Chapter 1. What is deep learning?

This chapter covers

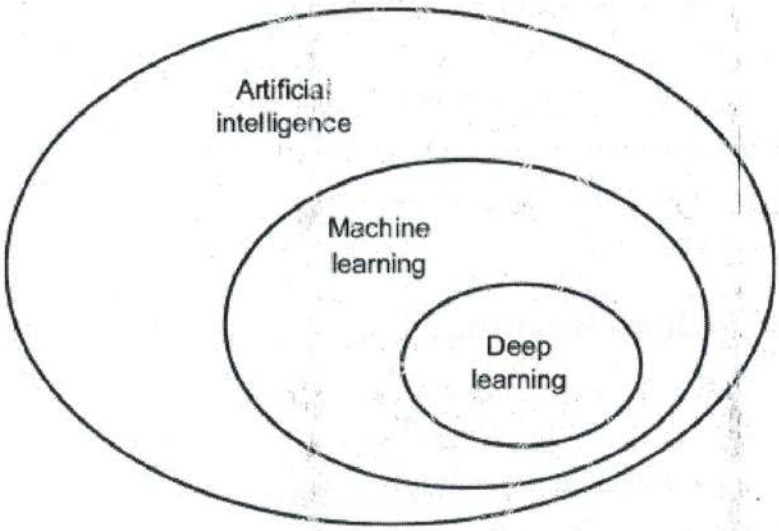
- High-level definitions of fundamental concepts
- Timeline of the development of machine learning
- Key factors behind deep learning's rising popularity and future potential

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 in 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 the AI agents. 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. and the differences between these terms.

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. A concise definition of the field would be as follows: *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 don't involve any learning. Early chess programs, for instance, only involved hardcoded rules crafted by programmers, and didn't qualify as machine learning. For a fairly long time, many experts believed that human-level artificial intelligence could be achieved by having programmers handcraft a sufficiently large set of explicit rules for manipulating knowledge. This approach is known as *symbolic AI*, and it was the dominant paradigm in AI from the 1950s to the late 1980s. 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, and 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. In 1843, Ada Lovelace remarked on the invention, "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."

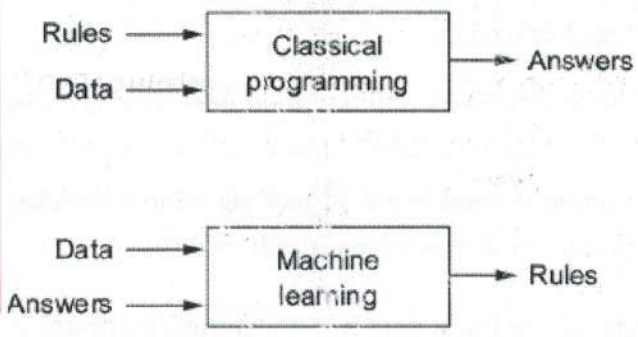
This remark was later quoted by AI pioneer Alan Turing as "Lady Lovelace's objection" in his landmark 1950 paper "Computing Machinery and Intelligence,"[1] which introduced the *Turing test* as well as key concepts that would come to shape AI. Turing was quoting Ada Lovelace while pondering whether general-purpose computers could be capable of learning and originality, and he came to the conclusion that they could.

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

Machine learning arises from this question: could a computer go beyond "whatever we know how to order it to perform" and learn on its own how to perform a specified task? Could a computer surprise us? Rather than programmers crafting data-processing rules by hand, could a computer automatically learn these rules by looking at data?

This question opens the door to a new programming paradigm. In classical programming, the paradigm of symbolic AI, humans input rules (a program) and data to be processed according to these rules, and out come answers (see figure 1.2). With machine learning, humans input data as well as the answers expected from the data, and out come the rules. These rules can then be applied to new data to produce original answers.

Figure 1.2. Machine learning: a new programming paradigm



A machine-learning system is *trained* rather than explicitly programmed. It's presented with many examples relevant to a task, and it finds statistical structure in these examples that eventually allows the system to come up with rules for automating the

task. For instance, if you wished to automate the task of tagging your vacation pictures, you could present a machine-learning system with many examples of pictures already tagged by humans, and the system would learn statistical rules for associating specific pictures to specific tags.

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 tightly related to mathematical statistics, but it differs from statistics in several important ways. 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 engineering oriented. It's a hands-on discipline in which ideas are proven empirically more often than theoretically.

1.1.3. Learning representations from data

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

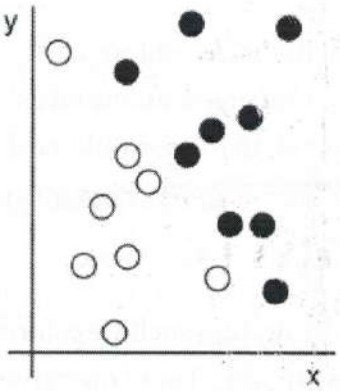
- *Input data points*—For instance, if the task is speech recognition, these data points could be sound files of people speaking. If the task is image tagging, they could be pictures.
- *Examples of the expected output*—In a speech-recognition task, these could be human-generated transcripts of sound files. In an image task, expected outputs could be tags such as "dog," "cat," and so on.
- *A way to measure whether the algorithm is doing a good job*—This is necessary 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, such as a classification task. ch 8 of ML textbook

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

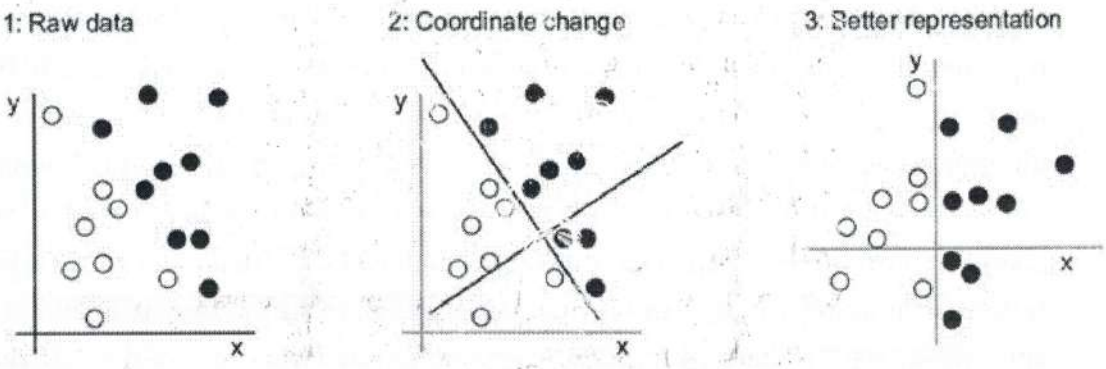


As you can see, we have a few white points and a few black points. Let's say we want to develop an algorithm that can take the coordinates (x, y) of a point and output whether that point is likely to be black or to be white. In this case,

- The inputs are the coordinates of our points. coord → □ → colors
- The expected outputs are the colors of our points. [matrix] [matrix]
- 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 basically solves the classification problem.

In this case, we defined the coordinate change by hand. But if instead we tried systematically searching for different possible coordinate changes, and used as feedback the percentage of points being correctly classified, then we would be doing machine learning. *Learning*, in the context of machine learning, describes an automatic search process for better representations.

All machine-learning algorithms consist of automatically finding such transformations that turn data into more-useful representations for a given task. These operations can be coordinate changes, as you just saw, or linear projections (which may destroy information), 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*.

So that's what machine learning is, technically: searching for useful representations of 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 car 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

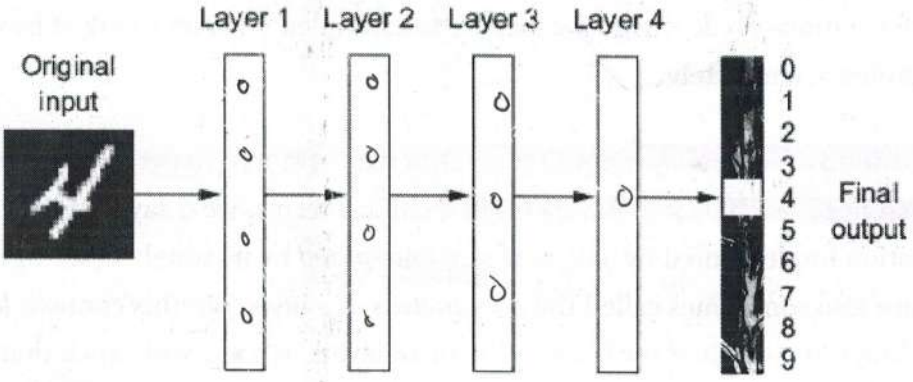
Deep learning is a specific subfield of machine learning: a new take on learning representations from data that puts an emphasis on learning successive *layers* of

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* and *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; hence, they're sometimes called *shallow learning*.

In deep learning, these layered representations are (almost always) learned via models called *neural networks*, structured in literal layers stacked on top of each other. The term *neural network* is a reference to neurobiology, but although some of the central concepts in deep learning were developed in part by drawing inspiration from our understanding of the brain, 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.

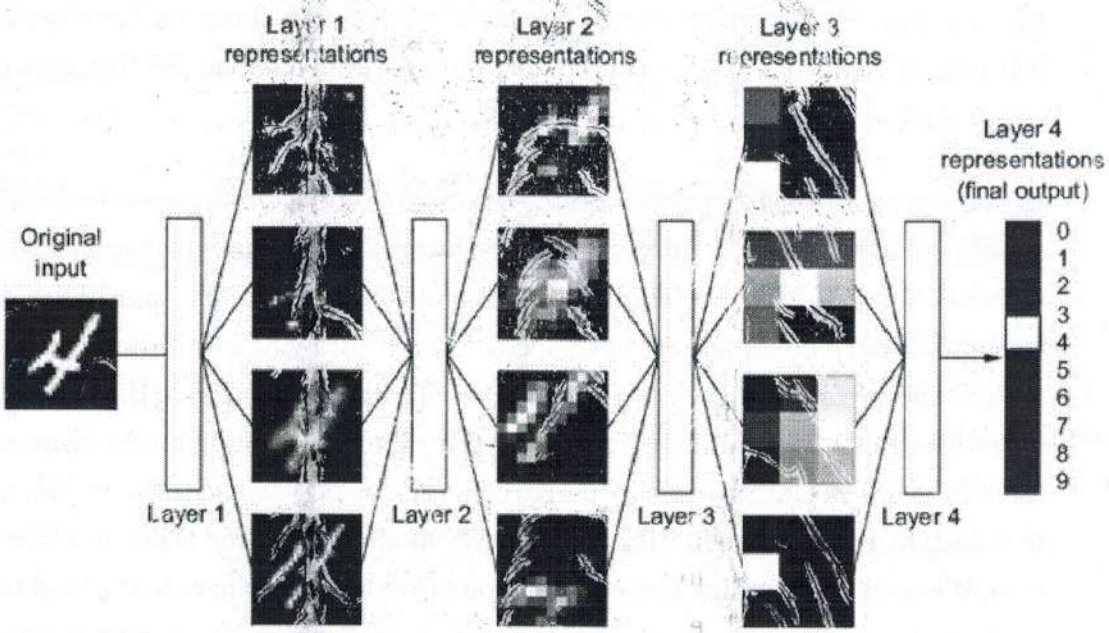
What do the representations learned by a deep-learning algorithm look like? Let's examine how a network several layers deep (see figure 1.5) transforms an image of a digit in order to recognize what digit it is.

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 operation, where information goes through successive filters and comes out increasingly *purified* (that is, useful with regard to some task).

Figure 1.6. Deep representations learned by a digit-classification model



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

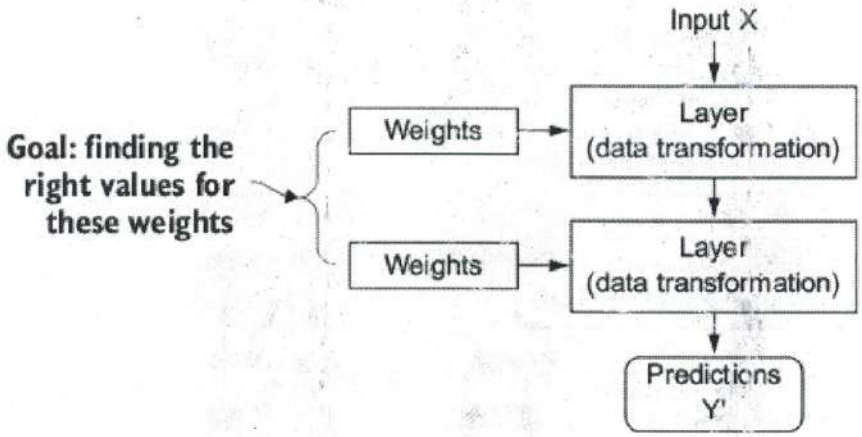
1.1.5. Understanding how deep learning works, in three figures

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 inputs and targets. You also know that deep neural networks do this input-to-target mapping via a deep sequence of simple data transformations (layers) and that these data transformations are learned by exposure to examples. Now let's look at how this learning happens, concretely.

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

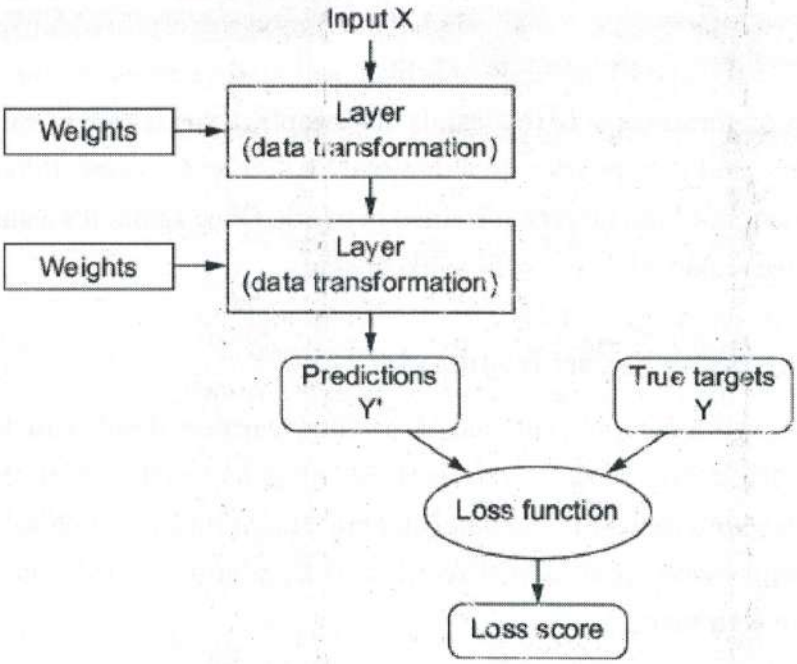
thing: a deep neural network can contain tens of millions of parameters. Finding the correct values for all of them may seem like a daunting task, especially given that modifying the value of one parameter will affect the behavior of all the others!

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 called the *objective 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).

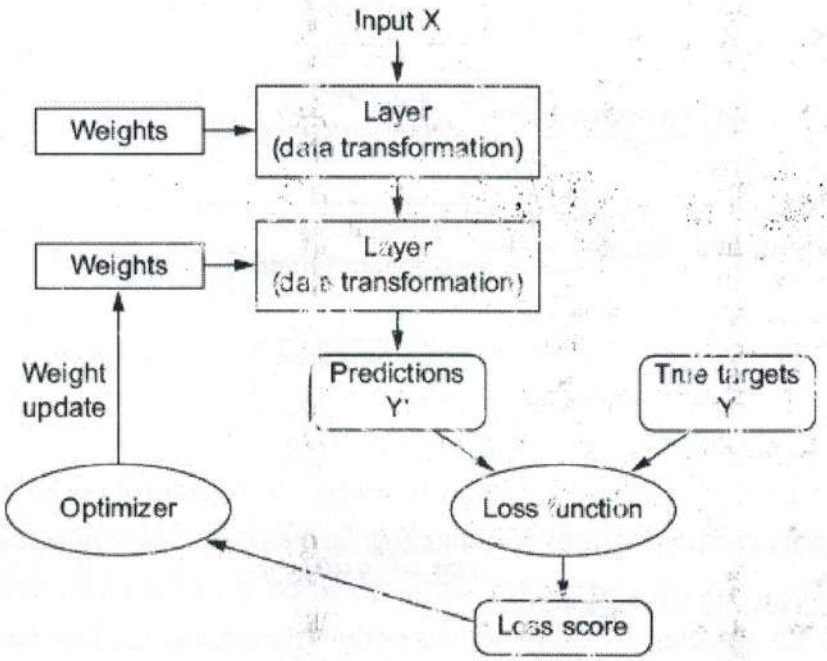
Figure 1.8. A loss function measures the quality of the network's output.



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, with remarkable results on perceptual problems such as seeing and hearing—problems involving skills that seem natural and intuitive to humans but have long been elusive for machines.

