DEEP LEARNING with Python

SECOND EDITION

François Chollet

MANNING

Deep Learning with Python

Deep Learning with Python

SECOND EDITION

FRANÇOIS CHOLLET

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: Jennifer Stout
Technical development editor: Frances Buontempo
Review editor: Aleksandar Dragosavljević
Production editor: Keri Hales
Copy editor: Andy Carroll
Proofreaders: Katie Tennant and Melody Dolab
Technical proofreader: Karsten Strøbæk
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617296864
Printed in the United States of America

To my son Sylvain: I hope you'll read this book someday!

brief contents

contents

preface

If you've picked up this book, you're probably aware of the extraordinary progress that deep learning has represented for the field of artificial intelligence in the recent past. 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. We're already applying deep learning to an amazing range of important problems across domains as different as medical imaging, agriculture, autonomous driving, education, disaster prevention, and manufacturing.

Yet, I believe deep learning is still in its early days. It has only realized a small fraction of its potential so far. Over time, it will make its way to every problem where it can help—a transformation that will take place over multiple decades.

In order to begin deploying deep learning technology to every problem that it could solve, we need to make it accessible to as many people as possible, including non-experts—people who aren't researchers or graduate students. For deep learning to reach its full potential, we need to radically democratize it. And today, I believe that we're at the cusp of a historical transition, where deep learning is moving out of academic labs and the R&D departments of large tech companies to become a ubiquitous part of the toolbox of every developer out there—not unlike the trajectory of web development in the late 1990s. Almost anyone can now build a website or web app for their business or community of a kind that would have required a small team of specialist engineers in 1998. In the not-so-distant future, anyone with an idea and basic coding skills will be able to build smart applications that learn from data.

When I released the first version of 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 me with my own experiments. But since 2015, hundreds of thousands of newcomers have entered the field of deep learning; many of them picked up Keras as their tool of choice. As I watched scores of smart people use Keras in unexpected, powerful ways, I came to care deeply about the accessibility and democratization of AI. I realized that the further we spread these technologies, the more useful and valuable they become. Accessibility quickly became an explicit goal in the development of Keras, and over a few short years, the Keras developer community has made fantastic achievements on this front. We've put deep learning into the hands of hundreds of thousands 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 to making deep learning available to as many people as possible. Keras had always needed a companion course to simultaneously cover the fundamentals of deep learning, deep learning best practices, and Keras usage patterns. In 2016 and 2017, I did my best to produce such a course, which became the first edition of this book, released in December 2017. It quickly became a machine learning best seller that sold over 50,000 copies and was translated into 12 languages.

However, the field of deep learning advances fast. Since the release of the first edition, many important developments have taken place—the release of TensorFlow 2, the growing popularity of the Transformer architecture, and more. And so, in late 2019, I set out to update my book. I originally thought, quite naively, that it would feature about 50% new content and would end up being roughly the same length as the first edition. In practice, after two years of work, it turned out to be over a third longer, with about 75% novel content. More than a refresh, it is a whole new book.

I wrote it with a focus on making the concepts behind deep learning, and their implementation, as approachable as possible. Doing so didn't require me to dumb down anything—I strongly believe that there are no difficult ideas in deep learning. I hope you'll find this book valuable and that it will enable you to begin building intelligent applications and solve the problems that matter to you.

acknowledgments

First of all, I'd like to thank the Keras community for making this book possible. Over the past six years, Keras has grown to have hundreds of open source contributors and more than one million users. Your contributions and feedback have turned Keras into what it is today.

On a more personal note, I'd like to thank my wife for her endless support during the development of Keras and the writing of this book.

I'd also like to thank Google for backing the Keras project. It has been fantastic to see Keras adopted as TensorFlow's high-level API. A smooth integration between Keras and TensorFlow greatly benefits both TensorFlow users and Keras users, and makes deep learning accessible to most.

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

Many thanks go to the technical peer reviewers: Billy O'Callaghan, Christian Weisstanner, Conrad Taylor, Daniela Zapata Riesco, David Jacobs, Edmon Begoli, Edmund Ronald PhD, Hao Liu, Jared Duncan, Kee Nam, Ken Fricklas, Kjell Jansson, Milan Šarenac, Nguyen Cao, Nikos Kanakaris, Oliver Korten, Raushan Jha, Sayak Paul, Sergio Govoni, Shashank Polasa, Todd Cook, and Viton Vitanis—and all the other people who sent us feedback on the draft on the book.

On the technical side, special thanks go to Frances Buontempo, who served as the book's technical editor, and Karsten Strøbæk, 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, then working up to state-of-the-art techniques. 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 machine learning and deep learning via detailed code snippets and intuitive mental models. You'll learn from abundant code examples that include extensive commentary, practical recommendations, and simple high-level explanations of everything you need to know to start using deep learning to solve concrete problems.

The code examples use the Python deep learning framework Keras, with TensorFlow 2 as its numerical engine. They demonstrate modern Keras and TensorFlow 2 best practices as of 2021.

After reading this book, you'll have a solid understand 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. You'll be able to use Keras to tackle real-world problems ranging from computer vision to natural language processing: image classification, image segmentation, timeseries forecasting, text classification, machine translation, text generation, and more.

Who should read this book

This book is written for people with 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.

In order to understand the code examples, you'll need reasonable Python proficiency. Additionally, familiarity with the NumPy library will be helpful, although it isn't required. 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 in order to follow along.

About the code

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

All code examples in this book are available from the Manning website at https://www.manning.com/books/deep-learning-with-python-second-edition, and as Jupyter notebooks on GitHub at https://github.com/fchollet/deep-learning-with-python-notebooks. They can be run directly in your browser via Google Colaboratory, a hosted Jupyter notebook environment that you can use for free. An internet connection and a desktop web browser are all you need to get started with deep learning.

liveBook discussion forum

Purchase of *Deep Learning with Python*, second edition, includes free access to a private web forum run by Manning Publications where you can make comments about the

book, ask technical questions, and receive help from the author and from other users. To access the forum, go to https://livebook.manning.com/#!/book/deep-learning-with-python-second-edition/discussion. You can also learn more about Manning's forums and the rules of conduct at https://livebook.manning.com/#!/discussion.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his 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 author

FRANÇOIS CHOLLET is the creator of Keras, one of the most widely used deep learning frameworks. He is currently a software engineer at Google, where he leads the Keras team. In addition, he does research on abstraction, reasoning, and how to achieve greater generality in artificial intelligence.

about the cover illustration

The figure on the cover of *Deep Learning with Python,* second 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* (four volumes), London, published between 1757 and 1772. The title page states that these are hand-colored copperplate engravings, heightened with gum arabic.

Thomas Jefferys (1719–1771) was called "Geographer to King George III." He was an English cartographer who was the leading map supplier of his day. He engraved and printed maps for government and other official bodies and produced a wide range of commercial maps and atlases, especially of North America. His work as a map maker sparked an interest in local dress customs of the lands he surveyed and mapped, which are brilliantly displayed in this collection. Fascination with faraway lands and travel for pleasure were relatively new phenomena in the late eighteenth century, and collections such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other countries.

The diversity of the drawings in Jefferys' volumes speaks vividly of the uniqueness and individuality of the world's nations some 200 years ago. Dress codes have changed since then, and the diversity by region and country, so rich at the time, has faded away. It's now often hard to tell the inhabitants of one continent from another. Perhaps, trying to view it optimistically, we've traded a cultural and visual diversity for a more varied personal life—or a more varied and interesting intellectual and technical life.

At a time when it's difficult to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Jefferys' pictures.

1 What is deep learning?

This chapter covers

- High-level definitions of fundamental concepts
- Timeline of the development of machine learning
- Key factors behind deep learning's rising popularity and future potential

In the past few years, 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 future or current 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 a future in which you have an active role to play: after reading this book, you'll be one of those who develop those 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?

This chapter provides essential context around artificial intelligence, machine learning, and deep learning.

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 (see figure 1.1)? How do they relate to each other?

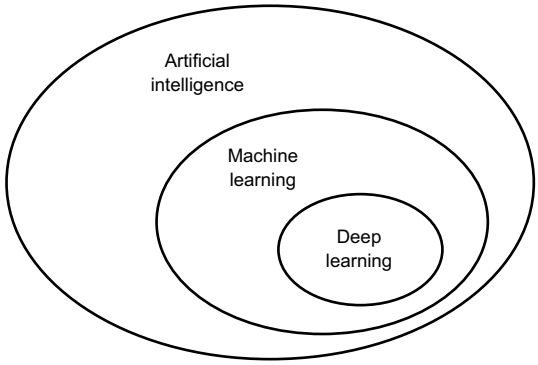


Figure 1.1 Artificial intelligence, machine learning, and deep learning

1.1.1 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*.

1.1.2 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 178 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* as well as key concepts that would come to shape AI.[2] Turing

[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.

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 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 (see figure 1.2). 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.

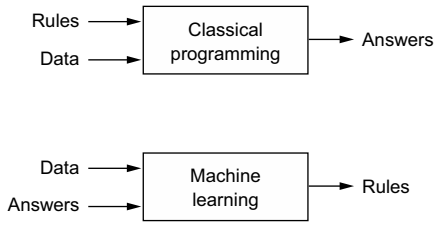


Figure 1.2 Machine learning: a new programming paradigm

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.

1.1.3 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 for executing 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 in order 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.

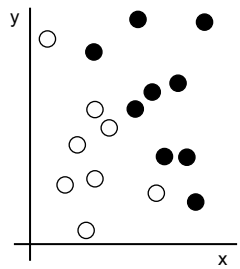


Figure 1.3 Some sample data

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 the coordinates (*x*, *y*) of a point and output whether that point is likely to be black or to be 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.

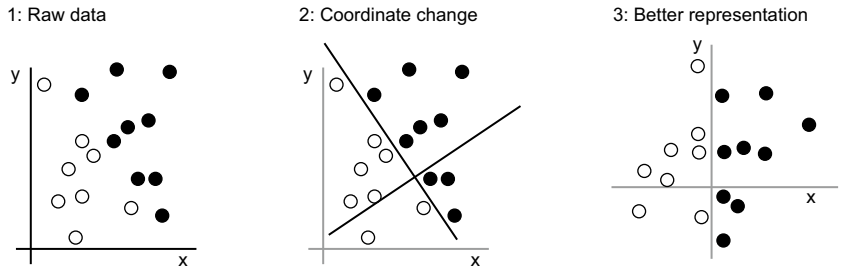


Figure 1.4 Coordinate change

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 down 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 handwriting?

This is possible to an extent. Rules based on representations of digits such as "number of closed loops" or vertical and horizontal pixel histograms can do a decent job of telling apart handwritten digits. But finding such useful representations by hand is hard work, and, as you can imagine, the resulting rule-based system is brittle—a nightmare to maintain. Every time you come across a new example of handwriting that breaks your carefully thought-out rules, you will 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 by using as feedback the percentage of digits being correctly classified in some development dataset? 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 taking a histogram of pixels and counting loops (like in our digits classification example), but they could also be linear projections, translations, 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 for solving a remarkably broad range of intellectual tasks, from speech recognition to autonomous driving.

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

1.1.4 The "deep" in "deep learning"

Deep learning is a specific subfield of machine learning: a new take on learning representations from data that puts an emphasis on 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" refers 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 was 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 in order to recognize what digit it is.

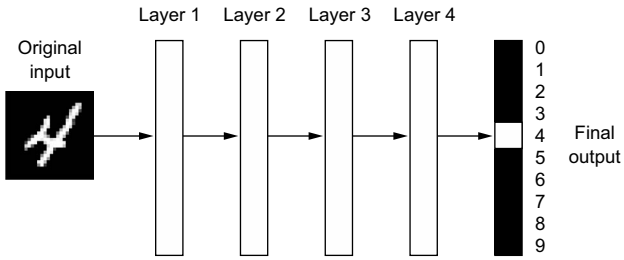


Figure 1.5 A deep neural network for digit classification

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).

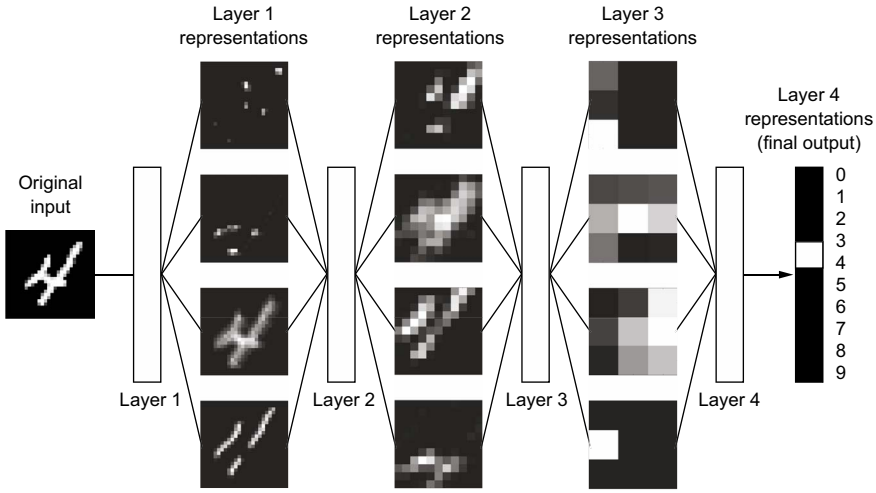


Figure 1.6 Data representations learned by a digit-classification model

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.

1.1.5 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 input 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 values 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!

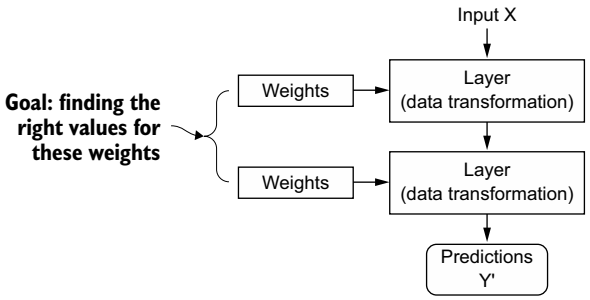


Figure 1.7 A neural network is parameterized by its weights.

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).

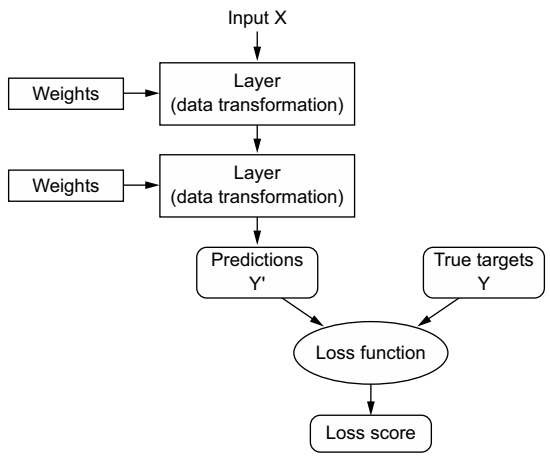


Figure 1.8 A loss function measures the quality of the network's output.

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.

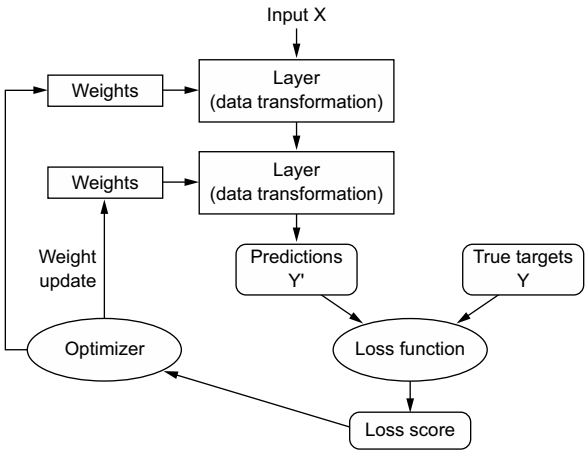


Figure 1.9 The loss score is used as a feedback signal to adjust the weights.

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 number of times (typically tens of iterations 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.

1.1.6 What deep learning has achieved so far

Although deep learning is a fairly old subfield of machine learning, it only rose to prominence in the early 2010s. In the few years since, it has achieved nothing short of a revolution in the field, producing remarkable results on perceptual tasks and even natural language processing tasks—problems involving skills that seem natural and intuitive to humans but have long been elusive for machines.

In particular, deep learning has enabled the following breakthroughs, all in historically difficult areas of machine learning:

- Near-human-level image classification
- Near-human-level speech transcription
- Near-human-level handwriting transcription

- Dramatically improved machine translation
- Dramatically improved text-to-speech conversion
- Digital assistants such as Google Assistant and Amazon Alexa
- Near-human-level autonomous driving
- Improved ad targeting, as used by Google, Baidu, or Bing
- Improved search results on the web
- Ability to answer natural language questions
- Superhuman Go 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's Apostolic 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, or even earthquakes, and so on. 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.

1.1.7 Don't believe the short-term hype

Although deep learning has led to remarkable achievements in recent years, expectations for what the field will be able to achieve in the next decade tend to run much higher than what will likely be possible. Although some world-changing applications like autonomous cars are already within reach, many more are likely to remain elusive for a long time, such as believable dialogue systems, human-level machine translation across arbitrary languages, and human-level natural language understanding. In particular, talk of human-level general intelligence shouldn't be taken too seriously. The risk with high expectations for the short term is that, as technology fails to deliver, research investment will 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 2021 such an achievement still appears to be far in the future—so far 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. It's best to moderate our expectations for the short term and make sure people less familiar with the technical side of the field have a clear idea of what deep learning can and can't deliver.

1.1.8 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. AI research has been moving forward amazingly quickly in the past ten years, in large part due to a level of funding never before seen in the short history of AI, but so far relatively little of this progress has made its way into the products and processes that form our world. Most of the research findings of deep learning aren't yet applied, or at least are not applied to the full range of problems they could solve across all industries. Your doctor doesn't yet use AI, and neither does your accountant. You probably don't use AI technologies very often in your day-to-day life. Of course, you can ask your smartphone simple questions and get reasonable answers, you can get fairly useful product recommendations on Amazon.com, and you can search for "birthday" on Google Photos and instantly find those pictures of your daughter's birthday party from last month. That's a far cry from where such technologies used to stand. But such tools are still only accessories to our daily lives. AI has yet to transition to being central to the way we work, think, and live.

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.

On the way, we may face a few setbacks and maybe even a new AI winter—in much the same way the internet industry was overhyped in 1998–99 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.

1.2 Before deep learning: A brief history of machine learning

Deep learning has reached a level of public attention and industry investment never before seen in the history of AI, but it isn't the first successful form of machine learning. It's safe to say that most of the machine learning algorithms used in the industry today aren't deep learning algorithms. Deep learning isn't always the right tool for the job—sometimes there isn't enough data for deep learning to be applicable, and sometimes the problem is better solved by a different algorithm. If deep learning is your first contact with machine learning, you may find yourself in a situation where all you have is the deep learning hammer, and every machine learning problem starts to look like a nail. The only way not to fall into this trap is to be familiar with other approaches and practice them when appropriate.

A detailed discussion of classical machine learning approaches is outside of the scope of this book, but I'll briefly go over them and describe the historical context in which they were developed. This will allow us to place deep learning in the broader context of machine learning and better understand where deep learning comes from and why it matters.

1.2.1 Probabilistic modeling

Probabilistic modeling is the application of the principles of statistics to data analysis. It is one of the earliest forms of machine learning, and it's still widely used to this day. One of the best-known algorithms in this category is the Naive Bayes algorithm.

Naive Bayes is a type of machine learning classifier based on applying Bayes' theorem while assuming that the features in the input data are all independent (a strong, or "naive" assumption, which is where the name comes from). This form of data analysis predates computers and was applied by hand decades before its first computer implementation (most likely dating back to the 1950s). Bayes' theorem and the foundations of statistics date back to the eighteenth century, and these are all you need to start using Naive Bayes classifiers.

A closely related model is *logistic regression* (logreg for short), which is sometimes considered to be the "Hello World" of modern machine learning. Don't be misled by its name—logreg is a classification algorithm rather than a regression algorithm. Much

like Naive Bayes, logreg predates computing by a long time, yet it's still useful to this day, thanks to its simple and versatile nature. It's often the first thing a data scientist will try on a dataset to get a feel for the classification task at hand.

1.2.2 Early neural networks

Early iterations of neural networks have been completely supplanted by the modern variants covered in these pages, but it's helpful to be aware of how deep learning originated. Although the core ideas of neural networks were investigated in toy forms as early as the 1950s, the approach took decades to get started. For a long time, the missing piece was an efficient way to train large neural networks. This changed in the mid-1980s, when multiple people independently rediscovered the Backpropagation algorithm—a way to train chains of parametric operations using gradient-descent optimization (we'll precisely define these concepts later in the book)—and started applying it to neural networks.

The first successful practical application of neural nets came in 1989 from Bell Labs, when Yann LeCun combined the earlier ideas of convolutional neural networks and backpropagation, and applied them to the problem of classifying handwritten digits. The resulting network, dubbed *LeNet*, was used by the United States Postal Service in the 1990s to automate the reading of ZIP codes on mail envelopes.

1.2.3 Kernel methods

As neural networks started to gain some respect among researchers in the 1990s, thanks to this first success, a new approach to machine learning rose to fame and quickly sent neural nets back to oblivion: kernel methods. *Kernel methods* are a group of classification algorithms, the best known of which is the *Support Vector Machine* (SVM). The modern formulation of an SVM was developed by Vladimir Vapnik and Corinna Cortes in the early 1990s at Bell Labs and published in 1995,[3] although an older linear formulation was published by Vapnik and Alexey Chervonenkis as early as 1963.[4]

SVM is a classification algorithm that works by finding "decision boundaries" separating two classes (see figure 1.10). SVMs proceed to find these boundaries in two steps:

1. The data is mapped to a new high-dimensional representation where the decision boundary can be expressed as a hyperplane (if the data was two-dimensional, as in figure 1.10, a hyperplane would be a straight line).
2. A good decision boundary (a separation hyperplane) is computed by trying to maximize the distance between the hyperplane and the closest data points from each class, a

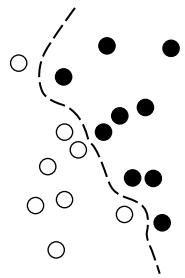


Figure 1.10
A decision boundary

[3] Vladimir Vapnik and Corinna Cortes, "Support-Vector Networks," *Machine Learning* 20, no. 3 (1995): 273–297.
[4] Vladimir Vapnik and Alexey Chervonenkis, "A Note on One Class of Perceptrons," *Automation and Remote Control* 25 (1964).

step called *maximizing the margin*. This allows the boundary to generalize well to new samples outside of the training dataset.

The technique of mapping data to a high-dimensional representation where a classification problem becomes simpler may look good on paper, but in practice it's often computationally intractable. That's where the *kernel trick* comes in (the key idea that kernel methods are named after). Here's the gist of it: to find good decision hyperplanes in the new representation space, you don't have to explicitly compute the coordinates of your points in the new space; you just need to compute the distance between pairs of points in that space, which can be done efficiently using a kernel function. A *kernel function* is a computationally tractable operation that maps any two points in your initial space to the distance between these points in your target representation space, completely bypassing the explicit computation of the new representation. Kernel functions are typically crafted by hand rather than learned from data—in the case of an SVM, only the separation hyperplane is learned.

At the time they were developed, SVMs exhibited state-of-the-art performance on simple classification problems and were one of the few machine learning methods backed by extensive theory and amenable to serious mathematical analysis, making them well understood and easily interpretable. Because of these useful properties, SVMs became extremely popular in the field for a long time.

But SVMs proved hard to scale to large datasets and didn't provide good results for perceptual problems such as image classification. Because an SVM is a shallow method, applying an SVM to perceptual problems requires first extracting useful representations manually (a step called *feature engineering*), which is difficult and brittle. For instance, if you want to use an SVM to classify handwritten digits, you can't start from the raw pixels; you should first find by hand useful representations that make the problem more tractable, like the pixel histograms I mentioned earlier.

1.2.4 Decision trees, random forests, and gradient boosting machines

Decision trees are flowchart-like structures that let you classify input data points or predict output values given inputs (see figure 1.11). They're easy to visualize and interpret. Decision trees learned from data began to receive significant research interest in the 2000s, and by 2010 they were often preferred to kernel methods.

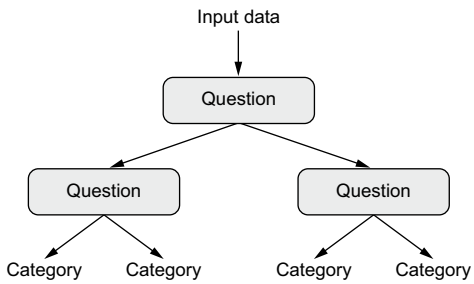


Figure 1.11 A decision tree: the parameters that are learned are the questions about the data. A question could be, for instance, "Is coefficient 2 in the data greater than 3.5?"

In particular, the *Random Forest* algorithm introduced a robust, practical take on decision-tree learning that involves building a large number of specialized decision trees and then ensembling their outputs. Random forests are applicable to a wide range of problems—you could say that they're almost always the second-best algorithm for any shallow machine learning task. When the popular machine learning competition website Kaggle (http://kaggle.com) got started in 2010, random forests quickly became a favorite on the platform—until 2014, when *gradient boosting machines* took over. A gradient boosting machine, much like a random forest, is a machine learning technique based on ensembling weak prediction models, generally decision trees. It uses *gradient boosting*, a way to improve any machine learning model by iteratively training new models that specialize in addressing the weak points of the previous models. Applied to decision trees, the use of the gradient boosting technique results in models that strictly outperform random forests most of the time, while having similar properties. It may be one of the best, if not *the* best, algorithm for dealing with nonperceptual data today. Alongside deep learning, it's one of the most commonly used techniques in Kaggle competitions.

1.2.5 Back to neural networks

Around 2010, although neural networks were almost completely shunned by the scientific community at large, a number of people still working on neural networks started to make important breakthroughs: the groups of Geoffrey Hinton at the University of Toronto, Yoshua Bengio at the University of Montreal, Yann LeCun at New York University, and IDSIA in Switzerland.

In 2011, Dan Ciresan from IDSIA began to win academic image-classification competitions with GPU-trained deep neural networks—the first practical success of modern deep learning. But the watershed moment came in 2012, with the entry of Hinton's group in the yearly large-scale image-classification challenge ImageNet (ImageNet Large Scale Visual Recognition Challenge, or ILSVRC for short). The ImageNet challenge was notoriously difficult at the time, consisting of classifying high-resolution color images into 1,000 different categories after training on 1.4 million images. In 2011, the top-five accuracy of the winning model, based on classical approaches to computer vision, was only 74.3%.[5] Then, in 2012, a team led by Alex Krizhevsky and advised by Geoffrey Hinton was able to achieve a top-five accuracy of 83.6%—a significant breakthrough. The competition has been dominated by deep convolutional neural networks every year since. By 2015, the winner reached an accuracy of 96.4%, and the classification task on ImageNet was considered to be a completely solved problem.

Since 2012, deep convolutional neural networks (*convnets*) have become the go-to algorithm for all computer vision tasks; more generally, they work on all perceptual

[5] "Top-five accuracy" measures how often the model selects the correct answer as part of its top five guesses (out of 1,000 possible answers, in the case of ImageNet).

tasks. At any major computer vision conference after 2015, it was nearly impossible to find presentations that didn't involve convnets in some form. At the same time, deep learning has also found applications in many other types of problems, such as natural language processing. It has completely replaced SVMs and decision trees in a wide range of applications. For instance, for several years, the European Organization for Nuclear Research, CERN, used decision tree–based methods for analyzing particle data from the ATLAS detector at the Large Hadron Collider (LHC), but CERN eventually switched to Keras-based deep neural networks due to their higher performance and ease of training on large datasets.

1.2.6 What makes deep learning different

The primary reason deep learning took off so quickly is that it offered better performance for many problems. But that's not the only reason. Deep learning also makes problem-solving much easier, because it completely 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, usually via simple transformations such as high-dimensional non-linear projections (SVMs) or decision trees. But the refined representations required by complex problems generally can't be attained by such techniques. 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 layers of 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.

You may ask, if the crux of the issue is to have multiple successive layers of representations, could shallow methods be applied repeatedly to emulate the effects of deep learning? In practice, successive applications of shallow-learning methods produce fast-diminishing returns, because the optimal first representation layer in a three-layer model isn't the optimal first layer in a one-layer or two-layer model. What is transformative about deep learning is that it allows a model to learn all layers of representation *jointly*, at the same time, rather than in succession (*greedily*, as it's called). With joint feature learning, whenever the model adjusts one of its internal features, all other features that depend on it automatically adapt to the change, without requiring human intervention. Everything is supervised by a single feedback signal: every change in the model serves the end goal. This is much more powerful than greedily stacking shallow models, because it allows for complex, abstract representations to be learned by breaking them down into long series of intermediate spaces (layers); each space is only a simple transformation away from the previous one.

These are the two essential characteristics of how deep learning learns from data: the *incremental, layer-by-layer way in which increasingly complex representations are developed*,

and the fact that *these intermediate incremental representations are learned jointly*, each layer being updated to follow both the representational needs of the layer above and the needs of the layer below. Together, these two properties have made deep learning vastly more successful than previous approaches to machine learning.

1.2.7 The modern machine learning landscape

A great way to get a sense of the current landscape of machine learning algorithms and tools is to look at machine learning competitions on Kaggle. Due to its highly competitive environment (some contests have thousands of entrants and million-dollar prizes) and to the wide variety of machine learning problems covered, Kaggle offers a realistic way to assess what works and what doesn't. So what kind of algorithm is reliably winning competitions? What tools do top entrants use?

In early 2019, Kaggle ran a survey asking teams that ended in the top five of any competition since 2017 which primary software tool they had used in the competition (see figure 1.12). It turns out that top teams tend to use either deep learning methods (most often via the Keras library) or gradient boosted trees (most often via the LightGBM or XGBoost libraries).

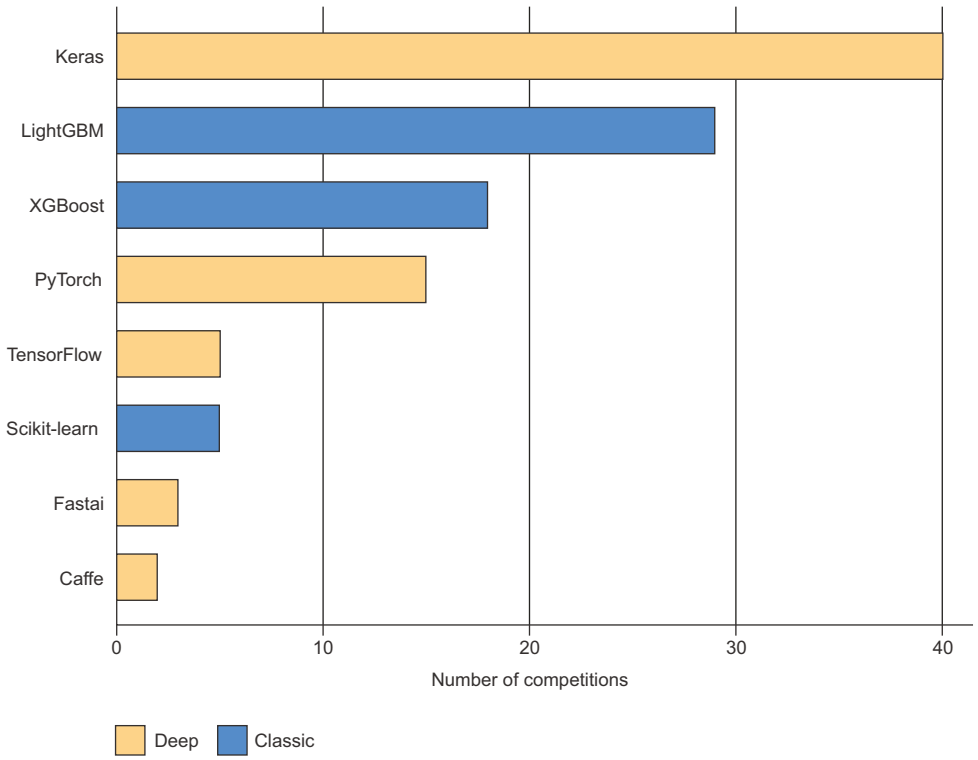


Figure 1.12 Machine learning tools used by top teams on Kaggle

It's not just competition champions, either. Kaggle also runs a yearly survey among machine learning and data science professionals worldwide. With tens of thousands of respondents, this survey is one of our most reliable sources about the state of the industry. Figure 1.13 shows the percentage of usage of different machine learning software frameworks.

