained on a large fraction of the entire internet.

- The availability of fast, highly parallel computation hardware at a low price, especially the GPUs produced by NVIDIA—first gaming GPUs and then chips designed from the ground up for deep learning. Early on, NVIDIA CEO Jensen Huang took note of the deep learning boom and decided to bet the company's future on it, which paid off in a big way.
- A complex stack of software layers that makes this computational power available to humans: the CUDA language, frameworks like TensorFlow, JAX, and PyTorch that do automatic differentiation, and Keras, which makes deep learning accessible to most people.

In the future, deep learning will not be used only by specialists such as researchers, graduate students, and engineers with an academic profile; it will be a tool in the toolbox of every developer, much like web technology today. Everyone needs to build intelligent apps: just as every business today needs a website, every product will need to intelligently make sense of user-generated data. Bringing about this future will require us to build tools that make deep learning radically easy to use and accessible to anyone with basic coding abilities. Keras has been the first major step in that direction.

20.1.5 The universal machine learning workflow

Having access to an extremely powerful tool for creating models that map any input space to any target space is great, but the difficult part of the machine learning workflow is often everything that comes before designing and training such models (and, for production models, what comes after, as well). Understanding the problem domain to be able to determine what to attempt to predict, given what data, and how to measure success is a prerequisite for any successful application of machine learning, and it isn't something that advanced tools like Keras and TensorFlow can help you with. As a reminder, here's a quick summary of the typical machine learning workflow as described in chapter 6:

- *Define the problem.* What data is available, and what are you trying to predict? Will you need to collect more data or hire people to manually label a dataset?
- *Identify a way to reliably measure success on your goal.* For simple tasks, this may be prediction accuracy, but in many cases, it will require sophisticated, domain-specific metrics.
- *Prepare the validation process that you'll use to evaluate your models.* In particular, you should define a training set, a validation set, and a test set. The validation-set and test-set labels shouldn't leak into the training data: for instance, with temporal prediction, the validation and test data should be posterior to the training data.
- *Vectorize the data by turning it into vectors and preprocessing it in a way that makes it more easily approachable by a neural network (normalization and so on).*

- *Develop a first model that beats a trivial common-sense baseline, thus demonstrating that machine learning can work on your problem.* This may not always be the case!
- *Gradually refine your model architecture by tuning hyperparameters and adding regularization.* Make changes based on performance on the validation data only, not the test data or the training data. Remember that you should get your model to overfit (thus identifying a model capacity level that's greater than you need) and only then begin to add regularization or downsize your model. Beware of validation-set overfitting when tuning hyperparameters—the fact that your hyperparameters may end up being overspecialized to the validation set. Avoiding this is the purpose of having a separate test set!
- *Deploy your final model in production—as a web API, as part of a JavaScript or C++ application, on an embedded device, etc.* Keep monitoring its performance on real-world data and use your findings to refine the next iteration of the model!

20.1.6 Key network architectures

The families of network architectures that you should be familiar with after reading this book are *densely connected networks*, *convolutional networks*, *recurrent networks*, *Diffusion Models*, and *Transformers*. Each type of model is meant for specific data modalities: a network architecture encodes *assumptions* about the structure of the data—a *hypothesis space* within which the search for a good model will proceed. Whether a given architecture will work on a given problem depends entirely on the match between the structure of the data and the assumptions of the network architecture.

These different network types can easily be combined to achieve larger multimodal models, much as you combine LEGO bricks. In a way, deep learning layers are LEGO bricks for information processing. Table 20.1 shows a quick overview of the mapping between input and output modalities and the appropriate network architectures.

Table 20.1 Model architectures for different data types

Input	Output	Model
Vector data	Class probability, Regression value	Densely connected network
Timeseries data	Class probability, Regression value	RNN, Transformer
Images	Class probability, Regression value	ConvNet
Text	Class probability, Regression value	Transformer
Text, Images	Text	Transformer
Text, Images	Images	VAE, Diffusion Model

Now let's quickly review the specificities of each network architecture.