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

- Near-human-level image classification
- Near-human-level speech recognition
- Near-human-level handwriting transcription
- Improved machine translation
- Improved text-to-speech conversion
- Digital assistants such as Google Now and Amazon Alexa
- Near-human-level autonomous driving
- Improved ad targeting, as used by Google, Baidu, and 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 to a wide variety of problems outside of machine perception and natural-language understanding, such as formal reasoning. If successful, this may herald an age where deep learning assists humans in science, software development, and more.

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 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 2016, 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 five 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 not applied to the full range of problems they can solve across all industries. Your doctor doesn't yet use AI, and neither does your accountant. You probably don't use AI technologies 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 great 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 a new AI winter—in much the same way the internet industry was overhyped in 1998–1999 and suffered from a crash that dried up investment throughout the early 2000s. But we'll get there eventually. AI will end up being applied to nearly every process that makes up our society and our daily lives, much like the internet is today.

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

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, then 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. ie be well-versed!

A detailed discussion of classical machine-learning approaches is outside of the scope of this book, but we'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 was 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 the *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 (later in the book, we'll precisely define these concepts)—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, [2] although an older linear formulation was published by Vapnik and Alexey Chervonenkis as early as 1963. [3]

haha always here :)

2

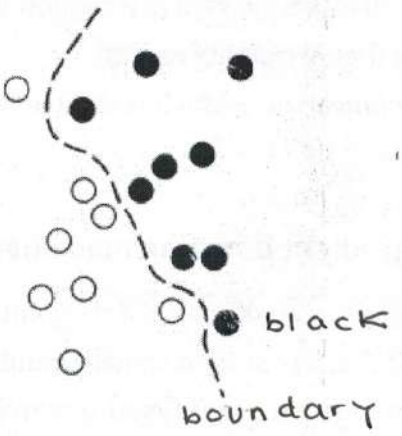
Vladimir Vapnik and Corinna Cortes, "Support-Vector Networks," *Machine Learning* 20, no. 3 (1995): 273–297.

3

Vladimir Vapnik and Alexey Chervonenkis, "A Note on One Class of Perceptrons," *Automation and Remote Control* 25 (1964).

SVMs aim at solving classification problems by finding good *decision boundaries* (see figure 1.10) between two sets of points belonging to two different categories. A decision boundary can be thought of as a line or surface separating your training data into two spaces corresponding to two categories. To classify new data points, you just need to check which side of the decision boundary they fall on.

Figure 1.10. A decision boundary



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 is 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 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.

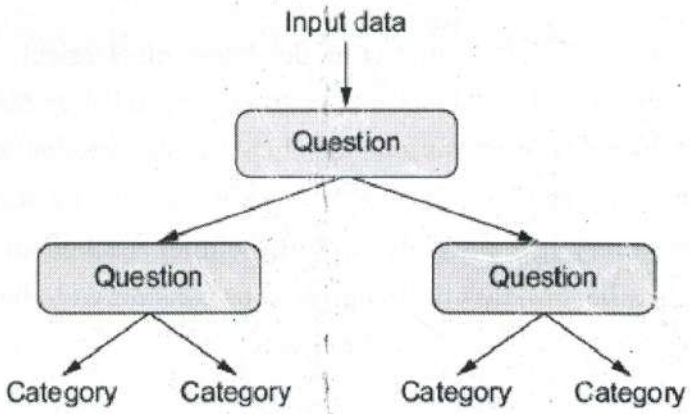
At the time they were developed, SVMs exhibited state-of-the-art performance on simple classification problems and were one of the few machine-learning methods backed by extensive theory and amenable to serious mathematical analysis, making them well understood and easily interpretable. Because of these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

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. Decisions 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 (https://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. 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%. 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 tasks. At major computer vision conferences in 2015 and 2016, 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 analysis of 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 on 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, there are fast-diminishing returns to successive applications of shallow-learning methods, 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 2016 and 2017, Kaggle was dominated by two approaches: gradient boosting machines and deep learning. Specifically, gradient boosting is used for problems where structured data is available, whereas deep learning is used for perceptual problems such as image classification. Practitioners of the former almost always use the excellent XGBoost library. Meanwhile, most Kaggle entrants incorporating deep learning use the Keras library, due to its ease of use and flexibility. XGBoost and Keras both support the two most popular data science languages: R and Python.

These are the two techniques you should be the most familiar with in order to be successful in applied machine learning today: gradient boosting machines, for shallow-learning problems; and deep learning, for perceptual problems. In technical terms, this means you'll need to be familiar with XGBoost and Keras—the two 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 drivers of deep learning for computer vision—convolutional neural networks and backpropagation—were already well understood in 1989. 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 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 what your laptop can deliver. Throughout the 2000s, companies like NVIDIA and AMD have been investing billions of dollars in developing fast, massively parallel chips (graphical processing units

[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[4] and Alex Krizhevsky[5] were among the first.

4

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.

5

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

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 X, a gaming GPU that cost $1,000 at the end of 2015, can deliver a peak of 6.6 TFLOPS in single precision: 6.6 trillion `float32` operations per second. That's about 350 times more than what you can get out of a modern laptop. On a TITAN X, it takes only a couple of days to train an ImageNet model of the sort that would have won the ILSVRC competition a few years ago. Meanwhile, large companies train deep-learning models on clusters of hundreds of GPUs of a type developed specifically for the needs of deep learning, such as the NVIDIA Tesla K80. The sheer computational power of such clusters is something that would never have been possible without modern GPUs.

What's more, the deep-learning industry is starting to go 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, which is reportedly 10 times faster and far more energy efficient than GPUs.

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 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 (1 category per image). But what makes ImageNet special isn't just its large size, but also the yearly competition associated with it.[6]

6

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

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 recent rise of deep learning.

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 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 help gradient propagation were discovered, such as batch normalization, residual connections, and depthwise separable convolutions. Today we can train from scratch models that are thousands of layers deep. can't even picture it

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.

In 2011, right before deep learning took the spotlight, the total venture capital investment in AI was around $19 million, which went almost entirely to practical applications of shallow machine-learning approaches. By 2014, it had risen to a staggering $394 million. Dozens of startups launched in these three years, trying to capitalize on the deep-learning hype. Meanwhile, large tech companies such as Google, Facebook, Baidu, and Microsoft have invested in internal research departments in amounts that would most likely dwarf the flow of venture-capital money. Only a few numbers have surfaced: In 2013, Google acquired the deep-learning startup DeepMind for a reported $500 million—the largest acquisition of an AI company in history. In 2014, Baidu started a deep-learning research center in Silicon Valley, investing $300 million in the project. The deep-learning hardware startup Nervana Systems was acquired by Intel in 2016 for over $400 million.

ie lots of $

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."[7]

lots of business

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

As a result of this wave of investment, the number of people working on deep learning rose in just five years from a few hundred to tens of thousands, and research progress

has reached a frenetic pace. There are currently no signs that this trend will slow any time soon.

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 or R scripting skills suffice to do advanced deep-learning research. This has been driven most notably by the development of Theano and then TensorFlow—two symbolic-tensor-manipulation frameworks 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 invest 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 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 haven't yet established the full scope of what it can do. With every passing month, we learn about new use cases and engineering improvements that lift previous limitations. In the wake of 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. Deep learning in 2017 seems to be in the first half of that sigmoid, with much more progress to come in the next few years.

excellent chapter; 10/10 for clarity;
I didn't have to look anything up & all concepts were well-explained.

Connections to previous knowledge:

1. Neural network → how it looks: INF 1339 w/ Dan Ryan
2. Large Hadron Collider's huge dataset of collisions, ~~ID~~ particles using keras
3. TensorFlow & keras frameworks were described in our Supervised Machine Learning textbook

UNIVERSITY OF TORONTO

Employment Application Form

(This form is to accompany a resume and cover letter.)

The University of Toronto is strongly committed to diversity within its community and especially welcomes applications from racialized persons, women, Indigenous / Aboriginal people, persons with disabilities, LGBTQ persons, and others who may contribute to the further diversification of ideas.

PERSONAL INFORMATION

Name: CHRISTINA NGUYEN
(Given name first, family name second)

Email address: christina.nguyen99@hotmail.com

Phone number Home: 6487 451 5121

Business: N/A

GENERAL INFORMATION

Position Applied for: Assistant

Position Title: Graduate student Library V Department: OISE Library

Job Opportunity No.: posting id 2954

Are you age 18 or older? Yes ✓ No ○

Are you legally entitled to work in Canada? Yes ✓ No ○

(If the University makes a conditional offer of employment, you may be asked to provide proof of your legal entitlement to work in Canada.)

Employment at the University of Toronto:

Current Employee: Yes ○ No ✓ Previously employed: Yes ○ No ✓

If currently or previously employed by the University of Toronto (including UTEMP), you must complete the following:

Department: N/A Reason Left: N/A

Date Left: N/A Personnel No. (if known): N/A Candidate ID (if known): N/A

If not currently employed by the University of Toronto, please complete the following:

Have you ever been convicted of a criminal offence for which a pardon has not been granted? Yes ○ No ✓

What date are you available to start work? immediately

WORK-RELATED REFERENCES

Please provide three employment related references, including your current supervisor. In addition to the references provided by the applicant, the University reserves the right to contact others who it deems relevant and appropriate in the assessment of this application.

Name and Title	Employment Relationship	Company and Phone Number
1. Please look to back of page		
2.		
3.		

All information provided in this form, my resume and cover letter, and information presented during the interview process is truthful to the best of my knowledge. I understand that falsification of any of this information or omission of any pertinent information may disqualify me from employment and/or will constitute grounds for dismissal. If employed, I agree to undergo medical examinations that may be required, which are relevant to the position for which I have applied, including medical examinations that may be required in accordance with University benefit plan requirements.

Date: Sept. 13, 2021 Signature: Christina Nguyen

Print Name: CHRISTINA NGUYEN

WORK - RELATED REFERENCES

	Name + title	Employment Relationship	Company + phone #
①	Prof. Anne MacLennan Professor at Dept. of communication & Media	Project manager of research	York University (416) 668 0507
* ②	Jess Jutras MS.	Intern supervisor	Antarctic Institute of Canada (780) 340 989
③	Prof. Elaina Hyde Professor of Physics & Astronomy	observatory Director	York Universit (647) 410 5881

(* Jess is in MST time zone!)

vector: quantity w/ direction & magnitude
tensor: an object that describes a linear mapping from one set of obj to another

Chapter 2. Before we begin: 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 your intuition about these notions without getting overly technical. In particular, we'll steer away from mathematical notation, which can be off-putting for those without any mathematics background and isn't strictly necessary to explain things well.

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

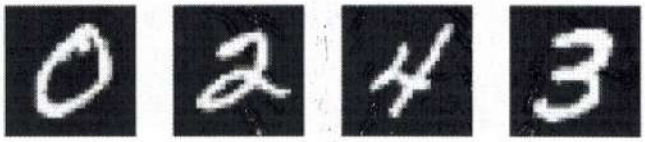
After reading this chapter, you'll have an intuitive understanding of how neural networks work, and you'll be able to move on to practical applications—which will start with chapter 3.

2.1. A FIRST LOOK AT A NEURAL NETWORK

Let's look at a concrete example of a neural network that uses the Keras R package 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. You probably haven't even installed Keras yet; 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



Classes and labels

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 Keras, which is covered in section 3.3.

The MNIST dataset comes preloaded in Keras, in the form of `train` and `test` lists, each of which includes a set of images (`x`) and associated labels (`y`).

Listing 2.1. Loading the MNIST dataset in Keras

```
library(keras)

mnist <- dataset_mnist()
train_images <- mnist$train$x
train_labels <- mnist$train$y
test_images <- mnist$test$x
test_labels <- mnist$test$y
```

`train_images` and `train_labels` form the *training set*: the data from which the model will learn. The model will then be tested on the *test set*: `test_images` and

`test_labels`. The images are encoded as 3D arrays, and the labels are a 1D array of digits, ranging from 0 to 9. The images and labels have a one-to-one correspondence.

The R `str()` function is a convenient way to get a quick glimpse at the structure of an array. Let's use it to look at the training data:

```
> str(train_images)
 int [1:60000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
> str(train_labels)
 int [1:60000(1d)] 5 0 4 1 9 2 1 3 1 4 ...
```

And here's the test data:

```
> str(test_images)
 int [1:10000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
> str(test_labels)
 int [1:10000(1d)] 7 2 1 0 4 1 4 9 5 9 ...
```

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

```
network <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(28 * 28)) %>%
  layer_dense(units = 10, activation = "softmax")
```

If you aren't familiar with the pipe operator (`%>%`) used to invoke methods on the `network` object, fear not: we'll cover this when we review this example again at the end of this chapter. For now, read it in your head as "then": start with a model, then add a layer, then add another layer, and so on.

The core building block of neural networks is the *layer*, a data-processing module that you can think of 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 network consists of a sequence of two layers, which are densely connected (also called *fully connected*) neural layers. The second (and last) layer is a 10-way *softmax* 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 network ready for training, we need to pick three more things, as part of the *compilation* step:

- *A loss function*—How the network will be able to measure its performance on the training data, and thus how it will be able to steer itself in the right direction.
- *An optimizer*—The mechanism through which the network will update itself based on the data it sees and its loss function.
- *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

```
network %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
```

You'll notice that the `compile()` function modifies the network in place (rather than returning a new network object, as is more conventional in R). We'll explain why when we revisit the example later in the chapter.

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

Listing 2.4. Preparing the image data

```
train_images <- array_reshape(train_images, c(60000, 28 * 28))
train_images <- train_images / 255

test_images <- array_reshape(test_images, c(10000, 28 * 28))
test_images <- test_images / 255
```

Note that we use the `array_reshape()` function rather than the `dim<-()` function to reshape the array. We'll explain why later, when we talk about tensor reshaping.

We also need to categorically encode the labels, a step that's explained in chapter 3.

Listing 2.5. Preparing the labels

```
train_labels <- to_categorical(train_labels)
test_labels <- to_categorical(test_labels)
```

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

```
> network %>% fit(train_images, train_labels, epochs = 5, batch_size = 128
Epoch 1/5
60000/60000 [==============================] - 9s - loss: 0.2575 -
    acc: 0.9255
Epoch 2/5
60000/60000 [==============================] - 10s - loss: 0.1038 -
    acc: 0.9687
Epoch 3/5
60000/60000 [==============================] - 10s - loss: 0.0688 -
    acc: 0.9793
Epoch 4/5
60000/60000 [==============================] - 9s - loss: 0.0496 -
    acc: 0.9855
Epoch 5/5
60000/60000 [==============================] - 9s - loss: 0.0372 -
    acc: 0.9883
```

Two quantities are displayed during training: the loss of the network over the training data, and the accuracy of the network over the training data.

We quickly reach an accuracy of 0.989 (98.9%) on the training data. Now let's check that the model performs well on the test set, too: evaluate

```
> metrics <- network %>% evaluate(test_images, test_labels)
> metrics
$loss
[1] 0.07519608

$acc
[1] 0.9785
```

The test-set accuracy turns out to be 97.8%—that's quite a bit lower than the training set accuracy. This gap between training accuracy and test accuracy is an example of *overfitting*: the fact that machine-learning models tend to perform worse on new data than on their training data. Overfitting is a central topic in chapter 3.

Let's generate predictions for the first 10 samples of the test set:

```
> network %>% predict_classes(test_images[1:10,])
 [1] 7 2 1 0 4 1 4 9 5 9
```

This concludes our first example—you just saw how you can build and train a neural network to classify handwritten digits in fewer than 20 lines of R code. In the next chapter, 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 network; tensor operations, which layers are made of; and gradient descent, which allows your network to learn from its training examples.

2.2. DATA REPRESENTATIONS FOR NEURAL NETWORKS

In the previous example, we started from data stored in multidimensional 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 Google's TensorFlow was named after them. So what's a tensor?

nana not again!

Tensors are a generalization of vectors and matrices to an arbitrary number of dimensions (note that in the context of tensors, a *dimension* is often called an *axis*). In R, vectors are used to create and manipulate 1D tensors, and matrices are used for 2D tensors. For higher-level dimensions, `array` objects (which support any number of dimensions) are used.

2.2.1. Scalars (0D tensors)

A tensor that contains only one number is called a *scalar* (or *scalar tensor*, or *zero-dimensional tensor*, or *0D tensor*). R doesn't have a data type to represent scalars (all

numeric objects are vectors, matrices, or arrays), but an R vector that's always length 1 is conceptually similar to a scalar.