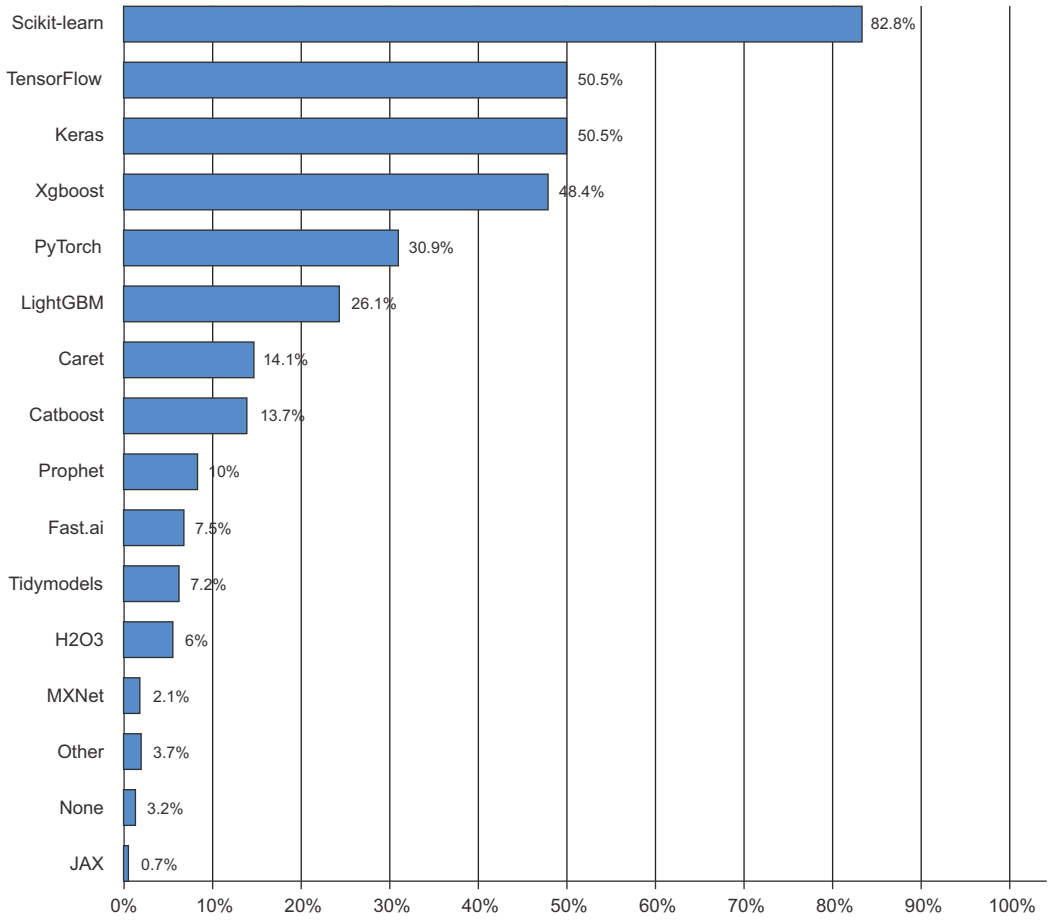


Figure 1.13 Tool usage across the machine learning and data science industry (Source: www.kaggle.com/kaggle-survey-2020)

From 2016 to 2020, the entire machine learning and data science industry has been dominated by these two approaches: deep learning and gradient boosted trees. Specifically, gradient boosted trees is used for problems where structured data is available, whereas deep learning is used for perceptual problems such as image classification.

Users of gradient boosted trees tend to use Scikit-learn, XGBoost, or LightGBM. Meanwhile, most practitioners of deep learning use Keras, often in combination with

its parent framework TensorFlow. The common point of these tools is they're all Python libraries: Python is by far the most widely used language for machine learning and data science.

These are the two techniques you should be the most familiar with in order to be successful in applied machine learning today: gradient boosted trees, for shallow-learning problems; and deep learning, for perceptual problems. In technical terms, this means you'll need to be familiar with Scikit-learn, XGBoost, and Keras—the three libraries that currently dominate Kaggle competitions. With this book in hand, you're already one big step closer.

1.3 Why deep learning? Why now?

The two key ideas of deep learning for computer vision—convolutional neural networks and backpropagation—were already well understood by 1990. The Long Short-Term Memory (LSTM) algorithm, which is fundamental to deep learning for timeseries, was developed in 1997 and has barely changed since. So why did deep learning only take off after 2012? What changed in these two decades?

In general, three technical forces are driving advances in machine learning:

- Hardware
- Datasets and benchmarks
- Algorithmic advances

Because the field is guided by experimental findings rather than by theory, algorithmic advances only become possible when appropriate data and hardware are available to try new ideas (or to scale up old ideas, as is often the case). Machine learning isn't mathematics or physics, where major advances can be done with a pen and a piece of paper. It's an engineering science.

The real bottlenecks throughout the 1990s and 2000s were data and hardware. But here's what happened during that time: the internet took off and high-performance graphics chips were developed for the needs of the gaming market.

1.3.1 Hardware

Between 1990 and 2010, off-the-shelf CPUs became faster by a factor of approximately 5,000. As a result, nowadays it's possible to run small deep learning models on your laptop, whereas this would have been intractable 25 years ago.

But typical deep learning models used in computer vision or speech recognition require orders of magnitude more computational power than your laptop can deliver. Throughout the 2000s, companies like NVIDIA and AMD invested billions of dollars in developing fast, massively parallel chips (graphical processing units, or GPUs) to power the graphics of increasingly photorealistic video games—cheap, single-purpose supercomputers designed to render complex 3D scenes on your screen in real time. This investment came to benefit the scientific community when, in 2007, NVIDIA launched CUDA (https://developer.nvidia.com/about-cuda), a programming interface

for its line of GPUs. A small number of GPUs started replacing massive clusters of CPUs in various highly parallelizable applications, beginning with physics modeling. Deep neural networks, consisting mostly of many small matrix multiplications, are also highly parallelizable, and around 2011 some researchers began to write CUDA implementations of neural nets—Dan Ciresan[6] and Alex Krizhevsky[7] were among the first.

What happened is that the gaming market subsidized supercomputing for the next generation of artificial intelligence applications. Sometimes, big things begin as games. Today, the NVIDIA Titan RTX, a GPU that cost $2,500 at the end of 2019, can deliver a peak of 16 teraFLOPS in single precision (16 trillion `float32` operations per second). That's about 500 times more computing power than the world's fastest supercomputer from 1990, the Intel Touchstone Delta. On a Titan RTX, it takes only a few hours to train an ImageNet model of the sort that would have won the ILSVRC competition around 2012 or 2013. Meanwhile, large companies train deep learning models on clusters of hundreds of GPUs.

What's more, the deep learning industry has been moving beyond GPUs and is investing in increasingly specialized, efficient chips for deep learning. In 2016, at its annual I/O convention, Google revealed its Tensor Processing Unit (TPU) project: a new chip design developed from the ground up to run deep neural networks significantly faster and far more energy efficient than top-of-the-line GPUs. Today, in 2020, the third iteration of the TPU card represents 420 teraFLOPS of computing power. That's 10,000 times more than the Intel Touchstone Delta from 1990.

These TPU cards are designed to be assembled into large-scale configurations, called "pods." One pod (1024 TPU cards) peaks at 100 petaFLOPS. For scale, that's about 10% of the peak computing power of the current largest supercomputer, the IBM Summit at Oak Ridge National Lab, which consists of 27,000 NVIDIA GPUs and peaks at around 1.1 exaFLOPS.

1.3.2 Data

AI is sometimes heralded as the new industrial revolution. If deep learning is the steam engine of this revolution, then data is its coal: the raw material that powers our intelligent machines, without which nothing would be possible. When it comes to data, in addition to the exponential progress in storage hardware over the past 20 years (following Moore's law), the game changer has been the rise of the internet, making it feasible to collect and distribute very large datasets for machine learning. Today, large companies work with image datasets, video datasets, and natural language datasets that couldn't have been collected without the internet. User-generated image tags on Flickr, for

---

[6] See "Flexible, High Performance Convolutional Neural Networks for Image Classification," *Proceedings of the 22nd International Joint Conference on Artificial Intelligence* (2011), www.ijcai.org/Proceedings/11/Papers/210.pdf.

[7] See "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems* 25 (2012), http://mng.bz/2286.

instance, have been a treasure trove of data for computer vision. So are YouTube videos. And Wikipedia is a key dataset for natural language processing.

If there's one dataset that has been a catalyst for the rise of deep learning, it's the ImageNet dataset, consisting of 1.4 million images that have been hand annotated with 1,000 image categories (one category per image). But what makes ImageNet special isn't just its large size, but also the yearly competition associated with it.[8]

As Kaggle has been demonstrating since 2010, public competitions are an excellent way to motivate researchers and engineers to push the envelope. Having common benchmarks that researchers compete to beat has greatly helped the rise of deep learning, by highlighting its success against classical machine learning approaches.

1.3.3 Algorithms

In addition to hardware and data, until the late 2000s, we were missing a reliable way to train very deep neural networks. As a result, neural networks were still fairly shallow, using only one or two layers of representations; thus, they weren't able to shine against more-refined shallow methods such as SVMs and random forests. The key issue was that of *gradient propagation* through deep stacks of layers. The feedback signal used to train neural networks would fade away as the number of layers increased.

This changed around 2009–2010 with the advent of several simple but important algorithmic improvements that allowed for better gradient propagation:

- Better *activation functions* for neural layers
- Better *weight-initialization schemes*, starting with layer-wise pretraining, which was then quickly abandoned
- Better *optimization schemes*, such as RMSProp and Adam

Only when these improvements began to allow for training models with 10 or more layers did deep learning start to shine.

Finally, in 2014, 2015, and 2016, even more advanced ways to improve gradient propagation were discovered, such as batch normalization, residual connections, and depthwise separable convolutions.

Today, we can train models that are arbitrarily deep from scratch. This has unlocked the use of extremely large models, which hold considerable representational power—that is to say, which encode very rich hypothesis spaces. This extreme scalability is one of the defining characteristics of modern deep learning. Large-scale model architectures, which feature tens of layers and tens of millions of parameters, have brought about critical advances both in computer vision (for instance, architectures such as ResNet, Inception, or Xception) and natural language processing (for instance, large Transformer-based architectures such as BERT, GPT-3, or XLNet).

[8] The ImageNet Large Scale Visual Recognition Challenge (ILSVRC), www.image-net.org/challenges/LSVRC.

1.3.4 A new wave of investment

As deep learning became the new state of the art for computer vision in 2012–2013, and eventually for all perceptual tasks, industry leaders took note. What followed was a gradual wave of industry investment far beyond anything previously seen in the history of AI (see figure 1.14).

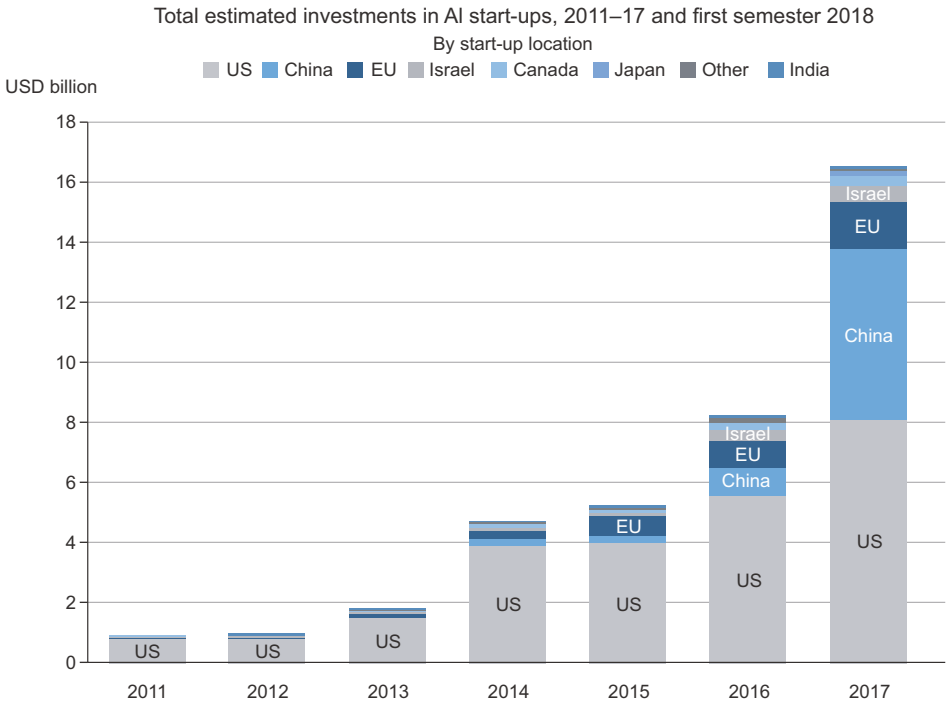


Figure 1.14 OECD estimate of total investments in AI startups (Source: http://mng.bz/zGN6)

In 2011, right before deep learning took the spotlight, the total venture capital investment in AI worldwide was less than a billion dollars, which went almost entirely to practical applications of shallow machine learning approaches. In 2015, it had risen to over $5 billion, and in 2017, to a staggering $16 billion. Hundreds of startups launched in these few years, trying to capitalize on the deep learning hype. Meanwhile, large tech companies such as Google, Amazon, and Microsoft have invested in internal research departments in amounts that would most likely dwarf the flow of venture-capital money.

Machine learning—in particular, deep learning—has become central to the product strategy of these tech giants. In late 2015, Google CEO Sundar Pichai stated, "Machine learning is a core, transformative way by which we're rethinking how we're

doing everything. We're thoughtfully applying it across all our products, be it search, ads, YouTube, or Play. And we're in early days, but you'll see us—in a systematic way—apply machine learning in all these areas." [9]

As a result of this wave of investment, the number of people working on deep learning went from a few hundred to tens of thousands in less than 10 years, and research progress has reached a frenetic pace.

1.3.5 The democratization of deep learning

One of the key factors driving this inflow of new faces in deep learning has been the democratization of the toolsets used in the field. In the early days, doing deep learning required significant C++ and CUDA expertise, which few people possessed.

Nowadays, basic Python scripting skills suffice to do advanced deep learning research. This has been driven most notably by the development of the now-defunct Theano library, and then the TensorFlow library—two symbolic tensor-manipulation frameworks for Python that support autodifferentiation, greatly simplifying the implementation of new models—and by the rise of user-friendly libraries such as Keras, which makes deep learning as easy as manipulating LEGO bricks. After its release in early 2015, Keras quickly became the go-to deep learning solution for large numbers of new startups, graduate students, and researchers pivoting into the field.

1.3.6 Will it last?

Is there anything special about deep neural networks that makes them the "right" approach for companies to be investing in and for researchers to flock to? Or is deep learning just a fad that may not last? 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 two 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 removes the need for feature engineering, replacing complex, brittle, engineering-heavy pipelines with simple, end-to-end trainable models that are typically built using only five or six different tensor operations.
- *Scalability*—Deep learning is highly amenable to parallelization on GPUs or TPUs, 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

[9] Sundar Pichai, Alphabet earnings call, Oct. 22, 2015.

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: for instance, it's possible to take a deep learning model trained for image classification and drop it into a video-processing pipeline. This allows us to reinvest previous work into increasingly complex and powerful models. This also makes deep learning applicable to fairly small datasets.

Deep learning has only been in the spotlight for a few years, and we may not yet have established the full scope of what it can do. With every passing year, we learn about new use cases and engineering improvements that lift previous limitations. Following a scientific revolution, progress generally follows a sigmoid curve: it starts with a period of fast progress, which gradually stabilizes as researchers hit hard limitations, and then further improvements become incremental.

When I was writing the first edition of this book, in 2016, I predicted that deep learning was still in the first half of that sigmoid, with much more transformative progress to come in the following few years. This has proven true in practice, as 2017 and 2018 have seen the rise of Transformer-based deep learning models for natural language processing, which have been a revolution in the field, while deep learning also kept delivering steady progress in computer vision and speech recognition. Today, in 2021, deep learning seems to have entered the second half of that sigmoid. We should still expect significant progress in the years to come, but we're probably out of the initial phase of explosive progress.

Today, I'm extremely excited about the deployment of deep learning technology to every problem it can solve—the list is endless. Deep learning is still a revolution in the making, and it will take many years to realize its full potential.

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 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 Keras and TensorFlow in chapter 3.

2.1 A first look at a neural network

Let's look at a concrete example of a neural network that uses the Python library Keras to learn to classify handwritten digits. 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 the next chapter, we'll review each element in the example and explain them 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.

Figure 2.1 MNIST sample digits

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*.

You don't need to try to reproduce this example on your machine just now. If you wish to, you'll first need to set up a deep learning workspace, which is covered in chapter 3.

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 tensorflow.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.

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

```
from tensorflow import keras
from tensorflow.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:

- *An optimizer*—The mechanism through which the model will update itself based on the training data it sees, so as to improve its performance.
- *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.
- *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="rmsprop",
              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'll 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)
Epoch 1/5
60000/60000 [==============================] - 5s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [========================>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

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]` belongs 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 quite a bit lower than the training-set accuracy (98.9%). 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 3.

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 TensorFlow 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 be already 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*).

2.2.1 Scalars (rank-0 tensors)

A tensor that contains only one number is called a *scalar* (or scalar tensor, or 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 is said to have exactly one axis. 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 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. 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 or TensorFlow.
- *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`, 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 tensorflow.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 Matplotlib library (a well-known Python data visualization library, which comes preinstalled in Colab); see figure 2.2.

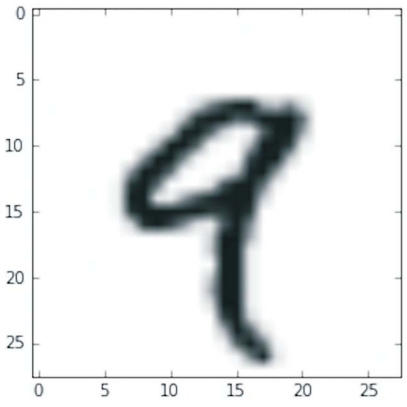


Figure 2.2 The fourth sample in our dataset

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()
```

Naturally, the corresponding label is 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, in order 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. In order to crop the images to patches of 14 × 14 pixels centered in the middle, you'd 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* (sometimes called the *samples dimension*). 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. 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 *n*th 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*. This is a term you'll frequently encounter 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

2.2.9 Vector data

This 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 3 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)`.

2.2.10 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).

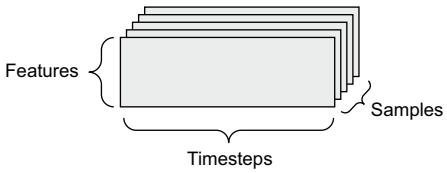


Figure 2.3 A rank-3 timeseries data tensor

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)`.

2.2.11 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).

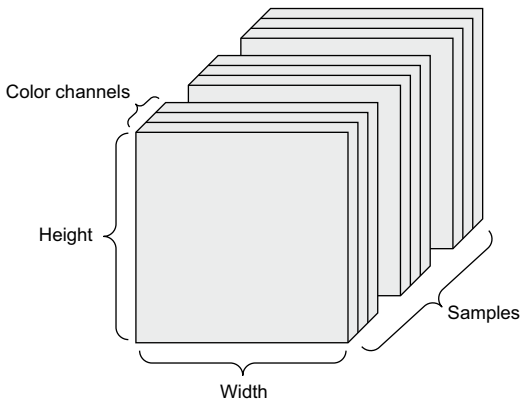


Figure 2.4 A rank-4 image data tensor

There are two conventions for shapes of image tensors: the *channels-last* convention (which is standard in TensorFlow) and the *channels-first* convention (which is increasingly falling out of favor).

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.

2.2.12 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`, each value would be stored in 32 bits, so the tensor would represent 405 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 in the MPEG format).

2.3 The gears of neural networks: Tensor operations

Much as 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 built 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(dot(input, W) + b)
```

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

- A dot product (`dot`) 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 here. 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 and TensorFlow 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 1970–90 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 is a rank-2 NumPy tensor.

```
    x = x.copy()                          # Avoid overwriting the input tensor.
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

You could do the same for addition:

```
def naive_add(x, y):
    assert len(x.shape) == 2              # x and y are rank-2 NumPy tensors.
    assert x.shape == y.shape
    x = x.copy()                          # Avoid overwriting the input tensor.
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

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.)                     # 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.)
print("Took: {0:.2f} s".format(time.time() - t0))
```

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

```
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 TensorFlow 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:

1. Axes (called *broadcast axes*) are added to the smaller tensor to match the `ndim` of the larger tensor.
2. 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.concatenate([y] * 32, axis=0)
```

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

At this point, we can proceed to 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 10 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]):
```

x is a rank-2 NumPy tensor.

y is a NumPy vector.

Avoid overwriting the input tensor.

```
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

With broadcasting, you can generally perform element-wise operations that take two inputs tensors 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*, or *dot product* (not to be confused with an element-wise product, the `*` operator), is one of the most common, most useful tensor operations.

In NumPy, a tensor product is done using the `np.dot` function (because the mathematical notation for tensor product is usually a dot):

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

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

```
z = x • y
```

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

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 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 dot product between two vectors is a scalar and that only vectors with the same number of elements are compatible for a dot product.

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

```
def naive_matrix_vector_dot(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 first dimension of x must be the same as the 0th dimension of y!

This operation returns a vector of 0s with the same shape as y.

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

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

Of course, a dot product generalizes to tensors with an arbitrary number of axes. The most common applications may be the dot product between two matrices. You can take the dot product of two matrices `x` and `y` (`dot(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_dot(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]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

x and y are NumPy matrices.

The first dimension of x must be the same as 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.

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

In the figure, `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.

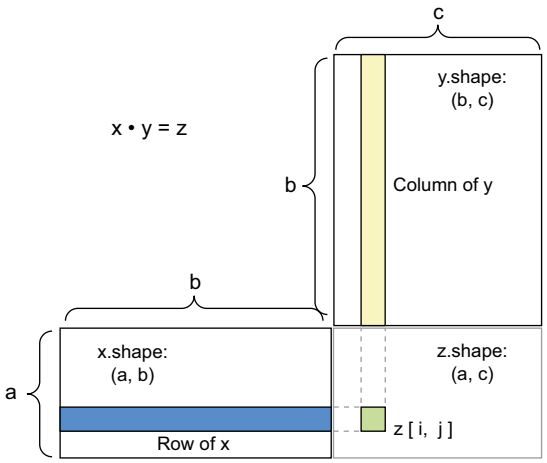


Figure 2.5 Matrix dot-product box diagram

More generally, you can take the dot 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.

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.

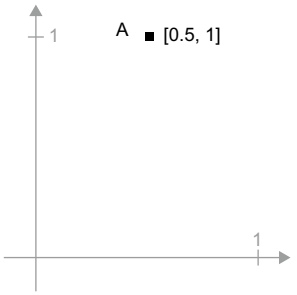


Figure 2.6 A point in a 2D space

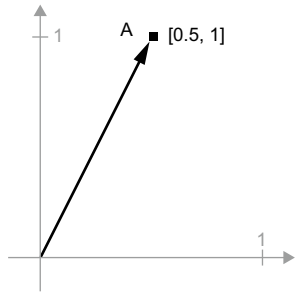


Figure 2.7 A point in a 2D space pictured as an arrow

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 a 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

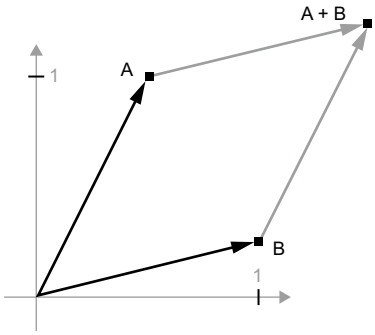


Figure 2.8 Geometric interpretation of the sum of two vectors

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.

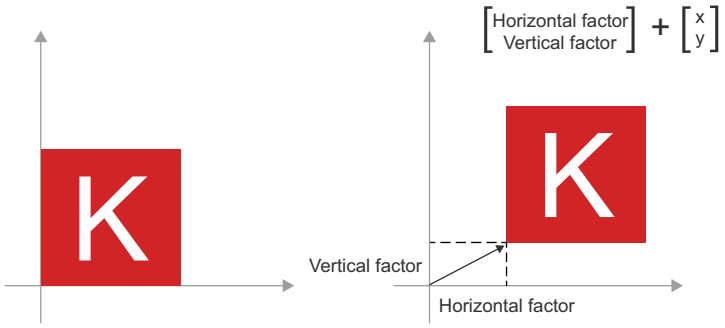


Figure 2.9 2D translation as a vector addition

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

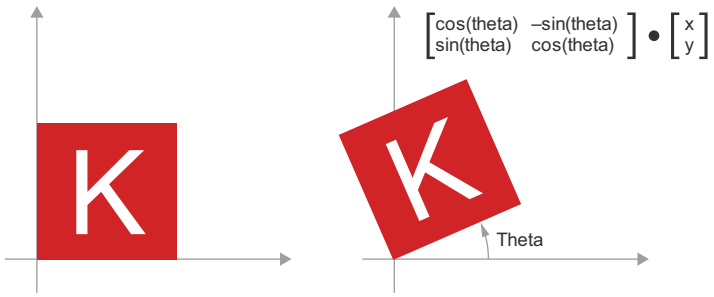


Figure 2.10 2D rotation (counterclockwise) as a dot product

- *Scaling*: A vertical and horizontal scaling of the image (see figure 2.11) can be achieved via a dot product with a 2 × 2 matrix `S = [[horizontal_factor, 0], [0, vertical_factor]]` (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).

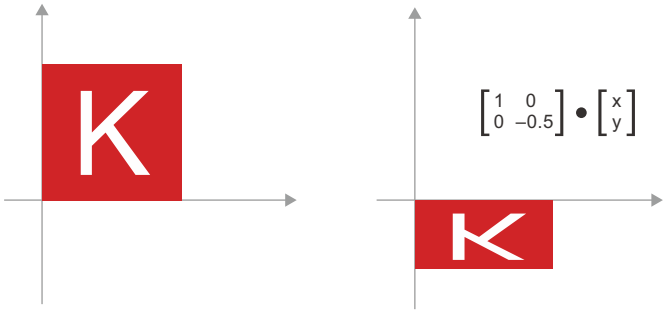


Figure 2.11 2D scaling as a dot product

- *Linear transform*: A dot product with an arbitrary matrix implements a linear transform. Note that *scaling* and *rotation*, listed 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 dot product with some matrix) 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.

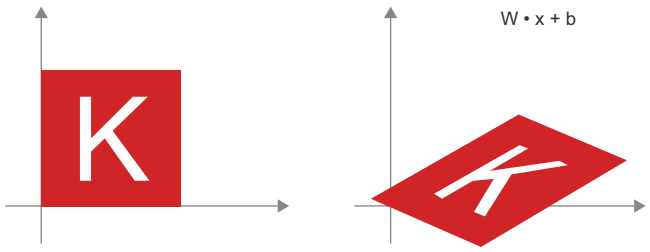


Figure 2.12 Affine transform in the plane

- *`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, non-linear geometric transformations, resulting in very rich hypothesis spaces for your deep neural networks. We'll cover this idea in more detail in the next chapter.

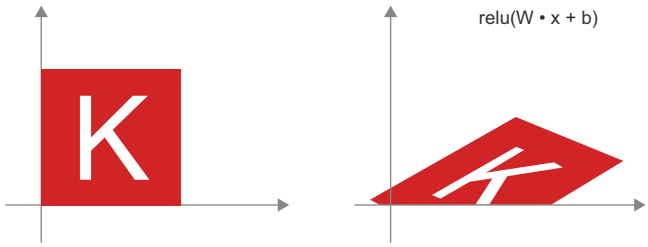


Figure 2.13 Affine transform followed by `relu` activation

2.3.6 A geometric interpretation of deep learning

You just learned that neural networks consist entirely of chains of tensor operations, and that 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, so as 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.

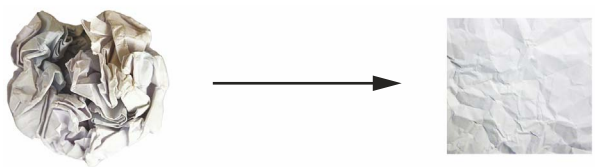


Figure 2.14 Uncrumpling a complicated manifold of data

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(dot(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(dot(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 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—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 thousands and sometimes up to millions). 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 `dot` 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 in the model's coefficients results in a small, predictable change in 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 to section 2.4.3. Otherwise, the following two sections 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.

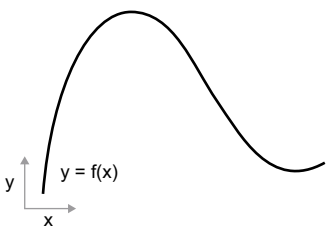


Figure 2.15 A continuous, smooth function

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.

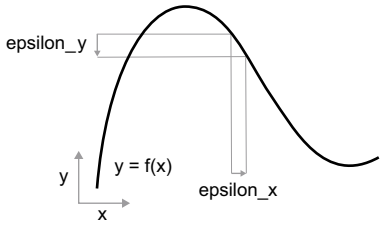


Figure 2.16 With a continuous function, a small change in `x` results in a small change in `y`.

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.

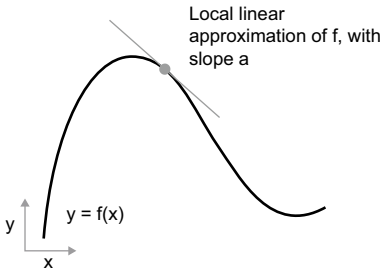


Figure 2.17 Derivative of `f` in `p`

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` in order 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 matrices as inputs, functions that take rank-3 tensors as inputs, 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 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 associate 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 = dot(W, x)
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` so as to make `loss_value` smaller. How do we do that?

Given fixed inputs `x` and `y_true`, the preceding 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` at 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`."

Partial derivatives

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 *direction of steepest ascent* of `loss_value = f(W)` around `W0`, as well as the slope of this ascent. Each partial derivative describes the slope of `f` in a specific direction.

For this reason, in much the same way that, 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, 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: for example, `W1 = W0 - step * grad(f(W0), W0)` (where `step` is a small scaling factor). That means going against the direction of steepest ascent of `f`, 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 often be several tens of millions.

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 for 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.

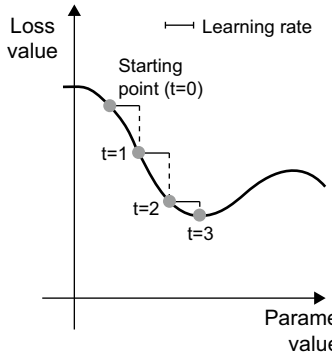


Figure 2.18 SGD down a 1D loss curve (one learnable parameter)

As you can see, intuitively 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.

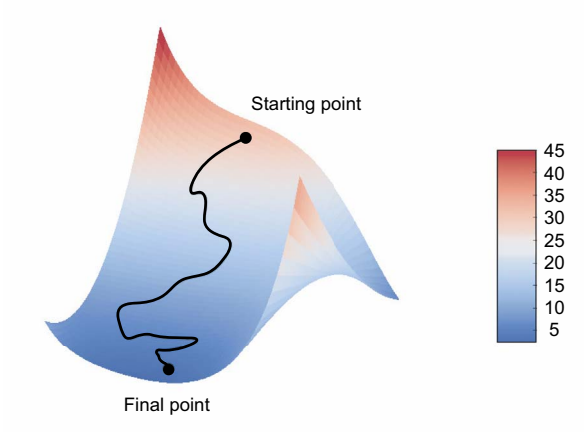


Figure 2.19 Gradient descent down a 2D loss surface (two learnable parameters)

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.

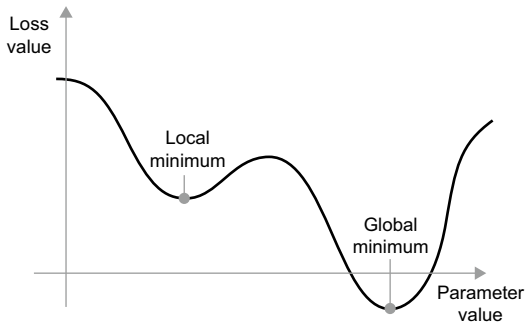


Figure 2.20 A local minimum and a global minimum

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, the optimization process could 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 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.
momentum = 0.1                                  Constant momentum factor
while loss > 0.01:                              Optimization loop
    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)
```