DENSELY CONNECTED NETWORKS

A densely connected network is a stack of `Dense` layers, meant to process vector data (where each sample is a vector of numerical or categorical attributes). Such networks

assume no specific structure in the input features: they're called *densely connected* because the units of a `Dense` layer are connected to every other unit. The layer attempts to map relationships between any two input features; this is unlike a 2D convolution layer, for instance, which only looks at *local* relationships.

Densely connected networks are most commonly used for categorical data (for example, where the input features are lists of attributes), such as the Boston Housing Price dataset used in chapter 4. They're also used as the final classification or regression stage of most networks. For instance, the ConvNets covered in chapter 8 typically end with one or two `Dense` layers, and so do the recurrent networks in chapter 13.

Remember, to perform *binary classification*, end your stack of layers with a `Dense` layer with a single unit and a `sigmoid` activation, and use `binary_crossentropy` as the loss. Your targets should be either 0 or 1:

```
import keras
from keras import layers

inputs = keras.Input(shape=(num_input_features,))
x = layers.Dense(32, activation="relu")(inputs)
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="binary_crossentropy")
```

To perform *single-label categorical classification* (where each sample has exactly one class, no more), end your stack of layers with a `Dense` layer with a number of units equal to the number of classes and a `softmax` activation. If your targets are one-hot encoded, use `categorical_crossentropy` as the loss; if they're integers, use `sparse_categorical_crossentropy`:

```
inputs = keras.Input(shape=(num_input_features,))
x = layers.Dense(32, activation="relu")(inputs)
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(num_classes, activation="softmax")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="categorical_crossentropy")
```

To perform *multilabel categorical classification* (where each sample can have several classes), end your stack of layers with a `Dense` layer with a number of units equal to the number of classes and a `sigmoid` activation, and use `binary_crossentropy` as the loss. Your targets should be k-hot encoded:

```
inputs = keras.Input(shape=(num_input_features,))
x = layers.Dense(32, activation="relu")(inputs)
```

```
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(num_classes, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="binary_crossentropy")
```

To perform *regression* toward a vector of continuous values, end your stack of layers with a `Dense` layer with a number of units equal to the number of values you're trying to predict (often a single one, such as the price of a house) and no activation. Various losses can be used for regression—most commonly `mean_squared_error` (MSE):

```
inputs = keras.Input(shape=(num_input_features,))
x = layers.Dense(32, activation="relu")(inputs)
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(num_values)(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="mse")
```

ConvNets

Convolution layers look at spatially local patterns by applying the same geometric transformation to different spatial locations (*patches*) in an input tensor. This results in representations that are *translation invariant*, making convolution layers highly data efficient and modular. This idea is applicable to spaces of any dimensionality: 1D (continuous sequences), 2D (images), 3D (volumes), and so on. You can use the `Conv1D` layer to process sequences, the `Conv2D` layer to process images, and the `Conv3D` layer to process volumes. As a leaner, more efficient alternative to convolution layers, you can also use *depthwise separable convolution* layers, such as `SeparableConv2D`.

ConvNets, or *convolutional networks*, consist of stacks of convolution and max-pooling layers. The pooling layers let you spatially downsample the data, which is required to keep feature maps to a reasonable size as the number of features grows and to allow subsequent convolution layers to "see" a greater spatial extent of the inputs. ConvNets are often ended with either a `Flatten` operation or a global pooling layer, turning spatial feature maps into vectors, followed by `Dense` layers to achieve classification or regression.