2.2.2. Vectors (1D tensors)

A one-dimensional array of numbers is called a *vector*, or *1D tensor*. A 1D tensor is said to have exactly one axis. We can convert the R vector to an `array` object to inspect its dimensions:

```
> x <- c(12, 3, 6, 14, 10)
> str(x)
 num [1:5] 12 3 6 14 10

> dim(as.array(x))
[1] 5
```

This vector has five entries and so is called a *five-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 (2D tensors)

A two-dimensional array of numbers is a *matrix*, 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:

```
> x <- matrix(rep(0, 3*5), nrow = 3, ncol = 5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0
> dim(x)
[1] 3 5
```

2.2.4. 3D tensors and higher-dimensional tensors

If you pack such matrices in a new array, you obtain a 3D tensor, which you can visually

interpret as a cube of numbers:

```
> x <- array(rep(0, 2*3*2), dim = c(2,3,2))
> str(x)
num [1:2, 1:3, 1:2] 0 0 0 0 0 0 0 0

> dim(x)
[1] 2 3 2
```

By packing 3D tensors in an array, you can create a 4D tensor, and so on. In deep learning, you'll generally manipulate tensors that are 0D to 4D, although you may go up to 5D 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 3D tensor has three axes, and a matrix has two axes.
- *Shape*—This is an integer vector that describes how many dimensions the tensor has along each axis. For instance, the previous matrix example has shape `(3, 5)`, and the 3D tensor example has shape `(2, 3, 2)`. A vector has a shape with a single element, such as `(5)`. You can access the dimensions of any array using the `dim()` function.
- *Data type*—This is the type of the data contained in the tensor; for instance, a tensor's type could be `integer` or `double`. On rare occasions, you may see a `character` tensor. But because tensors live in preallocated contiguous memory segments, and strings, being variable-length, would preclude the use of this implementation, they're rarely used.

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

```
library(keras)
mnist <- dataset_mnist()
train_images <- mnist$train$x
train_labels <- mnist$train$y
test_images <- mnist$test$x
test_labels <- mnist$test$y
```

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

```
> length(dim(train_images))
[1] 3
```

Here's its shape:

```
> dim(train_images)
[1] 60000    28    28
```

And this is its data type:

```
> typeof(train_images)
[1] "integer"
```

So what we have here is a 3D tensor of 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 plot the fifth digit in this 3D tensor (see figure 2.2):

Figure 2.2. The fifth sample in our dataset

```
digit <- train_images[5,,]
plot(as.raster(digit, max = 255))
```

2.2.6. Manipulating tensors in R

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 R arrays.

The following example selects digits #10 to #99 and puts them in an array of shape `(90, 28, 28)`:

```
> my_slice <- train_images[10:99,,]
> dim(my_slice)
[1] 90 28 28
```

It's equivalent to this more detailed notation, which specifies a start index and a stop index for the slice along each tensor axis:

```
> my_slice <- train_images[10:99,1:28,1:28]
> dim(my_slice)
[1] 90 28 28
```

In general, you may select 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 do this:

```
my_slice <- train_images[, 15:28, 15:28]
```

2.2.7. The notion of data batches

In general, the first axis 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 batch size of 128:

```
batch <- train_images[1:128,,]
```

And here's the next batch:

```
batch <- train_images[129:256,,]
```

When considering such a batch tensor, the first axis 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*—2D tensors of shape `(samples, features)`

- *Timeseries data or sequence data*—3D tensors of shape `(samples, timesteps, features)`
- *Images*—4D tensors of shape `(samples, height, width, channels)` or `(samples, channels, height, width)`
- *Video*—5D tensors of shape `(samples, frames, height, width, channels)` or `(samples, frames, channels, height, width)`

2.2.9. Vector data

This is the most common case. 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 2D tensor (that is, an array of vectors), where the first axis is the *samples axis* and the second axis is the *features axis*.

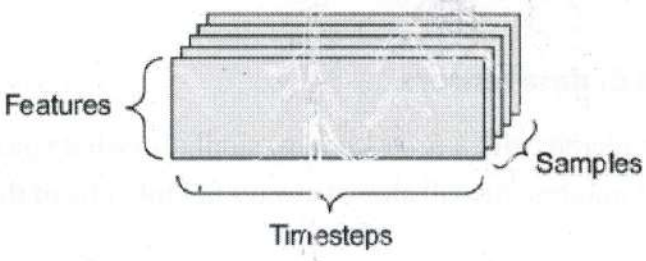
Let's take a look at two examples:

- An actuarial dataset of people, where we consider each person's age, ZIP code, 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 2D 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 3D tensor with an explicit time axis. Each sample can be encoded as a sequence of vectors (a 2D tensor), and thus a batch of data will be encoded as a 3D tensor (see figure 2.3).

Figure 2.3. A 3D timeseries data tensor



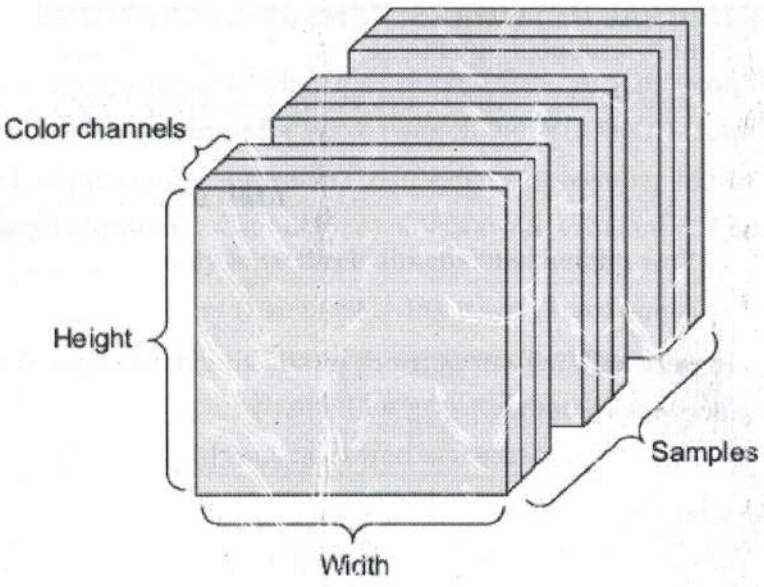
The time axis is always the second axis, by convention. Let's look at a few examples:

- 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 2D tensor of shape `(390, 3)` (there are 390 minutes in a trading day), and 250 days' worth of data can be stored in a 3D 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 140 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 2D tensor of shape `(140, 128)`, and a dataset of 1 million tweets can be stored in a tensor of shape `(1000000, 140, 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 2D tensors, by convention image tensors are always 3D, 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 4D image data tensor (channels-first convention)



There are two conventions for shapes of images tensors: the *channels-last* convention (used by TensorFlow) and the *channels-first* convention (used by Theano). The

TensorFlow machine-learning framework, from Google, places the color-depth axis at the end: `(samples, height, width, color_depth)`. Meanwhile, Theano places the color depth axis right after the batch axis: `(samples, color_depth, height, width)`. With the Theano convention, the previous examples would become `(128, 1, 256, 256)` and `(128, 3, 256, 256)`. The Keras framework provides support for both formats, TensorFlow & Theano.

2.2.12. Video data

Video data is one of the few types of real-world data for which you'll need 5D 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 3D tensor `(height, width, color_depth)`, a sequence of frames can be stored in a 4D tensor `(frames, height, width, color_depth)`, and thus a batch of different videos can be stored in a 5D 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 data type of the tensor is `double`, then each value is stored in 64 bits, so the tensor would represent 810 MB. Heavy! Videos you encounter in real life are much lighter, because they aren't stored as `double` 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* applied to tensors of numeric data. For instance, it's possible to add tensors, multiply tensors, and so on.

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

```
layer_dense(units = 512, activation = "relu")
```

This layer can be interpreted as a function, which takes as input a 2D tensor and returns another 2D tensor—a new representation for the input tensor. Specifically, the function is as follows (where `W` is a 2D tensor and `b` is a vector, both attributes of the

layer):

```
output = relu(dot(W, input) + 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 2D tensor and a vector `b`; and, finally, a `relu` operation. `relu(x)` is `max(x, 0)`.

Note

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

2.3.1. Element-wise operations

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

```
naive_relu <- function(x) {                    1
  for (i in nrow(x))
    for (j in ncol(x))
      x[i, j] <- max(x[i, j], 0)
  x
}
```

- ***1*** **`x` is a 2D tensor (R matrix).**

You do the same for addition:

```
naive_add <- function(x, y) {                  1
  for (i in nrow(x))
    for (j in ncol(x))
```

```
        x[i, j] = x[i, j] + y[i, j]
    x
}
```

- *1* **x and y are 2D tensors (matrices).**

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

In practice, when dealing with R arrays, these operations are available as well-optimized built-in R functions, which themselves delegate the heavy lifting to a BLAS implementation (Basic Linear Algebra Subprograms) if you have one installed (which you should). BLAS are low-level, highly parallel, efficient tensor-manipulation routines typically implemented in Fortran or C.

So in R you can do the following native element-wise operations, and they will be blazing fast:

```
z <- x + y          1
z <- pmax(z, 0)     2
```

- *1* **Element-wise addition**
- *2* **Element-wise relu**

2.3.2. Operations involving tensors of different dimensions

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

The R `sweep()` function enables you to perform operations between higher-dimension tensors and lower-dimension tensors. With `sweep()`, we could perform the matrix plus vector addition described earlier as follows:

```
sweep(x, 2, y, `+`)
```

The second argument (here, `2`) specifies the dimensions of `x` over which to sweep `y`. The last argument (here, `+`) is the operation to perform during the sweep, which should be a function of two arguments: `x` and an array of the same dimensions generated from `y` by `aperm()`.

You can apply a sweep in any number of dimensions and can apply any function that implements a vectorized operation over two arrays. The following example sweeps a 2D tensor over the last two dimensions of a 4D tensor using the `pmax()` function:

```
x <- array(round(runif(1000, 0, 9)), dim = c(64, 3, 32, 10))      1
y <- array(5, dim = c(32, 10))                                    2

z <- sweep(x, c(3, 4), y, pmax)                                   3
```

- **1 x is a tensor of random values with shape (64, 3, 32, 10).**
- **2 y is a tensor of 5s of shape (32, 10).**
- **3 The output z has shape (64, 3, 32, 10), like x.**

2.3.3. Tensor dot

The dot operation, also called a *tensor product* (not to be confused with an element-wise product) is the most common, most useful tensor operation. Contrary to element-wise operations, it combines entries in the input tensors.

An element-wise product is done with the * operator in R, whereas dot products use the %*% operator:

```
z <- 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:

```
naive_vector_dot <- function(x, y) {                              1
  z <- 0
  for (i in 1:length(x))
    z <- z + x[[i]] * y[[i]]
  z
}
```

- **1 x and y are 1D tensors (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 whose elements are the dot products between `y` and the rows of `x`. You implement it as follows:

```
naive_matrix_vector_dot <- function(x, y) {
  z <- rep(0, nrow(x))
  for (i in 1:nrow(x))
    for (j in 1:ncol(x))
      z[[i]] <- z[[i]] + x[[i, j]] * y[[j]]
  z
}
```

- ***1*** **`x` is a 2D tensor (matrix). `y` is a 1D tensor (vector).**

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

```
naive_matrix_vector_dot <- function(x, y) {
  z <- rep(0, nrow(x))
  for (i in 1:nrow(x))
    z[[i]] <- naive_vector_dot(x[i,], y)
  z
}
```

Note that as soon as one of the two tensors has more than one dimension, `%*%` is no longer symmetric, which is to say that `x %*% y` isn't the same as `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` (`x %*% y`) if and only if `ncol(x) == nrow(y)`. The result is a matrix with shape `(nrow(x), ncol(y))`, where the coefficients are the vector products between the rows of `x` and the columns of `y`. Here's the naive implementation:

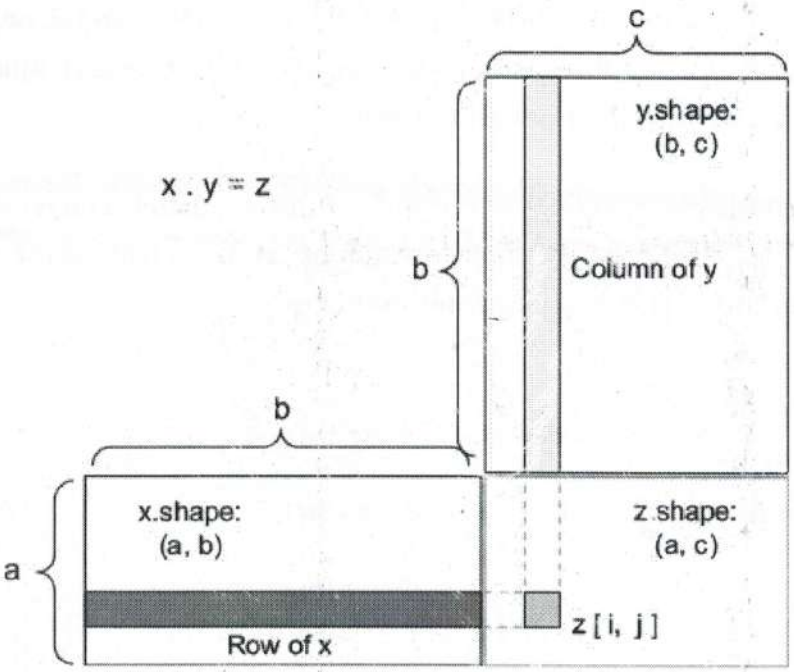
```
naive_matrix_dot <- function(x, y) {
  z <- matrix(0, nrow = nrow(x), ncol = ncol(y))
  for (i in 1:nrow(x))
    for (j in 1:ncol(y)) {
      row_x <- x[i,]
      column_y <- y[,j]
```

```
            z[i, j] <- naive_vector_dot(row_x, column_y)
        }
    z
}
```

- ***1* x and y are 2D tensors (matrices).**

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

Figure 2.5. Matrix dot-product box diagram



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

More generally, you can take the 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 network:

```
train_images <- array_reshape(train_images, c(60000, 28 * 28))
```

Note that we use the `array_reshape()` function rather than the `dim<-()` function to reshape the array. This is so that the data is reinterpreted using row-major semantics (as opposed to R's default column-major semantics), which is in turn compatible with the way the numerical libraries called by Keras (NumPy, TensorFlow, and so on) interpret array dimensions. You should always use the `array_reshape()` function when reshaping R arrays that will be passed to Keras.

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 <- matrix(c(0, 1,
                2, 3,
                4, 5),
              nrow = 3, ncol = 2, byrow = TRUE)

> x
     [,1] [,2]
[1,]    0    1
[2,]    2    3
[3,]    4    5

> x <- array_reshape(x, dim = c(6, 1))
> x
     [,1]
[1,]    0
[2,]    1
[3,]    2
[4,]    3
[5,]    4
[6,]    5

> x <- array_reshape(x, dim = c(2, 3))
> x
     [,1] [,2] [,3]
[1,]    0    1    2
[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]`. The `t()` function can be used to transpose a matrix:

```
> x <- matrix(0, nrow = 300, ncol = 20)
> dim(x)
[1] 300  20

> x <- t(x)
> dim(x)
[1]  20 300
```

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, 1.0]
```

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

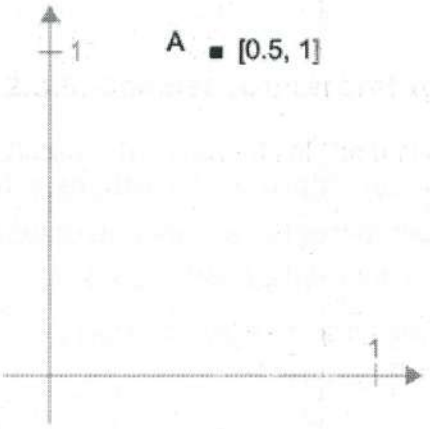
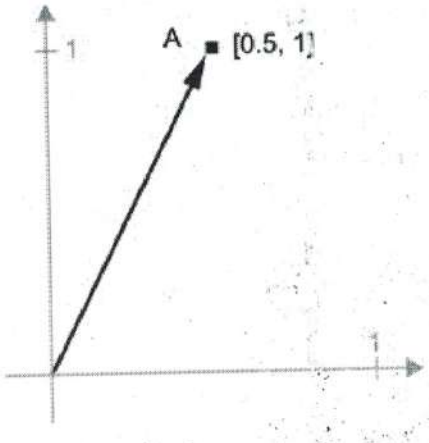
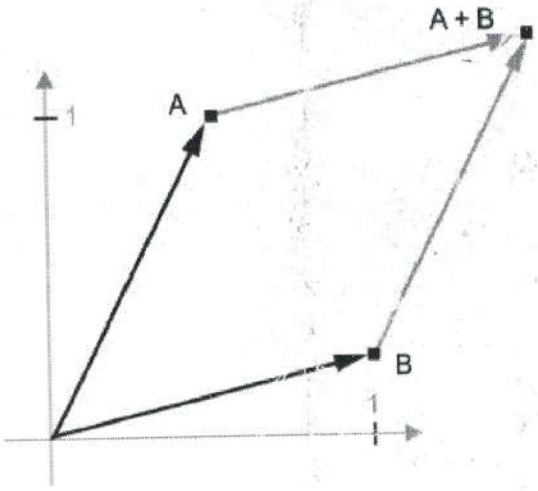


Figure 2.7. A point in a 2D space pictured as an arrow



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).

Figure 2.8. Geometric interpretation of the sum of two vectors



In general, elementary geometric operations such as affine transformations, rotations, scaling, and so on can be expressed as tensor operations. For instance, a rotation of a 2D vector by an angle theta can be achieved via a dot product with a 2×2 matrix `R = [u, v]`, where `u` and `v` are both vectors of the plane: `u = [cos(theta), sin(theta)]` and `v = [-sin(theta), cos(theta)]`.

2.3.6. A geometric interpretation of deep learning

You just learned that neural networks consist entirely of chains of tensor operations and that all of these tensor operations are just 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 long 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 (or any other machine-learning model) 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. 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 2.9. Uncrumpling a complicated manifold of data

Uncrumpling paper balls is what machine learning is about: finding neat representations for complex, highly folded data manifolds. 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 network example transforms its input data as follows:

```
output = relu(dot(W, input) + 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 network 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(W, input) + b)`, when `W` and `b` are random, will yield any useful representations. The resulting representations are meaningless—but they're a starting point. What comes next is to gradually adjust these weights, based on a feedback signal. This gradual adjustment, also called *training*, is basically the learning that machine learning is all about.