2.4.4 Chaining derivatives: The Backpropagation algorithm

In the preceding 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 the two-layer model we started the chapter with, 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 derivatives 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 defined in listing 2.2 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 `dot`, `relu`, `softmax`, and `+`, as well as our loss function `loss`, which are all easily differentiable:

```
loss_value = loss(y_true, softmax(dot(relu(dot(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 `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 as it is 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 TensorFlow and the deep learning revolution in general. It's a directed acyclic graph of operations—in our case, tensor operations. For instance, figure 2.21 shows 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

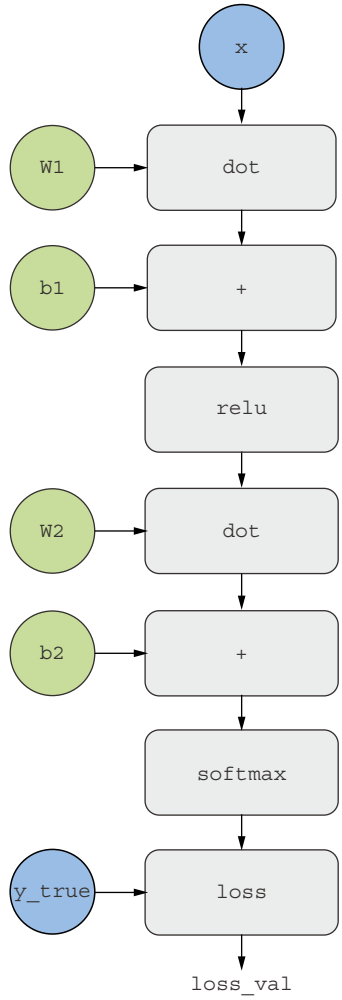


Figure 2.21 The computation graph representation of our two-layer model

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 (see figure 2.22). We'll consider a simplified version of figure 2.21, where we only have one linear layer and where all variables are scalar. We'll take two scalar variables `w` and `b`, a scalar input `x`, and apply some operations to them to combine them 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 will minimize `loss_val`, we are interested in computing `grad(loss_val, b)` and `grad(loss_val, w)`.

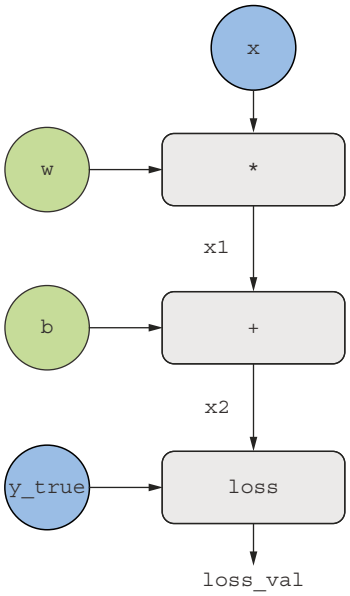


Figure 2.22 A basic example of a computation graph

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

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

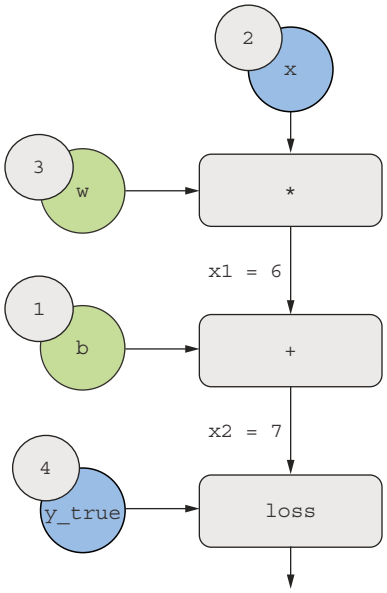


Figure 2.23 Running a forward pass

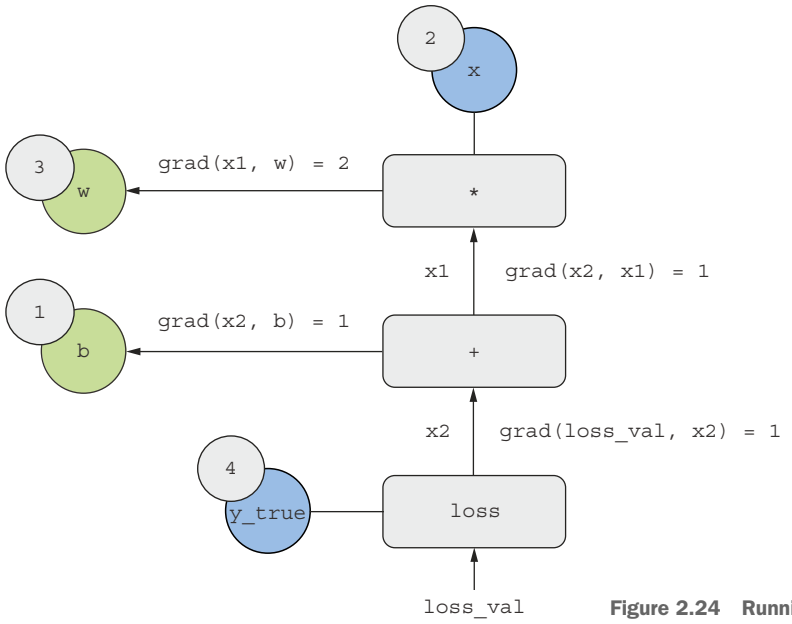


Figure 2.24 Running a backward pass

We have the following:

- `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).

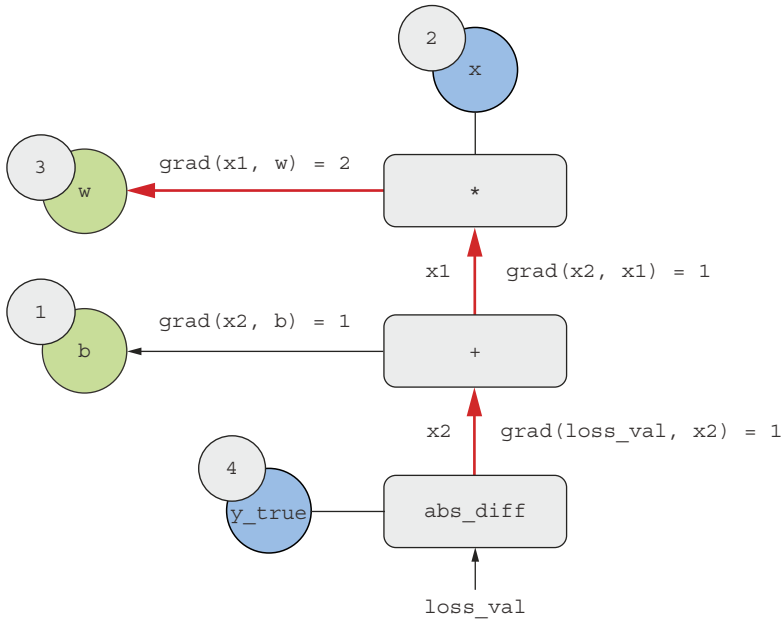


Figure 2.25 Path from `loss_val` to `w` in the backward graph

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`

NOTE If there are multiple paths linking the two nodes of interest, `a` and `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 TensorFlow. Automatic differentiation is implemented with the kind of computation graph you've just seen. 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!

THE GRADIENT TAPE IN TENSORFLOW

The API through which you can leverage TensorFlow's powerful automatic differentiation capabilities is the `GradientTape`. 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 output with respect to any variable or set of variables (instances of the `tf.Variable` class). A `tf.Variable` is a specific kind of tensor meant to hold mutable state—for instance, the weights of a neural network are always `tf.Variable` instances.

```
import tensorflow as tf
x = tf.Variable(0.)
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

Instantiate a scalar Variable with an initial value of 0.

Open a GradientTape scope.

Inside the scope, apply some tensor operations to our variable.

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

The `GradientTape` works with tensor operations:

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

Instantiate a Variable with shape (2, 2) and an initial value of all zeros.

grad_of_y_wrt_x is a tensor of shape (2, 2) (like x) describing the curvature of y = 2 * a + 3 around x = [[0, 0], [0, 0]].

It also works with lists of variables:

```
W = tf.Variable(tf.random.uniform((2, 2)))
b = tf.Variable(tf.zeros((2,)))
x = tf.random.uniform((2, 2))
with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])
```

matmul is how you say "dot product" in TensorFlow.

grad_of_y_wrt_W_and_b is a list of two tensors with the same shapes as W and b, respectively.

You will learn about the gradient tape in the next chapter.

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.

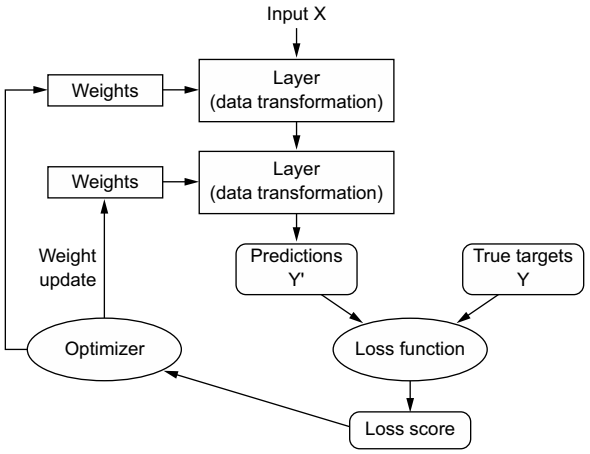


Figure 2.26 Relationship between the network, layers, loss function, and optimizer

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

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="rmsprop",
              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, and 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 `rmsprop` 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 "from scratch" in TensorFlow, step by step.

2.5.1 Reimplementing our first example from scratch in TensorFlow

What better demonstrates full, unambiguous understanding than implementing 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 we will barely use any Keras functionality at all.

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 TensorFlow API. For now, just try to follow the gist of what's going on—the intent of this example is to help crystalize 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`, but it would be `softmax` for the last layer):

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

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

```
import tensorflow as tf

class NaiveDense:
    def __init__(self, input_size, output_size, activation):
        self.activation = activation

        w_shape = (input_size, output_size)
        w_initial_value = tf.random.uniform(w_shape, minval=0, maxval=1e-1)
        self.W = tf.Variable(w_initial_value)

        b_shape = (output_size,)
        b_initial_value = tf.zeros(b_shape)
        self.b = tf.Variable(b_initial_value)

    def __call__(self, inputs):
        return self.activation(tf.matmul(inputs, self.W) + self.b)

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

Create a matrix, W, of shape (input_size, output_size), initialized with random values.

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

Apply the forward pass.

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=tf.nn.relu),
    NaiveDense(input_size=512, output_size=10, activation=tf.nn.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

1. Compute the predictions of the model for the images in the batch.
2. Compute the loss value for these predictions, given the actual labels.

3 Compute the gradient of the loss with regard to the model's weights.
4 Move the weights by a small amount in the direction opposite to the gradient.

To compute the gradient, we will use the TensorFlow `GradientTape` object we introduced in section 2.4.4:

```
def one_training_step(model, images_batch, labels_batch):
    with tf.GradientTape() as tape:
        predictions = model(images_batch)
        per_sample_losses = tf.keras.losses.sparse_categorical_crossentropy(
            labels_batch, predictions)
        average_loss = tf.reduce_mean(per_sample_losses)
    gradients = tape.gradient(average_loss, model.weights)
    update_weights(gradients, model.weights)
    return average_loss
```

Run the "forward pass" (compute the model's predictions under a GradientTape scope).

Compute 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.

Update the weights using the gradients (we will define this function shortly).

As you already know, the purpose of the "weight update" step (represented by the preceding `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_sub(g * learning_rate)
```

assign_sub is the equivalent of -= for TensorFlow variables.

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

```
from tensorflow.keras import optimizers

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

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

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 repeating 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 tensorflow.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)
predictions = predictions.numpy()
predicted_labels = np.argmax(predictions, axis=1)
matches = predicted_labels == test_labels
print(f"accuracy: {matches.mean():.2f}")
```

Calling .numpy() on a TensorFlow tensor converts it to a NumPy tensor.

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 leverage the high-level features of the Keras API.

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 stochastic 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 Keras and TensorFlow

This chapter covers

- A closer look at TensorFlow, Keras, and their relationship
- Setting up a deep learning workspace
- An overview of how core deep learning concepts translate to Keras and TensorFlow

This chapter is meant to give you everything you need to start doing deep learning in practice. I'll give you a quick presentation of Keras (https://keras.io) and TensorFlow (https://tensorflow.org), the Python-based deep learning tools that we'll use throughout the book. You'll find out how to set up a deep learning workspace, with TensorFlow, Keras, and GPU support. Finally, building on top of the first contact you had with Keras and TensorFlow in chapter 2, we'll review the core components of neural networks and how they translate to the Keras and TensorFlow 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 What's TensorFlow?

TensorFlow is a Python-based, free, open source machine learning platform, developed primarily by Google. Much like NumPy, the primary purpose of TensorFlow is to enable engineers and researchers to manipulate mathematical expressions over numerical tensors. But TensorFlow goes far beyond the scope of NumPy in the following ways:

- It can automatically compute the gradient of any differentiable expression (as you saw in chapter 2), making it highly suitable for machine learning.
- It can run not only on CPUs, but also on GPUs and TPUs, highly parallel hardware accelerators.
- Computation defined in TensorFlow can be easily distributed across many machines.
- TensorFlow programs can be exported to other runtimes, such as C++, JavaScript (for browser-based applications), or TensorFlow Lite (for applications running on mobile devices or embedded devices), etc. This makes TensorFlow applications easy to deploy in practical settings.

It's important to keep in mind that TensorFlow is much more than a single library. It's really a platform, home to a vast ecosystem of components, some developed by Google and some developed by third parties. For instance, there's TF-Agents for reinforcement-learning research, TFX for industry-strength machine learning workflow management, TensorFlow Serving for production deployment, and there's the TensorFlow Hub repository of pretrained models. Together, these components cover a very wide range of use cases, from cutting-edge research to large-scale production applications.

TensorFlow scales fairly well: for instance, scientists from Oak Ridge National Lab have used it to train a 1.1 exaFLOPS extreme weather forecasting model on the 27,000 GPUs of the IBM Summit supercomputer. Likewise, Google has used TensorFlow to develop very compute-intensive deep learning applications, such as the chess-playing and Go-playing agent AlphaZero. For your own models, if you have the budget, you can realistically hope to scale to around 10 petaFLOPS on a small TPU pod or a large cluster of GPUs rented on Google Cloud or AWS. That would still be around 1% of the peak compute power of the top supercomputer in 2019!

3.2 What's Keras?

Keras is a deep learning API for Python, built on top of TensorFlow, that provides a convenient way to define and train any kind of deep learning model. Keras was initially developed for research, with the aim of enabling fast deep learning experimentation.

Through TensorFlow, Keras can run on top of different types of hardware (see figure 3.1)—GPU, TPU, or plain CPU—and can be seamlessly scaled to thousands of machines.

Keras is known for prioritizing the developer experience. It's an API for human beings, not machines. It follows best practices for reducing cognitive load: it offers

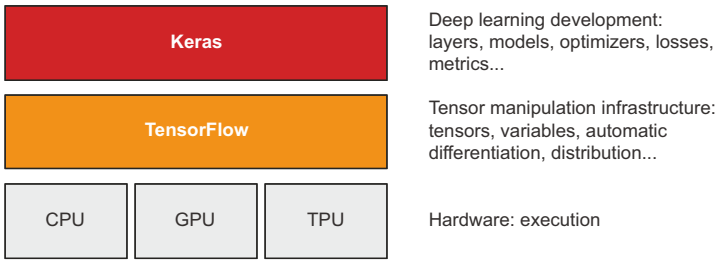


Figure 3.1 Keras and TensorFlow: TensorFlow is a low-level tensor computing platform, and Keras is a high-level deep learning API

consistent and simple workflows, it minimizes the number of actions required for common use cases, and it provides clear and actionable feedback upon user error. This makes Keras easy to learn for a beginner, and highly productive to use for an expert.

Keras has well over a million users as of late 2021, ranging 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, CERN, NASA, Yelp, Instacart, Square, and hundreds of startups working on a wide range of problems across every industry. Your YouTube recommendations originate from Keras models. The Waymo self-driving cars are developed with Keras models. Keras is also a popular framework on Kaggle, the machine learning competition website, where most deep learning competitions have been won using Keras.

Because Keras has a large and 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 trade-off between usability and flexibility. In chapter 5, we'll review in detail a good fraction of this spectrum of workflows. 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.

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 an array of possible usage patterns that all work nicely together. This makes Python suitable to 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 to different user profiles.

3.3 Keras and TensorFlow: A brief history

Keras predates TensorFlow by eight months. It was released in March 2015, and TensorFlow was released in November 2015. You may ask, if Keras is built on top of TensorFlow, how it could exist before TensorFlow was released? Keras was originally built on top of Theano, another tensor-manipulation library that provided automatic differentiation and GPU support—the earliest of its kind. Theano, developed at the Montréal Institute for Learning Algorithms (MILA) at the Université de Montréal, was in many ways a precursor of TensorFlow. It pioneered the idea of using static computation graphs for automatic differentiation and for compiling code to both CPU and GPU.

In late 2015, after the release of TensorFlow, Keras was refactored to a multibackend architecture: it became possible to use Keras with either Theano or TensorFlow, and switching between the two was as easy as changing an environment variable. By September 2016, TensorFlow had reached a level of technical maturity where it became possible to make it the default backend option for Keras. In 2017, two new additional backend options were added to Keras: CNTK (developed by Microsoft) and MXNet (developed by Amazon). Nowadays, both Theano and CNTK are out of development, and MXNet is not widely used outside of Amazon. Keras is back to being a single-backend API—on top of TensorFlow.

Keras and TensorFlow have had a symbiotic relationship for many years. Throughout 2016 and 2017, Keras became well known as the user-friendly way to develop TensorFlow applications, funneling new users into the TensorFlow ecosystem. By late 2017, a majority of TensorFlow users were using it through Keras or in combination with Keras. In 2018, the TensorFlow leadership picked Keras as TensorFlow's official high-level API. As a result, the Keras API is front and center in TensorFlow 2.0, released in September 2019—an extensive redesign of TensorFlow and Keras that takes into account over four years of user feedback and technical progress.

By this point, you must be eager to start running Keras and TensorFlow code in practice. Let's get you started.

3.4 Setting up a deep learning workspace

Before you can get started developing deep learning applications, you need to set up your development environment. It's highly recommended, although not strictly necessary, that you run deep learning code on a modern NVIDIA GPU rather than your computer's CPU. Some applications—in particular, image processing with convolutional networks—will be excruciatingly slow on CPU, even a fast multicore CPU. And even for applications that can realistically be run on CPU, you'll generally see the speed increase by a factor of 5 or 10 by using a recent GPU.

To do deep learning on a GPU, you have three options:

- Buy and install a physical NVIDIA GPU on your workstation.
- Use GPU instances on Google Cloud or AWS EC2.
- Use the free GPU runtime from Colaboratory, a hosted notebook service offered by Google (for details about what a "notebook" is, see the next section).

Colaboratory is the easiest way to get started, as it requires no hardware purchase and no software installation—just open a tab in your browser and start coding. It's the option we recommend for running the code examples in this book. However, the free version of Colaboratory is only suitable for small workloads. If you want to scale up, you'll have to use the first or second option.

If you don't already have a GPU that you can use for deep learning (a recent, high-end NVIDIA GPU), then running deep learning experiments in the cloud is a simple, low-cost way for you to move to larger workloads without having to buy any additional hardware. If you're developing using Jupyter notebooks, the experience of running in the cloud is no different from running locally.

But if you're a heavy user of deep learning, this setup isn't sustainable in the long term—or even for more than a few months. Cloud instances aren't cheap: you'd pay $2.48 per hour for a V100 GPU on Google Cloud in mid-2021. Meanwhile, a solid consumer-class GPU will cost you somewhere between $1,500 and $2,500—a price that has been fairly stable over time, even as the specs of these GPUs keep improving. If you're a heavy user of deep learning, consider setting up a local workstation with one or more GPUs.

Additionally, whether you're running locally or in the cloud, it's better to be using a Unix workstation. Although it's technically possible to run Keras on Windows directly, we don't recommend it. If you're a Windows user and you want to do deep learning on your own workstation, the simplest solution to get everything running is to set up an Ubuntu dual boot on your machine, or to leverage Windows Subsystem for Linux (WSL), a compatibility layer that enables you to run Linux applications from Windows. It may seem like a hassle, but it will save you a lot of time and trouble in the long run.

3.4.1 Jupyter notebooks: The preferred way to run deep learning experiments

Jupyter notebooks are a great way to run deep learning experiments—in particular, the many code examples in this book. They're widely used in the data science and machine learning communities. A *notebook* is a file generated by the Jupyter Notebook app (https://jupyter.org) that you can edit in your browser. It mixes the ability to execute Python code with rich text-editing capabilities for annotating what you're doing. A notebook also allows you to break up long experiments into smaller pieces that can be executed independently, which makes development interactive and means you don't have to rerun all of your previous code if something goes wrong late in an experiment.

I recommend using Jupyter notebooks to get started with Keras, although that isn't a requirement: you can also run standalone Python scripts or run code from within an IDE such as PyCharm. All the code examples in this book are available as open source notebooks; you can download them from GitHub at github.com/fchollet/deep-learning-with-python-notebooks.

3.4.2 Using Colaboratory

Colaboratory (or Colab for short) is a free Jupyter notebook service that requires no installation and runs entirely in the cloud. Effectively, it's a web page that lets you write and execute Keras scripts right away. It gives you access to a free (but limited) GPU runtime and even a TPU runtime, so you don't have to buy your own GPU. Colaboratory is what we recommend for running the code examples in this book.

First steps with Colaboratory

To get started with Colab, go to https://colab.research.google.com and click the New Notebook button. You'll see the standard Notebook interface shown in figure 3.2.

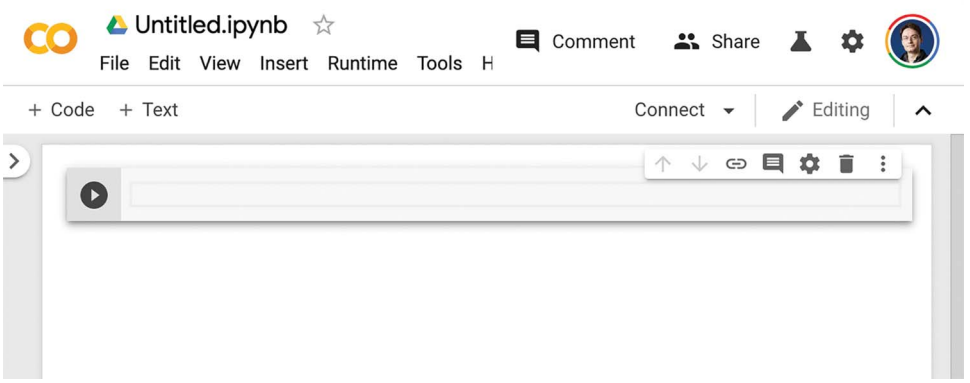


Figure 3.2 A Colab notebook

You'll notice two buttons in the toolbar: + Code and + Text. They're for creating executable Python code cells and annotation text cells, respectively. After entering code in a code cell, Pressing Shift-Enter will execute it (see figure 3.3).

In a text cell, you can use Markdown syntax (see figure 3.4). Pressing Shift-Enter on a text cell will render it.

Text cells are useful for giving a readable structure to your notebooks: use them to annotate your code with section titles and long explanation paragraphs or to embed figures. Notebooks are meant to be a multimedia experience!

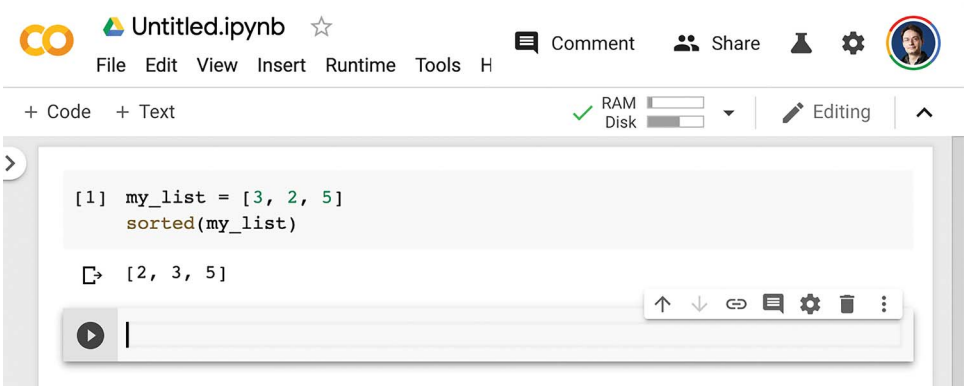


Figure 3.3 Creating a code cell

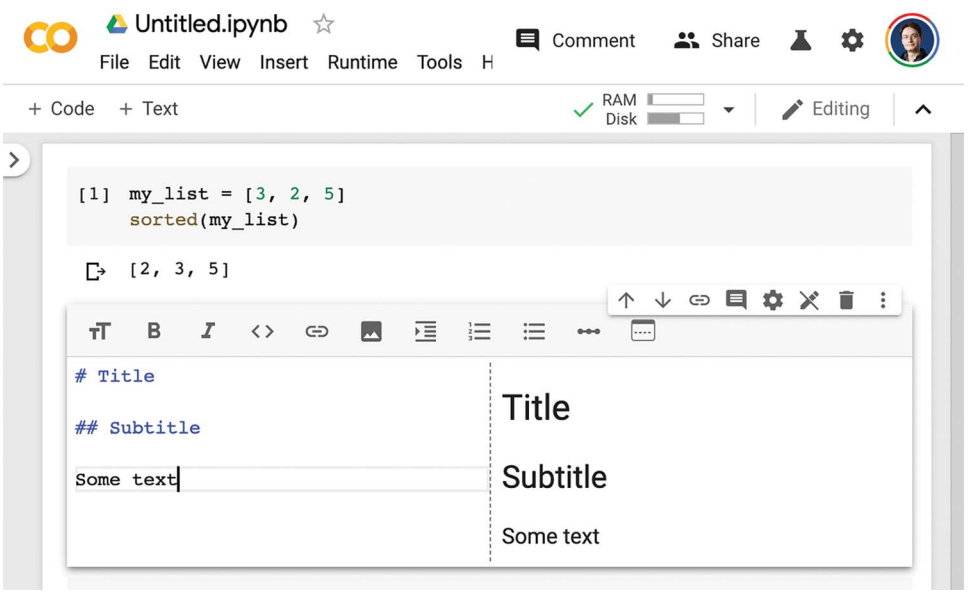


Figure 3.4 Creating a text cell

INSTALLING PACKAGES WITH PIP

The default Colab environment already comes with TensorFlow and Keras installed, so you can start using it right away without any installation steps required. But if you ever need to install something with `pip`, you can do so by using the following syntax in a code cell (note that the line starts with `!` to indicate that it is a shell command rather than Python code):

```
!pip install package_name
```

USING THE GPU RUNTIME

To use the GPU runtime with Colab, select Runtime > Change Runtime Type in the menu and select GPU for the Hardware Accelerator (see figure 3.5).

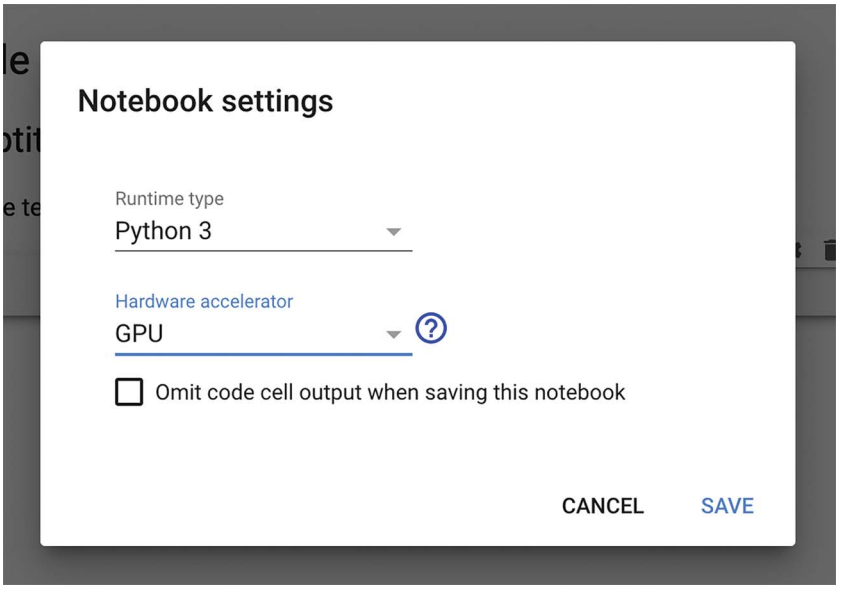


Figure 3.5 Using the GPU runtime with Colab

TensorFlow and Keras will automatically execute on GPU if a GPU is available, so there's nothing more you need to do after you've selected the GPU runtime.

You'll notice that there's also a TPU runtime option in that Hardware Accelerator dropdown menu. Unlike the GPU runtime, using the TPU runtime with TensorFlow and Keras does require a bit of manual setup in your code. We'll cover this in chapter 13. For the time being, we recommend that you stick to the GPU runtime to follow along with the code examples in the book.

You now have a way to start running Keras code in practice. Next, let's see how the key ideas you learned about in chapter 2 translate to Keras and TensorFlow code.

3.5 First steps with TensorFlow

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 TensorFlow APIs:
  - *Tensors*, including special tensors that store the network's state (*variables*)
  - *Tensor operations* such as addition, `relu`, `matmul`

  - *Backpropagation*, a way to compute the gradient of mathematical expressions (handled in TensorFlow via the `GradientTape` object)
- 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

In the previous chapter, you already had a first light contact with some of the corresponding TensorFlow and Keras APIs: you've briefly used TensorFlow's `Variable` class, the `matmul` operation, and the `GradientTape`. You've instantiated Keras `Dense` layers, packed them into a `Sequential` model, and trained that model with the `fit()` method.

Now let's take a deeper dive into how all of these different concepts can be approached in practice using TensorFlow and Keras.

3.5.1 Constant tensors and variables

To do anything in TensorFlow, we're going to need some tensors. Tensors need to be created with some initial value. For instance, you could create all-ones or all-zeros tensors (see listing 3.1), or tensors of values drawn from a random distribution (see listing 3.2).

Listing 3.1 All-ones or all-zeros tensors

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

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

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

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(
```

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.).

```
[[0.33779848]
 [0.06692922]
 [0.7749394 ]], shape=(3, 1), dtype=float32)
```

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.
```

Try to do the same thing in TensorFlow, and 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.
```

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? That's where *variables* come in. `tf.Variable` is the class meant to manage modifiable state in TensorFlow. You've already briefly seen it in action in the training loop implementation at the end of chapter 2.

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

Listing 3.5 Creating a TensorFlow 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, as follows.

Listing 3.6 Assigning a value to a TensorFlow variable

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

It also works for a subset of the coefficients.

Listing 3.7 Assigning a value to a subset of a TensorFlow variable

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

Similarly, `assign_add()` and `assign_sub()` are efficient equivalents of `+=` and `-=`, as shown next.

Listing 3.8 Using `assign_add()`

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

3.5.2 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

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

Take the square.

Take the square root.

Add two tensors (element-wise).

Take the product of two tensors (as discussed in chapter 2).

Multiply two tensors (element-wise).

Importantly, each of the preceding operations gets executed on the fly: at any point, you can print what the current result is, just like in NumPy. We call this *eager execution*.

3.5.3 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.)
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)`. You saw this in action in chapter 2.

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 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 `GradientTape` with constant tensor inputs

```
input_const = tf.constant(3.)
with tf.GradientTape() as tape:
   tape.watch(input_const)
   result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

Why is this necessary? 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 of the gradient tape.

The gradient tape is a powerful utility, even capable of computing *second-order gradients*, that is to say, 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.)
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.

3.5.4 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 machine learning model based on gradient descent. And you're only at chapter 3!

In a machine learning job interview, you may be asked to implement a linear classifier from scratch in TensorFlow: 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. We'll generate each class of points by drawing their coordinates from a random distribution with a specific covariance matrix and a specific mean. Intuitively, the covariance matrix describes the shape of the point cloud, and the mean describes its position in the plane (see figure 3.6). We'll reuse the same covariance matrix for both point clouds, but we'll use two different mean values—the point clouds will have the same shape, but different positions.

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

```
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)
```

Generate the first class of points: 1000 random 2D points. cov=[[1, 0.5],[0.5, 1]] corresponds to an oval-like point cloud oriented from bottom left to top right.

Generate the other class of points with a different mean and the same covariance matrix.

In the preceding code, `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 zeros and ones 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")))
```

Next, let's plot our data with Matplotlib.

Listing 3.16 Plotting the two point classes (see figure 3.6)

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

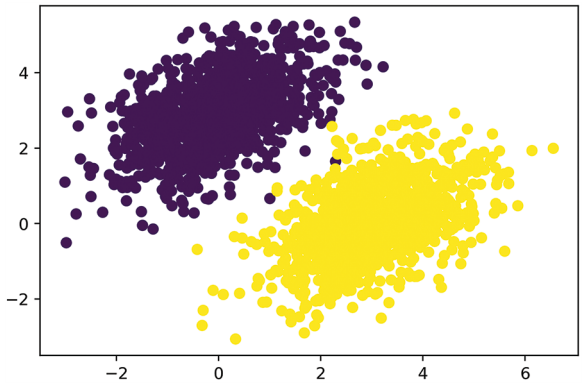


Figure 3.6 Our synthetic data: two classes of random points in the 2D plane

Now let's create a linear classifier that can learn to separate these two blobs. A linear classifier is an affine transformation (`prediction = W • input + 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 you saw at 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

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).

```
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,)))
```

Here's our forward pass function.

Listing 3.18 The forward pass function

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

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

The following listing shows our loss function.

Listing 3.19 The mean squared error loss function

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

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

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

Next is the training step, which receives some training data and updates the weights `W` and `b` so as to minimize the loss on the data.

Listing 3.20 The training step function

```
learning_rate = 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(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
```

Retrieve the gradient of the loss with regard to weights.

Forward pass, inside a gradient tape scope

Update 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) for all the 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'll 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 we would typically use for mini-batch training (we'll use `learning_rate = 0.1`, defined in listing 3.20).

Listing 3.21 The batch training loop

```
for step in range(40):
    loss = training_step(inputs, targets)
    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. Because our targets are zeros and ones, 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 (see figure 3.7):

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

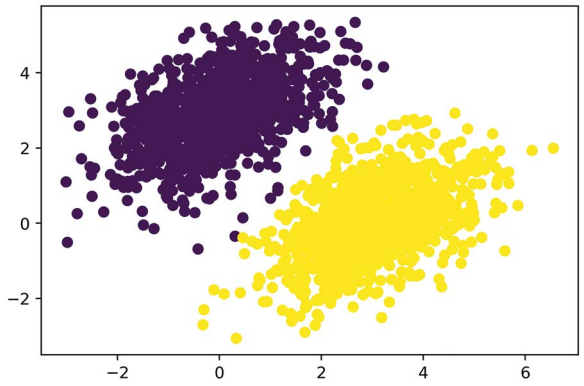


Figure 3.7 Our model's predictions on the training inputs: pretty similar to the training targets

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`. Above the line is class 1, and below the line is class 0. 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 (shown in figure 3.8):

Generate 100 regularly spaced numbers between –1 and 4, which we will use to plot our line.