Here's a typical image-classification network (categorical classification, in this case) using `SeparableConv2D` layers:

```
inputs = keras.Input(shape=(height, width, channels))
x = layers.SeparableConv2D(32, 3, activation="relu")(inputs)
x = layers.SeparableConv2D(64, 3, activation="relu")(x)
x = layers.MaxPooling2D(2)(x)
x = layers.SeparableConv2D(64, 3, activation="relu")(x)
x = layers.SeparableConv2D(128, 3, activation="relu")(x)
```

```
x = layers.MaxPooling2D(2)(x)
x = layers.SeparableConv2D(64, 3, activation="relu")(x)
x = layers.SeparableConv2D(128, 3, activation="relu")(x)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(num_classes, activation="softmax")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="categorical_crossentropy")
```

When building a very deep ConvNet, it's common to add *batch normalization* layers as well as *residual connections*—two architecture patterns that help gradient information flow smoothly through the network.

TRANSFORMERS

A Transformer looks at a set of vectors (such as word vectors) and uses *neural attention* to transform each vector into a representation that is aware of the *context* provided by the other vectors in the set. When the set in question is an ordered sequence, you can also use *positional encoding* to create Transformers that can take into account both global context and word order, capable of processing long text paragraphs much more effectively than RNNs or 1D ConvNets.

Transformers can be used for any set-processing or sequence-processing task, including text classification, but they excel especially at *sequence-to-sequence learning*, such as translating paragraphs in a source language into a target language.

A sequence-to-sequence Transformer is made of two parts:

- A `TransformerEncoder` that turns an input vector sequence into a context-aware, order-aware output vector sequence
- A `TransformerDecoder` that takes the output of the `TransformerEncoder`, as well as a target sequence, and predicts what should come next in the target sequence

If you're only processing a single sequence (or set) of vectors, you'd only use the `TransformerEncoder`.

Following is a sequence-to-sequence Transformer for mapping a source sequence to a target sequence (this setup could be used for machine translation or question-answering, for instance):

```
                                                          Source sequence
from keras_hub.layers import TokenAndPositionEmbedding
from keras_hub.layers import TransformerDecoder, TransformerEncoder

encoder_inputs = keras.Input(shape=(src_seq_length,), dtype="int64")
x = TokenAndPositionEmbedding(vocab_size, src_seq_length, embed_dim)(
    encoder_inputs
)
encoder_outputs = TransformerEncoder(intermediate_dim=256, num_heads=8)(x)
```

```
decoder_inputs = keras.Input(shape=(dst_seq_length,), dtype="int64")
x = TokenAndPositionEmbedding(vocab_size, dst_seq_length, embed_dim)(
    decoder_inputs
)
x = TransformerDecoder(intermediate_dim=256, num_heads=8)(x, encoder_outputs)
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
transformer = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)
transformer.compile(optimizer="adamw", loss="categorical_crossentropy")
```

Predictions for target sequence one step in the future

Target sequence so far

And this is a lone `TransformerEncoder` for binary classification of integer sequences:

```
inputs = keras.Input(shape=(seq_length,), dtype="int64")
x = TokenAndPositionEmbedding(vocab_size, seq_length, embed_dim)(inputs)
x = TransformerEncoder(intermediate_dim=256, num_heads=8)(x)
x = layers.GlobalMaxPooling1D()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="adamw", loss="binary_crossentropy")
```

RECURRENT NEURAL NETWORKS

Recurrent neural networks (RNNs) work by processing sequences of inputs one timestep at a time and maintaining a state throughout (a state is typically a vector or set of vectors). They should be used preferentially over 1D ConvNets in the case of sequences where patterns of interest aren't invariant by temporal translation (for instance, timeseries data where the recent past is more important than the distant past).

Three RNN layers are available in Keras: `SimpleRNN`, `GRU`, and `LSTM`. For most practical purposes, you should use either `GRU` or `LSTM`. `LSTM` is the more powerful of the two but is also more expensive; you can think of `GRU` as a simpler, cheaper alternative to it.

To stack multiple RNN layers on top of each other, each layer prior to the last layer in the stack should return the full sequence of its outputs (each input timestep will correspond to an output timestep); if you aren't stacking any further RNN layers, then it's common to return only the last output, which contains information about the entire sequence.