This happens within what's called a *training loop*, which works as follows. Repeat these steps in a loop, as long as necessary:

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

You'll eventually end up with a network that has a very low loss on its training data: a low mismatch between predictions `y_pred` and expected targets `y`. The network 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 network's weights. Given an individual weight coefficient in the network, 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 network 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 network 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 network.

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). A much better approach is to take advantage of the fact that all operations used in the network are *differentiable*, and compute the *gradient* of the loss with regard to the network's coefficients. You can then move the coefficients in the opposite direction from the gradient, thus decreasing 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 real number `x` to a new real number `y`. 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`:

```
f(x + epsilon_x) = y + epsilon_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
```

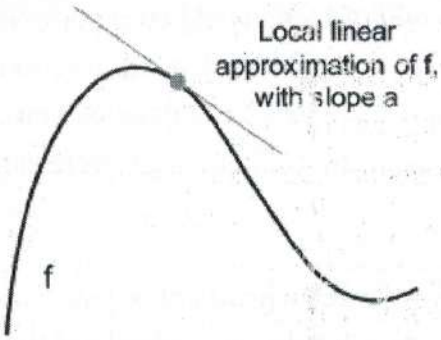
Obviously, this linear approximation is valid only when `x` is close enough to `p`.

The slope `a` is called the *derivative* of `f` in `p`. If `a` is negative, it means a small change of `x` around `p` will result in a decrease of `f(x)` (as shown in figure 2.10); and if `a` is positive, a small change 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.

yep, as usual with math

Figure 2.10. Derivative of `f` in `p`

|a| = steepness = how quick.



For every differentiable function `f(x)` (*differentiable* means "can be differentiated": for example, smooth, continuous functions can be differentiated), 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.

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

A *gradient* is the derivative of a tensor operation. It's the generalization of the concept of derivatives to functions of multidimensional inputs: that is, to functions that take tensors as inputs.

Consider an input vector `x`, a matrix `W`, a target `y`, and a loss function `loss`. You can use `W` to compute a target candidate `y_pred`, and compute the loss, or mismatch, between the target candidate `y_pred` and the target `y`:

```
y_pred = dot(W, x)
loss_value = loss(y_pred, y)
```

If the data inputs `x` and `y` are frozen, then this can be interpreted as a function mapping values of `W` to loss values:

```
loss_value = f(W)
```

Let's say the current value of `W` is `W0`. Then the derivative of `f` in the point `W0` is a tensor `gradient(f)(W0)` with the same shape as `W`, where each coefficient `gradient(f)`

`(W0) [i, j]` indicates the direction and magnitude of the change in `loss_value` you'd observe when modifying `W0[i, j]`. That tensor `gradient(f)(W0)` is the gradient of the function `f(W) = loss_value` in `W0`.

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, `gradient(f)(W0)` can be interpreted as the tensor describing the *curvature* of `f(W)` around `W0`.

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 `f(W)` by moving `W` in the opposite direction from the gradient: for example, `W1 = W0 - step * gradient(f)(W0)` (where `step` is a small scaling factor). That means going against the curvature, which intuitively should put you lower on the curve. Note that the scaling factor `step` is needed because `gradient(f)(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 `gradient(f)(W) = 0` for `W`. This is a polynomial equation of N variables, where N is the number of coefficients in the network. 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.

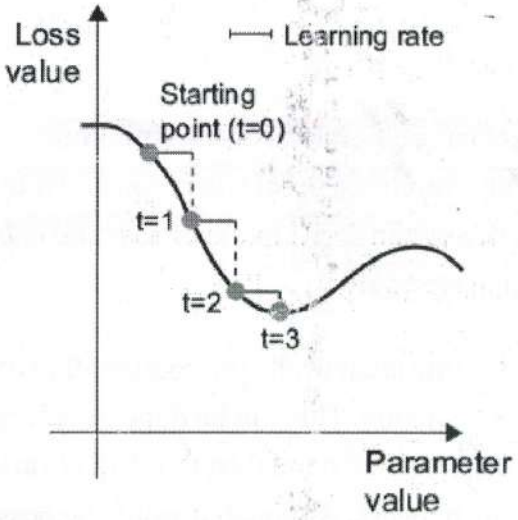
Instead, you can use the four-step algorithm outlined at the beginning of this section: modify the parameters little by little based on the current loss value on a random batch of data. Because you're dealing with a differentiable function, you can compute its gradient, which gives you an efficient way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

1. Draw a batch of training samples `x` and corresponding targets `y`.

2. Run the network on `x` to obtain predictions `y_pred`.
3. Compute the loss of the network on the batch, a measure of the mismatch between `y_pred` and `y`.
4. Compute the gradient of the loss with regard to the network's parameters (a *backward pass*).
5. Move the parameters a little in the opposite direction from the gradient—for example, `W = W - (step * gradient)`—thus reducing the loss on the batch a bit.

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 2.11 illustrates what happens in 1D, when the network has only one parameter and you have only one training sample.

Figure 2.11. SGD down a 1D loss curve (one learnable parameter)

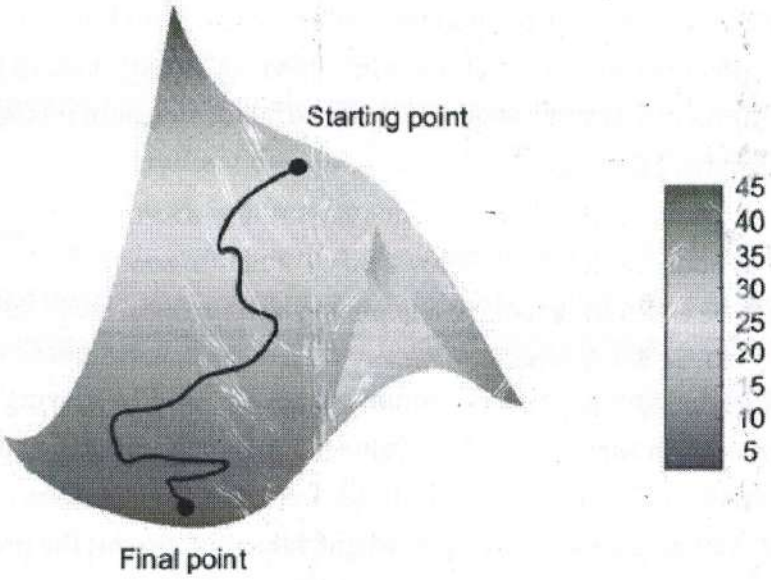


As you can see, intuitively it's important to pick a reasonable value for the `step` factor. If it's too small, the descent down the curve will take many iterations, and it could get stuck in a local minimum. If `step` is too large, your updates may end up taking you to completely random locations on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

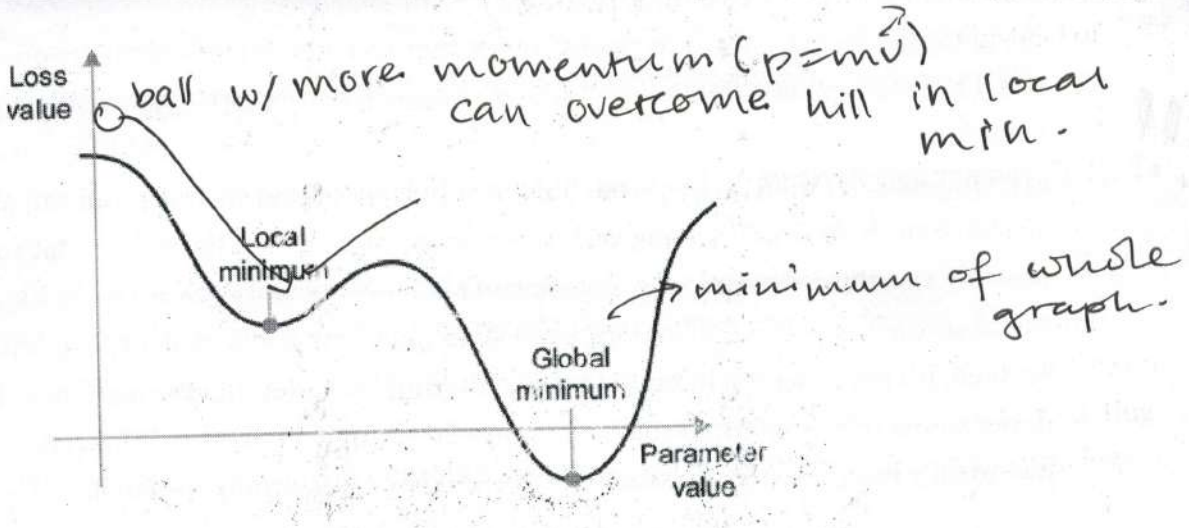
Although figure 2.11 illustrates gradient descent in a 1D parameter space, in practice you'll use gradient descent in highly dimensional spaces: every weight coefficient in a neural network is a free dimension in the space, and there may be tens of thousands or even millions of them. To help you build intuition about loss surfaces, you can also visualize gradient descent along a 2D loss surface, as shown in figure 2.12. 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 2.12. 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 2.13, which shows the curve of a loss as a function of a network parameter.

Figure 2.13. 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, then the optimization process would get stuck at the local minimum instead of making its way to the global minimum.

You can avoid such issues by using momentum, which draws inspiration from physics. A useful mental image here is to think of the optimization process as a small ball rolling down the loss curve. If it has enough 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
while (loss > 0.01) {
  params <- get_current_parameters()
  w <- params$w
  loss <- params$loss
  gradient <- params$gradient

  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 previous algorithm, we casually assumed that because a function is

differentiable, we can explicitly compute its derivative. In practice, a neural network function consists of many tensor operations chained together, each of which has a simple, known derivative. For instance, this is a network `f` composed of three tensor operations `a`, `b`, and `c`, with weight matrices `W1`, `W2`, and `W3`:

```
f(W1, W2, W3) = a(W1, b(W2, c(W3)))
```

Calculus tells us that such a chain of functions can be differentiated using the following identity, called the *chain rule*: `f(g(x)) = f'(g(x)) * g'(x)`. Applying the chain rule to the computation of the gradient values of a neural network gives rise to an algorithm called *Backpropagation* (also sometimes called *reverse-mode differentiation*). Backpropagation starts with the final loss value and works backward from the top layers to the bottom layers, applying the chain rule to compute the contribution that each parameter had in the loss value.

Nowadays and for years to come, people will implement networks in modern frameworks that are capable of *symbolic differentiation*, such as TensorFlow. This means that, given a chain of operations with a known derivative, they can compute a gradient *function* for the chain (by applying the chain rule) that maps network parameter values to gradient values. When you have access to such a function, the backward pass is reduced to a call to this gradient function. Thanks to symbolic differentiation, you'll never have to implement the Backpropagation algorithm by hand. For this reason, we won't waste your time and your focus on deriving the exact formulation of the Backpropagation algorithm in these pages. All you need is a good understanding of how gradient-based optimization works.

2.5. LOOKING BACK AT OUR FIRST EXAMPLE

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

This was the input data:

```
library(keras)
mnist <- dataset_mnist()

train_images <- mnist$train$x
train_images <- array_reshape(train_images, c(60000, 28 * 28))
train_images <- train_images / 255
```

```
test_images <- mnist$test$x
test_images <- array_reshape(test_images, c(10000, 28 * 28))
test_images <- test_images / 255
```

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

This was our network:

```
network <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(28*28)) %>%
  layer_dense(units = 10, activation = "softmax")
```

Now you understand that this network 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 network persists.

Using the pipe operator

You use the pipe (`%>%`) operator to add layers to a network. This operator comes from the `magrittr` package; it's shorthand for passing the value on its left as the first argument to the function on its right. We could have written the network code as follows:

```
network <- keras_model_sequential()
layer_dense(network, units = 512, activation = "relu",
            input_shape = c(28*28))
layer_dense(network, units = 10, activation = "softmax")
```

Using `%>%` results in code that's more readable and compact, so we'll use this form throughout the book.

If you're using RStudio, you can insert `%>%` using the Ctrl-Shift-M keyboard shortcut. To learn more about the pipe operator, see http://r4ds.had.co.nz/pipes.html.

This was the network-compilation step:

```
network %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
```

Now you understand that `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.

In-place modification of models

We're using the `%>%` operator to call `compile()`. We could have written the network compilation step as follows:

```
compile(
  network,
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
```

Using `%>%` for `compile` is less about compactness and more about providing a syntactic reminder of an important characteristic of Keras models: unlike most objects you work with in R, Keras models are modified in place. This is because Keras models are directed acyclic graphs of layers whose state is updated during training.

You don't operate on `network` and then return a new `network` object. Rather, you *do something to* the `network` object. Placing `network` to the left of `%>%` and not saving the results to a new variable signals to the reader that you're modifying in place.

Finally, this was the training loop:

```
network %>% fit(train_images, train_labels, epochs = 5, batch_size = 128)
```

Now you understand what happens when you call `fit`: the network 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*). At each iteration, the network will compute the gradients of the weights with regard to the loss on the batch, and update the weights accordingly. After these 5 epochs, the network will have performed 2,345 gradient updates (469 per epoch), and the loss of the network will be sufficiently low that the network 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.

2.6. SUMMARY

- *Learning* means finding a combination of model parameters 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 network parameters with respect to the loss on the batch. The network parameters are then moved a bit (the magnitude of the move is defined by the learning rate) in the opposite direction from the gradient.
- The entire learning process is made possible by the fact that neural networks are chains of differentiable tensor operations, 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.
- 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 network:
    - 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.

Chapter 3. Getting started with neural networks

This chapter covers

- Core components of neural networks
- An introduction to Keras
- Setting up a deep-learning workstation
- Using neural networks to solve basic classification and regression problems

This chapter is designed to get you started with using neural networks to solve real problems. You'll consolidate the knowledge you gained from our first practical example in chapter 2, and you'll apply what you've learned to three new problems covering the three most common use cases of neural networks: binary classification, multiclass classification, and scalar regression.

In this chapter, we'll take a closer look at the core components of neural networks that we introduced in chapter 2: layers, networks, objective functions, and optimizers.

We'll give you a quick introduction to Keras, the deep-learning library that we'll use throughout the book. You'll set up a deep-learning workstation with TensorFlow, Keras, and GPU support. We'll dive into three introductory examples of how to use neural networks to address real problems:

- 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 (regression)

By the end of this chapter, you'll be able to use neural networks to solve simple machine problems such as classification and regression over vector data. You'll then be ready to start building a more principled, theory-driven understanding of machine learning in chapter 4.

3.1. ANATOMY OF A NEURAL NETWORK

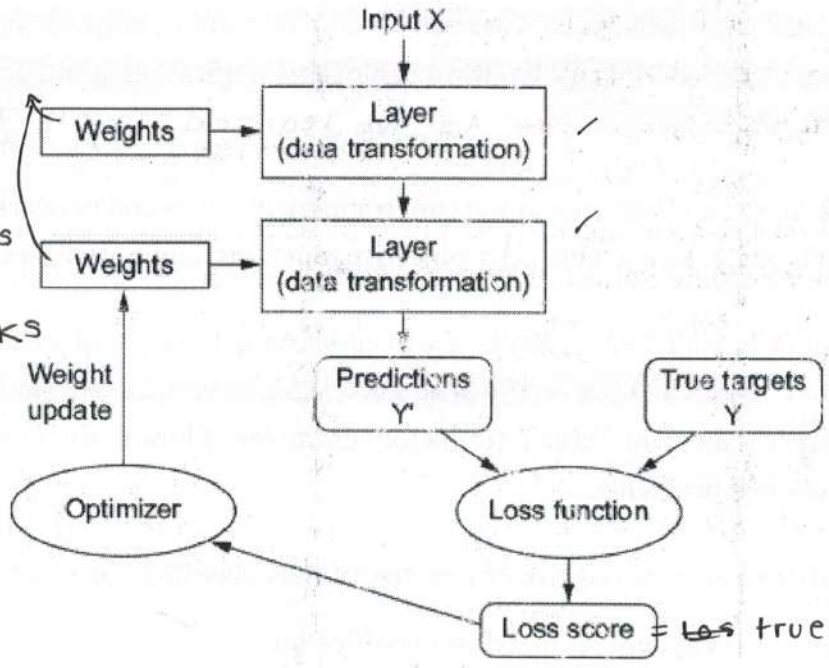
As you saw in the previous chapters, training a neural network revolves around the following objects:

- *Layers*, which are combined into a *network* (or *model*)
- The *input data* and corresponding *targets*
- The *loss function*, which defines the feedback signal used for learning
- The *optimizer*, which determines how learning proceeds

You can visualize their interaction as illustrated in figure 3.1: the network, 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 network's predictions match what was expected. The optimizer uses this loss value to update the network's weights.