This is our line's equation.

Plot 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)
```

Plot our model's predictions on the same plot.

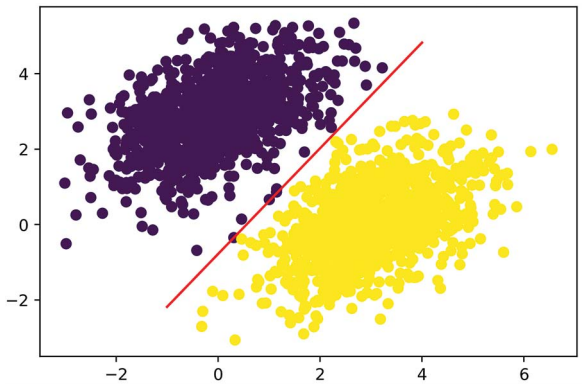


Figure 3.8 Our model, visualized as a line

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.

3.6 Anatomy of a neural network: Understanding core Keras APIs

At this point, you know the basics of TensorFlow, and you can use it to implement a toy model from scratch, such as the batch linear classifier in the previous section, or the toy neural network at the end of chapter 2. That's a solid foundation to build upon. It's now time to move on to a more productive, more robust path to deep learning: the Keras API.

3.6.1 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 rank-2 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 rank-3 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(dot(input, W) + b)`. This is what the same layer would look like in Keras.

Listing 3.22 A `Dense` layer implemented as a `Layer` subclass

```
from tensorflow import keras

class SimpleDense(keras.layers.Layer):
```

All Keras layers inherit from the base Layer class.

```
def __init__(self, units, activation=None):
    super().__init__()
    self.units = units
    self.activation = activation

def build(self, input_shape):
    input_dim = input_shape[-1]
    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 = tf.matmul(inputs, self.W) + self.b
    if self.activation is not None:
        y = self.activation(y)
    return y
```

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 is also possible to create standalone variables and assign them as layer attributes, like self.W = tf.Variable(tf.random.uniform(w_shape)).

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 TensorFlow tensor:

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

Instantiate our layer, defined previously.

Create some test inputs.

Call the layer on the inputs, just like a function.

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

A dense layer with 32 output units

This layer will return a tensor where the first dimension has been transformed to be 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 layer. For instance, suppose you write the following:

```
from tensorflow.keras import models
from tensorflow.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 the `Dense` layer we implemented in chapter 2 (which we named `NaiveDense`), 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 if 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 previous `SimpleDense` layer (see listing 3.22), with its `build()` and `call()` methods.

In `SimpleDense`, we no longer create weights in the constructor like in the `NaiveDense` example; instead, we create them in a dedicated state-creation method, `build()`, which receives as an 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 (a concept you'll learn about in chapter 7), and input masking (which we'll cover in chapter 11). For now, just remember: when implementing your own layers, put the forward pass in the `call()` method.

3.6.2 From layers to models

A deep learning model is a graph of layers. In Keras, that's the `Model` class. Until 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. These are some common ones:

- Two-branch networks
- Multihead networks
- Residual connections

Network topology can get quite involved. For instance, figure 3.9 shows the 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 could directly subclass the `Model` class, or you could 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

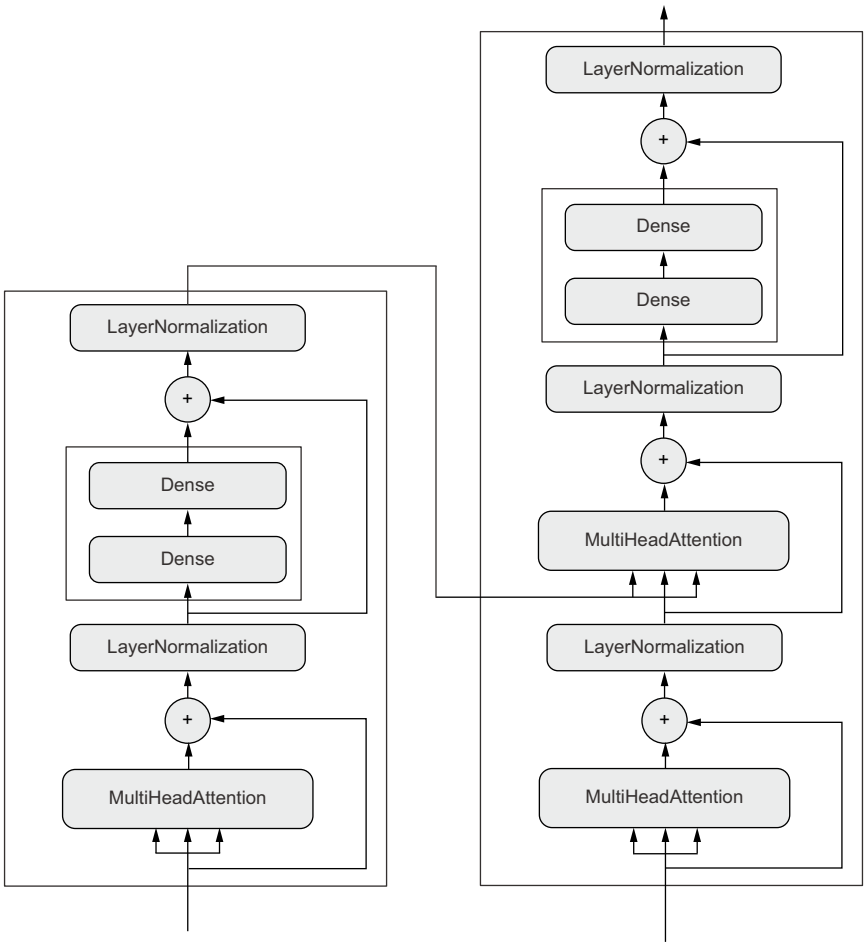


Figure 3.9 The Transformer architecture (covered in chapter 11). There's a lot going on here. Throughout the next few chapters, you'll climb your way up to understanding it.

you explicit principles for building neural networks and help you develop intuition as to 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.

3.6.3 The "compile" step: Configuring the learning process

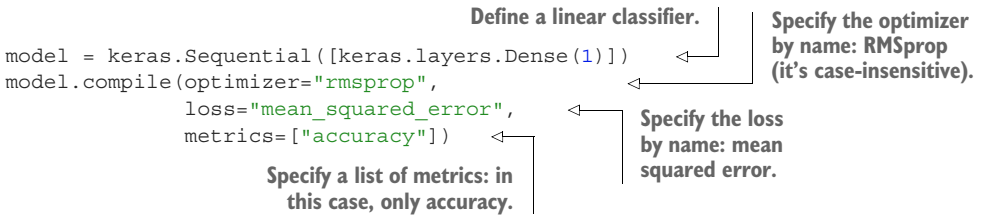
Once the model architecture is defined, you still have to choose three more things:

- *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 could also write your own custom training loops—we'll 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 preceding 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'll 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`

- `Adagrad`
- 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.

3.6.4 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: "maximizing 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.

3.6.5 Understanding the fit() method

After `compile()` comes `fit()`. The `fit()` method implements the training loop itself. These are its key arguments:

- 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 3.23 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 keys 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]}
```

3.6.6 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, and 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 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 3.24 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, shuffle the inputs and targets using a random indices permutation.

Reserve 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.

Note that if you want to compute the validation loss and metrics after the 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).

3.6.7 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 TensorFlow tensor and returns a TensorFlow tensor

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. You are ready to move on to solving real-world machine learning problems with Keras in the next chapter.

Summary

- TensorFlow is an industry-strength numerical computing framework that can run on CPU, GPU, or TPU. It can automatically compute the gradient of any differentiable expression, it can be distributed to many devices, and it can export programs to various external runtimes—even JavaScript.
- Keras is the standard API for doing deep learning with TensorFlow. It's what we'll use throughout this book.
- Key TensorFlow objects include tensors, variables, tensor operations, and the gradient tape.

- 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 use the `model.predict()` method to generate predictions on new inputs.

4 Getting started with neural networks: Classification and regression

This chapter covers

- Your first examples of real-world machine learning workflows
- Handling classification problems over vector data
- Handling continuous regression problems over vector data

This chapter is designed to get you started 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, multiclass classification, and scalar regression:

- Classifying movie reviews as positive or negative (binary classification)
- Classifying news wires by topic (multiclass 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.

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.
- *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 both with 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 *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.

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.

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 problems. 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 11, 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 tensorflow.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 us 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 lists of reviews; each review is a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists 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 we're restricting ourselves 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, here's how you can 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."

4.1.2 Preparing the data

You can't directly feed lists of integers into a neural network. They all have different lengths, but 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, 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. This would mean, for instance, turning the sequence `[8, 5]` into a 10,000-dimensional vector that would be all 0s except for indices 8 and 5, which would be 1s. Then you could use a `Dense` layer, capable of handling floating-point vector data, as the first layer in your model.

Let's go with the latter solution to vectorize the data, which you'll do manually for maximum clarity.

Listing 4.3 Encoding the integer sequences via multi-hot encoding

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 1.
    return results
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Creates an all-zero matrix of shape (len(sequences), dimension)

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.])
```

You should also vectorize your labels, which is straightforward:

```
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(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 me with the following architecture choices:

- 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. And the following listing shows the Keras implementation, similar to the MNIST example you saw previously.

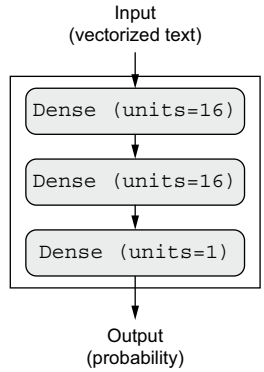


Figure 4.1 The three-layer model

Listing 4.4 Model definition

```
from tensorflow import keras
from tensorflow.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 so as to output a probability (a score between 0 and 1 indicating how likely the sample is to have the target "1": 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.

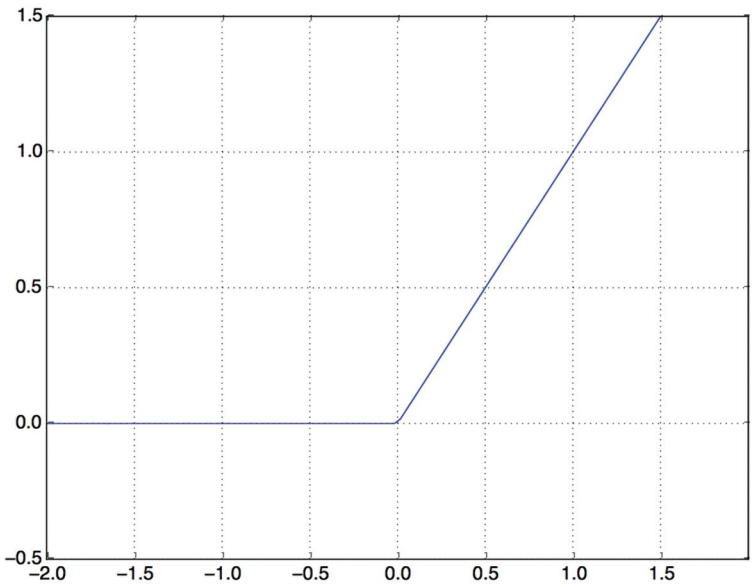


Figure 4.2 The rectified linear unit function

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: for instance, you could use `mean_squared_error`. But crossentropy is usually the best choice when you're dealing

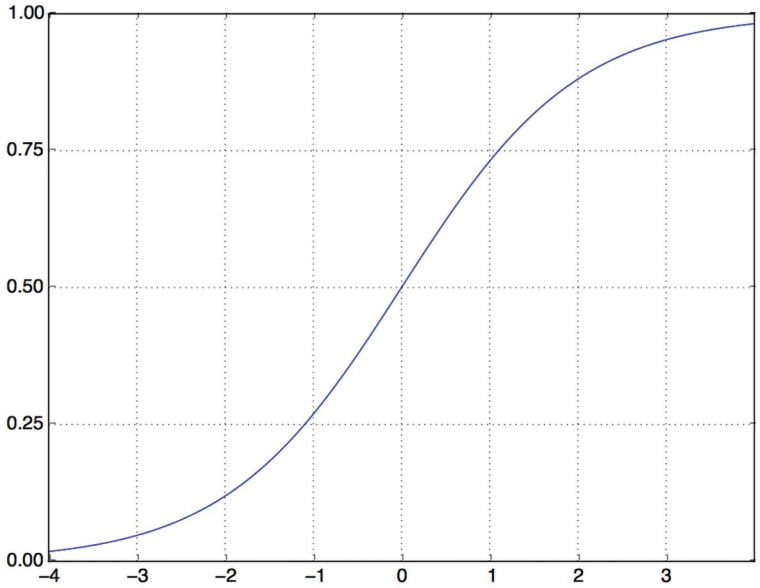


Figure 4.3 The sigmoid function

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:

```
output = dot(input, W) + b
```

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).

In order to get access to a much richer hypothesis space that will 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.

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 `rmsprop`, which is a usually a good default choice for virtually any problem.

Here's the step where we configure the model with the `rmsprop` optimizer and the `binary_crossentropy` loss function. Note that we'll also monitor accuracy during training.

Listing 4.5 Compiling the model

```
model.compile(optimizer="rmsprop",
              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, we'll create a validation set by setting apart 10,000 samples from the original training data.

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:]
```

We will 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, we will monitor loss and accuracy on the 10,000 samples that we set apart. We do so by passing the validation data as the `validation_data` argument.

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))
```

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 saw 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()
[u"accuracy", u"loss", u"val_accuracy", u"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.

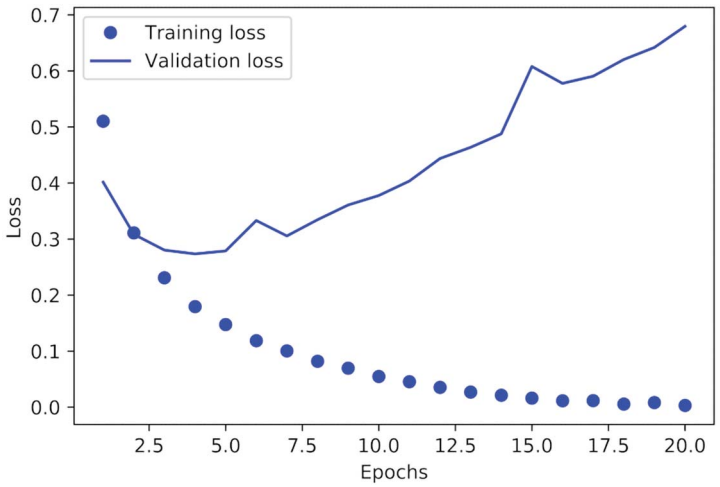


Figure 4.4 Training and validation loss

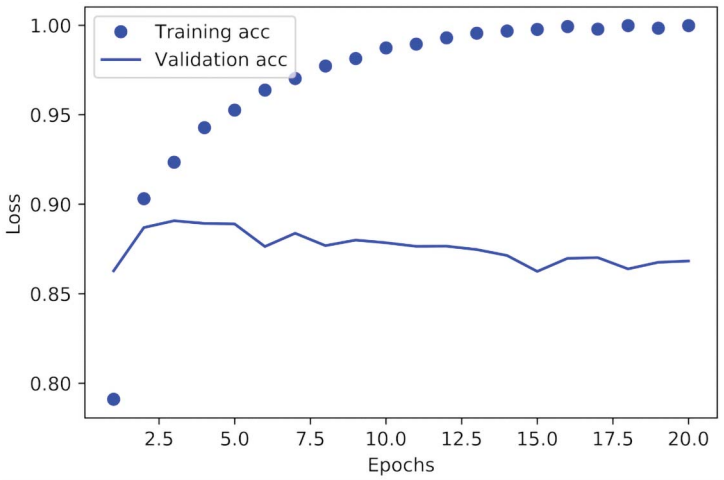


Figure 4.5 Training and validation accuracy

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, "bo", label="Training loss")
plt.plot(epochs, val_loss_values, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

"bo" is for "blue dot."

"b" is for "solid blue line."

Listing 4.9 Plotting the training and validation accuracy

```
plt.clf()
acc = history_dict["accuracy"]
val_acc = history_dict["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training acc")
plt.plot(epochs, val_acc, "b", label="Validation acc")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

Clears the figure

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 Retraining a model from scratch

```
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"])
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 `mse` 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 in order 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 likely 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 `rmsprop` 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, we'll build a model to classify Reuters newswires into 46 mutually exclusive topics. Because we 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), we'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 tensorflow.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[0]])
```

Note that 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 = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Vectorized training data

Vectorized test data

To vectorize the labels, there are two possibilities: you can cast the label list as an integer tensor, 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. The following listing shows an example.

Listing 4.14 Encoding the labels

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
```

```
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results
y_train = to_one_hot(train_labels)      <-- Vectorized training labels
y_test = to_one_hot(test_labels)        <-- Vectorized test labels
```

Note that there is a built-in way to do this in Keras:

```
from tensorflow.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, we'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 we'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, we 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 we'll use larger 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.

First, we 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.

Second, 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 output 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.

Listing 4.16 Compiling the model

```
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["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).

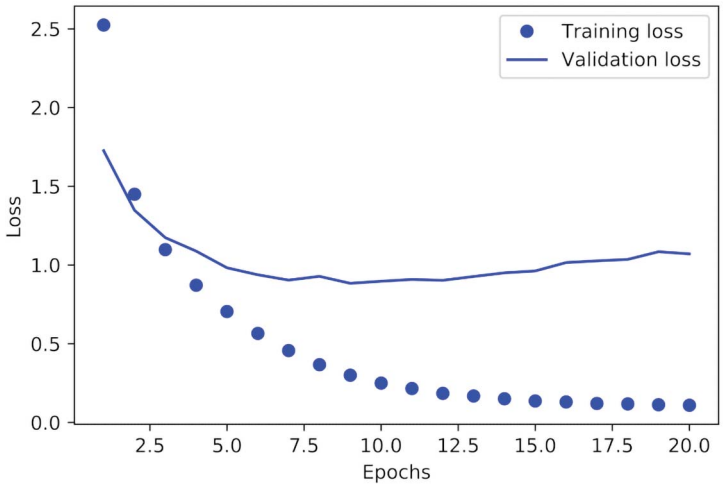


Figure 4.6 Training and validation loss

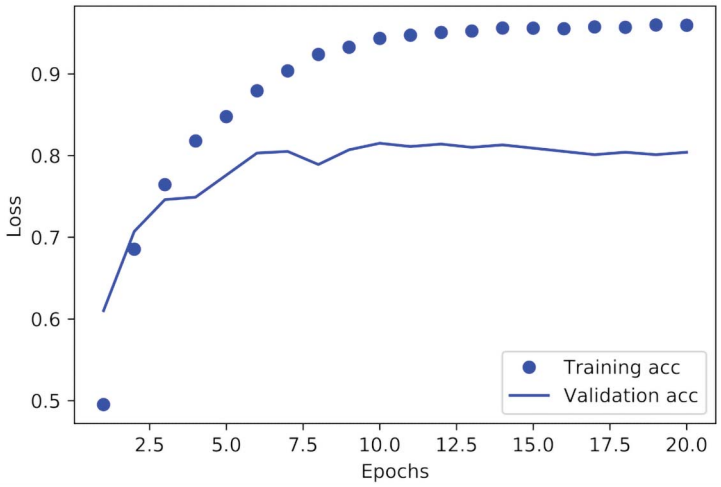


Figure 4.7 Training and validation accuracy

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, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

Listing 4.20 Plotting the training and validation accuracy

```
plt.clf()      <-- Clears the figure
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training accuracy")
plt.plot(epochs, val_acc, "b", label="Validation accuracy")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("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 4.21 Retraining a model from scratch

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
```

```
    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
model.compile(optimizer="rmsprop",
              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 ~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) == np.array(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 cast them as an integer tensor, like this:

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

The only thing this approach would change is the choice of the loss function. The loss function used in listing 4.21, `categorical_crossentropy`, expects the labels to follow a categorical encoding. With integer labels, you should use `sparse_categorical_crossentropy`:

```
model.compile(optimizer="rmsprop",
              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 many fewer than 46 units. Now let's see what happens when we introduce an information bottleneck by having intermediate layers that are significantly less than 46-dimensional: for example, 4-dimensional.

Listing 4.22 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="rmsprop",
              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 ~71% validation accuracy, an 8% absolute drop. This drop is mostly due to the fact that we'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 four-dimensional representations, but not all of it.

4.2.8 Further experiments

Like in the previous example, I 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 Don't confuse *regression* and the *logistic regression* algorithm. Confusingly, logistic regression isn't a regression algorithm—it's a classification algorithm.

4.3.1 The Boston housing price dataset

In this section, we'll attempt to predict the median price of homes in a given Boston suburb in the mid-1970s, given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on. The dataset we'll use has an interesting difference from the two previous examples. It has relatively few data points: only 506, split between 404 training samples and 102 test samples. And each *feature* in the input data (for example, the crime rate) has a different scale. For instance, some values are proportions, which take values between 0 and 1, others take values between 1 and 12, others between 0 and 100, and so on.

Listing 4.23 Loading the Boston housing dataset

```
from tensorflow.keras.datasets import boston_housing
(train_data, train_targets), (test_data, test_targets) = (
    boston_housing.load_data())
```

Let's look at the data:

```
>>> train_data.shape
(404, 13)
>>> test_data.shape
(102, 13)
```

As you can see, we have 404 training samples and 102 test samples, each with 13 numerical features, such as per capita crime rate, average number of rooms per dwelling, accessibility to highways, and so on.

The targets are the median values of owner-occupied homes, in thousands of dollars:

```
>>> train_targets
[ 15.2,  42.3,  50. ...  19.4,  19.4,  29.1]
```

The prices are typically between $10,000 and $50,000. If that sounds cheap, remember that this was the mid-1970s, 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 for dealing with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), we 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 4.24 Normalizing the data

```
mean = train_data.mean(axis=0)
train_data -= mean
```

```
std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```

Note that the quantities used for normalizing the test data are computed using the training data. You should never use any quantity computed on the test data in your workflow, even for something as simple as data normalization.

4.3.3 Building your model

Because so few samples are available, we'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 4.25 Model definition

```
def build_model():
    model = keras.Sequential([
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(1)
    ])
    model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
    return model
```

Because we need to instantiate the same model multiple times, we use a function to construct it.

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 we compile the model with the `mse` 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.

We'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 $500 on average.

4.3.4 Validating your approach using K-fold validation

To evaluate our model while we keep adjusting its parameters (such as the number of epochs used for training), we could split the data into a training set and a validation set, as we did in the previous examples. But because we 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 we chose for validation and which we chose for training: the validation scores

might have a high *variance* with regard to the validation split. This would prevent us from reliably evaluating our model.

The best practice in such situations is to use *K-fold* cross-validation (see figure 4.8).

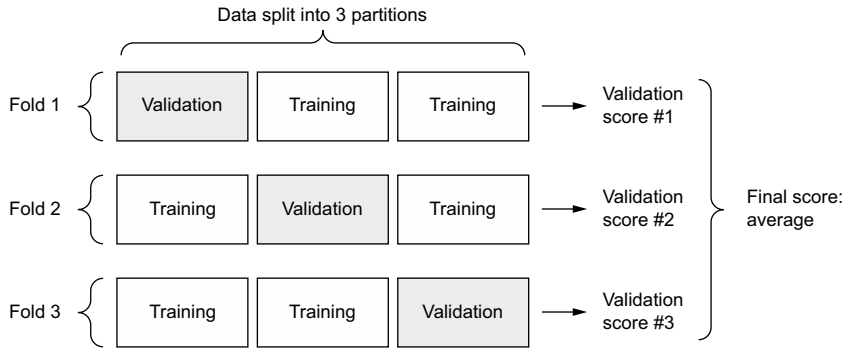


Figure 4.8 K-fold cross-validation with K=3

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.

Listing 4.26 K-fold validation

```
k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print(f"Processing fold #{i}")
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)
    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=16, verbose=0)
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose = 0)

Evaluates the model on the validation data

Running this with `num_epochs = 100` yields the following results:

```
>>> all_scores
[2.112449, 3.0801501, 2.6483836, 2.4275346]
>>> np.mean(all_scores)
2.5671294
```

The different runs do indeed show rather different validation scores, from 2.1 to 3.1. The average (2.6) is a much more reliable metric than any single score—that's the entire point of K-fold cross-validation. In this case, we're off by $2,600 on average, which is significant considering that the prices range from $10,000 to $50,000.

Let's try training the model a bit longer: 500 epochs. To keep a record of how well the model does at each epoch, we'll modify the training loop to save the per-epoch validation score log for each fold.

Listing 4.27 Saving the validation logs at each fold

```
num_epochs = 500
all_mae_histories = []
for i in range(k):
    print(f"Processing fold #{i}")
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)
    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=16, verbose=0)
    mae_history = history.history["val_mae"]
    all_mae_histories.append(mae_history)
```

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose=0)

We can then compute the average of the per-epoch MAE scores for all folds.

Listing 4.28 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 4.9.

Listing 4.29 Plotting validation scores

```
plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()
```

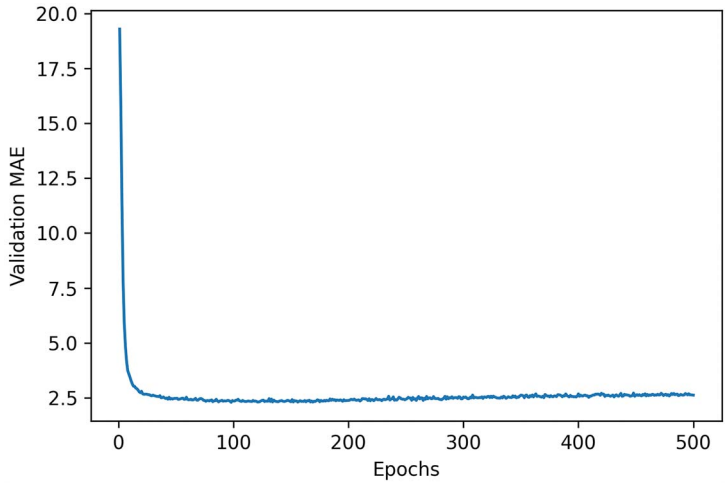


Figure 4.9 Validation MAE by epoch

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 4.30 Plotting validation scores, excluding the first 10 data points

```
truncated_mae_history = average_mae_history[10:]
plt.plot(range(1, len(truncated_mae_history) + 1), truncated_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()
```

As you can see in figure 4.10, validation MAE stops improving significantly after 120–140 epochs (this number includes the 10 epochs we omitted). Past that point, we 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.

Listing 4.31 Training the final model

```
model = build_model()
model.fit(train_data, train_targets,
          epochs=130, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

Gets a fresh, compiled model

Trains it on the entirety of the data

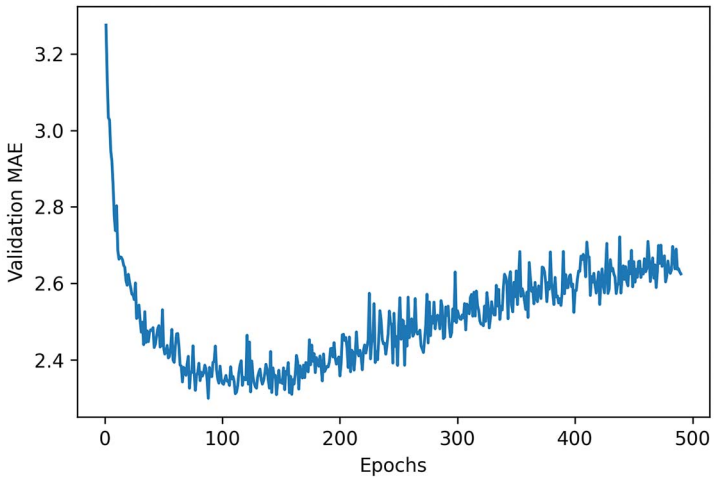


Figure 4.10 Validation MAE by epoch, excluding the first 10 data points

Here's the final result:

```
>>> test_mae_score
2.4642276763916016
```

We're still off by a bit under $2,500. It's an improvement! Just like with the two previous tasks, you can try varying the number of layers in the model, or the number of units per layer, to see if you can squeeze out a lower test error.

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 thousands of dollars:

```
>>> predictions = model.predict(test_data)
>>> predictions[0]
array([9.990133], dtype=float32)
```

The first house in the test set is predicted to have a price of about $10,000.

4.3.6 Wrapping up

Here's what you should take away from this scalar regression example:

- Regression is done using different loss functions than 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 mean absolute error (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.
  - The "Wrapping up" sections earlier in the chapter summarize the important points you've learned regarding each task.
  - Regression uses different loss functions and different evaluation metrics than classification.
- 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 the models you saw in the previous chapter, performance on the held-out validation data started improving 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.

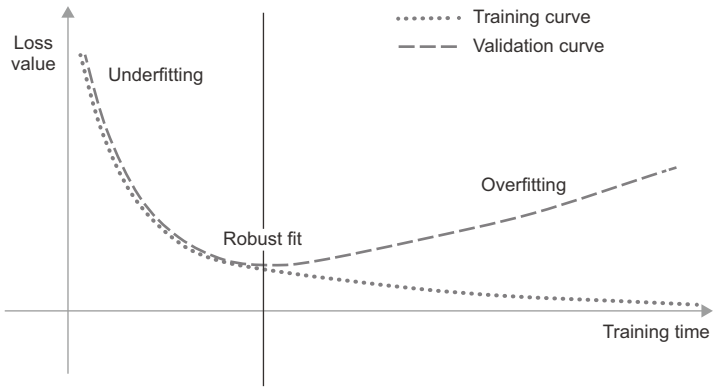


Figure 5.1 Canonical overfitting behavior

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, 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.

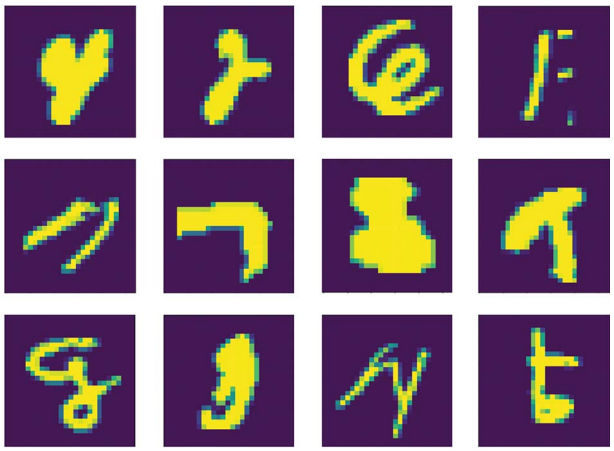


Figure 5.2 Some pretty weird MNIST training samples

What are these? I 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 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.

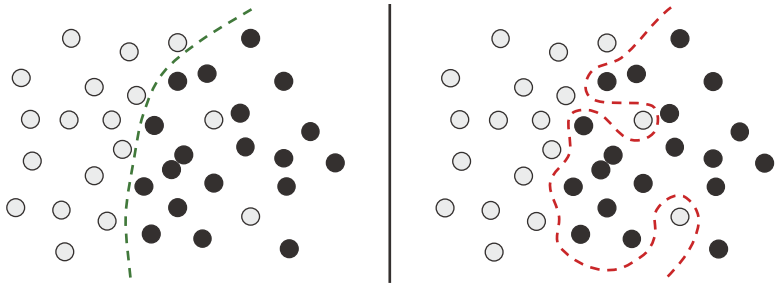


Figure 5.4 Dealing with outliers: robust fit vs. overfitting

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. 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 unripe, ripe, or rotten. These categories have no objective boundaries, so the same picture might be classified as either unripe 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 and sometimes by a clear sky, with some probability.

A model could overfit to such probabilistic data by being too confident about ambiguous regions of the feature space, like in figure 5.5. A more robust fit would ignore individual data points and look at the bigger picture.

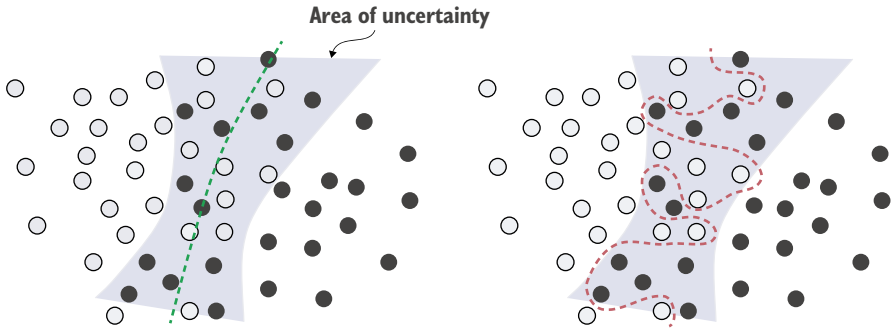


Figure 5.5 Robust fit vs. overfitting giving an ambiguous area of the feature space

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 leverage 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 something. 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 tensorflow.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

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model():
    model = keras.Sequential([
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    model.compile(optimizer="rmsprop",
                  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)
```

Let's compare how the validation accuracy of each model evolves over time.

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, "b--",
         label="Validation accuracy with zeros channels")
plt.title("Effect of noise channels on validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
```

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 (see figure 5.6)—purely through the influence of spurious correlations. The more noise channels you add, the further accuracy will degrade.