Following is a single RNN layer for binary classification of vector sequences:

```
inputs = keras.Input(shape=(num_timesteps, num_features))
x = layers.LSTM(32)(inputs)
outputs = layers.Dense(num_classes, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="binary_crossentropy")
```

And this is a stacked RNN layer for binary classification of vector sequences:

```
inputs = keras.Input(shape=(num_timesteps, num_features))
x = layers.LSTM(32, return_sequences=True)(inputs)
x = layers.LSTM(32, return_sequences=True)(x)
x = layers.LSTM(32)(x)
outputs = layers.Dense(num_classes, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="binary_crossentropy")
```

20.2 Limitations of deep learning

Building deep learning models is like playing with LEGO bricks: layers can be plugged together to map essentially anything to anything, given that you have appropriate training data available and that the mapping is achievable via a continuous geometric transformation of reasonable complexity.

Here's the catch, though—this mapping is often not learnable in a way that will generalize. Deep learning models operate like vast, interpolative databases of patterns. Their pattern-matching strength is also their core weakness:

- *They fundamentally struggle to adapt to novelty.* Because their parameters are fixed after training, they can only retrieve or replicate patterns similar to their training data. Faced with inputs significantly outside this familiar distribution—no matter how simple the underlying task—their performance degrades drastically, as they lack mechanisms for fluid generalization beyond their memorized experience. This explains why even large models fail on novel tasks or simple variations of familiar problems—like ARC-AGI tasks.
- *They're very sensitive to phrasing and other distractors.* Deep learning models exhibit high sensitivity to superficial variations in input presentation, such as minor phrasing changes (consider prompt sensitivity in LLMs) or imperceptible perturbations (consider adversarial examples in vision), indicating a lack of robust, human-like understanding.
- *They often can't learn generalizable algorithms.* The continuous, geometric nature of deep learning models makes them fundamentally ill-suited for learning exact, discrete, step-by-step algorithms, such as those that are the bread and butter of classical computer science. Models approximate such processes through interpolation rather than implementing robust, generalizable procedures.

You should always resist the temptation to anthropomorphize deep learning models. Their performance is built on pointwise statistical patterns rather than human-like experiential grounding, making it brittle when encountering deviations from training data.

The narrative that simply scaling up model size and training data would lead to general intelligence has proven insufficient. While scaling enhances performance on

benchmarks that amount to memorization tests, it fails to address the fundamental limitations of deep learning, which stem from the core paradigm of fitting static, interpolative curves to data. Five years of exponential scaling of base LLMs haven't overcome these constraints because the underlying approach remains unchanged.

By 2024, this realization spurred a transition toward test-time adaptation (TTA), where models perform search or fine-tuning during the inference phase to adapt to novel problems. While TTA methods have yielded major breakthroughs, such as OpenAI's o3 surpassing human baseline on ARC-AGI-1 in late 2024, this performance has come at an extreme computational cost. Efficient, human-like adaptation is still a completely open problem, and the slightly harder ARC-AGI-2 benchmark remains completely unsolved as of today. We still need further conceptual advances beyond mere scaling or brute-force search.

20.3 What might lie ahead

Solving human-like fluid intelligence (and ARC-AGI-2) requires moving beyond the limitations inherent in current approaches. While deep learning excels at *value-centric abstraction*, which enables pattern recognition and intuition, it fundamentally lacks capabilities for *program-centric abstraction,* which underpins discrete reasoning, planning, and causal understanding. Human intelligence seamlessly integrates both—future AI must do the same.