Let's take a closer look at layers, networks, loss functions, and optimizers.

Figure 3.1. Relationship between the network, layers, loss function, and optimizer



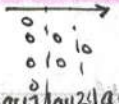
3.1.1. Layers: the building blocks of deep learning

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 layers are appropriate for different tensor formats and different types of data processing. For instance, simple vector data, stored in 2D tensors of shape `(samples, features)`, is often processed by *densely connected* layers, also called *fully connected* or *dense* layers (the `layer_dense` function in Keras). Sequence data, stored in 3D tensors of shape `(samples, timesteps, features)`, is typically processed by *recurrent* layers such as `layer_lstm`. Image data, stored in 4D tensors, is usually processed by 2D convolution layers (`layer_conv_2d`).

DEFINE LAYER !

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



```
layer <- layer_dense(units = 32, input_shape = c(784))
```

We're creating a layer that will only accept as input 2D tensors where the first dimension is 784 (the first dimension, the batch dimension, is unspecified, and thus any value would be accepted). This layer will return a tensor where the first dimension has been transformed to be 32.

Thus this layer 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 compatibility, 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:

↳ eg you feed a [30×30×30] matrix into a [21×21×21] just fine

```
model <- keras_model_sequential() %>%
  layer_dense(units = 32, input_shape = c(784)) %>%
  layer_dense(units = 32)
```

is that what it means?

The second layer didn't receive an input shape argument—instead, it automatically inferred its input shape as being the output shape of the layer that came before.

3.1.2. Models: networks of layers

not cyclical.

A deep-learning model is a directed, acyclic graph of layers. The most common instance is a linear stack of layers, mapping a single input to a single output.

!

But as you move forward, you'll be exposed to a much broader variety of network

topologies. Some common ones include the following:

- Two-branch networks
- Multihead networks
- Inception blocks

The topology of a network defines a *hypothesis space*. You may remember that in chapter 1, we defined 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.

Picking the right network architecture is more an art than a science; and although there are some best practices and principles you can rely on, only practice can help you become a proper neural-network architect. The next few chapters will both teach you explicit principles for building neural networks and help you develop intuition as to what works or doesn't work for specific problems.

3.1.3. Loss functions and optimizers: keys to configuring the learning process

Once the network architecture is defined, you still have to choose two 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).

A neural network that has multiple outputs may have multiple loss functions (one per output). But the gradient-descent process must be based on a *single* scalar loss value; so, for multiloss networks, all losses are combined (via averaging) into a single scalar quantity.

Choosing the right objective 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, mean-squared error for a regression problem, connectionist temporal classification (CTC) for a sequence-learning problem, and so on. Only when you're working on truly new research problems will you have to develop your own objective functions. In the next few chapters, we'll detail explicitly which loss functions to choose for a wide range of common tasks.

3.2. INTRODUCTION TO KERAS

Throughout this book, the code examples use Keras (https://keras.rstudio.com). Keras is a deep-learning framework that provides a convenient way to define and train almost any kind of deep-learning model. Keras was initially developed for researchers, with the aim of enabling fast experimentation.

Keras has the following key features:

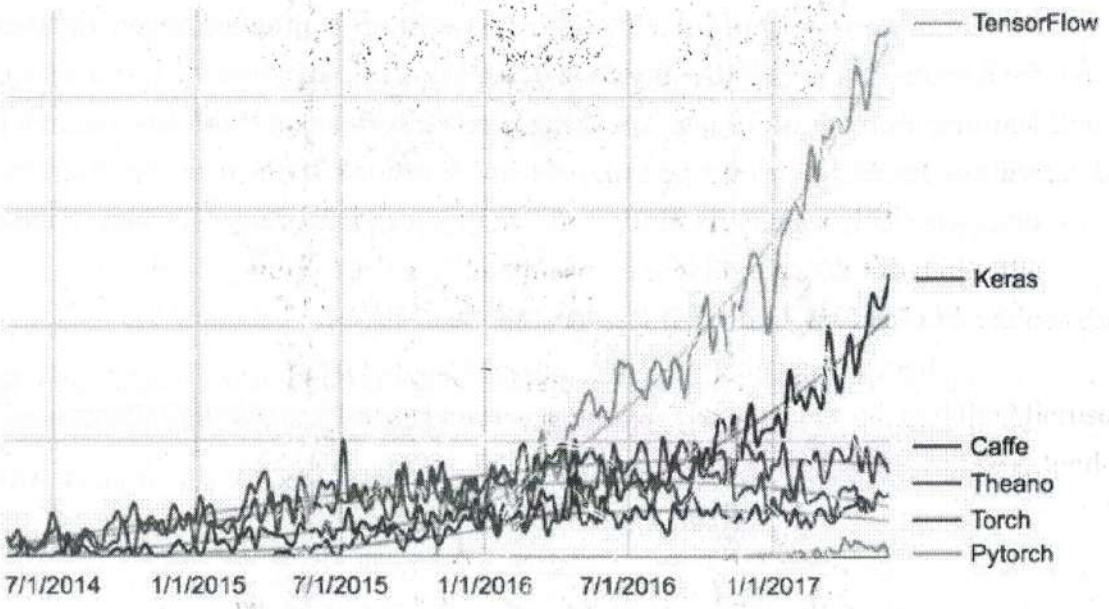
CPU is designed to handle a wide-range of tasks quickly; GPU is designed to quickly render high-res images & videos concurrently

- It allows the same code to run seamlessly on CPU or GPU.
- It has a user-friendly API that makes it easy to quickly prototype deep-learning models.
- It has built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.
- It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, and so on. This means Keras is appropriate for building essentially any deep-learning model, from a generative adversarial network to a neural Turing machine.

Keras and its R interface are distributed under the permissive MIT license, which means they can be freely used in commercial projects. The Keras R package is compatible with R versions 3.2 and higher. The documentation for the R interface is available at https://keras.rstudio.com. The main Keras project website can be found at https://keras.io.

Keras has well over 150,000 users, ranging from academic researchers and engineers at both startups and large companies to graduate students and hobbyists. Keras is used at Google, Netflix, Uber, CERN, Yelp, Square, and hundreds of startups working on a wide range of problems. Keras is also a popular framework on Kaggle, the machine-learning competition website, where almost every recent deep-learning competition has been won using Keras models (see figure 3.2).

Figure 3.2. Google web search interest for different deep-learning frameworks over time



3.2.1. Keras, TensorFlow, Theano, and CNTK

Keras is a model-level library, providing high-level building blocks for developing deep-learning models. It doesn't handle low-level operations such as tensor manipulation and differentiation. Instead, it relies on a specialized, well-optimized tensor library to do so, serving as the *backend engine* of Keras. Rather than choosing a single tensor library and tying the implementation of Keras to that library, Keras handles the problem in a modular way (see figure 3.3); thus, several different backend engines can be plugged seamlessly into Keras. Currently, the three existing backend implementations are the TensorFlow backend, the Theano backend, and the Microsoft Cognitive Toolkit (CNTK) backend. In the future, it's likely that Keras will be extended to work with even more deep-learning execution engines.

Figure 3.3. The deep-learning software and hardware stack

Keras
(which
includes
TensorFlow,
Theano,
CNTK)

TensorFlow, CNTK, and Theano are some of the primary platforms for deep learning today. Theano (http://deeplearning.net/software/theano) is developed by the MILA lab at *Université de Montréal*, TensorFlow (www.tensorflow.org) is developed by Google, and CNTK (https://github.com/Microsoft/CNTK) is developed by Microsoft. Any piece of code that you write with Keras can be run with any of these backends without having to change anything in the code: you can seamlessly switch between the two during development, which often proves useful—for instance, if one of these backends proves to be faster for a specific task. We recommend using the TensorFlow backend as the default for most of your deep-learning needs, because it's the most widely adopted, scalable, and production ready.

Via TensorFlow (or Theano, or CNTK), Keras is able to run seamlessly on both CPUs and GPUs. When running on CPU, TensorFlow is itself wrapping a low-level library for tensor operations, called Eigen (http://eigen.tuxfamily.org). On GPU, TensorFlow wraps a library of well-optimized deep-learning operations called the NVIDIA CUDA Deep Neural Network library (cuDNN).

3.2.2. Installing Keras

To get started with Keras, you need to install the Keras R package, the core Keras library, and a backend tensor engine (such as TensorFlow). You can do so as follows:

```
install.packages("keras")                    1

library(keras)                               2
install_keras()                              2
```

- **1 Installs the Keras R package**
- **2 Installs the core Keras library and TensorFlow**

This will provide you with a default CPU-based installation of Keras and TensorFlow.

As noted in the next section on setting up a deep-learning workstation, you'll probably

want to train your deep learning models on a GPU. If you're running on a system with an NVIDIA GPU and properly configured CUDA and cuDNN libraries, you can install the GPU-based version of the TensorFlow backend engine as follows:

```
install_keras(tensorflow = "gpu")
```

Note that you should do this only if your workstation has an NVIDIA GPU and required software (CUDA and cuDNN), because the GPU version of TensorFlow will fail to load if these prerequisites aren't met. The next section covers GPU configurations in more detail.

3.2.3. Developing with Keras: a quick overview

You've already seen one example of a Keras model: the MNIST example. The typical Keras workflow looks just like that example:

1. Define your training data: input tensors and target tensors.
2. Define a network of layers (or *model*) that maps your inputs to your targets.
3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
4. Iterate on your training data by calling the `fit()` method of your model.

There are two ways to define a model: using the `keras_model_sequential()` function (only for linear stacks of layers, which is the most common network architecture by far) or the *functional API* (for directed acyclic graphs of layers, which let you build completely arbitrary architectures).

As a refresher, here's a two-layer model defined using `keras_model_sequential` (note that we're passing the expected shape of the input data to the first layer):

```
model <- keras_model_sequential() %>%
  layer_dense(units = 32, input_shape = c(784)) %>%
  layer_dense(units = 10, activation = "softmax")
```

And here's the same model defined using the functional API:

```
input_tensor <- layer_input(shape = c(784))

output_tensor <- input_tensor %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")
```

```
model <- keras_model(inputs = input_tensor, outputs = output_tensor)
```

With the functional API, you're manipulating the data tensors that the model processes and applying layers to this tensor as if they were functions.

Note

A detailed guide to what you can do with the functional API can be found in chapter 7. Until chapter 7, we'll only be using `keras_model_sequential` in our code examples.

Once your model architecture is defined, it doesn't matter whether you used `keras_model_sequential` or the functional API. All of the following steps are the same.

The learning process is configured in the compilation step, where you specify the optimizer and loss function(s) that the model should use, as well as the metrics you want to monitor during training. Here's an example with a single loss function, which is by far the most common case:

```
model %>% compile(
  optimizer = optimizer_rmsprop(lr = 0.0001),
  loss = "mse",
  metrics = c("accuracy")
)
```

Finally, the learning process consists of passing arrays of input data (and the corresponding target data) to the model via the `fit()` method, similar to what you'd do with other machine-learning libraries:

```
model %>% fit(input_tensor, target_tensor, batch_size = 128, epochs = 10)
```

Over the next few chapters, you'll build a solid intuition about what type of network architectures work for different kinds of problems, how to pick the right learning configuration, and how to tweak a model until it gives the results you want to see.

We'll look at three basic examples in sections 3.4, 3.5, and 3.6: a two-class classification example, a many-class classification example, and a regression example. All the code

examples in this book are available as open source notebooks: you can download them from the book's website at www.manning.com/books/deep-learning-with-r.

3.3. SETTING UP A DEEP-LEARNING WORKSTATION

Before you can get started developing deep-learning applications, you need to set up your workstation. It's highly recommended, although not strictly necessary, that you run deep-learning code on a modern NVIDIA GPU. Some applications—in particular, image processing with convolutional networks and sequence processing with recurrent neural 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 speed increase by a factor or 5 or 10 by using a modern GPU. If you don't want to install a GPU on your machine, you can alternatively consider running your experiments on an AWS EC2 GPU instance or on Google Cloud Platform. But note that cloud GPU instances can become expensive over time.

Whether you're running locally or in the cloud, it's better to be using a Unix workstation. Although it's technically possible to use Keras on Windows (all three Keras backends support Windows), we don't recommend it. If you're a Windows user, the simplest solution to get everything running is to set up an Ubuntu dual boot on your machine. It may seem like a hassle, but using Ubuntu will save you a lot of time and trouble in the long run. Unix is an operating system. Linux is a unix operating system.

Note that in order to use Keras, you need to install TensorFlow *or* CNTK *or* Theano (or all of them, if you want to be able to switch back and forth among the three backends). In this book, we'll focus on TensorFlow, with some light instructions relative to Theano. We won't cover CNTK.

3.3.1. Getting Keras running: two options

To get started in practice, we recommend one of the following two options:

- Use the official EC2 Deep Learning AMI (https://aws.amazon.com/amazon-ai/amis), and run Keras experiments within RStudio Server on EC2. You can find details on this and other cloud GPU options at https://tensorflow.rstudio.com/tools/cloud_gpu.
- Install everything from scratch on a local Unix workstation. Do this if you already have a high-end NVIDIA GPU. You can find details on setting up a local GPU workstation at https://tensorflow.rstudio.com/tools/local_gpu. NOPE

Let's take a closer look at some of the compromises involved in picking one option over the other.

3.3.2. Running deep-learning jobs in the cloud: pros and cons

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 get started without having to buy any additional hardware. If you're using RStudio Server, the experience of running in the cloud is no different from running locally. As of mid-2017, the cloud offering that makes it easiest to get started with deep learning is definitely AWS EC2.

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 weeks. EC2 instances are expensive: for example, the `p2.xlarge` instance, which won't provide you with much power, costs $0.90 per hour as of mid-2017. Meanwhile, a solid consumer-class GPU will cost you somewhere between $1,000 and $1,500—a price that has been fairly stable over time, even as the specs of these GPUs keep improving. If you're serious about deep learning, you should set up a local workstation with one or more GPUs.

In short, EC2 is a great way to get started. You could follow the code examples in this book entirely on an EC2 GPU instance. But if you're going to be a power user of deep learning, get your own GPUs.

3.3.3. What is the best GPU for deep learning?

If you're going to buy a GPU, which one should you choose? The first thing to note is that it must be an NVIDIA GPU. NVIDIA is the only graphics computing company that has invested heavily in deep learning so far, and modern deep-learning frameworks can only run on NVIDIA cards.

As of mid-2017, we recommend the NVIDIA TITAN Xp as the best card on the market for deep learning. For lower budgets, you may want to consider the GTX 1060. If you're reading these pages in 2018 or later, take the time to look online for fresher recommendations, because new models come out every year.

From this section onward, we'll assume that you have access to a machine with Keras and its dependencies installed—preferably with GPU support. Make sure you finish this step before you proceed. There is no shortage of tutorials on how to install Keras and common deep-learning dependencies.

We can now dive into practical Keras examples.

3.4. CLASSIFYING MOVIE REVIEWS: A BINARY CLASSIFICATION EXAMPLE

Two-class classification, or binary classification, may be the most widely applied kind of machine-learning problem. In this example, you'll learn to classify movie reviews as positive or negative, based on the text content of the reviews.

3.4.1. 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.

Why use separate training and test sets? Because you should never test a machine-learning model on the same data that you used to train it! Just because a model performs well on its training data doesn't mean it will perform well on data it has never seen; and what you care about is your model's performance on new data (because you already know the labels of your training data—obviously you don't need your model to predict those). For instance, it's possible that your model could end up merely *memorizing* a mapping between your training samples and their targets, which would be useless for the task of predicting targets for data the model has never seen before. We'll go over this point in much more detail in the next chapter.

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.

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 3.1. Loading the IMDB dataset

```
library(keras)

imdb <- dataset_imdb(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% imdb
```

Using the multi-assignment (%<-%) operator

The datasets built into Keras are all nested lists of training and test data. Here, we use the multi-assignment operator (`%<-%`) from the `zeallot` package to unpack the list into a set of distinct variables. This could equally be written as follows:

```
imdb <- dataset_imdb(num_words = 10000)
```

```
train_data <- imdb$train$x
train_labels <- imdb$train$y
test_data <- imdb$test$x
test_labels <- imdb$test$y
```

The multi-assignment version is preferable because it's more compact. The `%<-%` operator is automatically available whenever the R Keras package is loaded.

The argument `num_words = 10000` means you'll keep only the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows you to work with vector data of a manageable size.