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*

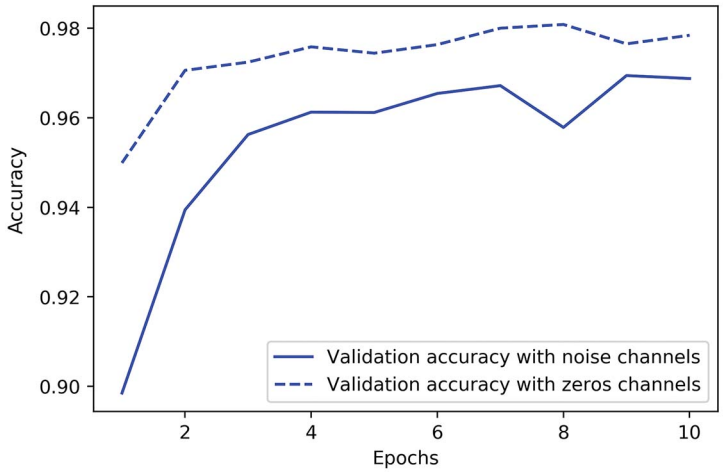


Figure 5.6 Effect of noise channels on validation accuracy

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 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)
```

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 how come 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.

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 (Euclidian) 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 at every point). A smooth surface within a 3D space is a 2D manifold. And so on.

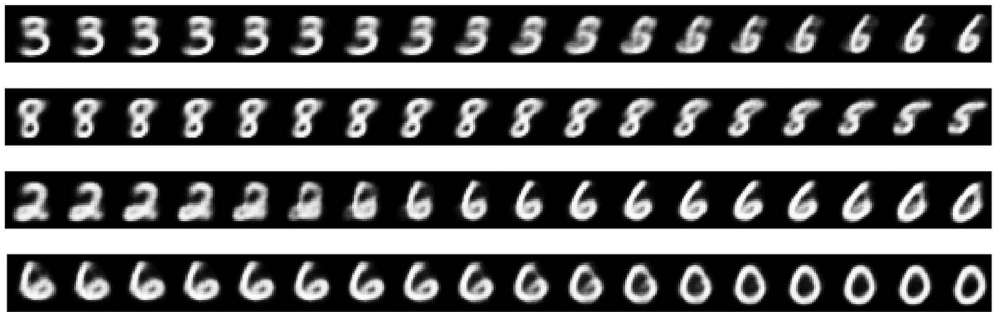


Figure 5.7 Different MNIST digits gradually morphing into one another, showing that the space of handwritten digits forms a "manifold." This image was generated using code from chapter 12.

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, but also for human faces, tree morphology, the sounds of the human voice, and even natural language.

The manifold hypothesis implies that

- 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 to say, 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.

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:

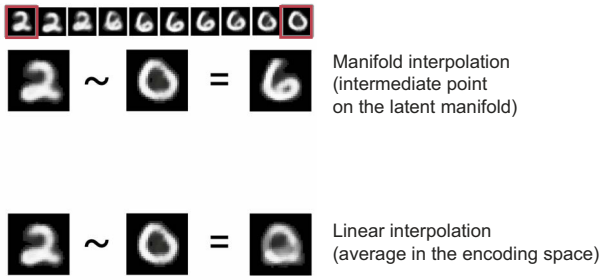


Figure 5.8 Difference between linear interpolation and interpolation on the latent manifold. Every point on the latent manifold of digits is a valid digit, but the average of two digits usually isn't.

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 14.

WHY DEEP LEARNING WORKS

Remember the crumpled paper ball metaphor from chapter 2? A sheet of paper represents a 2D manifold within 3D space (see figure 5.9). A deep learning model is a tool for uncrumpling paper balls, that is, for disentangling latent manifolds.

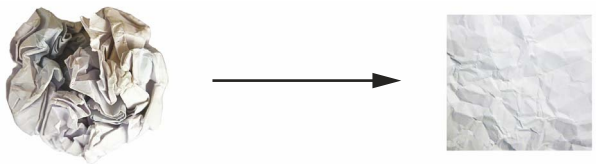


Figure 5.9 Uncrumpling a complicated manifold of data

A deep learning model is basically a very high-dimensional curve—a curve that is smooth and continuous (with additional constraints on its structure, originating from model architecture priors), since it needs to be differentiable. And that curve is fitted to data points via gradient descent, smoothly and incrementally. By its very nature, 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.

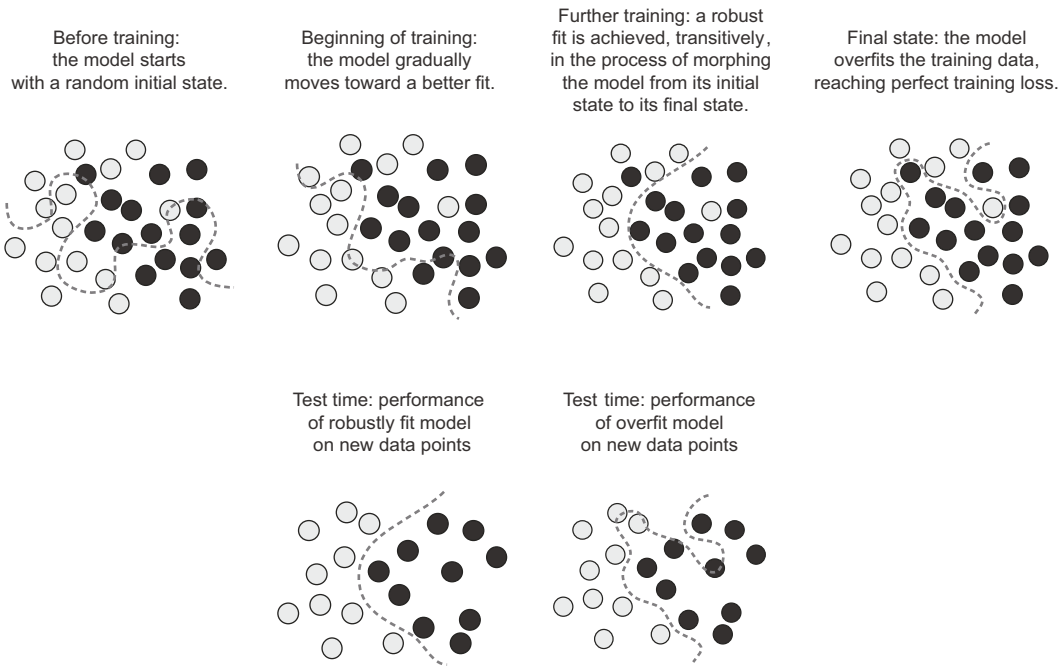


Figure 5.10 Going from a random model to an overfit model, and achieving a robust fit as an intermediate state

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.

- 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 particularly the case for image-processing models (discussed in chapters 8 and 9) and sequence-processing models (chapter 10). 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 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.

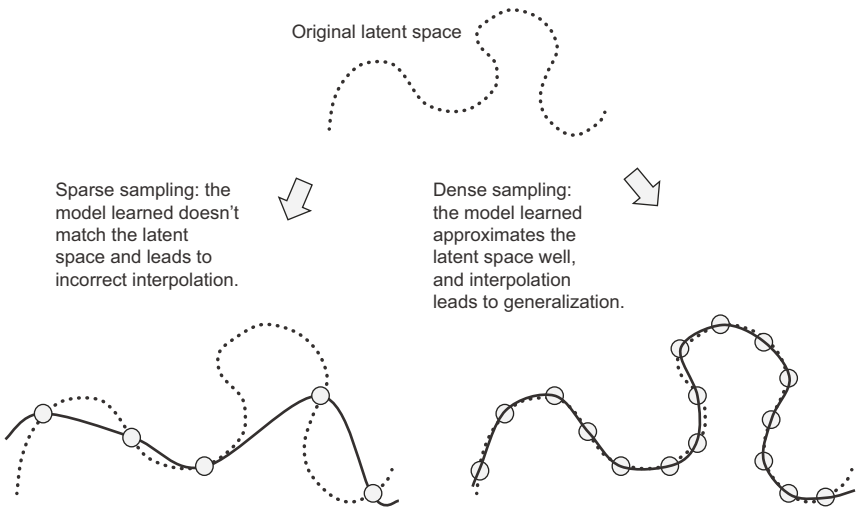


Figure 5.11 A dense sampling of the input space is necessary in order to learn a model capable of accurate generalization.

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'll need a way to assess how your model is currently doing. In the following section, you'll learn 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, I'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 for evaluating 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 on 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 holdout 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 HOLDOUT 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, in order 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, holdout validation looks like figure 5.12. Listing 5.5 shows a simple implementation.

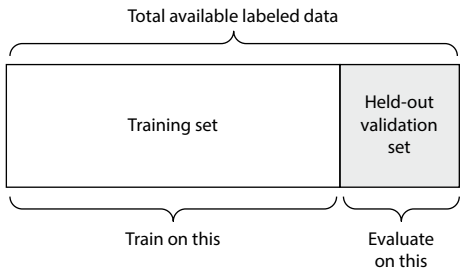


Figure 5.12 Simple holdout validation split

Listing 5.5 Holdout validation (note that labels are omitted for simplicity)

```
num_validation_samples = 10000
np.random.shuffle(data)
```

Shuffling the data is usually appropriate.

Defines the validation set

```
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

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.

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 holdout 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.

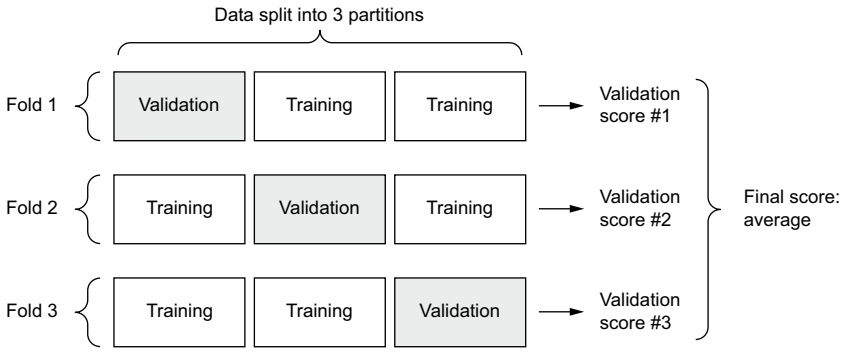


Figure 5.13 K-fold cross-validation with K=3

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

Creates a brand-new instance of the model (untrained)

Validation score: average of the validation scores of the k folds

Trains the final model on all non-test data available

Uses the remainder of the data as training data. Note that the + operator represents list concatenation, not summation.

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.

It's particularly important 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, and you can keep going. This baseline could be

the 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 will contain 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 that is able to overfit.

5.3.1 Tuning key gradient descent parameters

Sometimes training doesn't get started, or it 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 keeping 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(1.),
              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 30%–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(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 preceding 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 Leveraging 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 10, 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 leverage 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 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("Effect of insufficient model capacity on validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
```

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 where the training loss doesn't go down, this is an issue that can always be solved. If

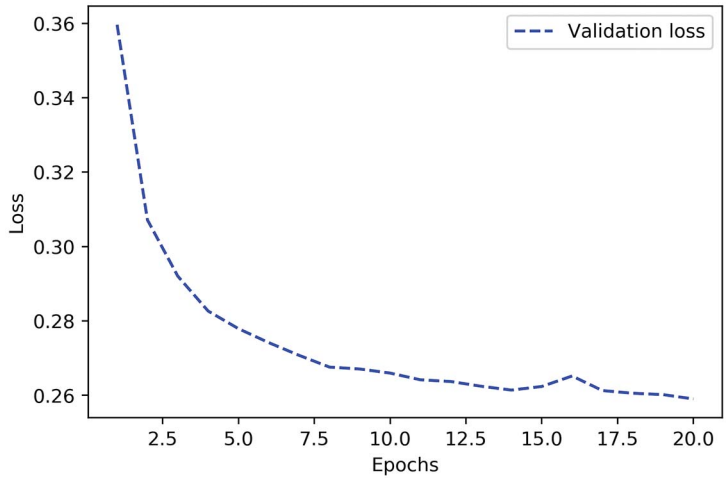


Figure 5.14 Effect of insufficient model capacity on loss curves

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 to say, one 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 96 units each:

```
model = keras.Sequential([
    layers.Dense(96, activation="relu"),
    layers.Dense(96, 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 validation curve now looks exactly like it should: the model fits fast and starts overfitting after 8 epochs (see figure 5.15).

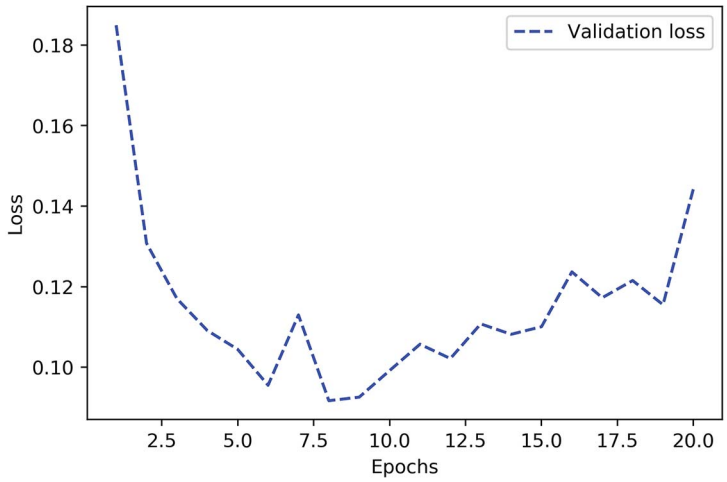


Figure 5.15 Validation loss for a model with appropriate capacity

5.4 Improving generalization

Once your model has shown itself to have some generalization power and to be able to overfit, it's time to switch your focus to 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, 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'll 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 to 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 5.16).

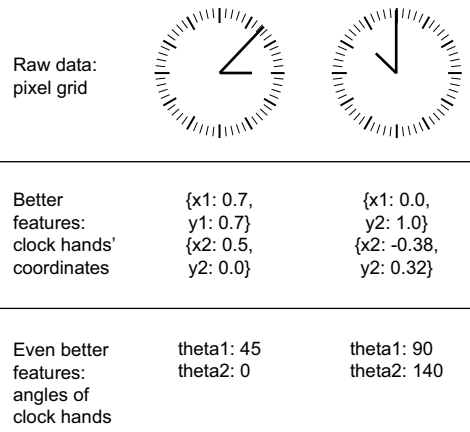


Figure 5.16 Feature engineering for reading the time on a clock

If you choose to use the raw pixels of the image as input data, 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), 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.

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, better organized. Doing so 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, 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 in 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, and then we would retrain a new model for exactly that number of epochs. This is pretty standard, but 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, and 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, more "generic"; thus it is 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; thus, in order 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) in order to find the correct model size for your data. The general workflow for finding 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. The following listing shows our original model.

Listing 5.10 Original model

```
from tensorflow.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.
    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 5.17 shows a comparison of the validation losses of the original model and the smaller model.

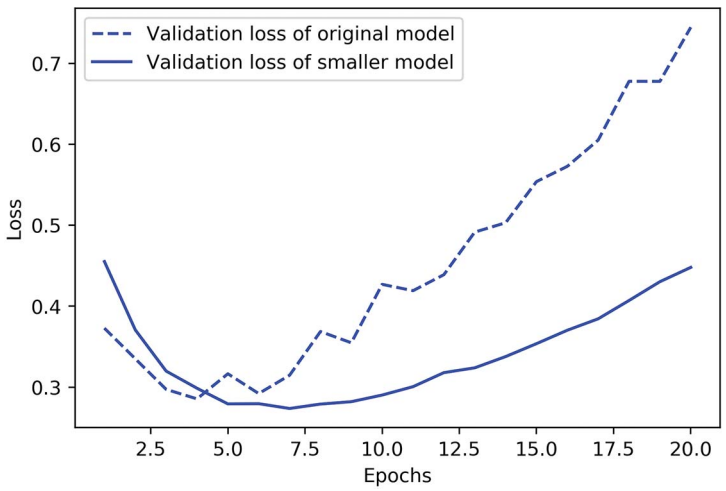


Figure 5.17 Original model vs. smaller model on IMDB review classification

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. While it is standard to work with models that are significantly overparameterized for what they're trying to learn, 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 and if its validation loss curve looks choppy with high-variance (although choppy validation metrics could also be a symptom of using an unreliable validation process, such as a validation split that's too small).

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 5.18 shows how the bigger model fares compared with the reference model.

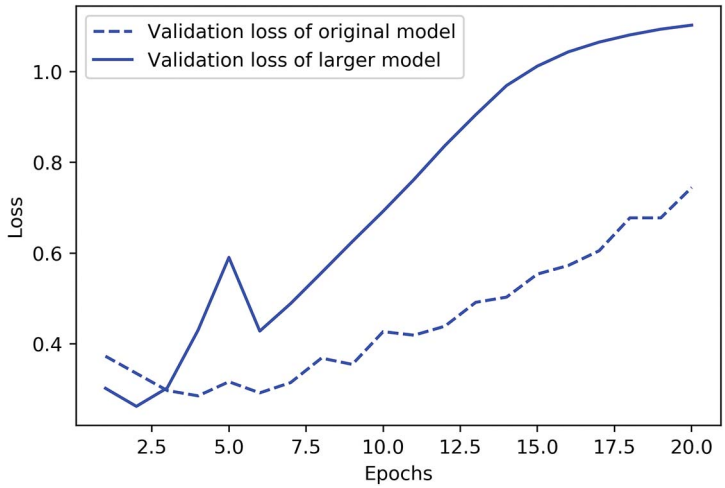


Figure 5.18 Original model vs. much larger model on IMDB review classification

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).

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 our initial movie-review classification model.

Listing 5.13 Adding L2 weight regularization to the model

```
from tensorflow.keras import regularizers
model = keras.Sequential([
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(16,
                 kernel_regularizer=regularizers.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)
```

In the preceding listing, `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 5.19 shows the impact 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.

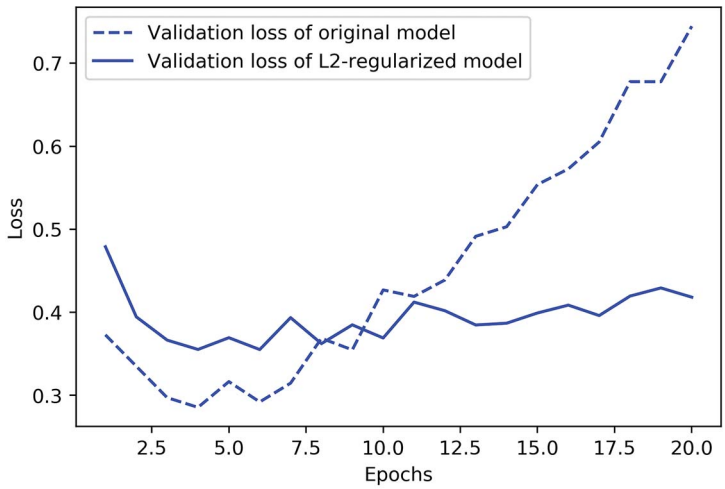


Figure 5.19 Effect of L2 weight regularization on validation loss

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 tensorflow.keras import regularizers
regularizers.l1(0.001)
regularizers.l1_l2(l1=0.001, l2=0.001)
```

L1 regularization

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 hasn't much impact on model capacity and generalization. In these cases, a different regularization technique is preferred: dropout.

ADDING DROPOUT

Dropout is one of the most effective and most commonly used regularization techniques for neural networks; it was developed by Geoff Hinton and his students at the University of Toronto. 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(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 5.20):

At training time

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
layer_output /= 0.5
```

Note that we're scaling up rather than scaling down in this case.

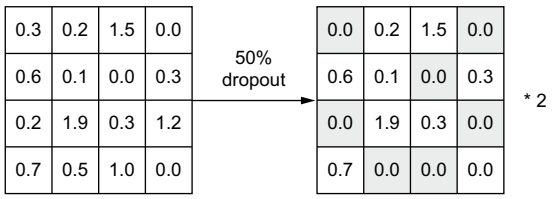


Figure 5.20 Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time the activation matrix is unchanged.

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. In his own words, "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 5.21 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.

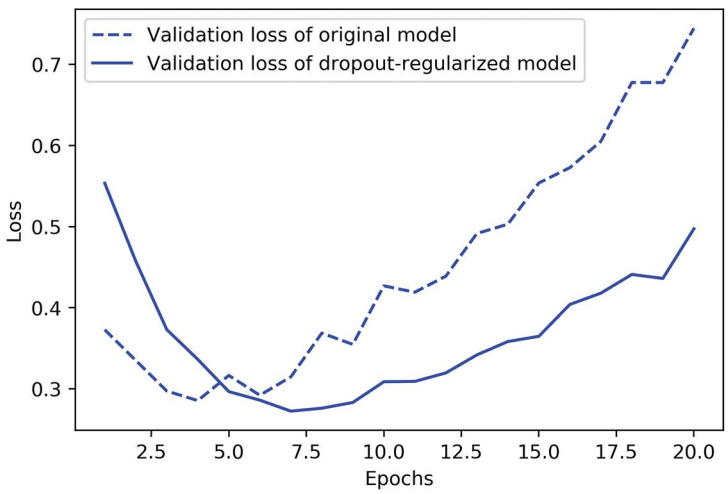


Figure 5.21 Effect of dropout on validation loss

To recap, these are the most common ways to maximize generalization and prevent overfitting in neural networks:

- Get more training data, or better training data.
- Develop better features.
- Reduce the capacity of the model.
- Add weight regularization (for smaller models).
- Add 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 holdout 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 for doing this include tuning your learning rate and batch size, leveraging 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

- Steps for framing a machine learning problem
- Steps for developing a working model
- Steps for 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 starting 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 archeological sites.

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 merely encode these same biases into a black-box algorithm that 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, this impact has a moral direction: technical choices are also ethical choices. Always be deliberate about the values you want your work to support.

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 a universal step-by-step blueprint that you can use to approach and solve any machine learning problem, like those in the previous list. This template will bring together and consolidate everything you've learned in chapters 4 and 5, and will give you the wider context that should anchor what you'll learn in the next chapters.

The universal workflow of machine learning is broadly structured in three parts:

1. *Define the task*—Understand the problem domain and the business logic underlying what the customer asked for. Collect a dataset, understand what the data represents, and choose how you will measure success on the task.
2. *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 and that can overfit, and then regularize and tune your model until you achieve the best possible generalization performance.
3. *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-generation model.

Let's dive in.

6.1 Define 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, and 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 Frame the problem

Framing a machine learning problem usually involves many detailed discussions with stakeholders. Here are the questions that should be on the 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 training data available: 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'll 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 the 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 in order 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 for finding new archeological sites from satellite images is an image-similarity ranking task: you need to retrieve new images that look the most like known archeological sites.

- What do existing solutions look like? Perhaps your customer already has a handcrafted 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 under consideration—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 you 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.

6.1.2 Collect 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's 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 archeologists 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 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 article "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 made the importance of data greater.

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, such as those for the music recommendation task or the 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 determine 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 it 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 worth investing in it early in a project.

BEWARE OF NON-REPRESENTATIVE 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 your data collection work.

Suppose you're developing an app where users can take pictures of a plate of food to find out the name of the dish. You train a model using pictures from an image-sharing social network that's popular with foodies. Come deployment time, 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 for 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 non-representative 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.

"DEWEY DEFEATS TRUMAN": A famous example of sampling bias

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.

6.1.3 Understand your data

It's pretty 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'll 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 and which 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 Choose 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 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 the area under a *receiver operating characteristic* (ROC) curve, abbreviated as ROC AUC, are common metrics. For class-imbalanced problems, ranking problems, or multilabel classification, you can use precision and recall, as well as a weighted form of accuracy or ROC AUC. 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); they showcase a wide range of problems and evaluation metrics.

6.2 Develop 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 already done, and skipping model deployment and maintenance, which are assumed to be handled by someone else. In fact, model development is only one step in the machine learning workflow, and if you ask me, 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 Prepare 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 typically be 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 in chapter 4, we started with text represented as lists of integers (standing for sequences of words), and we used one-hot encoding to turn them into a tensor of `float32` data. In the examples of classifying digits and predicting house prices, the data came in vectorized form, so we were able to skip this step.

VALUE NORMALIZATION

In the MNIST digit-classification example from chapter 2, we started with image data encoded as integers in the 0–255 range, encoding grayscale values. Before we fed this data into our network, we had to cast it to `float32` and divide by 255 so we'd end up with floating-point values in the 0–1 range. Similarly, when predicting house prices, we started with features that took a variety of ranges—some features had small floating-point values, and others had fairly large integer values. Before we fed this data into our network, we 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 homogenous*—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, we 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 first feature (the column of index 0 in the data) was the per capita crime rate. 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 categorial 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 Choose an evaluation protocol

As you 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 holdout validation set*—This is the way to go when you have plenty of data.
- *Doing K-fold cross-validation*—This is the right choice when you have too few samples for holdout validation to be reliable.
- *Doing iterated K-fold validation*—This is for performing highly accurate model evaluation when little data is available.

Pick one of these. In most cases, the first will work well enough. As you learned, though, always be mindful of the *representativity* of your validation set, and be careful not to have redundant samples between your training set and your validation set.

6.2.3 Beat 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.

The following table can help you choose a last-layer activation and a loss function for a few common problem types.

(continued)

Choosing the right last-layer activation and loss function for your model

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`

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 you 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 Scale up: Develop 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:

1. Add layers.
2. Make the layers bigger.
3. 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 Regularize and tune 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 13.

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; 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 Deploy the model

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 Explain your work to stakeholders and set 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 non-specialists towards 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 trade-offs that can only be handled with a deep understanding of the business context.

6.3.2 Ship 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 in 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

This is perhaps the common way to turn a model into a product: install TensorFlow on a server or cloud instance, and query the model's predictions via a REST API. You could build your own serving app using something like Flask (or any other Python web development library), or use TensorFlow's own library for shipping models as APIs, called *TensorFlow Serving* (www.tensorflow.org/tfx/guide/serving). With TensorFlow Serving, you can deploy a Keras model in minutes.

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 good fits 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 it 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. You may have 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. You can use the TensorFlow Model Optimization Toolkit to help with this (www.tensorflow.org/model_optimization).
- Getting the highest possible accuracy isn't mission critical for your task. There is always a trade-off 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.

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. 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, your go-to solution is TensorFlow Lite (www.tensorflow.org/lite). It's a framework for efficient on-device deep learning inference that runs on Android and iOS smartphones, as well as ARM64-based computers, Raspberry Pi, or certain microcontrollers. It includes a converter that can straightforwardly turn your Keras model into the TensorFlow Lite format.

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 having the model run directly in the browser, on the user's computer (utilizing GPU resources if they're 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.

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 (www.tensorflow.org/js), a JavaScript library for deep learning that implements almost all of the Keras API (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 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 deciding how much pruning you want to apply, you are in control of the trade-off 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 a quarter the size but remains near the accuracy of the original model.

The TensorFlow ecosystem includes a weight pruning and quantization toolkit (www.tensorflow.org/model_optimization) that is deeply integrated with the Keras API.

6.3.3 Monitor 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 the 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 Maintain 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 systems, it will 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. As such,

- 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 for your current model to classify—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 model development, 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 for 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: holdout 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 you set appropriate expectations with stakeholders.
  - Optimize a final model for inference, and ship a 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 Working with Keras: A deep dive

This chapter covers

- Creating Keras models with the `Sequential` class, the Functional API, and model subclassing
- Using built-in Keras training and evaluation loops
- Using Keras callbacks to customize training
- Using TensorBoard to monitor training and evaluation metrics
- Writing training and evaluation loops from scratch

You've now got some 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 even learned in chapter 3 how to inherit from the `Layer` class to create custom layers, and how to use the TensorFlow `GradientTape` 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. 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.

7.2 Different ways to build Keras models

There are three APIs for building models in Keras (see figure 7.1):

- The *Sequential model*, 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.

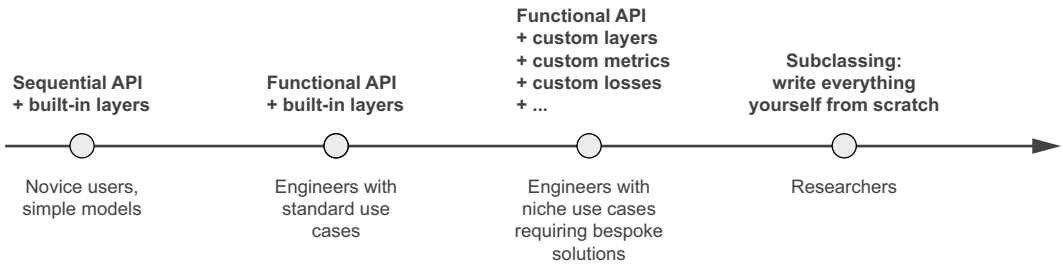


Figure 7.1 Progressive disclosure of complexity for model building

7.2.1 The Sequential model

The simplest way to build a Keras model is to use the Sequential model, which you already know about.

Listing 7.1 The `Sequential` class

```
from tensorflow import keras
from tensorflow.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, which is 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 saw in chapter 4 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 preceding Sequential model does not have any weights (listing 7.3) until you actually call it on some data, or call its `build()` method with an input shape (listing 7.4).

Listing 7.3 Models that aren't yet built have no weights

```
>>> model.weights
ValueError: Weights for model sequential_1 have not yet been created.
```

At that point, the model isn't built yet.

Listing 7.4 Calling a model for the first time to build it

```
>>> model.build(input_shape=(None, 3))
>>> model.weights
[<tf.Variable "dense_2/kernel:0" shape=(3, 64) dtype=float32, ... >,
 <tf.Variable "dense_2/bias:0" shape=(64,) dtype=float32, ... >
 <tf.Variable "dense_3/kernel:0" shape=(64, 10) dtype=float32, ... >,
 <tf.Variable "dense_3/bias:0" shape=(10,) dtype=float32, ... >]
```

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.

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
Trainable params: 906
Non-trainable params: 0
_________________________________________________________________
```

As you can see, this model happens to be named "sequential_1." You can 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
Trainable params: 906
Non-trainable params: 0
_________________________________________________________________
```

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` 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 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
Trainable params: 256
Non-trainable params: 0
_________________________________________________________________
>>> 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
Trainable params: 906
Non-trainable params: 0
_________________________________________________________________
```

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 stack of two layers 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)
```

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 and 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)
```

Here's the summary of our model:

```
>>> model.summary()
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
my_input (InputLayer)        [(None, 3)]               0
_________________________________________________________________
```

```
dense_6 (Dense)              (None, 64)                256
_________________________________________________________________
dense_7 (Dense)              (None, 10)                650
=================================================================
Total params: 906
Trainable params: 906
Non-trainable params: 0
_________________________________________________________________
```

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 one-hot encoded)

We can encode the text inputs as arrays of ones and zeros of size `vocabulary_size` (see chapter 11 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")(features)

priority = layers.Dense(1, activation="sigmoid", name="priority")(features)
department = layers.Dense(
    num_departments, activation="softmax", name="department")(features)

model = keras.Model(inputs=[title, text_body, tags],
                    outputs=[priority, department])
```

Combine input features into a single tensor, features, by concatenating them.

Define model inputs.

Apply an intermediate layer to recombine input features into richer representations.

Define model outputs.

Create 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 be in 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

# Dummy input data
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))

# Dummy target data
priority_data = np.random.random(size=(num_samples, 1))
department_data = np.random.randint(0, 2, size=(num_samples, num_departments))

model.compile(optimizer="rmsprop",
              loss=["mean_squared_error", "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])
```

If you don't want to rely on input order (for instance, because you have many inputs or outputs), you can also leverage the names you gave to the `Input` objects and 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="rmsprop",
              loss={"priority": "mean_squared_error", "department":
                    "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 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 (see figure 7.2).

```
keras.utils.plot_model(model, "ticket_classifier.png")
```

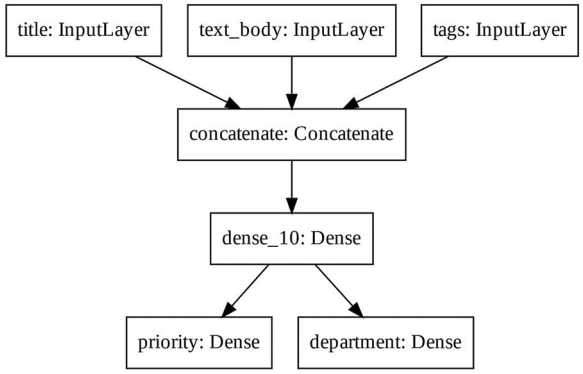


Figure 7.2 Plot generated by `plot_model()` on our ticket classifier model

You can add to this plot the input and output shapes of each layer in the model, which can be helpful during debugging (see figure 7.3).