Future key developments may include

- *Hybrid models*—Future models will likely integrate learned algorithmic modules (providing reasoning and symbolic manipulation) with deep learning modules (providing perception and intuition). These systems might learn to use programming primitives like control flow, variables, recursion, and complex data structures dynamically.
- *Deep-learning guided program search*—Program synthesis—automatically discovering executable code that meets specifications—offers a route to program-centric abstraction. However, its reliance on inefficient discrete search is a major bottleneck. A crucial advance will be using deep learning to guide this search, utilizing learned intuition about program structure to navigate vast, combinatorial spaces of programs efficiently, much like human developers use experience and intuition to narrow down their choices.
- *Modular recombination and lifelong learning*—We'll move away from monolithic, end-to-end models trained from scratch. Instead, future AI systems will use massive libraries of reusable, modular components that can be repurposed across many problems, acquired from experience. These libraries will feature both "geometric" (deep learning based) and "algorithmic" modules. When faced with a new problem, such AI systems will fetch relevant modules and dynamically recombine them into a new model adapted to the situation at hand. Whenever the system ends up developing a reusable component as a by-product of

this problem-solving loop, the new component would get added to the library, becoming available for every future task the system might encounter.

Ultimately, developing AI that mirrors human-like fluid intelligence will require blending continuous pattern recognition together with discrete, symbolic programs and fully embracing the paradigm of on-the-fly adaptation.

20.4 Staying up to date in a fast-moving field

As final parting words, I want to give you some pointers about how to keep learning and updating your knowledge and skills after you've turned the last page of this book. The field of modern deep learning, as we know it today, is only a few years old, despite a long, slow prehistory stretching back decades. With an exponential increase in financial resources and research headcount since 2013, the field as a whole is now moving at a frenetic pace. What you've learned in this book won't stay relevant forever, and it isn't all you'll need for the rest of your career.

Fortunately, there are plenty of free online resources that you can use to stay up to date and expand your horizons. Here are a few.

20.4.1 Practice on real-world problems using Kaggle

An effective way to acquire real-world experience is to try your hand at machine learning competitions on Kaggle (https://kaggle.com). The only real way to learn is through practice and actual coding—that's the philosophy of this book, and Kaggle competitions are the natural continuation of this. On Kaggle, you'll find an array of constantly renewed data science competitions, many of which involve deep learning, prepared by companies interested in obtaining novel solutions to some of their most challenging machine learning problems. Fairly large monetary prizes are offered to top entrants.

By participating in a few competitions, maybe as part of a team, you'll become more familiar with the practical side of some of the advanced best practices described in this book, especially hyperparameter tuning, avoiding validation-set overfitting, and model ensembling.

20.4.2 Read about the latest developments on arXiv

Deep learning research, in contrast with some other scientific fields, takes place completely in the open. Papers are made publicly and freely accessible as soon as they're finalized, and a lot of related software is open source. arXiv (https://arxiv.org)—pronounced "archive" (the X stands for the Greek *chi*)—is an open access preprint server for physics, mathematics, and computer science research papers. It has become the de facto way to stay up to date on the cutting edge of machine learning and deep learning. The large majority of deep learning researchers upload any paper they write to arXiv shortly after completion. This allows them to plant a flag and claim a specific finding without waiting for a conference acceptance (which takes months), which is necessary given the fast pace of research and the intense competition in the field. It

also allows the field to move extremely fast: all new findings are immediately available for all to see and to build on.

An important downside is that the sheer quantity of new papers posted every day on arXiv makes it impossible to even skim them all, and the fact that they aren't peer-reviewed makes it difficult to identify those that are both important and high quality. It's challenging, and becoming increasingly more so, to find the signal in the noise. But some tools can help: in particular, you can use Google Scholar (https://scholar.google.com) to keep track of publications by your favorite authors.

20.4.3 Explore the Keras ecosystem

With over 2.5 million users as of early 2025 and still growing, Keras has a large ecosystem of tutorials, guides, and related open source projects:

- Your main reference for working with Keras is the online documentation at https://keras.io. In particular, you'll find extensive developer guides at https://keras.io/guides, and you'll find dozens of high-quality Keras code examples at https://keras.io/examples. Make sure to check them out!
- The Keras source code can be found at https://github.com/keras-team/keras, and Keras Hub can be found at https://github.com/keras-team/keras-hub.
- You can follow François (@fchollet) and Matt (@mattdangerw) on X (formerly Twitter).