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*:

```
> str(train_data[[1]])
int [1:218] 1 14 22 16 43 530 973 1622 1385 65 ...

> train_labels[[1]]
[1] 1
```

Because you're restricting yourself to the top 10,000 most frequent words, no word index will exceed 10,000:

```
> max(sapply(train_data, max))
[1] 9999
```

For kicks, here's how you can quickly decode one of these reviews back to English words:

```
word_index <- dataset_imdb_word_index()
reverse_word_index <- names(word_index)
names(reverse_word_index) <- word_index
decoded_review <- sapply(train_data[[1]], function(index) {
  word <- if (index >= 3) reverse_word_index[[as.character(index - 3)]]
  if (!is.null(word)) word else "?"
})
```

- *1* **word_index is a named list mapping words to an integer index.**

- **2 Reverses it, mapping integer indices to words**
- **3 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."**

3.4.2. Preparing the data

You can't feed lists of integers into a neural network. 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, word_indices)`, and then use as the first layer in your network a layer capable of handling such integer tensors (the "embedding" layer, which we'll cover in detail later in the book).
- One-hot encode your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence `[3, 5]` into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s. Then you could use as the first layer in your network a dense layer, capable of handling floating-point vector data.

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

Listing 3.2. Encoding the integer sequences into a binary matrix

```
vectorize_sequences <- function(sequences, dimension = 10000) {
  results <- matrix(0, nrow = length(sequences), ncol = dimension)      1
  for (i in 1:length(sequences))
    results[i, sequences[[i]]] <- 1                                     2
  results
}

x_train <- vectorize_sequences(train_data)
x_test <- vectorize_sequences(test_data)
```

- **1 Creates an all-zero matrix of shape (length(sequences), dimension)**
- **2 Sets specific indices of results[i] to 1s**

Here's what the samples look like now:

```
> str(x_train[1,])
 num [1:10000] 1 1 0 1 1 1 1 1 1 0 ...
```

You should also convert your labels from integer to numeric, which is straightforward:

```
y_train <- as.numeric(train_labels)
y_test <- as.numeric(test_labels)
```

Now the data is ready to be fed into a neural network.

3.4.3. Building your network

The input data is vectors, and the labels are scalars (1s and 0s): this is the easiest setup you'll ever encounter. A type of network that performs well on such a problem is a simple stack of fully connected (dense) layers with `relu` activations: `layer_dense(units = 16, activation = "relu")`.

The argument being passed to each dense layer (`16`) is the number of hidden units of the layer. A *hidden unit* is a dimension in the representation space of the layer. You may remember from chapter 2 that each such dense layer with a `relu` activation implements the following chain of tensor operations:

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

Having 16 hidden 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 network to have when learning internal representations." Having more hidden units (a higher-dimensional representation space) allows your network to learn more complex representations, but it makes the network 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).

There are two key architecture decisions to be made about such a stack of dense layers:

- How many layers to use
- How many hidden units to choose for each layer

How do we make the decisions?

In chapter 4, you'll learn formal principles to guide you in making these choices. For the time being, you'll have to trust us with the following architecture choice:

- Two intermediate layers with 16 hidden units each
- A third layer that will output the scalar prediction regarding the sentiment of the current review

The intermediate layers will use `relu` as their activation function, and the final layer will use 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 3.4), whereas a sigmoid "squashes" arbitrary values into the `[0, 1]` interval (see figure 3.5), outputting something that can be interpreted as a probability.

Figure 3.4. The rectified linear unit function

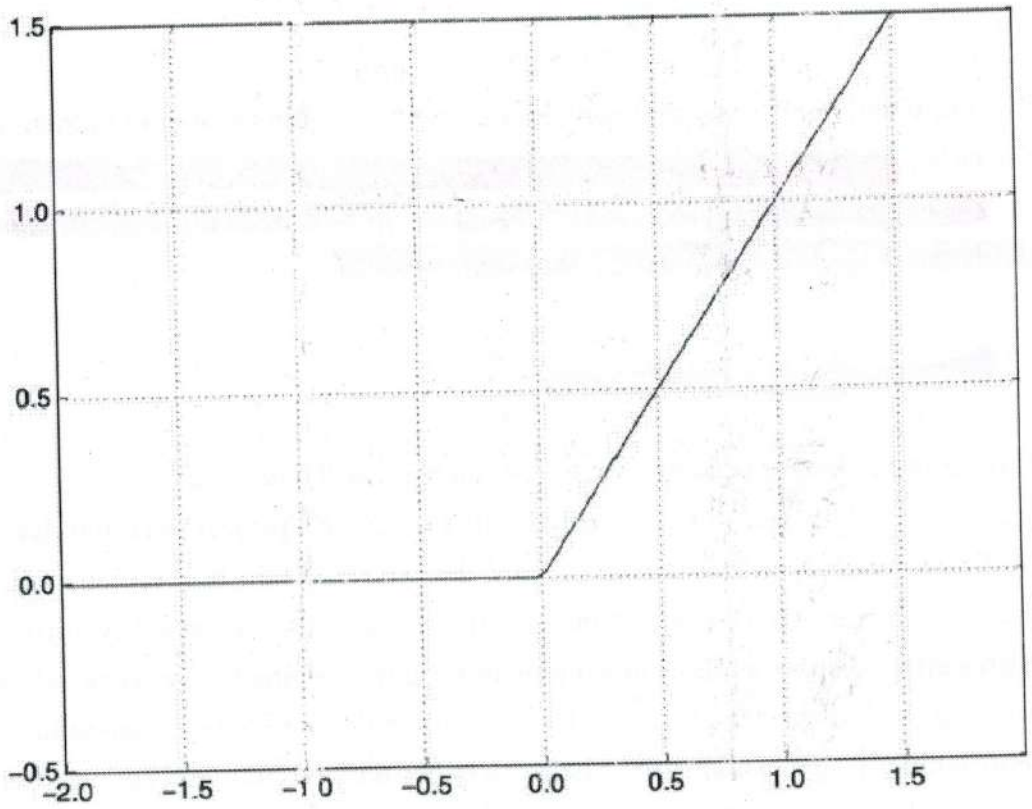


Figure 3.5. The sigmoid function

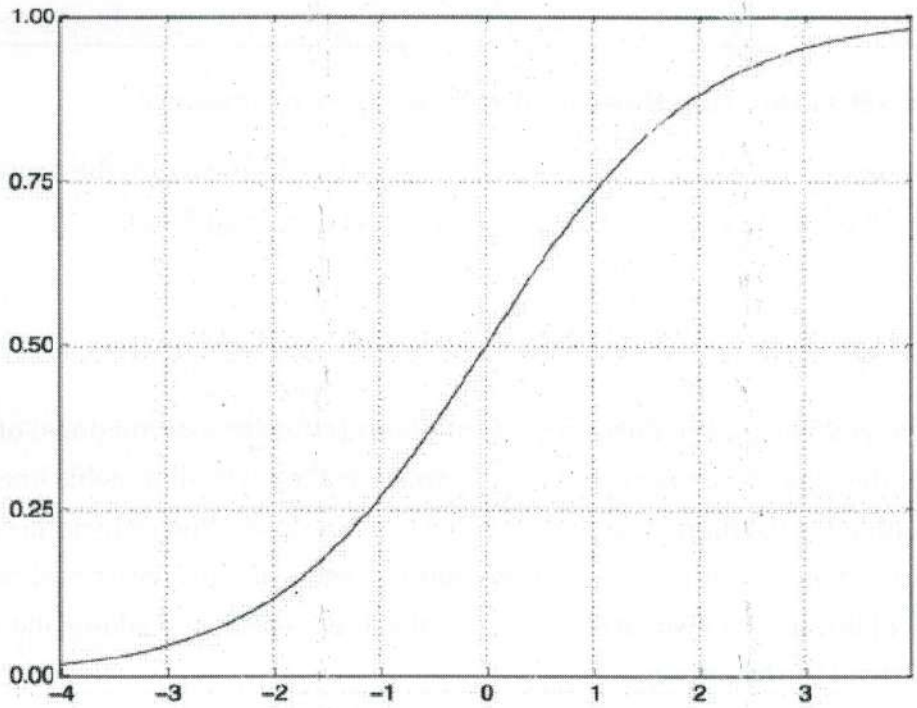
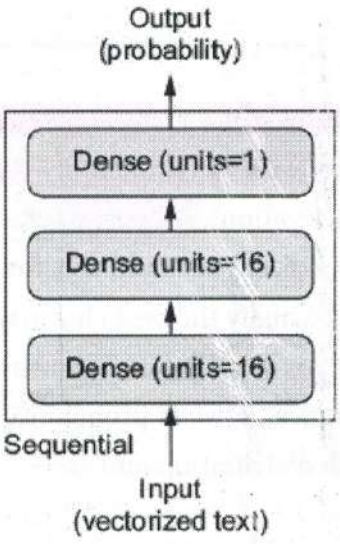


Figure 3.6 show what the network looks like. And listing 3.3 shows the Keras implementation, similar to the MNIST example you saw previously.

Figure 3.6. The three-layer network



Listing 3.3. The model definition

```
library(keras)

model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

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(W, input) + b
```

So the layer could only learn *linear transformations* (affine transformations) of the input data: the *hypothesis space* of the layer would be the set of all possible linear transformations of the input data into a 16-dimensional space. Such a hypothesis space is too restricted and wouldn't benefit from multiple layers of representations, because a deep stack of linear layers would still implement a linear operation: adding more layers wouldn't extend the hypothesis space.

In order to get access to a much richer hypothesis space that would benefit from deep representations, you need a non-linearity, or activation function. `relu` is the most popular activation function in deep learning, but there are many other candidates, which all come with similarly strange names: `prelu`, `elu`, and so on.

[→Need to read more about making these decisions

Finally, you need to choose a loss function and an optimizer. Because you're facing a binary classification problem and the output of your network is a probability (you end your network with a single-unit layer with a sigmoid activation), it's best to use the `binary_crossentropy` loss. It isn't the only viable choice: you could use, for instance, `mean_squared_error`. But crossentropy is usually the best choice when you're dealing with models that output probabilities. *Crossentropy* is a quantity from the field of Information Theory that measures the distance between probability distributions or, in this case, between the ground-truth distribution and your predictions.

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

Listing 3.4. Compiling the model

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
```

```
  metrics = c("accuracy")
)
```

You're passing your optimizer, loss function, and metrics as strings, which is possible because `rmsprop`, `binary_crossentropy`, and `accuracy` are packaged as part of Keras. Sometimes you may want to configure the parameters of your optimizer or pass a custom loss function or metric function. The former can be done by passing an optimizer instance as the `optimizer` argument, as shown in listing 3.5; the latter can be done by passing function objects as the `loss` and/or `metrics` arguments, as shown in listing 3.6.

Listing 3.5. Configuring the optimizer

```
model %>% compile(
  optimizer = optimizer_rmsprop(lr=0.001),
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

Listing 3.6. Using custom losses and metrics

```
model %>% compile(
  optimizer = optimizer_rmsprop(lr = 0.001),
  loss = loss_binary_crossentropy,
  metrics = metric_binary_accuracy
)
```

3.4.4. Validating your approach

In order to monitor during training the accuracy of the model on data it has never seen before, you'll create a validation set by setting apart 10,000 samples from the original training data.

Listing 3.7. Setting aside a validation set

```
val_indices <- 1:10000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]
y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
```

You'll now train the model for 20 epochs (20 iterations over all samples in the `x_train` and `y_train` tensors), in mini-batches of 512 samples. At the same time,

you'll monitor loss and accuracy on the 10,000 samples that you set apart. You do so by passing the validation data as the `validation_data` argument.

Listing 3.8. Training your model

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(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 `fit()` returns a `history` object. Let's take a look at it:

```
> str(history)
List of 2
 $ params :List of 8
  ..$ metrics          : chr [1:4] "loss" "acc" "val_loss" "val_acc"
  ..$ epochs           : int 20
  ..$ steps            : NULL
  ..$ do_validation    : logi TRUE
  ..$ samples          : int 15000
  ..$ batch_size       : int 512
  ..$ verbose          : int 1
  ..$ validation_samples: int 10000
 $ metrics:List of 4
  ..$ acc     : num [1:20] 0.783 0.896 0.925 0.941 0.952 ...
  ..$ loss    : num [1:20] 0.532 0.331 0.24 0.186 0.153 ...
  ..$ val_acc : num [1:20] 0.832 0.882 0.886 0.888 0.888 ...
  ..$ val_loss: num [1:20] 0.432 0.323 0.292 0.278 0.278 ...
 - attr(*, "class")= chr "keras_training_history"
```

The `history` object includes parameters used to fit the model (`history-$params`) as well as data for each of the metrics being monitored (`history$metrics`).

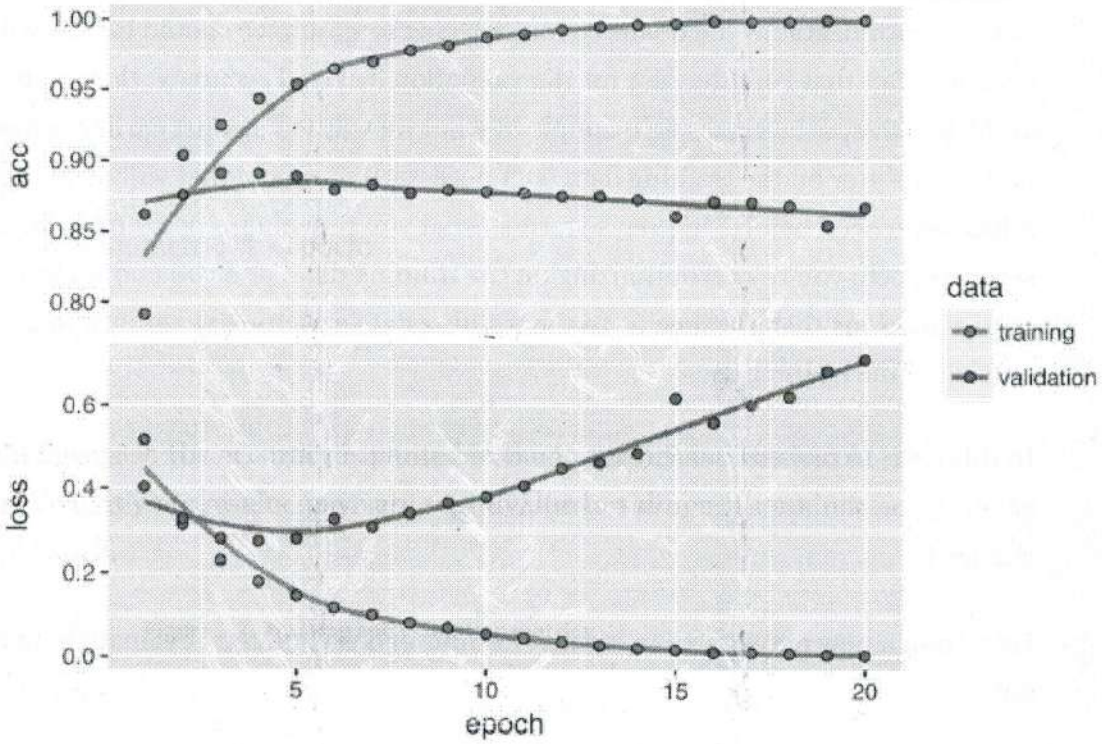
The `history` object has a `plot()` method that enables you to visualize training and

validation metrics by epoch:

```
plot(history)
```

In figure 3.7, the accuracy is plotted in the top panel and the loss in the bottom panel. Note that your results may vary slightly due to a different random initialization of your network.

Figure 3.7. Training and validation metrics



Training history with the plot() method

The `plot()` method for training history objects uses `ggplot2` for plotting if it's available (if it isn't, base graphics are used). The plot includes all specified metrics as well as the loss; it draws a smoothing line if there are 10 or more epochs. You can customize all of this behavior via various arguments to the `plot()` method.

If you want to create a custom visualization, call the `as.data.frame()` method on the history to obtain a data frame with factors for each metric as well as training versus validation:

```
> history_df <- as.data.frame(history)
> str(history_df)
```

```
'data.frame': 120 obs. of 4 variables:
 $epoch  : int 1 2 3 4 5 6 7 8 9 10...
 $ value : num 0.87 0.941 0.954 0.962 0.965 ...
 $ metric: Factor w/ 2 levels "acc","loss": 1 1 1 1 1 1 1 1 1 1...
 $ data  : Factor w/ 2 levels "training","validation": 1 1 1 1 1 1 1 1 1
 1...
```

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 a 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 second 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 three epochs. In general, you can use a range of techniques to mitigate overfitting, which we'll cover in chapter 4.

Let's train a new network from scratch for four epochs and then evaluate it on the test data.