```
keras.utils.plot_model(
    model, "ticket_classifier_with_shape_info.png", show_shapes=True)
```

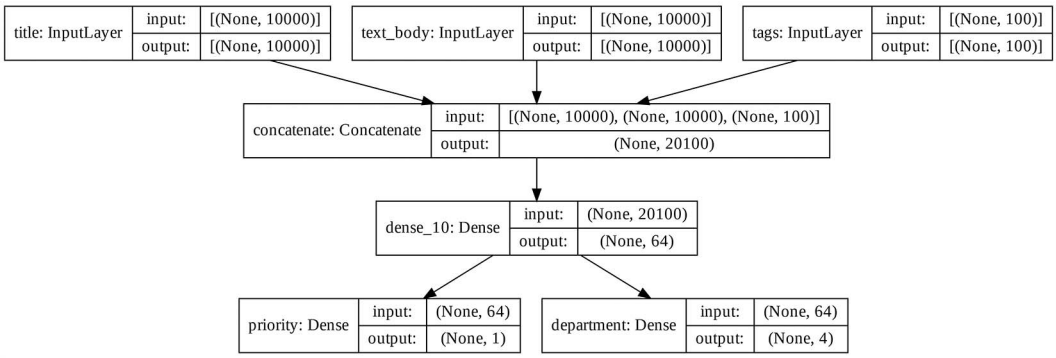


Figure 7.3 Model plot with shape information added

The "None" in the tensor shapes represents the batch size: this model allows batches of any size.

Access to layer connectivity also means that you can inspect and reuse individual nodes (layer calls) in the graph. The `model.layers` model property 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
[<tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d358>,
 <tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d2e8>,
 <tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d470>,
 <tensorflow.python.keras.layers.merge.Concatenate at 0x7fa963f9d860>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa964074390>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa963f9d898>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa963f95470>]
>>> model.layers[3].input
[<tf.Tensor "title:0" shape=(None, 10000) dtype=float32>,
 <tf.Tensor "text_body:0" shape=(None, 10000) dtype=float32>,
 <tf.Tensor "tags:0" shape=(None, 100) dtype=float32>]
>>> model.layers[3].output
<tf.Tensor "concatenate/concat:0" shape=(None, 20100) dtype=float32>
```

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 previous model—you want to estimate 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 start from the intermediate features of your previous model, since you have access to them, like this.

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 (see figure 7.4):

```
keras.utils.plot_model(
    new_model, "updated_ticket_classifier.png", show_shapes=True)
```

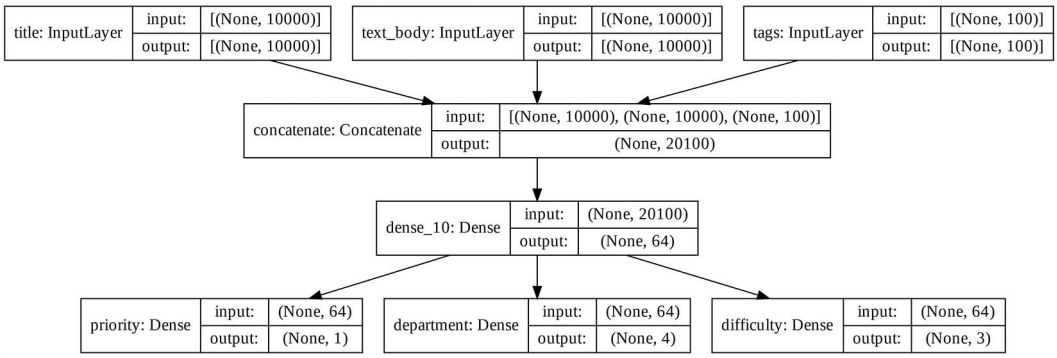


Figure 7.4 Plot of our new model

7.2.3 Subclassing the Model class

The last model-building pattern you should know about is the most advanced one: `Model` subclassing. You 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"]
        text_body = inputs["text_body"]
        tags = inputs["tags"]

        features = self.concat_layer([title, text_body, tags])
        features = self.mixing_layer(features)
```

Don't forget to call the super() constructor!

Define sublayers in the constructor.

Define the forward pass in the call() method.

```
        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 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 `fit()`, `evaluate()`, and `predict()` methods. 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 arguments must match exactly what gets returned by call()—here, a list of two elements.

```
model.compile(optimizer="rmsprop",
              loss=["mean_squared_error", "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 structure of the target data must match exactly what is returned by the call() method—here, a list of two elements.

The structure of the input data must match exactly what is expected by the call() method—here, a dict with keys title, text_body, and tags.

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. 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 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, or `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 trade-off 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—I recommend using it over model subclassing.

Going forward, all examples in this book will use the Functional API, simply because all 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. They 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, take a look at the following listing.

Listing 7.17 The standard workflow: `compile()`, `fit()`, `evaluate()`, `predict()`

```
from tensorflow.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)
```

Create a model (we factor this into a separate function so as to reuse it later).

```
    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="rmsprop",
              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)
```

Load your data, reserving some for validation.

Compile the model by specifying its optimizer, the loss function to minimize, and the metrics to monitor.

Use fit() to train the model, optionally providing validation data to monitor performance on unseen data.

Use evaluate() to compute the loss and metrics on new data.

Use predict() to compute classification probabilities on new data.

There are a couple of ways you can customize this simple workflow:

- Provide your own custom metrics.
- Pass *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 measuring 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, and 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. Like layers, a metric has an internal state stored in TensorFlow 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

```
import tensorflow as tf

class RootMeanSquaredError(keras.metrics.Metric):
```

Subclass the Metric class.

Define the state variables in the constructor. Like for layers, you have access to the add_weight() method.

To match our MNIST model, we expect categorical predictions and integer labels.

```
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", dtype="int32")

def update_state(self, y_true, y_pred, sample_weight=None):
    y_true = tf.one_hot(y_true, depth=tf.shape(y_pred)[1])
    mse = tf.reduce_sum(tf.square(y_true - y_pred))
    self.mse_sum.assign_add(mse)
    num_samples = tf.shape(y_pred)[0]
    self.total_samples.assign_add(num_samples)
```

Implement 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. 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 tf.sqrt(self.mse_sum / tf.cast(self.total_samples, tf.float32))
```

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 with 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="rmsprop",
              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 displaying 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 from 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 proper number of epochs to train for, 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="val_accuracy",
        patience=2,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath="checkpoint_path.keras",
        monitor="val_loss",
        save_best_only=True,
    )
]
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              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 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.

Note that 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')`. To reload the model you've saved, just 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 `keras.callbacks.Callback` class. 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)
on_batch_begin(batch, logs)
on_batch_end(batch, logs)
on_train_begin(logs)
on_train_end(logs)
```

Called at the start of every epoch

Called at the end of every epoch

Called right before processing each batch

Called right after processing each batch

Called at the start of training

Called at the end of training

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 their first argument (an integer).

Here's a simple example that saves a list of per-batch loss values during training and saves a graph of 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}")
        self.per_batch_losses = []
```

Let's test-drive it:

```
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              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.

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. This inspires your next idea. The more iterations of this loop you're able to run, the more refined and

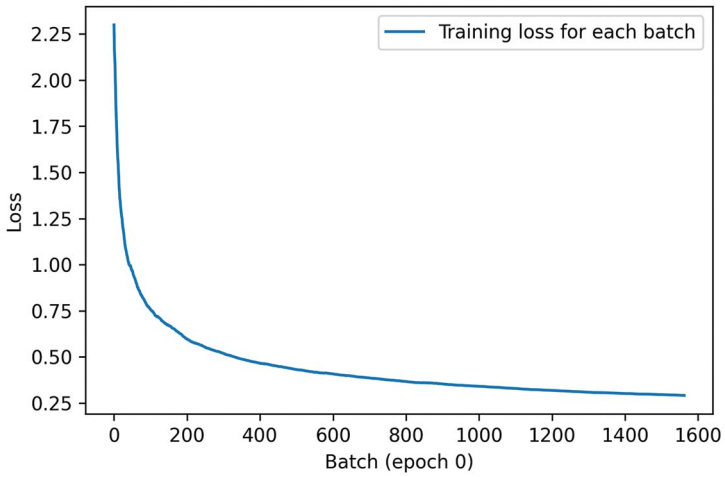


Figure 7.5 The output of our custom history plotting callback

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's results? That's where TensorBoard comes in (see figure 7.6).

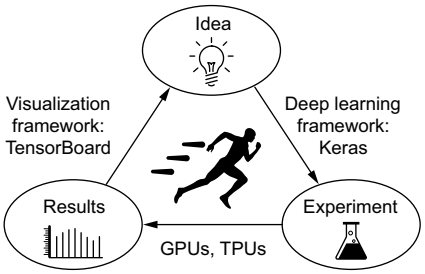


Figure 7.6 The loop of progress

TensorBoard (www.tensorflow.org/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 to use 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="rmsprop",
              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 be already 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 in order 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 (see figure 7.7).

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

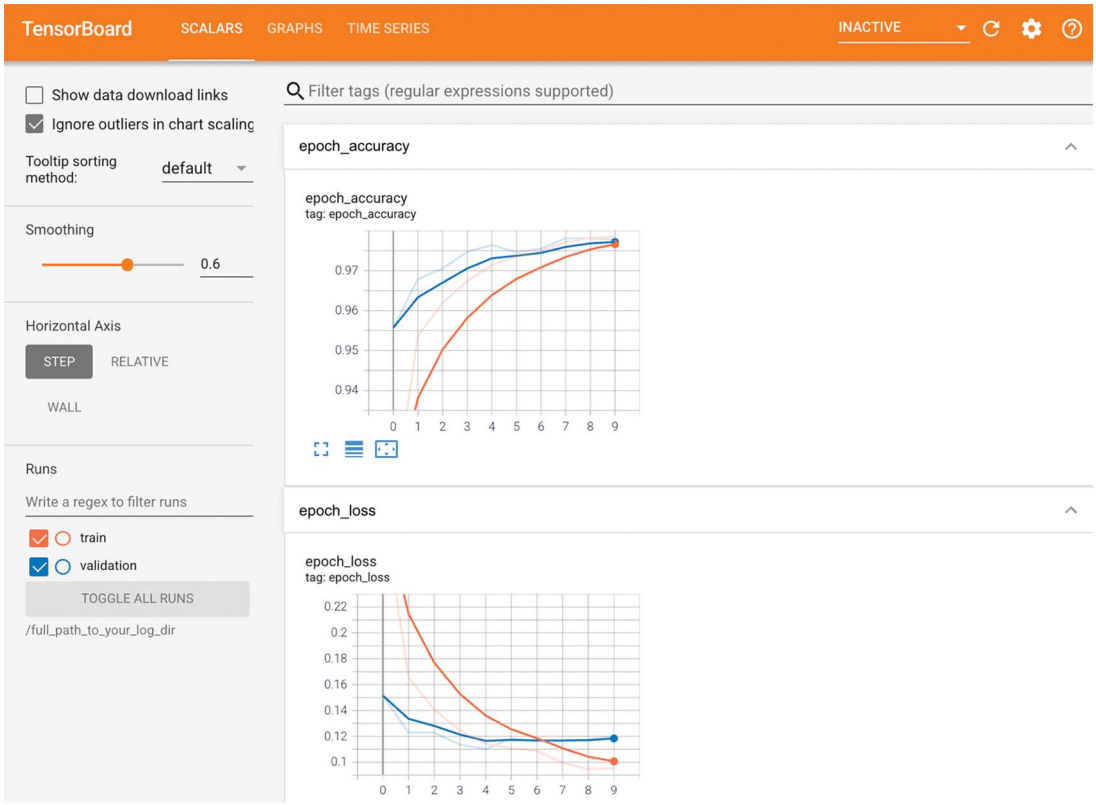


Figure 7.7 TensorBoard can be used for easy monitoring of training and evaluation metrics.

category. There are other setups where no explicit targets are present, such as *generative learning* (which we will discuss in chapter 12), *self-supervised learning* (where targets are obtained from the inputs), and *reinforcement learning* (where learning is driven by occasional "rewards," much like training a dog). 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 already saw 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) inside a gradient tape 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.

7.4.1 Training versus 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 `gradients = tape.gradient(loss, model.weights)`. 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 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 `tape.gradients(loss, model.weights)`, but rather `tape.gradients(loss, model.trainable_weights)`. Indeed, layers and models own two kinds of weights:

- *Trainable weights*—These are 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*—These 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 discuss in chapter 9. The `BatchNormalization` layer needs non-trainable weights in order 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 learned about in chapter 6).

Taking into account these two details, a supervised-learning training step ends up looking like this:

```
def train_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs, training=True)
        loss = loss_fn(targets, predictions)
```

```
    gradients = tape.gradients(loss, model.trainable_weights)
    optimizer.apply_gradients(zip(model.trainable_weights, gradients))
```

7.4.2 Low-level usage of metrics

In a low-level training loop, you will probably want to leverage 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:

```
metric = keras.metrics.SparseCategoricalAccuracy()
targets = [0, 1, 2]
predictions = [[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 = [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).

7.4.3 A complete training and evaluation loop

Let's combine the forward pass, backward pass, and metrics tracking into a `fit()`-like training step function that takes a batch of data and targets and returns the logs that would get displayed by the `fit()` progress bar.

Listing 7.21 Writing a step-by-step training loop: the training step function

```
model = get_mnist_model()

loss_fn = keras.losses.SparseCategoricalCrossentropy()
optimizer = keras.optimizers.RMSprop()
metrics = [keras.metrics.SparseCategoricalAccuracy()]
loss_tracking_metric = keras.metrics.Mean()

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(zip(gradients, model.trainable_weights))
```

Prepare the loss function.

Prepare the optimizer.

Prepare the list of metrics to monitor.

Prepare a Mean metric tracker to keep track of the loss average.

Run the forward pass. Note that we pass training=True.

Run the backward pass. Note that we use model.trainable_weights.

```
    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs[metric.name] = metric.result()

    loss_tracking_metric.update_state(loss)
    logs["loss"] = loss_tracking_metric.result()
    return logs
```

Keep track of metrics.

Keep track of the loss average.

Return the current values of the metrics and the loss.

We will need to reset the state of our metrics at the start of each epoch and before running evaluation. Here's a utility function to do it.

Listing 7.22 Writing a step-by-step training loop: resetting the metrics

```
def reset_metrics():
    for metric in metrics:
        metric.reset_state()
    loss_tracking_metric.reset_state()
```

We can now lay out our complete training loop. Note that we use a `tf.data.Dataset` object to turn our NumPy data into an iterator that iterates over the data in batches of size 32.

Listing 7.23 Writing a step-by-step training loop: the loop itself

```
training_dataset = tf.data.Dataset.from_tensor_slices(
    (train_images, train_labels))
training_dataset = training_dataset.batch(32)
epochs = 3
for epoch in range(epochs):
    reset_metrics()
    for inputs_batch, targets_batch in training_dataset:
        logs = train_step(inputs_batch, targets_batch)
    print(f"Results at the end of epoch {epoch}")
    for key, value in logs.items():
        print(f"...{key}: {value:.4f}")
```

And here's the evaluation loop: a simple `for` loop that repeatedly calls a `test_step()` function, which processes a single batch of data. The `test_step()` function is just a subset of the logic of `train_step()`. It omits the code that deals with updating the weights of the model—that is to say, everything involving the `GradientTape` and the optimizer.

Listing 7.24 Writing a step-by-step evaluation loop

```
def test_step(inputs, targets):
    predictions = model(inputs, training=False)
    loss = loss_fn(targets, predictions)

    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs["val_" + metric.name] = metric.result()
```

Note that we pass training=False.

```
    loss_tracking_metric.update_state(loss)
    logs["val_loss"] = loss_tracking_metric.result()
    return logs

val_dataset = tf.data.Dataset.from_tensor_slices((val_images, val_labels))
val_dataset = val_dataset.batch(32)
reset_metrics()
for inputs_batch, targets_batch in val_dataset:
    logs = test_step(inputs_batch, targets_batch)
print("Evaluation results:")
for key, value in logs.items():
    print(f"...{key}: {value:.4f}")
```

Congrats—you've just reimplemented `fit()` and `evaluate()`! Or almost: `fit()` and `evaluate()` support many more features, including large-scale distributed computation, which requires a bit more work. It also includes several key performance optimizations.

Let's take a look at one of these optimizations: TensorFlow function compilation.

7.4.4 Make it fast with tf.function

You may have noticed that your custom loops are running significantly slower than the built-in `fit()` and `evaluate()`, despite implementing essentially the same logic. That's because, by default, TensorFlow code is executed line by line, *eagerly*, much like NumPy code or regular Python code. Eager execution makes it easier to debug your code, but it is far from optimal from a performance standpoint.

It's more performant to *compile* your TensorFlow code into a *computation graph* that can be globally optimized in a way that code interpreted line by line cannot. The syntax to do this is very simple: just add a `@tf.function` to any function you want to compile before executing, as shown in the following listing.

Listing 7.25 Adding a `@tf.function` decorator to our evaluation-step function

```
@tf.function                                  <-- This is the only line that changed.
def test_step(inputs, targets):
    predictions = model(inputs, training=False)
    loss = loss_fn(targets, predictions)

    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs["val_" + metric.name] = metric.result()

    loss_tracking_metric.update_state(loss)
    logs["val_loss"] = loss_tracking_metric.result()
    return logs

val_dataset = tf.data.Dataset.from_tensor_slices((val_images, val_labels))
val_dataset = val_dataset.batch(32)
reset_metrics()
```

```
for inputs_batch, targets_batch in val_dataset:
    logs = test_step(inputs_batch, targets_batch)
print("Evaluation results:")
for key, value in logs.items():
    print(f"...{key}: {value:.4f}")
```

On the Colab CPU, we go from taking 1.80 s to run the evaluation loop to only 0.8 s. Much faster!

Remember, while you are debugging your code, prefer running it eagerly, without any `@tf.function` decorator. It's easier to track bugs this way. Once your code is working and you want to make it fast, add a `@tf.function` decorator to your training step and your evaluation step—or any other performance-critical function.

7.4.5 Leveraging fit() with a custom training loop

In the previous sections, we were writing our own training loop 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 or built-in support for distributed training.

What if you need a custom training algorithm, but you still want to leverage the power of the built-in Keras training logic? 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 a simple example:

- We create a new class that subclasses `keras.Model`.
- We override the method `train_step(self, data)`. Its contents are nearly identical to what we used in the previous section. It returns a dictionary mapping metric names (including the loss) to their current values.
- We implement a `metrics` property that tracks the model's `Metric` instances. This enables the model to automatically call `reset_state()` on the model's metrics 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.

Listing 7.26 Implementing a custom training step to use with `fit()`

```
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)
```

This metric object will be used to track the average of per-batch losses during training and evaluation.

We override the train_step method.

We use self(inputs, training=True) instead of model(inputs, training=True), since our model is the class itself.

```
        gradients = tape.gradient(loss, model.trainable_weights)
        optimizer.apply_gradients(zip(gradients, model.trainable_weights))

        loss_tracker.update_state(loss)
        return {"loss": loss_tracker.result()}

    @property
    def metrics(self):
        return [loss_tracker]
```

We update the loss tracker metric that tracks the average of the loss.

We return the average loss so far by querying the loss tracker metric.

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:

```
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.RMSprop())
model.fit(train_images, train_labels, epochs=3)
```

There are a couple of points to note:

- 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` decorator when you override `train_step`—the framework does it for you.

Now, what about metrics, and what about configuring the loss via `compile()`? After you've called `compile()`, you get access to the following:

- `self.compiled_loss`—The loss function you passed to `compile()`.
- `self.compiled_metrics`—A wrapper for the list of metrics you passed, which allows you to call `self.compiled_metrics.update_state()` to update all of your metrics at once.
- `self.metrics`—The actual list of metrics you passed to `compile()`. Note that it also includes a metric that tracks the loss, similar to what we did manually with our `loss_tracking_metric` earlier.

We can thus write

```
class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        with tf.GradientTape() as tape:
            predictions = self(inputs, training=True)
            loss = self.compiled_loss(targets, predictions)
        gradients = tape.gradient(loss, model.trainable_weights)
```

Compute the loss via self.compiled_loss.

```
        optimizer.apply_gradients(zip(gradients, model.trainable_weights))
        self.compiled_metrics.update_state(targets, predictions)
        return {m.name: m.result() for m in self.metrics}
```

Update the model's metrics via self.compiled_metrics.

Return a dict mapping metric names to their current value.

Let's try it:

```
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.RMSprop(),
              loss=keras.losses.SparseCategoricalCrossentropy(),
              metrics=[keras.metrics.SparseCategoricalAccuracy()])
model.fit(train_images, train_labels, epochs=3)
```

That was a lot of information, but you now 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 inter-operate together.
- 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 take control of what `fit()` does by overriding the `train_step()` method.
- Beyond `fit()`, you can also write your own training loops entirely from scratch. This is useful for researchers implementing brand-new training algorithms.

8 Introduction to deep learning for computer vision

This chapter covers

- Understanding convolutional neural networks (convnets)
- Using data augmentation to mitigate overfitting
- Using a pretrained convnet to do feature extraction
- Fine-tuning a pretrained convnet

Computer vision is the earliest and biggest success story of deep learning. Every day, you're interacting with deep vision models—via Google Photos, Google image search, YouTube, video filters in camera apps, 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.

Computer vision is the problem domain that led to the initial rise of deep learning between 2011 and 2015. A type of deep learning model 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. I remember exhorting an ex-professor of mine, in February 2014, to pivot to deep learning. "It's the next big thing!" I would say. "Well, maybe it's just a fad," he replied. By 2016, his entire lab was doing deep learning. There's no stopping an idea whose time has come.

This chapter introduces convolutional neural networks, also known as *convnets*, the type of deep learning model that is now used almost universally in 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 that classifies 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 our densely connected model from chapter 2 out of the water.

The following listing shows 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

```
from tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
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.Flatten()(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: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 128)         73856
_________________________________________________________________
flatten (Flatten)            (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 10)                11530
=================================================================
Total params: 104,202
Trainable params: 104,202
Non-trainable params: 0
_________________________________________________________________
```

You can see that the output of every `Conv2D` and `MaxPooling2D` layer is a rank-3 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 (32, 64, or 128).

After the last `Conv2D` layer, we end up with an output of shape `(3, 3, 128)`—a 3×3 feature map of 128 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 `Flatten` layer before adding the `Dense` layers.

Finally, we 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 tensorflow.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="rmsprop",
              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—in the case of images, patterns found in small 2D windows of the inputs (see figure 8.1). In the previous example, these windows were all 3×3.

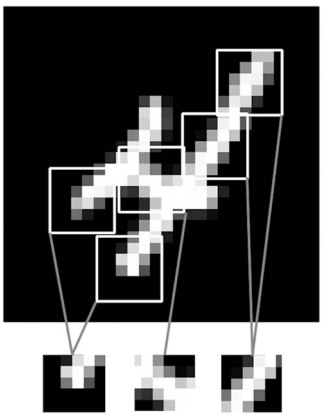


Figure 8.1 Images can be broken into local patterns such as edges, textures, and so on.

This key characteristic gives convnets two interesting properties:

- *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.* 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 (see figure 8.2). This allows convnets to efficiently learn increasingly complex and abstract visual concepts, because *the visual world is fundamentally spatially hierarchical.*

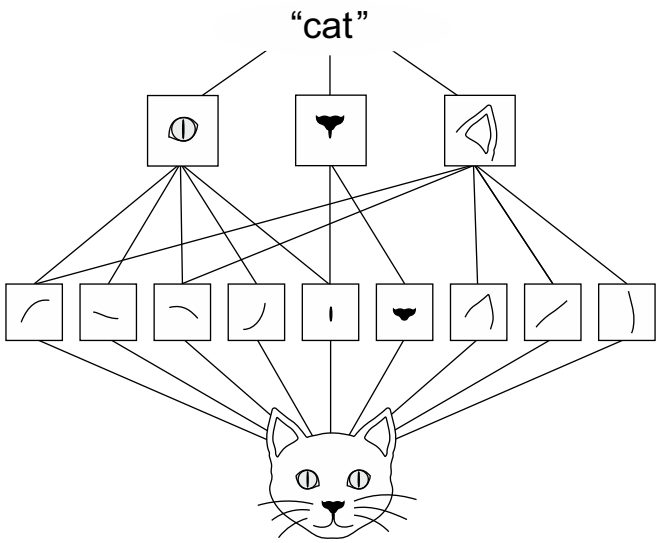


Figure 8.2 The visual world forms a spatial hierarchy of visual modules: elementary lines or textures combine into simple objects such as eyes or ears, which combine into high-level concepts such as "cat."

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, 32)`: it computes 32 filters over its input. Each of these 32 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).

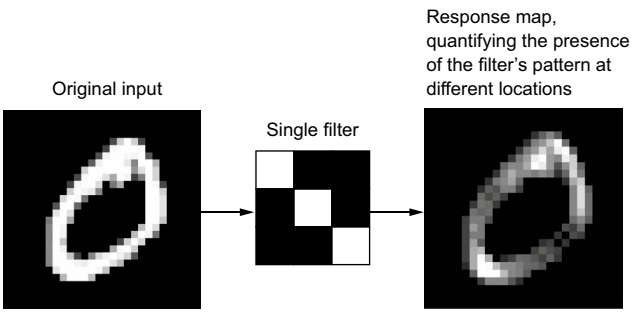


Figure 8.3 The concept of a response map: a 2D map of the presence of a pattern at different locations in an input

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.

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*—This is 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 (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.

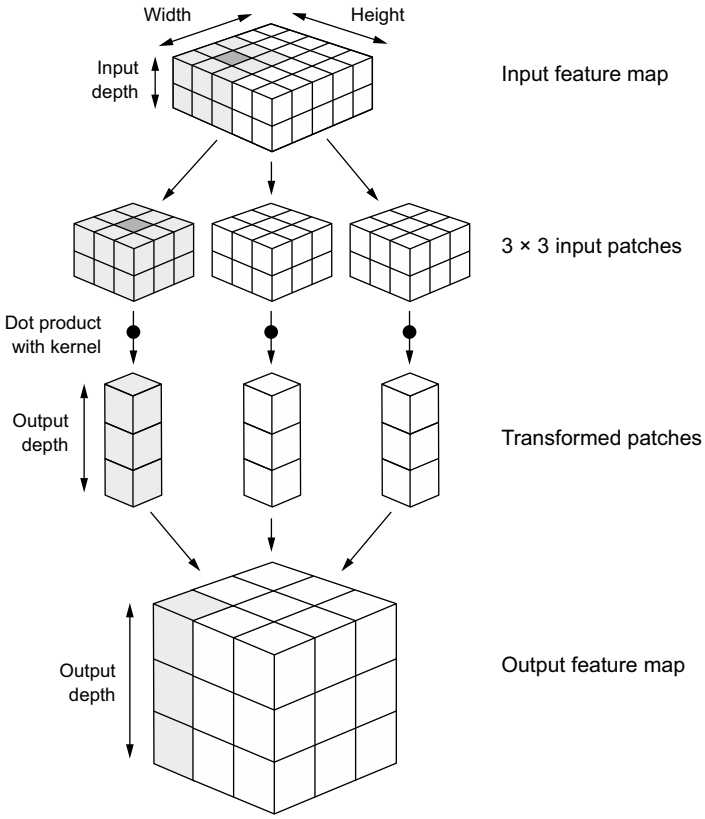


Figure 8.4 How convolution works

Note that the output width and height may differ from the input width and height for two reasons:

- Border effects, which can be countered by padding the input feature map
- The use of *strides*, which I'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.

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

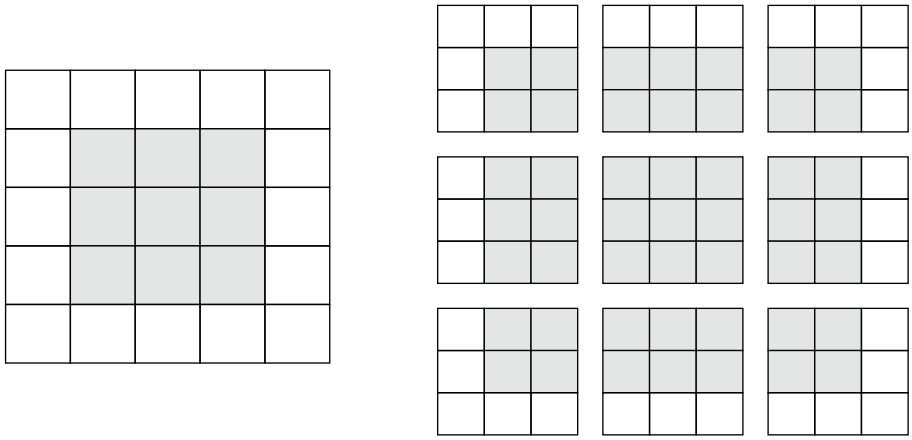


Figure 8.5 Valid locations of 3 × 3 patches in a 5 × 5 input feature map

and columns on each side of the input feature map so as to make it possible to fit center 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).

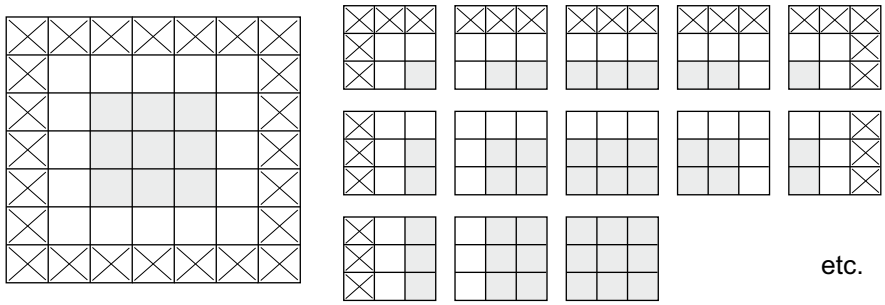


Figure 8.6 Padding a 5 × 5 input in order to be able to extract 25 3 × 3 patches

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*. Our 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).

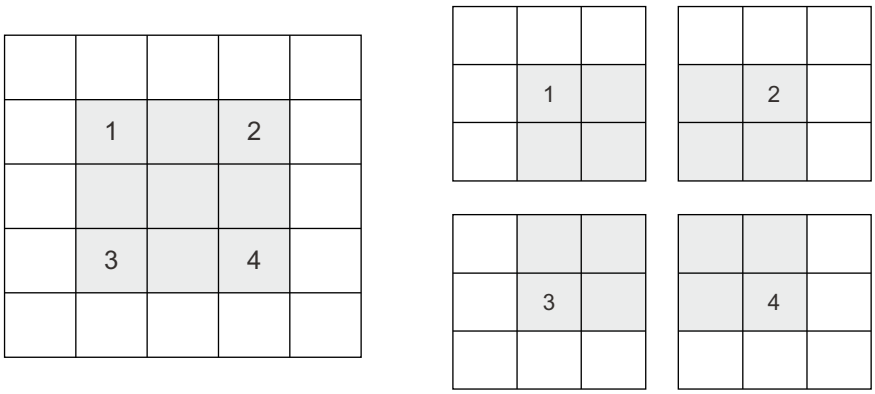


Figure 8.7 3 × 3 convolution patches with 2 × 2 strides

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 see 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, in order 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 the following listing.

Listing 8.5 An incorrectly structured convnet missing its max-pooling layers

```
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(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: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 22, 22, 128)       73856
_________________________________________________________________
flatten_1 (Flatten)          (None, 61952)             0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                619530
=================================================================
Total params: 712,202
Trainable params: 712,202
Non-trainable params: 0
_________________________________________________________________
```

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 22 × 22 × 128 = 61,952 total coefficients per sample. This is huge. When you flatten it to stick a `Dense` layer of size 10 on top, that layer would have over half a million parameters. This is far too large for such a small model and would result in intense overfitting.

In short, the reason to use downsampling is to reduce the number of feature-map coefficients to process, as well as to induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows (in terms of the fraction of the original input 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. 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*. 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 should 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 in a dataset containing 5,000 pictures of cats and dogs (2,500 cats, 2,500 dogs). 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 you 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 70%. 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 an accuracy of 80–85%.

In the next section, we'll review two more essential techniques for applying deep learning to small datasets: *feature extraction with a pretrained model* (which will get us to an accuracy of 97.5%) 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 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 models (usually trained on the ImageNet dataset) are now 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 (see the "Downloading a Kaggle dataset in Google Colaboratory" sidebar).

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 as the `kaggle` package, which is preinstalled on Colab. Downloading this dataset is as easy as running the following command in a Colab cell:

```
!kaggle competitions download -c dogs-vs-cats
```

However, access to the API is restricted to Kaggle users, so in order to run the preceding command, you first need to authenticate yourself. The `kaggle` package will look for your login credentials in a JSON file located at ~/.kaggle/kaggle.json. Let's create this file.

First, you need to create a Kaggle API key and download it to your local machine. Just navigate to the Kaggle website in a web browser, log in, and go to the My Account page. In your account settings, you'll find an API section. Clicking the Create New API Token button will generate a kaggle.json key file and will download it to your machine.

Second, go to your Colab notebook, and upload the API's key JSON file to your Colab session by running the following code in a notebook cell:

```
from google.colab import files
files.upload()
```

When you run this cell, you will see a Choose Files button appear. Click it and select the kaggle.json file you just downloaded. This uploads the file to the local Colab runtime.