20.5 Final words

This is the end of *Deep Learning with Python*! I hope you've learned a thing or two about machine learning, deep learning, Keras, and maybe even cognition in general. Learning is a lifelong journey, especially in the field of AI, where we have far more unknowns on our hands than certitudes. So please go on learning, questioning, and researching. Never stop. Because even given the progress made so far, most of the fundamental questions in AI remain unanswered. Many haven't even been properly asked yet.

index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

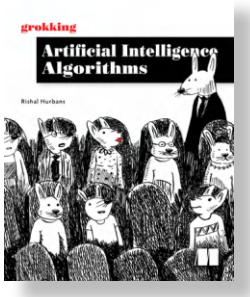
W

X

Y

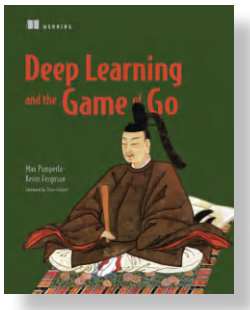
Z

RELATED MANNING TITLES



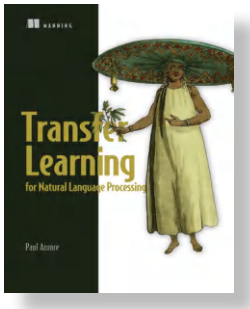
Grokking Artificial Intelligence Algorithms
by Rishal Hurbans

ISBN 9781617296185
392 pages, $59.99
July 2020



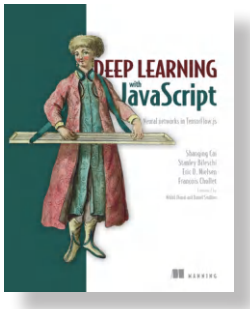
Deep Learning and the Game of Go
by Max Pumperla and Kevin Ferguson
Foreword by Thore Graepel

ISBN: 9781617295324
384 pages, $54.99
January 2019



Transfer Learning for Natural Language Processing
by Paul Azunre

ISBN: 9781617297267
272 pages, $49.99
July 2021



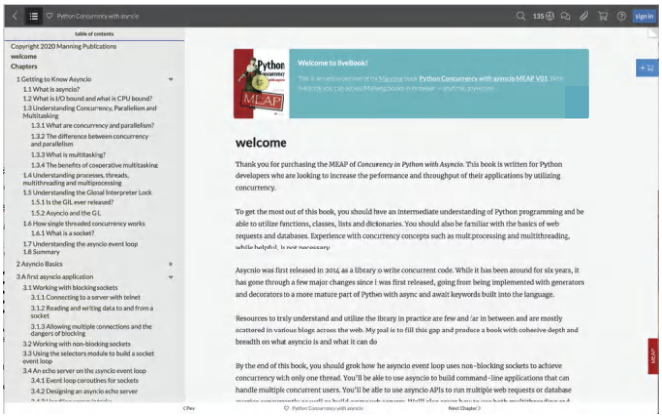
Deep Learning with JavaScript
by Shanqing Cai, Stanley Bileschi, Eric D. Nielsen, and François Chollet
Foreword by Nikhil Thorat and Daniel Smilkov

ISBN 9781617296178
560 pages, $49.99
January 2020

For ordering information, go to www.manning.com

MANNING

liveBook



A new online reading experience

liveBook, our online reading platform, adds a new dimension to your Manning books, with features that make reading, learning, and sharing easier than ever. A liveBook version of your book is included FREE with every Manning book.

This next generation book platform is more than an online reader. It's packed with unique features to upgrade and enhance your learning experience.