Listing 3.9. Retraining a model from scratch

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("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
$loss
[1] 0.2900235

$acc
[1] 0.88512
```

This fairly naive approach achieves an accuracy of 88%. With state-of-the-art approaches, you should be able to get close to 95%.

3.4.5. Using a trained network to generate predictions on new data

After having trained a network, you'll want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the `predict` method:

```
> model %>% predict(x_test[1:10,])
 [1,] 0.92306918
 [2,] 0.84061098
 [3,] 0.99952853
 [4,] 0.67913240
 [5,] 0.73874789
 [6,] 0.23108074
 [7,] 0.01230567
 [8,] 0.04898361
 [9,] 0.99017477
[10,] 0.72034937
```

As you can see, the network is confident for some samples (0.99 or more, or 0.01 or less) but less confident for others (0.7, 0.2).

3.4.6. 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 hidden layers. Try using one or three hidden layers, and see how doing so affects validation and test accuracy.
- Try using layers with more hidden units or fewer hidden 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`.

3.4.7. 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 network should end with a dense layer with one unit and a `sigmoid` activation: the output of your network 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.

3.5. CLASSIFYING NEWSWIRES: A MULTICLASS CLASSIFICATION EXAMPLE

In the previous section, you saw how to classify vector inputs into two mutually exclusive classes using a densely connected neural network. But what happens when you have more than two classes?

In this section, you'll build a network to classify Reuters newswires into 46 mutually exclusive topics. Because you have many classes, this problem is an instance of *multiclass classification*; and because each data point should be classified into only one category, the problem is more specifically an instance of *single-label, multiclass classification*. If each data point could belong to multiple categories (in this case, topics), you'd be facing a *multilabel, multiclass classification* problem.

3.5.1. 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 3.10. Loading the Reuters dataset

```
library(keras)

reuters <- dataset_reuters(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% reuters
```

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:

```
> length(train_data)
[1] 8982
> length(test_data)
[1] 2246
```

As with the IMDB reviews, each example is a list of integers (word indices):

```
> train_data[[1]]
 [1]    1    2    2    8   43   10  447    5   25  207  270    5 3095  111
[16]  369  186   90   67    7   89    5   19  102    6   19  124   15   90
[31]   84   22  482   26    7   48    4   49    8  864   39  209  154    6
[46]    6   83   11   15   22  155   11   15    7   48    9 4579 1005  504
[61]  258    6  272   11   15   22  134   44   11   15   16    8  197 1245
[76]   67   52   29  209   30   32  132    6  109   15   17   12
```

Here's how you can decode it back to words, in case you're curious.

Listing 3.11. Decoding newswires back to text

```
word_index <- dataset_reuters_word_index()
reverse_word_index <- names(word_index)
names(reverse_word_index) <- word_index
decoded_newswire <- sapply(train_data[[1]], function(index) {
  word <- if (index >= 3) reverse_word_index[[as.character(index - 3)]]
  if (!is.null(word)) word else "?"
})
```

- ***1*** **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[[1]]
3
```

3.5.2. Preparing the data

You can vectorize the data with the exact same code as in the previous example.

Listing 3.12. Encoding the data

```
vectorize_sequences <- function(sequences, dimension = 10000) {
  results <- matrix(0, nrow = length(sequences), ncol = dimension)
  for (i in 1:length(sequences))
    results[i, sequences[[i]]] <- 1
  results
}

x_train <- vectorize_sequences(train_data)                    1
x_test <- vectorize_sequences(test_data)                      2
```

- *1* **Vectorized training data**
- *2* **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*. For a more detailed explanation of one-hot encoding, see section 6.1. In this case, one-hot encoding of the labels consists of embedding each label as an all-zero vector with a 1 in the place of the label index. Here's an example:

```
to_one_hot <- function(labels, dimension = 46) {
  results <- matrix(0, nrow = length(labels), ncol = dimension)
  for (i in 1:length(labels))
    results[i, labels[[i]] + 1] <- 1
  results
}

one_hot_train_labels <- to_one_hot(train_labels)              1
one_hot_test_labels <- to_one_hot(test_labels)                2
```

- *1* **Vectorized training labels**

- **2 Vectorized test labels**

Note that there is a built-in way to do this in Keras, which you've already seen in action in the MNIST example:

```
one_hot_train_labels <- to_categorical(train_labels)
one_hot_test_labels <- to_categorical(test_labels)
```

3.5.3. Building your network

This topic-classification problem looks similar to the previous movie-review classification problem: in both cases, you're trying to classify short snippets of text. But there is a new constraint here: the number of output classes has gone from 2 to 46. The dimensionality of the output space is much larger.

In a stack of dense layers like that you've been using, each layer can only access information present in the output of the previous layer. If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers: each layer can potentially become an information bottleneck. In the movie-review example, you used 16-dimensional intermediate layers, but a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information.

For this reason you'll use larger layers. Let's go with 64 units.

Listing 3.13. Model definition

```
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 46, activation = "softmax")
```

There are two other things you should note about this architecture:

- You end the network with a dense layer of size 46. This means for each input sample, the network will output a 46-dimensional vector. Each entry in this vector (each dimension) will encode a different output class.
- The last layer uses a `softmax` activation. You saw this pattern in the MNIST example. It means the network will output a *probability distribution* over the 46 different output classes—for every input sample, the network 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 network and the true distribution of the labels. By minimizing the distance between these two distributions, you train the network to output something as close as possible to the true labels.

Listing 3.14. Compiling the model

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
```

3.5.4. Validating your approach

Let's set apart 1,000 samples in the training data to use as a validation set.

Listing 3.15. Setting aside a validation set

```
val_indices <- 1:1000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- one_hot_train_labels[val_indices,]
partial_y_train = one_hot_train_labels[-val_indices,]
```

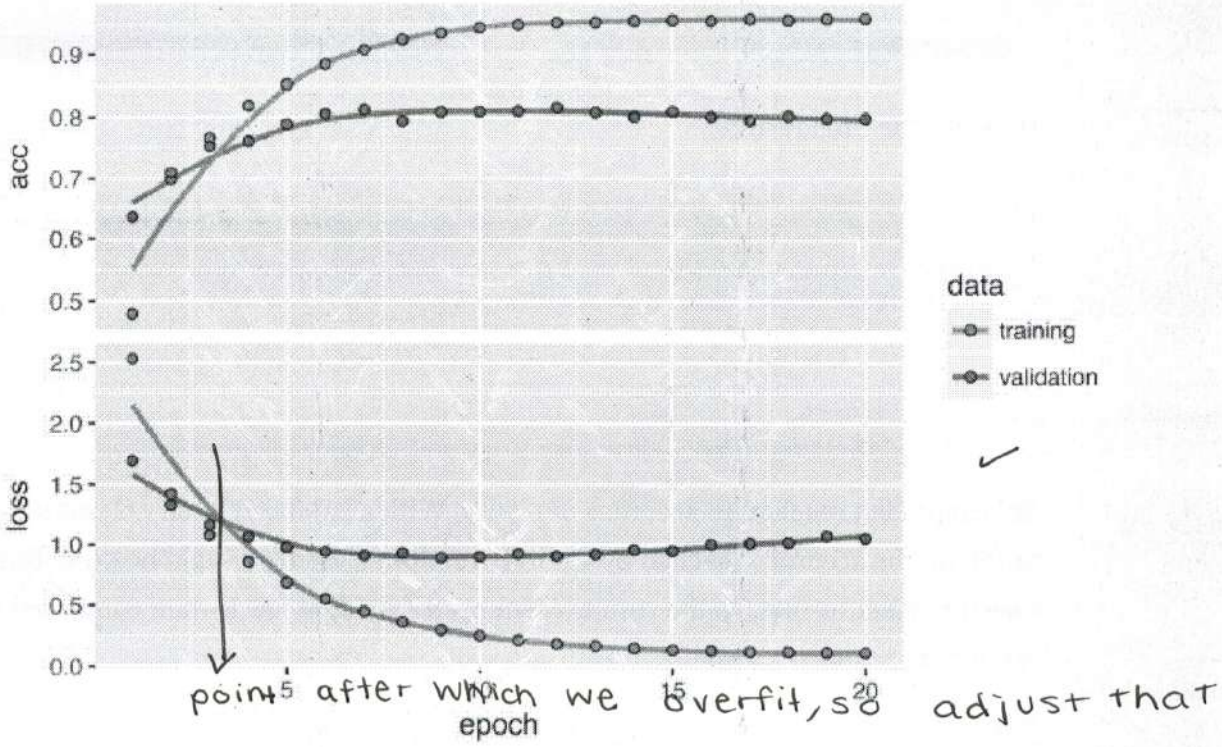
Now, let's train the network for 20 epochs.

Listing 3.16. Training the model

```
history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
```

And finally, let's display its loss and accuracy curves (see figure 3.8).

Figure 3.8. Training and validation metrics



Listing 3.17. Plotting the training and validation metrics

```
plot(history)
```

The network begins to overfit after nine epochs. Let's train a new network from scratch for nine epochs and then evaluate it on the test set.

Listing 3.18. Retraining a model from scratch

```
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 46, activation = "softmax")

model %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 9,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

results <- model %>% evaluate(x_test, one_hot_test_labels)
```

Here are the final results:

```
> results
$loss
[1] 0.9834202

$acc
[1] 0.7898486
```

This approach reaches an accuracy of ~ 79%. With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%. But in this case it's closer to 18%, so the results seem pretty good, at least when compared to a random baseline:

```
> test_labels_copy <- test_labels
> test_labels_copy <- sample(test_labels_copy)
> length(which(test_labels == test_labels_copy)) / length(test_labels)
[1] 0.1821015
```

3.5.5. Generating predictions on new data

You can verify that the `predict` method of the model instance returns a probability distribution over all 46 topics. Let's generate topic predictions for all of the test data.

Listing 3.19. Generating predictions for new data

```
predictions <- model %>% predict(x_test)
```

Each entry in `predictions` is a vector of length 46:

```
> dim(predictions)
[1] 2246   46
```

The coefficients in this vector sum to 1:

```
> sum(predictions[1,])
[1] 1
```

The largest entry is the predicted class—the class with the highest probability:

```
> which.max(predictions[1,])
[1] 4
```

3.5.6. A different way to handle the labels and the loss

We mentioned earlier that another way to encode the labels would be to preserve their integer values. The only thing this approach would change is the choice of the loss function. The previous loss function, `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 = c("accuracy")
)
```

This new loss function is still mathematically the same as `categorical_crossentropy`; it just has a different interface.

3.5.7. 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 significantly fewer than 46 hidden units. Now let's see what happens when you introduce an information bottleneck by having intermediate layers that are significantly less than 46-dimensional: for example, 4-dimensional.

Listing 3.20. A model with an information bottleneck

```
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 4, activation = "relu") %>%
  layer_dense(units = 46, activation = "softmax")

model %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 128,
  validation_data = list(x_val, y_val)
```

```
)
```

The network now peaks at ~71% validation accuracy, an 8% absolute drop. This drop is mostly due to the fact that you're trying to compress a lot of information (enough information to recover the separation hyperplanes of 46 classes) into an intermediate space that is too low-dimensional. The network is able to cram *most* of the necessary information into these eight-dimensional representations, but not all of it.

3.5.8. Further experiments

- Try using larger or smaller layers: 32 units, 128 units, and so on.
- You used two hidden layers. Now try using a single hidden layer, or three hidden layers.

3.5.9. Wrapping up

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

- If you're trying to classify data points among N classes, your network should end with a dense layer of size N.
- In a single-label, multiclass classification problem, your network 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 network 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 network due to intermediate layers that are too small.

3.6. 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* with the algorithm *logistic regression*. Confusingly, logistic regression isn't a regression algorithm—it's a classification algorithm.

3.6.1. The Boston Housing Price dataset

You'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 you'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 3.21. Loading the Boston housing dataset

```
library(keras)

dataset <- dataset_boston_housing()
c(c(train_data, train_targets), c(test_data, test_targets)) %<-% dataset
```

Let's look at the data:

```
> str(train_data)
 num [1:404, 1:13] 1.2325 0.0218 4.8982 0.0396 3.6931 ...
> str(test_data)
 num [1:102, 1:13] 18.0846 0.1233 0.055 1.2735 0.0715 ...
```

As you can see, you 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:

```
> str(train_targets)
num [1:404(1d)] 15.2 42.3 50 21.1 17.7 18.5 11.3 15.6 15.6 14.4 ...
```

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.

3.6.2. Preparing the data

It would be problematic to feed into a neural network values that all take wildly different ranges. The network might be able to automatically adapt to such heterogeneous data, but it would definitely make learning more difficult. A widespread best practice to deal with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), you subtract the mean of the feature and divide by the standard deviation, so that the feature is centered around 0 and has a unit standard deviation. This is easily done in R using the `scale()` function.

Listing 3.22. Normalizing the data

```
mean <- apply(train_data, 2, mean)                                    1
std <- apply(train_data, 2, sd)
train_data <- scale(train_data, center = mean, scale = std)           2
test_data <- scale(test_data, center = mean, scale = std)
```

- **1 Calculates the mean and standard deviation on the training data**
- **2 Scales the training and test data using the mean and standard deviation from the training data**

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

3.6.3. Building your network

Because so few samples are available, you'll use a very small network with two hidden layers, each with 64 units. In general, the less training data you have, the worse overfitting will be, and using a small network is one way to mitigate overfitting.

Listing 3.23. Model definition

```
build_model <- function() {                                    1
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(train_data)[[2]]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)
  model %>% compile(
    optimizer = "rmsprop",
    loss = "mse",
    metrics = c("mae")
  )
}
```

- ***1* Because you'll need to instantiate the same model multiple times, you use a function to construct it.**

The network 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 network could only learn to predict values between 0 and 1. Here, because the last layer is purely linear, the network is free to learn to predict values in any range.

Note that you compile the network 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.

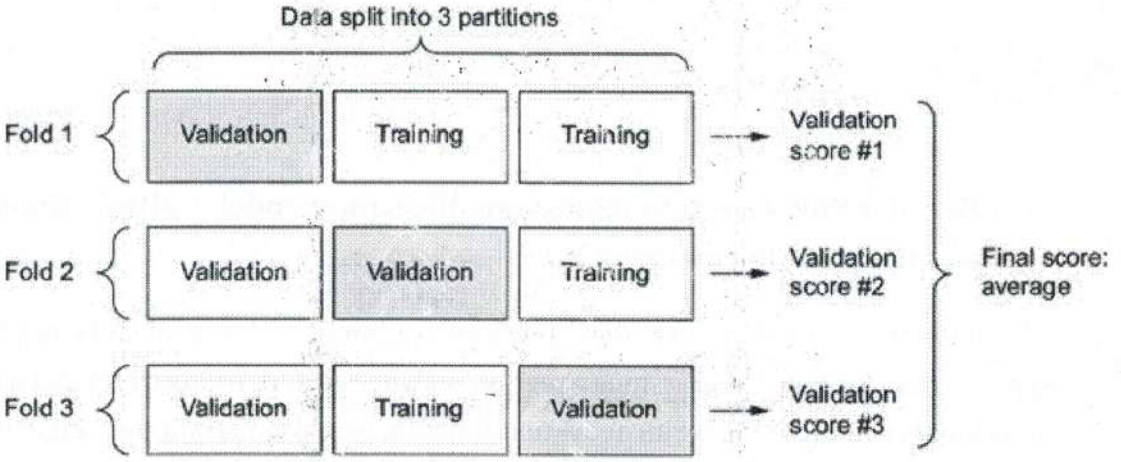
You're also monitoring a new metric during training: *mean absolute error* (MAE). It's the absolute value of the difference between the predictions and the targets. For instance, an MAE of 0.5 on this problem would mean your predictions are off by $500 on average.

3.6.4. Validating your approach using K-fold validation

To evaluate your network while you keep adjusting its parameters (such as the number of epochs used for training), you could split the data into a training set and a validation set, as you did in the previous examples. But because you have so few data points, the validation set would end up being very small (for instance, about 100 examples). As a consequence, the validation scores might change a lot depending on which data points you chose to use for validation and which you chose for training: the validation scores might have a high *variance* with regard to the validation split. This would prevent you from reliably evaluating your model.

The best practice in such situations is to use *K-fold* cross-validation (see figure 3.9). It consists of splitting the available data into K partitions (typically $K = 4$ or 5), instantiating K identical models, and training each one on $K - 1$ partitions while evaluating on the remaining partition. The validation score for the model used is then the average of the K validation scores obtained. In terms of code, this is straightforward.

Figure 3.9. 3-fold cross-validation



Listing 3.24. K-fold validation

```
k <- 4
indices <- sample(1:nrow(train_data))
folds <- cut(indices, breaks = k, labels = FALSE)

num_epochs <- 100
all_scores <- c()
for (i in 1:k) {
  cat("processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE)                    1
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]
  partial_train_data <- train_data[-val_indices,]                     2
  partial_train_targets <- train_targets[-val_indices]

  model <- build_model()                                              3

  model %>% fit(partial_train_data, partial_train_targets,            4
                epochs = num_epochs, batch_size = 1, verbose = 0)

  results <- model %>% evaluate(val_data, val_targets, verbose = 0)   5
  all_scores <- c(all_scores, results$mean_absolute_error)
}
```

- **1 Prepares the validation data: data from partition #k**
- **2 Prepares the training data: data from all other partitions**
- **3 Builds the Keras model (already compiled)**
- **4 Trains the model (in silent mode, verbose = 0)**
- **5 Evaluates the model on the validation data**

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

```
> all_scores
[1] 2.065541 2.270200 2.838082 2.381782
> mean(all_scores)
[1] 2.388901
```

The different runs do indeed show rather different validation scores, from 2.1 to 2.8. The average (2.4) is a much more reliable metric than any single score—that's the entire point of K-fold cross-validation. In this case, you're off by $2,400 on average, which is significant considering that the prices range from $10,000 to $50,000.