Finally, create a ~/.kaggle folder (`mkdir ~/.kaggle`), and copy the key file to it (`cp kaggle.json ~/.kaggle/`). As a security best practice, you should also make sure that the file is only readable by the current user, yourself (`chmod 600`):

```
!mkdir ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
```

You can now download the data we're about to use:

```
!kaggle competitions download -c dogs-vs-cats
```

The first time you try to download the data, you may get a "403 Forbidden" error. That's because you need to accept the terms associated with the dataset before you download it—you'll have to go to www.kaggle.com/c/dogs-vs-cats/rules (while logged into your Kaggle account) and click the I Understand and Accept button. You only need to do this once.

Finally, the training data is a compressed file named train.zip. Make sure you uncompress it (`unzip`) silently (`-qq`):

```
!unzip -qq train.zip
```

The pictures in our dataset are medium-resolution color JPEGs. Figure 8.8 shows some examples.

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,

Figure 8.8 Samples from the Dogs vs. Cats dataset. Sizes weren't modified: the samples come in different sizes, colors, backgrounds, etc.

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:

```
cats_vs_dogs_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 calls to `shutil`.

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("cats_vs_dogs_small")
```

Path to the directory where the original dataset was uncompressed

Directory where we will store our smaller dataset

```
def make_subset(subset_name, start_index, end_index):
    for category in ("cat", "dog"):
        dir = new_base_dir / subset_name / category
        os.makedirs(dir)
        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)
```

Utility function to copy cat (and dog) images from index start_index to index end_index to the subdirectory new_base_dir/{subset_name}/cat (and /dog). The "subset_name" will be either "train", "validation", or "test".

Create the training subset with the first 1,000 images of each category.

Create the validation subset with the next 500 images of each category.

Create the test subset with the next 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 the 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` and `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 `Flatten` layer. Here, because we start from inputs of size 180 pixels × 180 pixels (a somewhat arbitrary choice), we end up with feature maps of size 7 × 7 just before the `Flatten` layer.

> **NOTE** The depth of the feature maps progressively increases in the model (from 32 to 256), 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

```
from tensorflow import keras
from tensorflow.keras import layers
```

The model expects RGB images of size 180 × 180.

```
inputs = keras.Input(shape=(180, 180, 3))
x = layers.Rescaling(1./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=256, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Rescale inputs to the [0, 1] range by dividing them by 255.

Let's look at how the dimensions of the feature maps change with every successive layer:

```
>>> model.summary()
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 180, 180, 3)]     0
_________________________________________________________________
rescaling (Rescaling)        (None, 180, 180, 3)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 178, 178, 32)      896
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 89, 89, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 87, 87, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 43, 43, 64)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 41, 41, 128)       73856
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 20, 20, 128)       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 18, 18, 256)       295168
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 9, 9, 256)         0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 7, 7, 256)         590080
_________________________________________________________________
flatten_2 (Flatten)          (None, 12544)             0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 12545
=================================================================
Total params: 991,041
Trainable params: 991,041
Non-trainable params: 0
_________________________________________________________________
```

For the compilation step, we'll go with the `RMSprop` optimizer, as usual. Because we ended the model with a single sigmoid unit, we'll use binary crossentropy as the loss (as a reminder, check out table 6.1 in chapter 6 for a cheat sheet on which loss function to use in various situations).

Listing 8.8 Configuring the model for training

```
model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              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 × 180).
5. Pack them into batches (we'll use batches of 32 images).

It 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 we'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 our 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 tensorflow.keras.utils import image_dataset_from_directory

train_dataset = image_dataset_from_directory(
    new_base_dir / "train",
    image_size=(180, 180),
    batch_size=32)
validation_dataset = image_dataset_from_directory(
    new_base_dir / "validation",
    image_size=(180, 180),
    batch_size=32)
test_dataset = image_dataset_from_directory(
    new_base_dir / "test",
    image_size=(180, 180),
    batch_size=32)
```

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`.

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, 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:

```
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:

```
>>> for i, element in enumerate(dataset):
>>>     print(element.shape)
>>>     if i >= 2:
>>>         break
(16,)
(16,)
(16,)
```

We can use the `.batch()` method to batch the data:

```
>>> 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, we have access to a range of useful dataset methods, such as

- `.shuffle(buffer_size)`—Shuffles elements within a buffer
- `.prefetch(buffer_size)`—Prefetches a buffer of elements in GPU memory to achieve better device utilization.
- `.map(callable)`—Applies an arbitrary transformation to each element of the dataset (the function `callable`, which expects to take as input a single element yielded by the dataset).

The `.map()` method, in particular, is one that you will use often. Here's an example. We'll use it to reshape the elements in our toy dataset from shape `(16,)` to shape `(4, 4)`:

```
>>> reshaped_dataset = dataset.map(lambda x: tf.reshape(x, (4, 4)))
>>> 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 in this chapter.

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 8.10 Displaying the shapes of the data and labels 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'll use the `validation_data` argument in `fit()` to monitor validation metrics on a separate `Dataset` object.

Note that we'll also use a `ModelCheckpoint` callback to save the model after each epoch. We'll configure it with the path specifying 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 8.11 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=30,
    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 8.12 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, "bo", 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, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.legend()
plt.show()
```

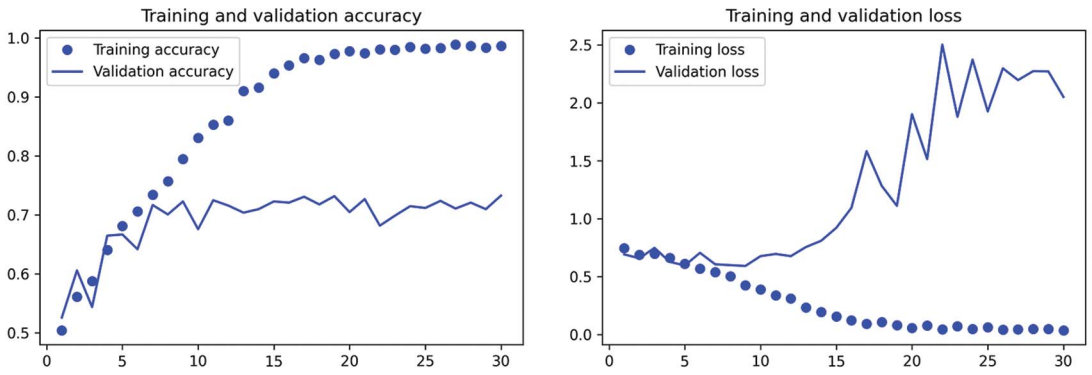


Figure 8.9 Training and validation metrics for a simple convnet

These plots are characteristic of overfitting. The training accuracy increases linearly over time, until it reaches nearly 100%, whereas the validation accuracy peaks at 75%. The validation loss reaches its minimum after only ten 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 8.13 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 69.5%. (Due to the randomness of neural network initializations, you may get numbers within one percentage point of that.)

Because we have relatively few training samples (2,000), overfitting will be our 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 so it can generalize better.

In Keras, this can be done by adding a number of *data augmentation layers* at the start of your model. Let's get started with an example: the following Sequential model chains several random image transformations. In our model, we'd include it right before the `Rescaling` layer.

Listing 8.14 Define a data augmentation stage to add to an image model

```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
    ]
)
```

These are just a few of the layers available (for more, see the Keras documentation). Let's quickly go over this code:

- `RandomFlip("horizontal")`—Applies horizontal flipping to a random 50% of the images that go through it
- `RandomRotation(0.1)`—Rotates 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)`—Zooms 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).

Listing 8.15 Displaying some randomly augmented training images

```
plt.figure(figsize=(10, 10))
for images, _ in train_dataset.take(1):
    for i in range(9):
        augmented_images = data_augmentation(images)
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(augmented_images[0].numpy().astype("uint8"))
        plt.axis("off")
```

We can use take(N) to only sample *N* batches from the dataset. This is equivalent to inserting a break in the loop after the *N*th batch.

Apply the augmentation stage to the batch of images.

Display the first image in the output batch. For each of the nine iterations, this is a different augmentation of the same image.

Figure 8.10 Generating variations of a very good boy via random data augmentation

If we 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—we can't produce new information; we can only remix existing information. As such, this may not be enough to completely get rid of overfitting. To further fight overfitting, we'll also add a `Dropout` layer to our model right before the densely connected classifier.

One last thing you should know about random image augmentation layers: just like `Dropout`, they're inactive during inference (when we call `predict()` or `evaluate()`). During evaluation, our model will behave just the same as when it did not include data augmentation and dropout.

Listing 8.16 Defining a new convnet that includes image augmentation and dropout

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = layers.Rescaling(1./255)(x)
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=256, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              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 three times as many epochs—one hundred.

Listing 8.17 Training the regularized convnet

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch_with_augmentation.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=100,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

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

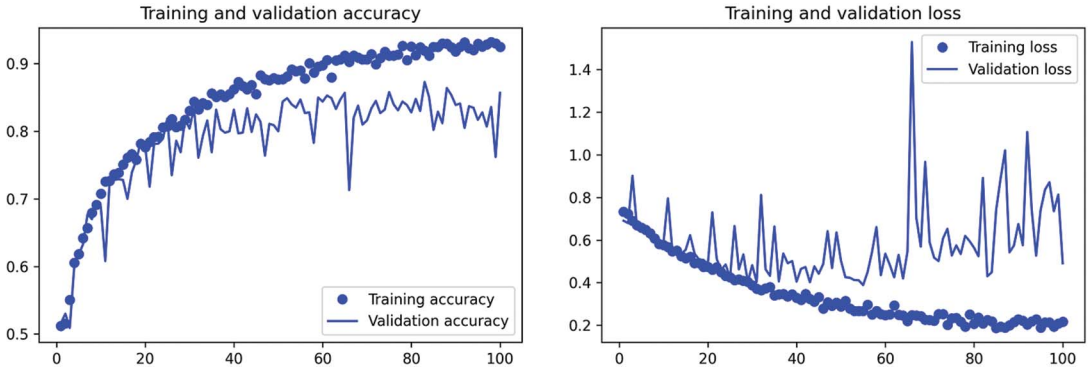


Figure 8.11 Training and validation metrics with data augmentation

the original model). The validation accuracy ends up consistently in the 80–85% range—a big improvement over our first try.

Let's check the test accuracy.

Listing 8.18 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 83.5%. It's starting to look good! If you're using Colab, make sure you 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), we might be able to get an even better accuracy, likely up to 90%. But it would prove difficult to go any higher just by training our own convnet from scratch, because we have so little data to work with. As a next step to improve our accuracy on this problem, we'll have to use a pretrained model, which is the focus of the next two sections.

8.3 Leveraging 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, 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 VGG16 architecture, developed by Karen Simonyan and Andrew Zisserman in 2014.[1] Although it's an older model, far from the current state of the art and somewhat heavier than many other recent models, I chose it because its architecture is similar to what you're already familiar with, and it's easy to understand without introducing any new concepts. This may be your first encounter with one of these cutesy model names—VGG, ResNet, Inception, Xception, and so on; you'll get used to them because they will come up frequently if you keep doing deep learning for computer vision.

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* 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).

Why only reuse the convolutional base? Could we 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 are 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

[1] Karen Simonyan and Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv (2014), https://arxiv.org/abs/1409.1556.

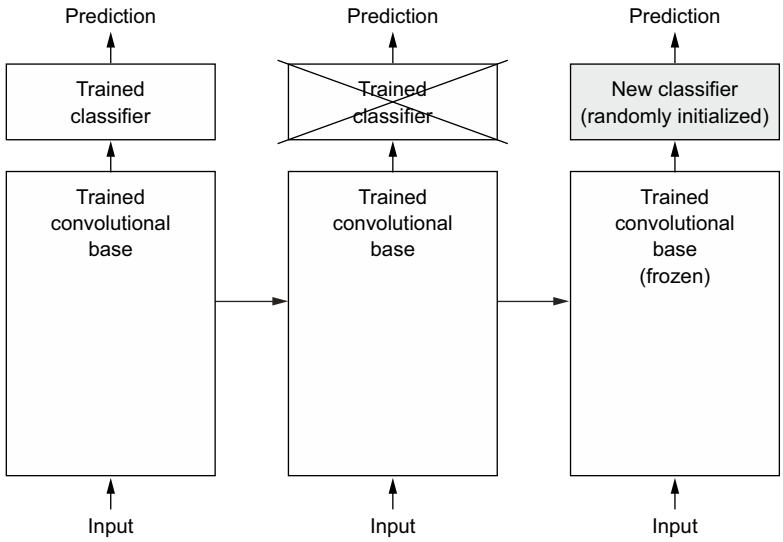


Figure 8.12 Swapping classifiers while keeping the same convolutional base

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, in order to 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 into practice by using the convolutional base of the VGG16 network, trained on ImageNet, to extract interesting features from cat and dog images, and then train a dogs-versus-cats classifier on top of these features.

The VGG16 model, among others, comes prepackaged with Keras. You can import it from the `keras.applications` module. Many other image-classification models (all pretrained on the ImageNet dataset) are available as part of `keras.applications`:

- Xception
- ResNet
- MobileNet
- EfficientNet
- DenseNet
- etc.

Let's instantiate the VGG16 model.

Listing 8.19 Instantiating the VGG16 convolutional base

```
conv_base = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False,
    input_shape=(180, 180, 3))
```

We pass three arguments to the constructor:

- `weights` specifies the weight checkpoint from which to initialize the model.
- `include_top` refers to including (or not) the densely connected classifier on top of the network. By default, this densely connected classifier corresponds to the 1,000 classes from ImageNet. Because we intend to use our own densely connected classifier (with only two classes: `cat` and `dog`), we don't need to include it.
- `input_shape` is the shape of the image tensors that we'll feed to the network. This argument is purely optional: if we don't pass it, the network will be able to process inputs of any size. Here we pass it so that we can visualize (in the following summary) how the size of the feature maps shrinks with each new convolution and pooling layer.

Here's the detail of the architecture of the VGG16 convolutional base. It's similar to the simple convnets you're already familiar with:

```
>>> conv_base.summary()
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_19 (InputLayer)        [(None, 180, 180, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 180, 180, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 180, 180, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 90, 90, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 90, 90, 128)       73856
_________________________________________________________________
```

```
block2_conv2 (Conv2D)        (None, 90, 90, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 45, 45, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 45, 45, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 45, 45, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 45, 45, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 22, 22, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 22, 22, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 22, 22, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 22, 22, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 11, 11, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 11, 11, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 11, 11, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 11, 11, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 5, 5, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

The final feature map has shape `(5, 5, 512)`. That's the feature map on top of which we'll stick a densely connected classifier.

At this point, there are two ways we could proceed:

- Run the convolutional base over our dataset, record its output to a NumPy array on disk, and then use this data as input to a standalone, densely connected classifier similar to those you saw in chapter 4 of this book. 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 us to use data augmentation.
- Extend the model we have (`conv_base`) by adding `Dense` layers on top, and run the whole thing from end to end on the input data. This will allow us 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 our 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 VGG16 features.

Listing 8.20 Extracting the VGG16 features and corresponding labels

```
import numpy as np

def get_features_and_labels(dataset):
    all_features = []
    all_labels = []
    for images, labels in dataset:
        preprocessed_images = keras.applications.vgg16.preprocess_input(images)
        features = conv_base.predict(preprocessed_images)
        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. Moreover, the `VGG16` model expects inputs that are preprocessed with the function `keras.applications.vgg16.preprocess_input`, which scales pixel values to an appropriate range.

The extracted features are currently of shape `(samples, 5, 5, 512)`:

```
>>> train_features.shape
(2000, 5, 5, 512)
```

At this point, we can define our densely connected classifier (note the use of dropout for regularization) and train it on the data and labels that we just recorded.

Listing 8.21 Defining and training the densely connected classifier

```
inputs = keras.Input(shape=(5, 5, 512))
x = layers.Flatten()(inputs)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
```

Note the use of the Flatten layer before passing the features to a Dense layer.

```
model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              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=20,
    validation_data=(val_features, val_labels),
    callbacks=callbacks)
```

Training is very fast because we only have to deal with two `Dense` layers—an epoch takes less than one second even on CPU.

Let's look at the loss and accuracy curves during training (see figure 8.13).

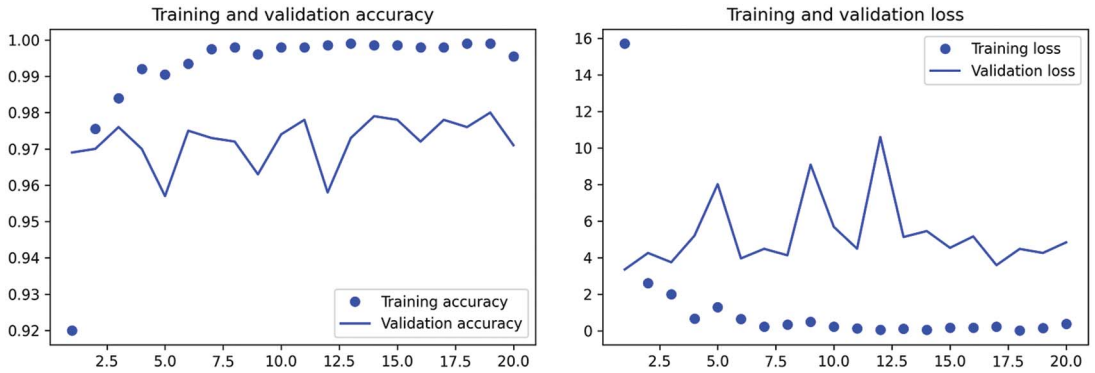


Figure 8.13 Training and validation metrics for plain feature extraction

Listing 8.22 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, "bo", 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, "bo", label="Training loss")
```

```
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.legend()
plt.show()
```

We reach a validation accuracy of about 97%—much better than we 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 we'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.

Feature extraction together with data augmentation

Now let's review the second technique I mentioned for doing feature extraction, which is much slower and more expensive, but which allows us 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.

In order 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. If we don't do this, 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, we freeze a layer or model by setting its `trainable` attribute to `False`.

Listing 8.23 Instantiating and freezing the VGG16 convolutional base

```
conv_base  = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False)
conv_base.trainable = False
```

Setting `trainable` to `False` empties the list of trainable weights of the layer or model.

Listing 8.24 Printing the list of trainable weights before and after freezing

```
>>> conv_base.trainable = True
>>> print("This is the number of trainable weights "
          "before freezing the conv base:", len(conv_base.trainable_weights))
This is the number of trainable weights before freezing the conv base: 26
>>> conv_base.trainable = False
>>> print("This is the number of trainable weights "
          "after freezing the conv base:", len(conv_base.trainable_weights))
This is the number of trainable weights after freezing the conv base: 0
```

Now we can create a new model that chains together

1. A data augmentation stage
2. Our frozen convolutional base
3. A dense classifier

Listing 8.25 Adding a data augmentation stage and a classifier to the convolutional base

```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
    ]
)

inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = keras.applications.vgg16.preprocess_input(x)
x = conv_base(x)
x = layers.Flatten()(x)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              metrics=["accuracy"])
```

Apply data augmentation.

Apply input value scaling.

With this setup, only the weights from the two `Dense` layers that we 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 in order 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. 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 50.

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.

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="feature_extraction_with_data_augmentation.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 results again (see figure 8.14). As you can see, we reach a validation accuracy of over 98%. This is a strong improvement over the previous model.

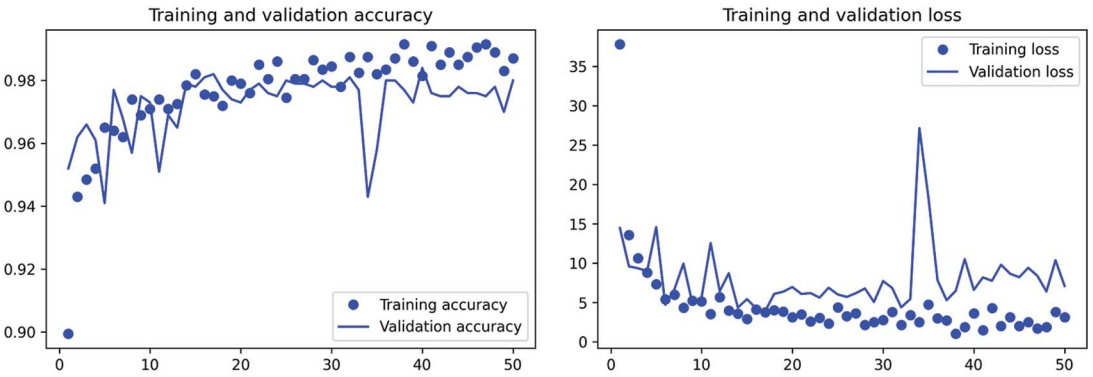


Figure 8.14 Training and validation metrics for feature extraction with data augmentation

Let's check the test accuracy.

Listing 8.26 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}")
```

We get a test accuracy of 97.5%. This is only a modest improvement compared to the previous test accuracy, which is a bit disappointing given the strong results on the validation data. A model's accuracy always depends on the set of samples you evaluate it on! Some sample sets may be more difficult than others, and strong results on one set won't necessarily fully translate to all other sets.

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 a few of the top layers of a 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 these top layers. This is called *fine-tuning* because it slightly adjusts the more abstract representations of the model being reused in order to make them more relevant for the problem at hand.

I stated earlier that it's necessary to freeze the convolution base of VGG16 in order to be able to train a randomly initialized classifier on top. For the same reason, it's only possible to fine-tune the top layers of the convolutional base once the classifier on top has already been trained. If the classifier isn't already trained, 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 our custom network on top of an already-trained base network.
2. Freeze the base network.
3. Train the part we added.
4. Unfreeze some layers in the base network. (Note that you should not unfreeze "batch normalization" layers, which are not relevant here since there are no such layers in VGG16. Batch normalization and its impact on fine-tuning is explained in the next chapter.)
5. Jointly train both these layers and the part we added.

You already completed the first three steps when doing feature extraction. Let's proceed with step 4: we'll unfreeze our `conv_base` and then freeze individual layers inside it.

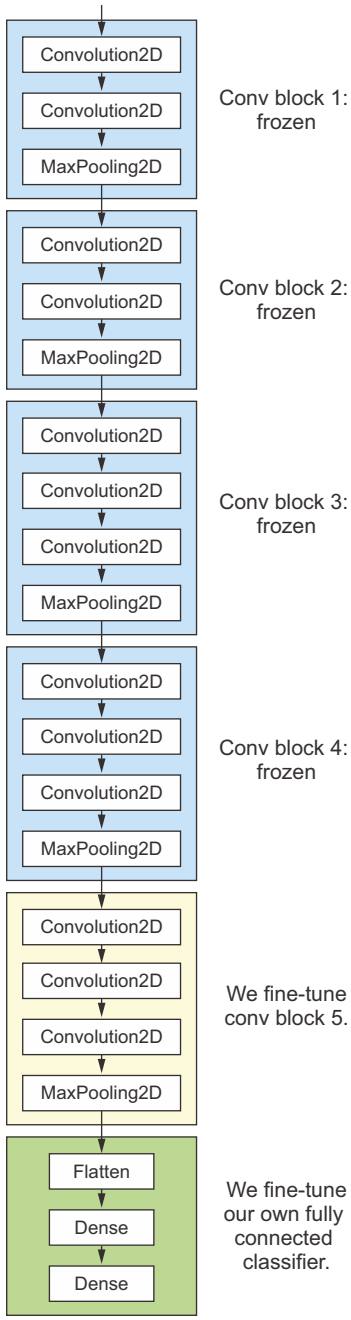


Figure 8.15 Fine-tuning the last convolutional block of the VGG16 network

As a reminder, this is what our convolutional base looks like:

```
>>> conv_base.summary()
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_19 (InputLayer)        [(None, 180, 180, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 180, 180, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 180, 180, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 90, 90, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 90, 90, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 90, 90, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 45, 45, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 45, 45, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 45, 45, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 45, 45, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 22, 22, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 22, 22, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 22, 22, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 22, 22, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 11, 11, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 11, 11, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 11, 11, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 11, 11, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 5, 5, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

We'll fine-tune the last three convolutional layers, which means all layers up to `block4_pool` should be frozen, and the layers `block5_conv1`, `block5_conv2`, and `block5_conv3` should be trainable.

Why not fine-tune more layers? Why not fine-tune the entire convolutional base? You could. But you need to consider the following:

- 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, in this situation, it's a good strategy to fine-tune only the top two or three layers in the convolutional base. Let's set this up, starting from where we left off in the previous example.

Listing 8.27 Freezing all layers until the fourth from the last

```
conv_base.trainable = True
for layer in conv_base.layers[:-4]:
    layer.trainable = False
```

Now we can begin fine-tuning the model. We'll do this with the RMSprop optimizer, using a very low learning rate. The reason for using a low learning rate is that we want to limit the magnitude of the modifications we make to the representations of the three layers we're fine-tuning. Updates that are too large may harm these representations.

Listing 8.28 Fine-tuning the model

```
model.compile(loss="binary_crossentropy",
              optimizer=keras.optimizers.RMSprop(learning_rate=1e-5),
              metrics=["accuracy"])

callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="fine_tuning.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

We can finally evaluate this model on the test data:

```
model = keras.models.load_model("fine_tuning.keras")
test_loss, test_acc = model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

Here, we get a test accuracy of 98.5% (again, your own results may be within one 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 we 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 leveraging modern deep learning techniques, we 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 are the best type of machine learning models for 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 Advanced deep learning for computer vision

This chapter covers

- The different branches of computer vision: image classification, image segmentation, object detection
- Modern convnet architecture patterns: residual connections, batch normalization, depthwise separable convolutions
- Techniques for visualizing and interpreting what convnets learn

The previous chapter gave you a first introduction to deep learning for computer vision via simple models (stacks of `Conv2D` and `MaxPooling2D` layers) and a simple use case (binary image classification). But there's more to computer vision than image classification! This chapter dives deeper into more diverse applications and advanced best practices.

9.1 Three essential 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 (an image can only be in one category, excluding the others), or multi-label classification (tagging all categories that an image belongs to, as seen in figure 9.1). 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 seen in figure 9.1). For instance, when Zoom or Google Meet diplays a custom background behind you in a video call, it's using an image segmentation model to tell your face apart from what's behind it, at pixel 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.

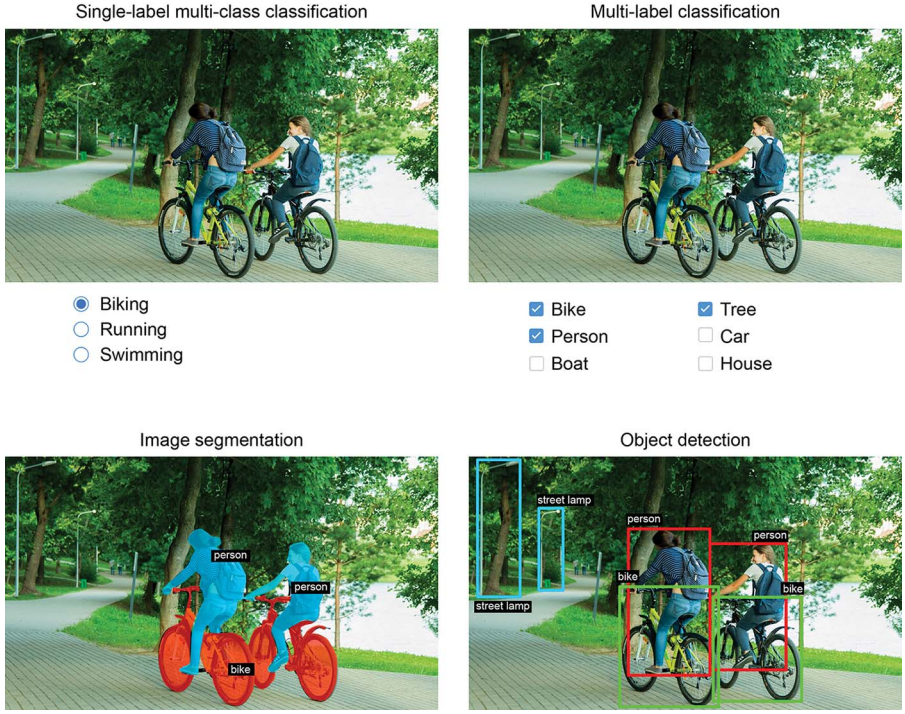


Figure 9.1 The three main computer vision tasks: classification, segmentation, detection

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, 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. Most computer vision applications boil down to one of these three.

You've seen image classification in action in the previous chapter. Next, let's dive into image segmentation. It's a very useful and versatile technique, and you can straightforwardly approach it with what you've already learned so far.

Note that we won't cover object detection, because it would be too specialized and too complicated for an introductory book. However, you can check out the RetinaNet example on keras.io, which shows how to build and train an object detection model from scratch in Keras in around 450 lines of code (https://keras.io/examples/vision/retinanet/).

9.2 An image segmentation example

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 two 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 9.2).
- *Instance segmentation*, which seeks not only to classify image pixels by category, but also to parse out individual object instances. In an image with two cats in it, instance segmentation would treat "cat 1" and "cat 2" as two separate classes of pixels (see figure 9.2).

In this example, we'll focus on semantic segmentation: we'll be looking once again at images of cats and dogs, and this time we'll learn how to tell apart the main subject and its background.

We'll work with the Oxford-IIIT Pets dataset (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

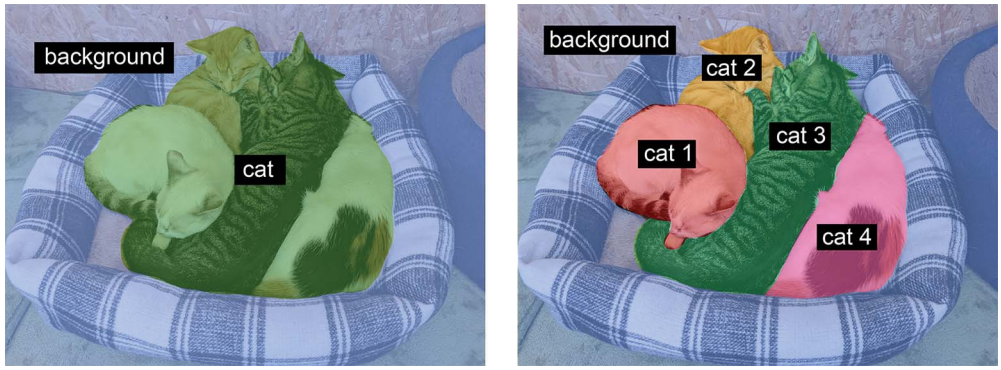


Figure 9.2 Semantic segmentation vs. instance segmentation

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 os

input_dir = "images/"
target_dir = "annotations/trimaps/"

input_img_paths = sorted(
    [os.path.join(input_dir, fname)
     for fname in os.listdir(input_dir)
     if fname.endswith(".jpg")])
```

```
target_paths = sorted(
    [os.path.join(target_dir, fname)
     for fname in os.listdir(target_dir)
     if fname.endswith(".png") and not fname.startswith(".")])
```

Now, what does one of these inputs and its mask look like? Let's take a quick look. Here's a sample image (see figure 9.3):

```
import matplotlib.pyplot as plt
from tensorflow.keras.utils import load_img, img_to_array

plt.axis("off")
plt.imshow(load_img(input_img_paths[9]))
```

Display input image number 9.

Figure 9.3 An example image

And here is its corresponding target (see figure 9.4):

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 9.4 The corresponding target mask

Next, let's load our inputs and targets into two NumPy arrays, and let's split the arrays into a training and a validation set. Since the dataset is very small, we can just 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

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])

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:]
```

We resize everything to 200 × 200.

Total number of samples in the data

Shuffle 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.

Subtract 1 so that our labels become 0, 1, and 2.

Load all images in the input_imgs float32 array and their masks in the targets uint8 array (same order). The inputs have three channels (RBG values) and the targets have a single channel (which contains integer labels).

Reserve 1,000 samples for validation.

Split the data into a training and a validation set.