- Add your own notes and bookmarks
- One-click code copy
- Learn from other readers in the discussion forum
- Audio recordings and interactive exercises
- Read all your purchased Manning content in any browser, anytime, anywhere

As an added bonus, you can search every Manning book and video in liveBook—even ones you don't yet own. Open any liveBook, and you'll be able to browse the content and read anything you like.*

Find out more at www.manning.com/livebook-program.

**Open reading is limited to 10 minutes per book daily*

MANNING

The Manning Early Access Program

Don't wait to start learning! In MEAP, the Manning Early Access Program, you can read books as they're being created and long before they're available in stores.

Here's how MEAP works.

- **Start now.** Buy a MEAP and you'll get all available chapters in PDF, ePub, Kindle, and liveBook formats.
- **Regular updates.** New chapters are released as soon as they're written. We'll let you know when fresh content is available.
- **Finish faster.** MEAP customers are the first to get final versions of all books! Pre-order the print book, and it'll ship as soon as it's off the press.
- **Contribute to the process.** The feedback you share with authors makes the end product better.
- **No risk.** You get a full refund or exchange if we ever have to cancel a MEAP.

Explore dozens of titles in MEAP at www.manning.com.

Loss, last-layer activation, and metrics to use for different tasks (chapter 6)

Task	Last-layer activation	Loss function	Metrics
Binary classification	Sigmoid	Binary crossentropy	Binary accuracy, ROC AUC
Multiclass, single-label classification	Softmax	Categorical crossentropy	Categorical accuracy, Top-k categorical accuracy, ROC AUC
Multiclass, multi-label classification	Sigmoid	Binary crossentropy	Binary accuracy, ROC AUC
Regression	None	Mean squared error	Mean absolute error

Model architecture for different data types (chapter 20)

Input	Output	Model
Vector data	Class probability, regression value	Densely connected network
Timeseries	Class probability, regression value	RNN, Transformer
Images	Class probability, regression value	ConvNet
Text	Class probability, regression value	Transformer
Text, images	Text	Transformer
Text, images	Images	VAE, diffusion model

PYTHON/DEEP LEARNING

DEEP LEARNING with Python Third Edition

Chollet • Watson

In less than a decade, deep learning has changed the world—twice. First, Python-based libraries like Keras, TensorFlow, and PyTorch elevated neural networks from lab experiments to high-performance production systems deployed at scale. And now, through Large Language Models and other generative AI tools, deep learning is again transforming business and society. In this new edition, Keras creator François Chollet invites you into this amazing subject in the fluid, mentoring style of a true insider.

Deep Learning with Python, Third Edition makes the concepts behind deep learning and generative AI understandable and approachable. This complete rewrite of the bestselling original includes fresh chapters on transformers, building your own GPT-like LLM, and generating images with diffusion models. Each chapter introduces practical projects and code examples that build your understanding of deep learning, layer by layer.

What's Inside

- Hands-on, code-first learning
- Comprehensive, from basics to generative AI
- Intuitive and easy math explanations
- Examples in Keras, PyTorch, JAX, and TensorFlow

For readers with intermediate Python skills. No previous experience with machine learning or linear algebra required.

François Chollet is the co-founder of Ndea and the creator of Keras. **Matthew Watson** is a software engineer at Google working on Gemini and a core maintainer of Keras.

For print book owners, all digital formats are free:
https://www.manning.com/freebook

MANNING

"Perfect for anyone interested in learning by doing from one of the industry greats."
—Anthony Goldbloom
Founder of Kaggle

"A sharp, deeply practical guide that teaches you how to think from first principles to build models that actually work."
—Santiago Valdarrama
Founder of ml.school

"The most up-to-date and complete guide to deep learning you'll find today!"
—Aran Komatsuzaki
EleutherAI

"Masterfully conveys the true essence of neural networks. A rare case in recent years of outstanding technical writing."
—Salvatore Sanfilippo
Creator of Redis

ISBN-13: 978-1-63343-658-9