Now it's time to define our model:

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

Here's the output of the `model.summary()` call:

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 200, 200, 3)]     0
_________________________________________________________________
rescaling (Rescaling)        (None, 200, 200, 3)       0
_________________________________________________________________
conv2d (Conv2D)              (None, 100, 100, 64)      1792
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 100, 100, 64)      36928
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 50, 50, 128)       73856
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 50, 50, 128)       147584
_________________________________________________________________
```

```
conv2d_4 (Conv2D)            (None, 25, 25, 256)       295168
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 25, 25, 256)       590080
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 25, 25, 256)       590080
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 50, 50, 256)       590080
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 50, 50, 128)       295040
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 100, 100, 128)     147584
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 100, 100, 64)      73792
_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 200, 200, 64)      36928
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 200, 200, 3)       1731
=================================================================
Total params: 2,880,643
Trainable params: 2,880,643
Non-trainable params: 0
_________________________________________________________________
```

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 the last chapter, 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 "Understanding convolution strides" in 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 12.

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 have the same shape as the target masks, `(200, 200, 3)`. 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 to get back to images of shape `(200, 200, 3)`.

We can now compile and fit our model:

```
model.compile(optimizer="rmsprop", loss="sparse_categorical_crossentropy")

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 9.5):

```
epochs = range(1, len(history.history["loss"]) + 1)
loss = history.history["loss"]
```

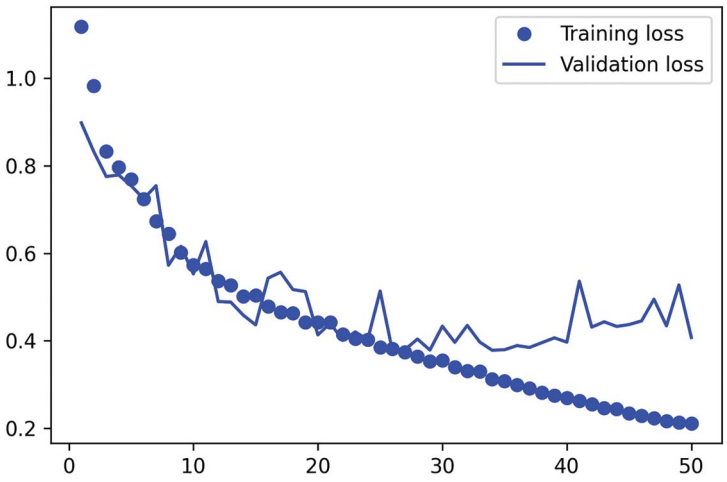


Figure 9.5 Displaying training and validation loss curves

```
val_loss = history.history["val_loss"]
plt.figure()
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.legend()
```

You can see that we start overfitting midway, around epoch 25. Let's reload our best performing model according to the validation loss, and demonstrate how to use it to predict a segmentation mask (see figure 9.6):

```
from tensorflow.keras.utils import array_to_img

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 9.6 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.

By this point, throughout chapter 8 and the beginning of chapter 9, you've learned the basics of how to perform image classification and image segmentation: you can already accomplish a lot with what you know. However, the convnets that experienced engineers develop to solve real-world problems aren't quite as simple as those we've been using in our demonstrations so far. You're still lacking the essential mental models and thought processes that enable experts to make quick and accurate decisions about how to put together state-of-the-art models. To bridge that gap, you need to learn about *architecture patterns*. Let's dive in.

9.3 Modern convnet architecture patterns

A model's "architecture" is the sum of the choices that went into creating it: which layers to use, how to configure them, and 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. In order 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 as 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 convolutions*. Once you master how to use them, you will be able to build highly effective image models. We will apply them to our cat vs. dog classification problem.

Let's start from the bird's-eye view: the modularity-hierarchy-reuse (MHR) formula for system architecture.

9.3.1 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* in this context). That's the MHR formula (modularity-hierarchy-reuse), 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 (see figure 9.7).

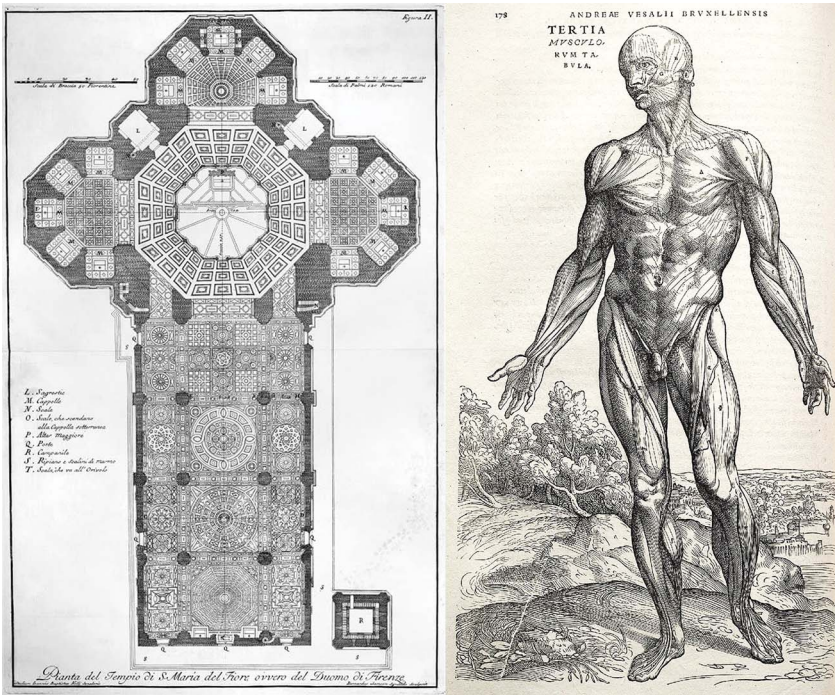


Figure 9.7 Complex systems follow a hierarchical structure and are organized into distinct modules, which are reused multiple times (such as your four limbs, which are all variants of the same blueprint, or your 20 "fingers").

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 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, the popular VGG16 architecture we used in the previous chapter is structured into repeated "conv, conv, max pooling" blocks (see figure 9.8).

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 VGG16 model (see figure 9.8).

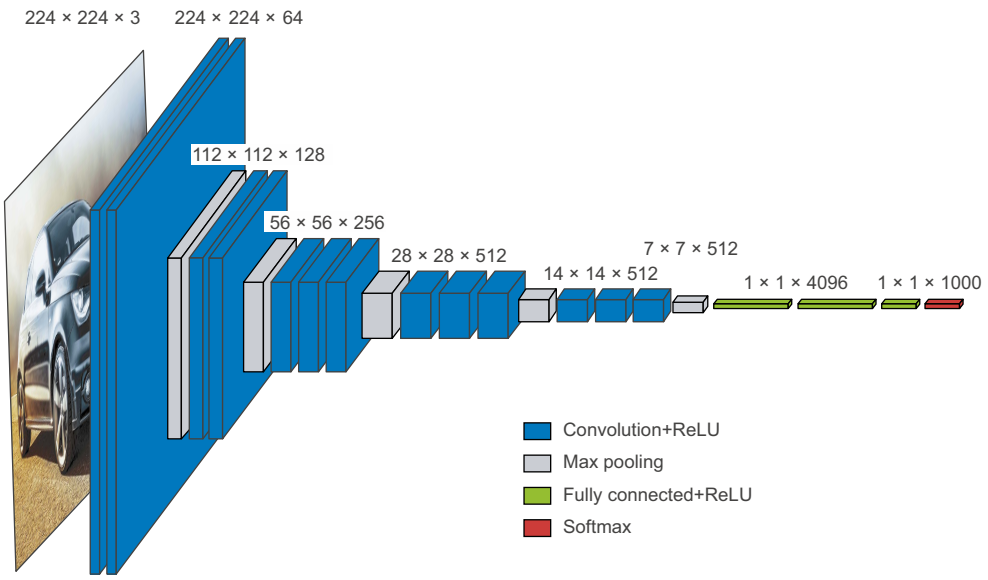


Figure 9.8 The VGG16 architecture: note the repeated layer blocks and the pyramid-like structure of the feature maps

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, due to 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, 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 non-obvious results—rather, it gets leveraged as a *signal of seriousness*, like an expensive suit on a salesman.

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—making 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, and 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?"

9.3.2 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. 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 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 to use a *residual connection*. It's dead easy: just add the input of a layer or block of layers back to its output (see figure 9.9). 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]

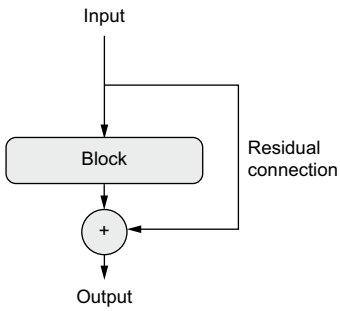


Figure 9.9 A residual connection around a processing block

In practice, you'd implement a residual connection as follows.

Listing 9.1 A residual connection in pseudocode

```
x = ...
residual = x
x = block(x)
x = add([x, residual])
```

Some input tensor

Save a pointer to the original input. This is called the residual.

This computation block can potentially be destructive or noisy, and that's fine.

Add the original input to the layer's output: the final output will thus always preserve full information about the original input.

[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.

Note that adding the input back to the output of a block implies that the output should have the same shape as the input. However, 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 (see listing 9.2). You'd typically use `padding="same"` in the convolution layers in your target block so as 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 (see listing 9.3).

Listing 9.2 Residual block where the number of filters changes

```
from tensorflow import keras
from tensorflow.keras import layers

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)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

Set 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. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.

Listing 9.3 Case where the target block includes a max pooling layer

```
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])
```

Set aside the residual.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

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:

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./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)
```

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

```
        if pooling:
            x = layers.MaxPooling2D(2, padding="same")(x)
            residual = layers.Conv2D(filters, 1, strides=2)(residual)
        elif filters != residual.shape[-1]:
            residual = layers.Conv2D(filters, 1)(residual)
        x = layers.add([x, residual])
        return x

inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)
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)
model.summary()
```

If we use max pooling, we add a strided convolution to project the residual to the expected shape.

First block

If we don't use max pooling, we only project the residual if the number of channels has changed.

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.

This is the model summary we get:

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 32, 32, 3)]       0
_________________________________________________________________
rescaling (Rescaling)        (None, 32, 32, 3)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 32)        128
_________________________________________________________________
add (Add)                    (None, 16, 16, 32)        0
                                                       max_pooling2d[0][0]
                                                       conv2d_2[0][0]
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2D) (None, 8, 8, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 64)          2112
_________________________________________________________________
add_1 (Add)                  (None, 8, 8, 64)          0
                                                       max_pooling2d_1[0][0]
                                                       conv2d_5[0][0]
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 8, 8, 128)         147584
_________________________________________________________________
```

```
conv2d_8 (Conv2D)                (None, 8, 8, 128)     8320        add_1[0][0]
__________________________________________________________________________________________________
add_2 (Add)                      (None, 8, 8, 128)     0           conv2d_7[0][0]
                                                                   conv2d_8[0][0]
__________________________________________________________________________________________________
global_average_pooling2d (Globa (None, 128)           0           add_2[0][0]
__________________________________________________________________________________________________
dense (Dense)                    (None, 1)             129         global_average_pooling2d[0][0]
==================================================================================================
Total params: 297,697
Trainable params: 297,697
Non-trainable params: 0
__________________________________________________________________________________________________
```

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*.

9.3.3 Batch normalization

Normalization 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 already seen several times in this book: 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 in this book normalized data before feeding it into models. But data normalization may be of interest after every transformation operated 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 batch-wise mean and variance of the data seen during training.

[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.

Although the original paper stated 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, I 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`, etc.:

```
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 would 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 what is shown in listing 9.4, you would do what's shown in listing 9.5.

Listing 9.4 How not to use batch normalization

```
x = layers.Conv2D(32, 3, activation="relu")(x)
x = layers.BatchNormalization()(x)
```

Listing 9.5 How to use batch normalization: the activation comes last

```
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 for this approach 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, then activation, and then batch normalization, your model will still train, and you won't necessarily see worse results.

> **On batch normalization and fine-tuning**
>
> 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.

9.3.4 Depthwise separable convolutions

What if I 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) and 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 9.10.

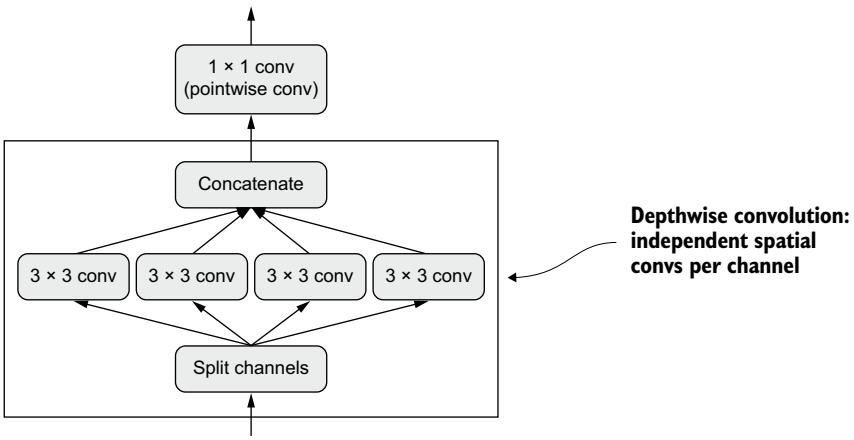


Figure 9.10 Depthwise separable convolution: a depthwise convolution followed by a pointwise convolution

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 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. It results 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 × 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.

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.

[3] François Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," Conference on Computer Vision and Pattern Recognition (2017), https://arxiv.org/abs/1610.02357.

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 multi-year re-engineering of the entire ecosystem.

9.3.5 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 these ideas together into a single model. Its architecture will resemble a smaller version of Xception, and we'll apply it to the dogs vs. cats task from the last chapter. For data loading and model training, we'll simply reuse the setup we used in section 8.2.5, but we'll replace the model definition with the following convnet:

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)

x = layers.Rescaling(1./255)(x)
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)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

We use the same data augmentation configuration as before.

Don't forget input rescaling!

In the original model, we used a Flatten layer before the Dense layer. Here, we go with a GlobalAveragePooling2D layer.

Like in the original model, we add a dropout layer for regularization.

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.

Note that 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 afterwards.

This convnet has a trainable parameter count of 721,857, slightly lower than the 991,041 trainable parameters of the original model, but still in the same ballpark. Figure 9.11 shows its training and validation curves.

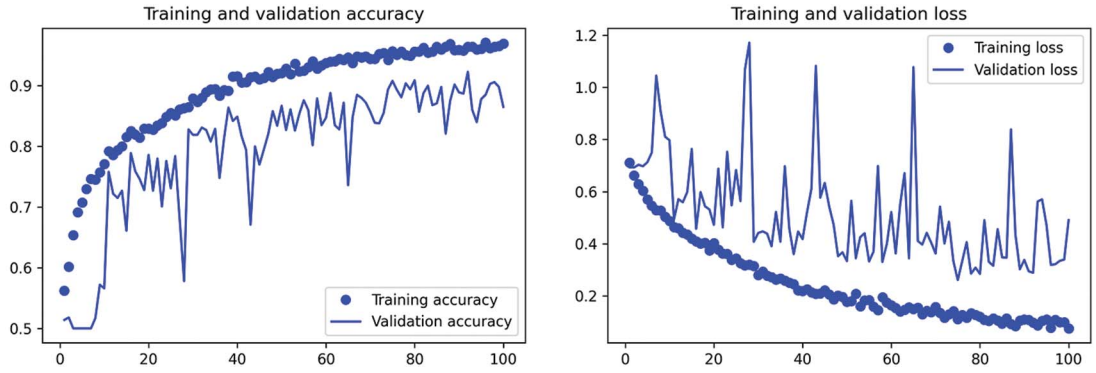


Figure 9.11 Training and validation metrics with an Xception-like architecture

You'll find that our new model achieves a test accuracy of 90.8%, compared to 83.5% for the naive model in the last chapter. As you can see, following architecture best practices does have an immediate, sizable impact 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'll cover in detail in chapter 13. We haven't gone through this step here, so the configuration of the preceding model is purely based on the best practices we discussed, plus, when it comes to gauging model size, a small amount of intuition.

Note that these architecture best practices are relevant to computer vision in general, not just image classification. For example, Xception is used as the standard convolutional base in DeepLabV3, a popular state-of-the-art image segmentation solution.[4]

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.

9.4 Interpreting what convnets learn

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 in medical imaging use cases. We will end this chapter by getting 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 convnet filters*—Useful for understanding precisely what visual pattern or concept each filter in a convnet is receptive to

[4] Liang-Chieh Chen et al., "Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation," ECCV (2018), https://arxiv.org/abs/1802.02611.

- *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

For the first method—activation visualization—we'll use the small convnet that we trained from scratch on the dogs-versus-cats classification problem in section 8.2. For the next two methods, we'll use a pretrained Xception model.

9.4.1 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. We 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:

```
>>> from tensorflow import keras
>>> model = keras.models.load_model(
    "convnet_from_scratch_with_augmentation.keras")
>>> model.summary()
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 180, 180, 3)]     0
_________________________________________________________________
sequential (Sequential)      (None, 180, 180, 3)       0
_________________________________________________________________
rescaling_1 (Rescaling)      (None, 180, 180, 3)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 178, 178, 32)      896
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 89, 89, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 87, 87, 64)        18496
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 43, 43, 64)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 41, 41, 128)       73856
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 20, 20, 128)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 18, 18, 256)       295168
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 9, 9, 256)         0
_________________________________________________________________
```

```
conv2d_9 (Conv2D)            (None, 7, 7, 256)         590080
_________________________________________________________________
flatten_1 (Flatten)          (None, 12544)             0
_________________________________________________________________
dropout (Dropout)            (None, 12544)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 12545
=================================================================
Total params: 991,041
Trainable params: 991,041
Non-trainable params: 0
_________________________________________________________________
```

Next, we'll get an input image—a picture of a cat, not part of the images the network was trained on.

Listing 9.6 Preprocessing a single image

```
from tensorflow 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))
```

Download a test image.

Open the image file and resize it.

Turn the image into a float32 NumPy array of shape (180, 180, 3).

Add a dimension to transform the array into a "batch" of a single sample. Its shape is now (1, 180, 180, 3).

Let's display the picture (see figure 9.12).

Listing 9.7 Displaying the test picture

```
import matplotlib.pyplot as plt
plt.axis("off")
plt.imshow(img_tensor[0].astype("uint8"))
plt.show()
```

In order to extract the feature maps we want to look at, we'll create a Keras model that takes batches of images as input, and that outputs the activations of all convolution and pooling layers.

Figure 9.12 The test cat picture

Listing 9.8 Instantiating a model that returns layer activations

```
from tensorflow.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)
```

Extract the outputs of all Conv2D and MaxPooling2D layers and put them in a list.

Save the layer names for later.

Create a model that will return these outputs, given the model input.

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 9.9 Using the model to compute layer activations

```
activations = activation_model.predict(img_tensor)
```

Return 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 fifth channel of the activation of the first layer of the original model (see figure 9.13).

Listing 9.10 Visualizing the fifth channel

```
import matplotlib.pyplot as plt
plt.matshow(first_layer_activation[0, :, :, 5], cmap="viridis")
```

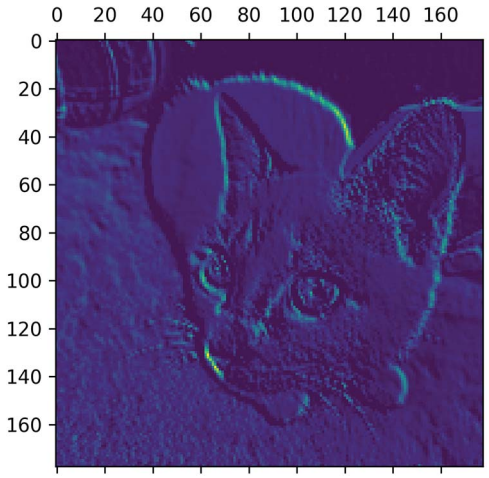


Figure 9.13 Fifth channel of the activation of the first layer on the test cat picture

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 9.14). 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 9.11 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()
```

Iterate over the activations (and the names of the corresponding layers).

The layer activation has shape (l, size, size, n_features).

Prepare an empty grid for displaying all the channels in this activation.

This is a single channel (or feature).

Normalize channel values within the [0, 255] range. All-zero channels are kept at zero.

```
            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. / 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")
```

Place the channel matrix in the empty grid we prepared.

Display the grid for the layer.

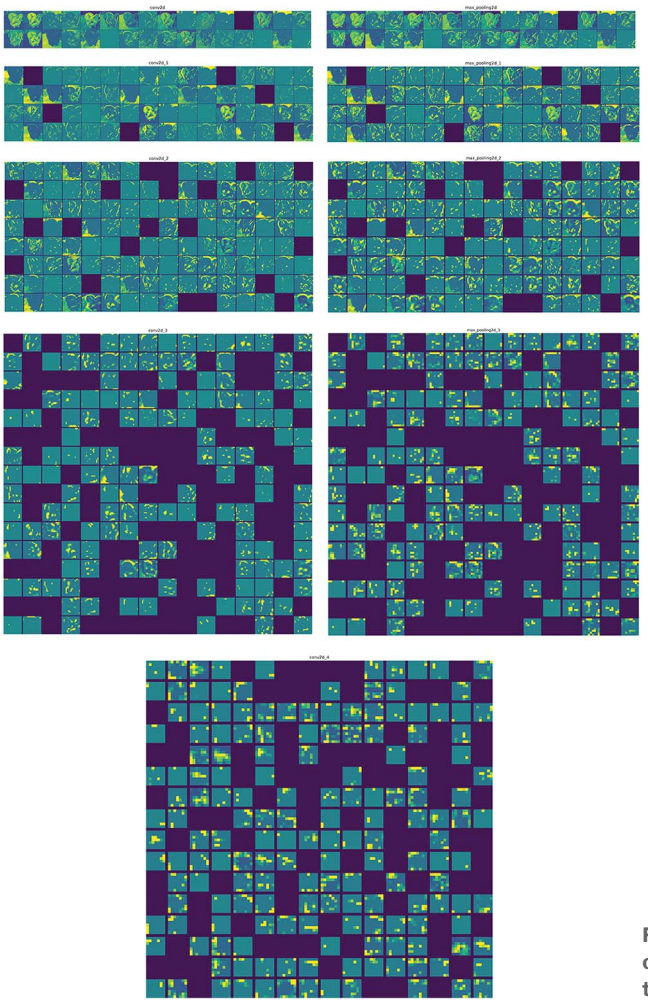


Figure 9.14 Every channel of every layer activation on the test cat picture

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 deeper, the activations become increasingly abstract and less visually interpretable. They begin to encode higher-level concepts such as "cat ear" and "cat eye." Deeper presentations 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, almost 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 evidenced 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 9.15). Try it right now: this effect is absolutely real. Your brain has learned to completely abstract its visual input—to transform it

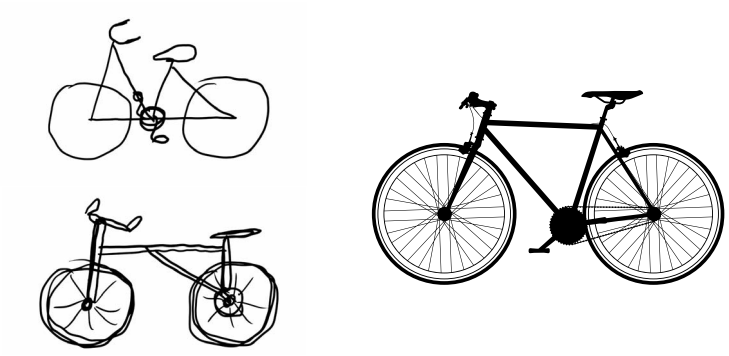


Figure 9.15 Left: attempts to draw a bicycle from memory. Right: what a schematic bicycle should look like.

into high-level visual concepts while filtering out irrelevant visual details—making it tremendously difficult to remember how things around you look.

9.4.2 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, pretrained on ImageNet. 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 our second example of a low-level gradient descent loop leveraging the `GradientTape` object (the first one was in chapter 2).

First, let's instantiate the Xception model, loaded with weights pretrained on the ImageNet dataset.

Listing 9.12 Instantiating the Xception convolutional base

```
model = keras.applications.xception.Xception(
    weights="imagenet",
    include_top=False)
```

The classification layers are irrelevant for this use case, so we don't include the top stage of the model.

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 9.13 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: we 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 9.14 Creating a feature extractor model

```
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, simply call it on some input data (note that Xception requires inputs to be preprocessed via the `keras.applications.xception.preprocess_input` function).

Listing 9.15 Using the feature extractor

```
activation = feature_extractor(
    keras.applications.xception.preprocess_input(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:

```
import tensorflow as tf

def compute_loss(image, filter_index):
    activation = feature_extractor(image)
    filter_activation = activation[:, 2:-2, 2:-2, filter_index]
    return tf.reduce_mean(filter_activation)
```

The loss function takes an image tensor and the index of the filter we are considering (an integer).

Return the mean of the activation values for the filter.

Note that we avoid border artifacts by only involving non-border pixels in the loss; we discard the first two pixels along the sides of the activation.

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 it extracts the NumPy value of the outputs. It's schematically equivalent to this:

```
def predict(x):
    y_batches = []
    for x_batch in get_batches(x):
```

(continued)

```
        y_batch = model(x).numpy()
        y_batches.append(y_batch)
    return np.concatenate(y_batches)
```

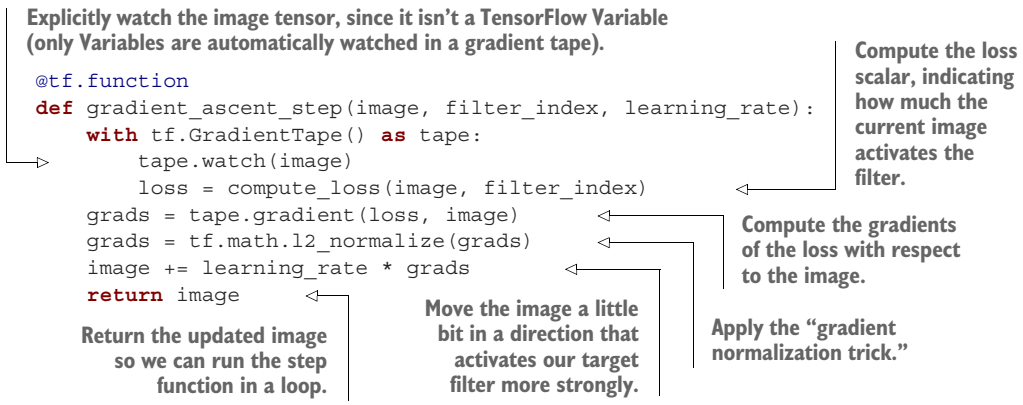
This means that `predict()` calls can scale to very large arrays. Meanwhile, `model(x)` happens in-memory and doesn't scale. On the other hand, `predict()` is not differentiable: you cannot retrieve its gradient if you call it in a `GradientTape` scope.

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).

Let's set up the gradient ascent step function, using the `GradientTape`. Note that we'll use a `@tf.function` decorator to speed it up.

A non-obvious trick to help the gradient descent process go smoothly is to normalize the gradient tensor by dividing it by its L2 norm (the square root of the average of the square 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.

Listing 9.16 Loss maximization via stochastic gradient ascent



Now we have all the pieces. Let's put them together into a Python function that takes as input a layer name and a filter index, and returns a tensor representing the pattern that maximizes the activation of the specified filter.

Listing 9.17 Function to generate filter visualizations

```
img_width = 200
img_height = 200
```

```
def generate_filter_pattern(filter_index):
    iterations = 30
    learning_rate = 10.
    image = tf.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].numpy()
```

Amplitude of a single step

Number of gradient ascent steps to apply

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).

Repeatedly update the values of the image tensor so as 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, we need to post-process this tensor to turn it into a displayable image. We do so with the following straightforward utility function.

Listing 9.18 Utility function to convert a tensor into a valid image

```
def deprocess_image(image):
    image -= image.mean()
    image /= image.std()
    image *= 64
    image += 128
    image = np.clip(image, 0, 255).astype("uint8")
    image = image[25:-25, 25:-25, :]
    return image
```

Normalize image values within the [0, 255] range.

Center crop to avoid border artifacts.

Let's try it (see figure 9.16):

```
>>> plt.axis("off")
>>> plt.imshow(deprocess_image(generate_filter_pattern(filter_index=2)))
```

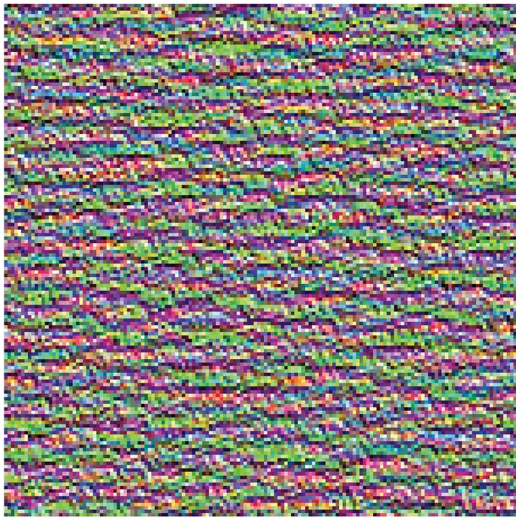


Figure 9.16 Pattern that the second channel in layer `block3_sepconv1` responds to maximally

It seems that filter 0 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.

Listing 9.19 Generating a grid of all filter response patterns in a layer

```
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
cropped_width = img_width - 25 * 2
cropped_height = img_height - 25 * 2
width = n * cropped_width + (n - 1) * margin
height = n * cropped_height + (n - 1) * margin
stitched_filters = np.zeros((width, height, 3))

for i in range(n):
    for j in range(n):
        image = all_images[i * n + j]
        stitched_filters[
            row_start = (cropped_width + margin) * i
            row_end = (cropped_width + margin) * i + cropped_width
            column_start = (cropped_height + margin) * j
            column_end = (cropped_height + margin) * j + cropped_height

            stitched_filters[
                row_start: row_end,
                column_start: column_end, :] = image

keras.utils.save_img(
    f"filters_for_layer_{layer_name}.png", stitched_filters)
```

Generate and save visualizations for the first 64 filters in the layer.

Prepare a blank canvas for us to paste filter visualizations on.

Fill the picture with the saved filters.

Save the canvas to disk.

These filter visualizations (see figure 9.17) 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 deeper 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.

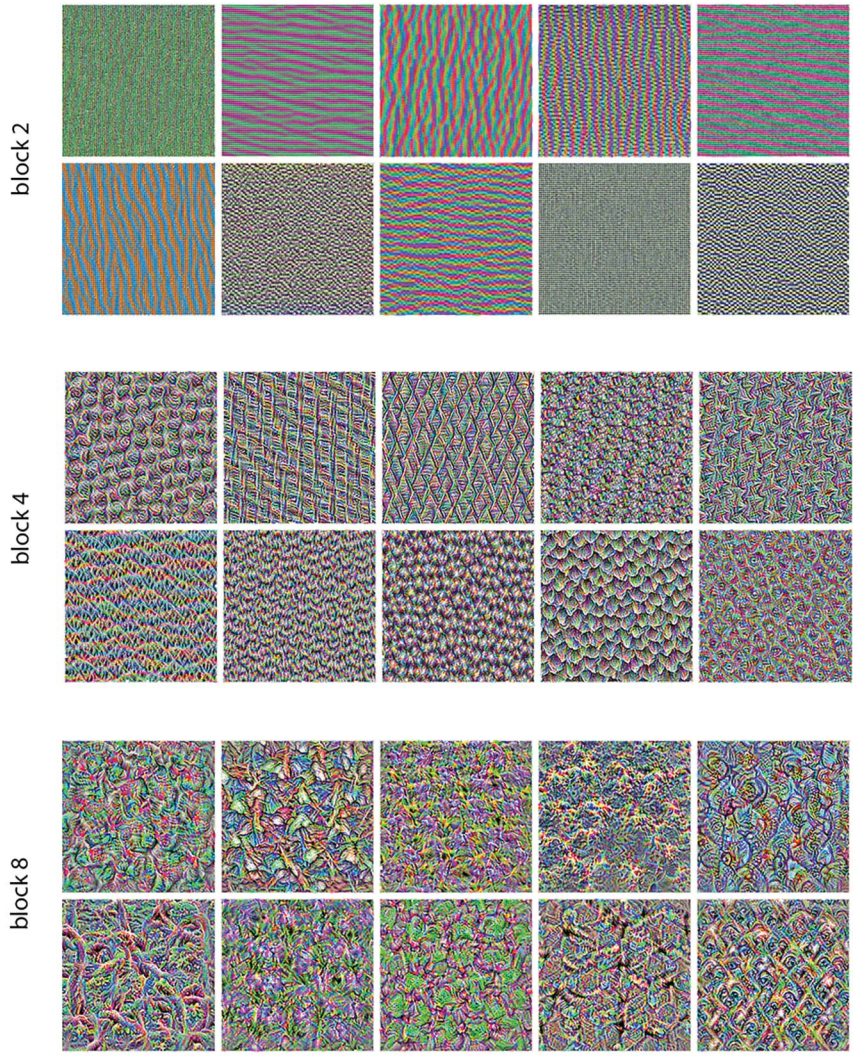


Figure 9.17 Some filter patterns for layers `block2_sepconv1`, `block4_sepconv1`, and `block8_sepconv1`

9.4.3 Visualizing heatmaps of class activation

We'll introduce 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 an article titled "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization."[5]

Grad-CAM consists of taking the output feature map of a convolution layer, given an input image, and weighing 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 to imagine 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.

Listing 9.20 Loading the Xception network with pretrained weights

```
model = keras.applications.xception.Xception(weights="imagenet")
```

Note that we include the densely connected classifier on top; in all previous cases, we discarded it.

Consider the image of two African elephants shown in figure 9.18, possibly a mother and her calf, strolling on the savanna. Let's convert this image into something the Xception model can read: the model was trained on images of size 299 × 299, preprocessed according to a few rules that are packaged in the `keras.applications.xception.preprocess_input` utility function. So we need to load the image, resize it to 299 × 299, convert it to a NumPy `float32` tensor, and apply these preprocessing rules.

Listing 9.21 Preprocessing an input image for Xception

```
img_path = keras.utils.get_file(
    fname="elephant.jpg",
    origin="https://img-datasets.s3.amazonaws.com/elephant.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)
    array = keras.applications.xception.preprocess_input(array)
    return array

img_array = get_img_array(img_path, target_size=(299, 299))
```

Download the image and store it locally under the path img_path.

Return a Python Imaging Library (PIL) image of size 299 × 299.

Return a float32 NumPy array of shape (299, 299, 3).

Add a dimension to transform the array into a batch of size (1, 299, 299, 3).

Preprocess the batch (this does channel-wise color normalization).

[5] Ramprasaath R. Selvaraju et al., arXiv (2017), https://arxiv.org/abs/1610.02391.