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

Listing 3.25. Saving the validation logs at each fold

```
num_epochs <- 500
all_mae_histories <- NULL
for (i in 1:k) {
  cat("processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE)                    1
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,]                     2
  partial_train_targets <- train_targets[-val_indices]

  model <- build_model()                                              3

  history <- model %>% fit(                                           4
    partial_train_data, partial_train_targets,
    validation_data = list(val_data, val_targets),
    epochs = num_epochs, batch_size = 1, verbose = 0
  )
  mae_history <- history$metrics$val_mean_absolute_error
  all_mae_histories <- rbind(all_mae_histories, mae_history)
```

```
}
```

- *1* **Prepares the validation data: data from partition #k**
- *2* **Prepares the training data: data from all other partitions**
- *3* **Builds the Keras model (already compiled)**
- *4* **Trains the model (in silent mode, verbose=0)**

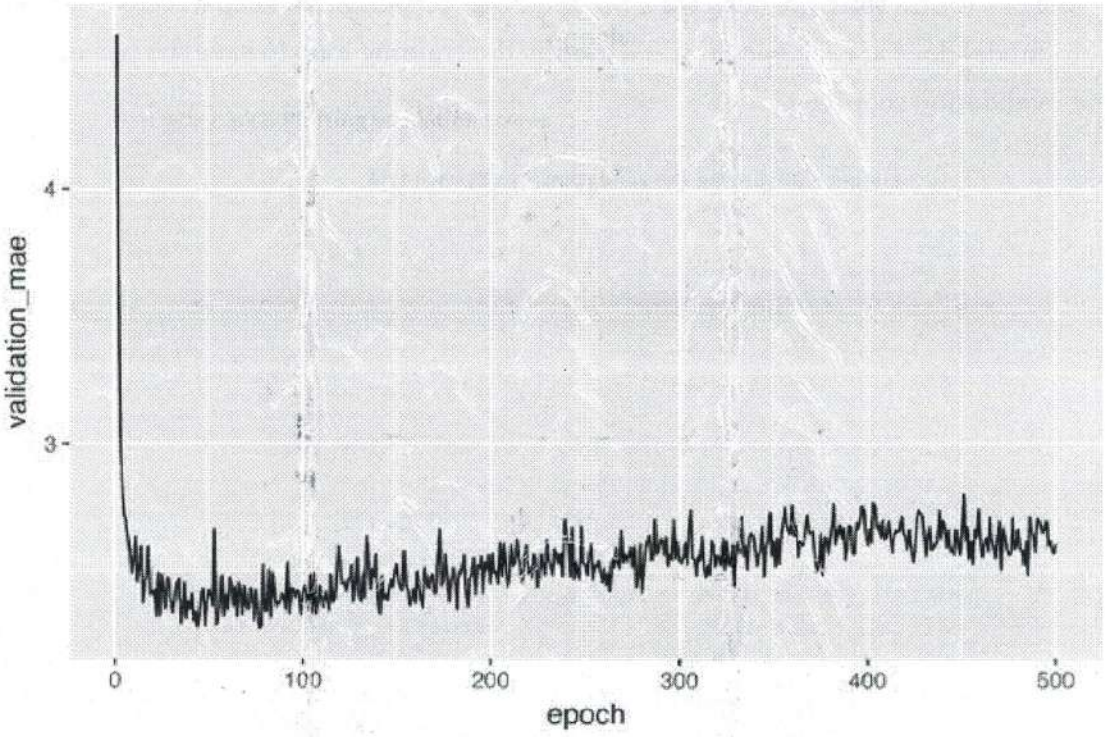
You can then compute the average of the per-epoch MAE scores for all folds.

Listing 3.26. Building the history of successive mean K-fold validation scores

```
average_mae_history <- data.frame(
  epoch = seq(1:ncol(all_mae_histories)),
  validation_mae = apply(all_mae_histories, 2, mean)
)
```

Let's plot this; see figure 3.10.

Figure 3.10. Validation MAE by epoch

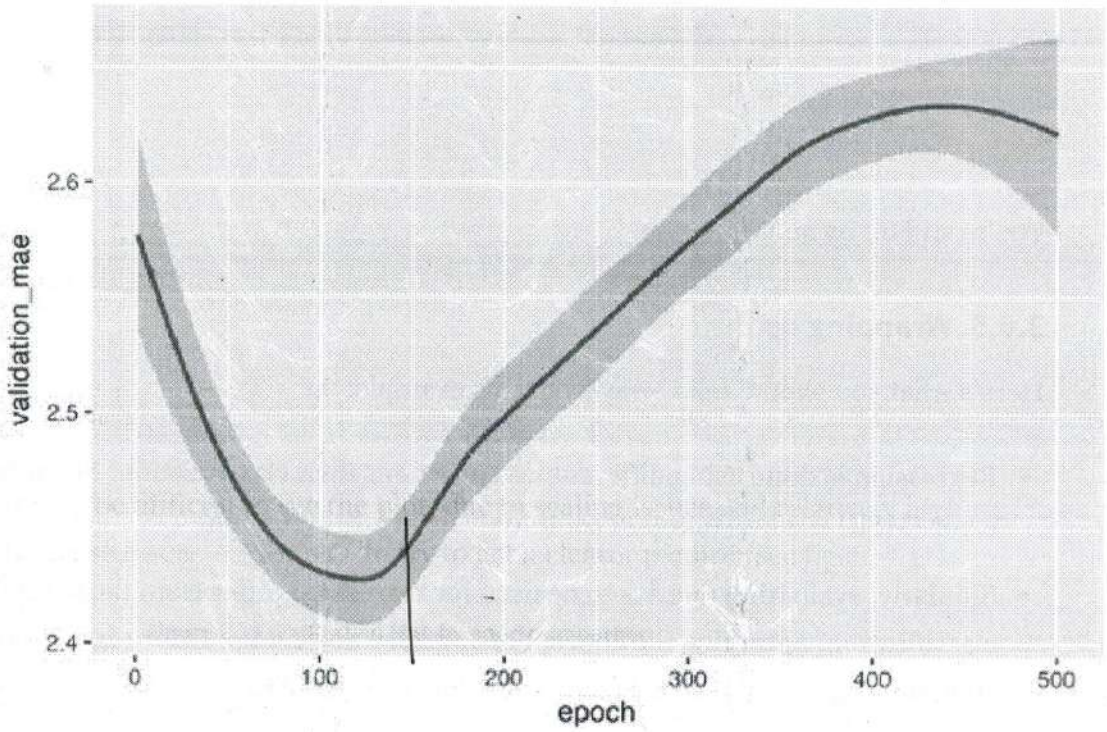


Listing 3.27. Plotting validation scores

```
library(ggplot2)
ggplot(average_mae_history, aes(x = epoch, y = validation_mae)) + geom_lin
```

It may be difficult to see the plot, due to scaling issues and relatively high variance. Let's use `geom_smooth()` to try to get a clearer picture (see figure 3.11).

Figure 3.11. Validation MAE by epoch: smoothed



Listing 3.28. Plotting validation scores with `geom_smooth()`

```
ggplot(average_mae_history, aes(x = epoch, y = validation_mae)) + geom_smo
```

According to this plot, validation MAE stops improving significantly after 125 epochs. Past that point, you start overfitting.

Once you're finished tuning other parameters of the model (in addition to the number of epochs, you could also adjust the size of the hidden 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 3.29. Training the final model

```
model <- build_model()
model %>% fit(train_data, train_targets,                    1
          epochs = 80, batch_size = 16, verbose = 0)
result <- model %>% evaluate(test_data, test_targets)
```

- ***1*** **Trains the model on the entirety of the data**

Here's the final result:

```
> result
$loss
[1] 15.58299

$mean_absolute_error
[1] 2.54131
```

You're still off by about $2,540.

3.6.5. Wrapping up

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

- Regression is done using different loss functions than 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 network with few hidden layers (typically only one or two), in order to avoid severe overfitting.

3.7. SUMMARY

- You're now able to handle the most common kinds of machine-learning tasks on vector data: binary classification, multiclass classification, and scalar regression. The "Wrapping up" sections earlier in the chapter summarize the important points you've learned regarding these types of tasks.
- 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 network with only one or two hidden 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.
- Regression uses different loss functions and different evaluation metrics than classification.
- When you're working with little data, K-fold validation can help reliably evaluate your model. (prevents overfitting, which tends to happen more on smaller data sets)

Chapter 4. Fundamentals of machine learning

This chapter covers

- Forms of machine learning beyond classification and regression
- Formal evaluation procedures for machine-learning models
- Preparing data for deep learning
- Feature engineering
- Tackling overfitting
- The universal workflow for approaching machine-learning problems

After the three practical examples in chapter 3, 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 into a solid conceptual framework for attacking and solving deep-learning problems. In section 4.5, we'll consolidate all of these concepts—model evaluation, data preprocessing and feature engineering, and tackling overfitting—into a detailed seven-step workflow for tackling any machine-learning task.

4.1. FOUR BRANCHES OF MACHINE LEARNING

In our previous examples, you've become familiar with three specific types of machine-learning problems: binary classification, multiclass classification, and scalar regression. All three are instances of *supervised learning*, where the goal is to learn the relationship between training inputs and training targets.

Supervised learning is just the tip of the iceberg—machine learning is a vast field with a complex subfield taxonomy. Machine-learning algorithms generally fall into four broad categories, described in the following sections.

4.1.1. Supervised learning

This is by far the most common case It consists of learning to map input data to known targets (also called *annotations*), given a set of examples (often annotated by humans). All four examples you've encountered in this book so far were canonical examples of supervised learning. Generally, almost all applications of deep learning that are in the spotlight these days belong in this category, such as optical character recognition, speech recognition, image classification, and language translation.

Although supervised learning mostly consists of classification and regression, there are more exotic variants as well, including the following (with examples):

- *Sequence generation*—Given a picture, predict a caption describing it. Sequence generation can sometimes be reformulated as a series of classification problems (such as repeatedly predicting a word or token in a sequence).
- *Syntax tree prediction*—Given a sentence, predict its decomposition into a syntax tree.
- *Object detection*—Given a picture, draw a bounding box around certain objects inside the picture. This can also be expressed as a classification problem (given many candidate bounding boxes, classify the contents of each one) or as a joint classification and regression problem, where the bounding-box coordinates are predicted via vector regression.
- *Image segmentation*—Given a picture, draw a pixel-level mask on a specific object.

4.1.2. Unsupervised learning

This branch of machine learning consists of finding interesting transformations of the input data without the help of any targets, for the purposes of data visualization, data compression, or data denoising, or to better understand the correlations present in the data at hand. Unsupervised learning is the bread and butter of data analytics, and it's often a necessary step in better understanding a dataset before attempting to solve a supervised-learning problem. *Dimensionality reduction* and *clustering* are well-known categories of unsupervised learning.

4.1.3. Self-supervised learning

This is a specific instance of supervised learning, but it's different enough that it deserves its own category. Self-supervised learning is supervised learning without human-annotated labels—you can think of it as supervised learning without any humans in the loop. There are still labels involved (because the learning has to be supervised by something), but they're generated from the input data, typically using a heuristic algorithm.

Chapter 4. Fundamentals of machine learning

This chapter covers

- Forms of machine learning beyond classification and regression
- Formal evaluation procedures for machine-learning models
- Preparing data for deep learning
- Feature engineering
- Tackling overfitting
- The universal workflow for approaching machine-learning problems

After the three practical examples in chapter 3, 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 into a solid conceptual framework for attacking and solving deep-learning problems. In section 4.5, we'll consolidate all of these concepts—model evaluation, data preprocessing and feature engineering, and tackling overfitting—into a detailed seven-step workflow for tackling any machine-learning task.

4.1. FOUR BRANCHES OF MACHINE LEARNING

In our previous examples, you've become familiar with three specific types of machine-learning problems: binary classification, multiclass classification, and scalar regression. All three are instances of *supervised learning*, where the goal is to learn the relationship between training inputs and training targets.

Supervised learning is just the tip of the iceberg—machine learning is a vast field with a complex subfield taxonomy. Machine-learning algorithms generally fall into four broad categories, described in the following sections.

4.1.1. Supervised learning

This is by far the most common case It consists of learning to map input data to known targets (also called *annotations*), given a set of examples (often annotated by humans). All four examples you've encountered in this book so far were canonical examples of supervised learning. Generally, almost all applications of deep learning that are in the spotlight these days belong in this category, such as optical character recognition, speech recognition, image classification, and language translation.

Although supervised learning mostly consists of classification and regression, there are more exotic variants as well, including the following (with examples):

- *Sequence generation*—Given a picture, predict a caption describing it. Sequence generation can sometimes be reformulated as a series of classification problems (such as repeatedly predicting a word or token in a sequence).
- *Syntax tree prediction*—Given a sentence, predict its decomposition into a syntax tree.
- *Object detection*—Given a picture, draw a bounding box around certain objects inside the picture. This can also be expressed as a classification problem (given many candidate bounding boxes, classify the contents of each one) or as a joint classification and regression problem, where the bounding-box coordinates are predicted via vector regression.
- *Image segmentation*—Given a picture, draw a pixel-level mask on a specific object.

4.1.2. Unsupervised learning

This branch of machine learning consists of finding interesting transformations of the input data without the help of any targets, for the purposes of data visualization, data compression, or data de noising, or to better understand the correlations present in the data at hand. Unsupervised learning is the bread and butter of data analytics, and it's often a necessary step in better understanding a dataset before attempting to solve a supervised-learning problem. *Dimensionality reduction* and *clustering* are well-known categories of unsupervised learning.

4.1.3. Self-supervised learning

This is a specific instance of supervised learning, but it's different enough that it deserves its own category. Self-supervised learning is supervised learning without human-annotated labels—you can think of it as supervised learning without any humans in the loop. There are still labels involved (because the learning has to be supervised by something), but they're generated from the input data, typically using a heuristic algorithm.

For instance, *autoencoders* are a well-known example of self-supervised learning, where the generated targets are the input, unmodified. In the same way, trying to predict the next frame in a video, given past frames, or the next word in a text, given previous words, are instances of self-supervised learning (*temporally supervised learning*, in this case: supervision comes from future input data). Note that the distinction between supervised, self-supervised, and unsupervised learning can be blurry sometimes—these categories are more of a continuum without solid borders. Self-supervised learning can be reinterpreted as either supervised or unsupervised learning, depending on whether you pay attention to the learning mechanism or to the context of its application.

Note

In this book, we'll focus specifically on supervised learning, because it's by far the dominant form of deep learning today, with a wide range of industry applications. We'll also take a briefer look at self-supervised learning in later chapters.

4.1.4. Reinforcement learning

Long overlooked, this branch of machine learning recently started to get a lot of attention after Google DeepMind successfully applied it to learning to play Atari games (and, later, learning to play Go at the highest level). In reinforcement learning, an *agent* receives information about its environment and learns to choose actions that will maximize some reward. For instance, a neural network that "looks" at a video-game screen and outputs game actions in order to maximize its score can be trained via reinforcement learning.

Currently, reinforcement learning is mostly a research area and hasn't yet had significant practical successes beyond games. In time, however, we expect to see reinforcement learning take over an increasingly large range of real-world applications: self-driving cars, robotics, resource management, education, and so on. It's an idea whose time has come, or will come soon.

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.

4.2. EVALUATING MACHINE-LEARNING MODELS

In the three examples presented in chapter 3, 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, all three models began to *overfit*. That is, their performance on never-before-seen data started stalling (or worsening) compared to their performance on the training data—which always improves as training progresses.

In machine learning, the goal is to achieve models that *generalize*—that perform well on never-before-seen data—and overfitting is the central obstacle. You can only control that which you can observe, so it's crucial to be able to reliably measure the generalization power of your model. The following sections look at strategies for mitigating overfitting and maximizing generalization. In this section, we'll focus on how to measure generalization: how to evaluate machine-learning models.

4.2.1. 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.

You may ask, why not have two sets: a training set and a test set? You'd train on the training data and evaluate on the test data. Much simpler!

The reason is that developing a model always involves tuning its configuration: for example, choosing the number of layers or the size of the layers (called the *hyperparameters* of the model, to distinguish them from the *parameters*, which are the network's weights). You do this tuning by using as a feedback signal the performance of the model on the validation data. In essence, this tuning is a form of *learning*: a search for a good configuration in some parameter space. As a result, tuning the configuration of the model based on its performance on the validation set can quickly result in *overfitting to the validation set*, even though your model is never directly trained on it.

Central to this phenomenon is the notion of *information leaks*. Every time you tune a hyperparameter of your model based on the model's performance on the validation set, some information about the validation data leaks into the model. If you do this only once, for one parameter, then very few bits of information will leak, and your validation set will remain reliable to evaluate the model. But if you repeat this many times—

running one experiment, evaluating on the validation set, and modifying your model as a result—then you'll leak an increasingly significant amount of information about the validation set into the model.

At the end of the day, you'll end up with a model that performs artificially well on the validation data, because that's what you optimized it for. You care about performance on completely new data, not the validation data, so you need to use a completely different, never-before-seen dataset to evaluate the model: the test dataset. Your model shouldn't have had access to *any* information about the test set, even indirectly. If anything about the model has been tuned based on test set performance, then your measure of generalization will be flawed.

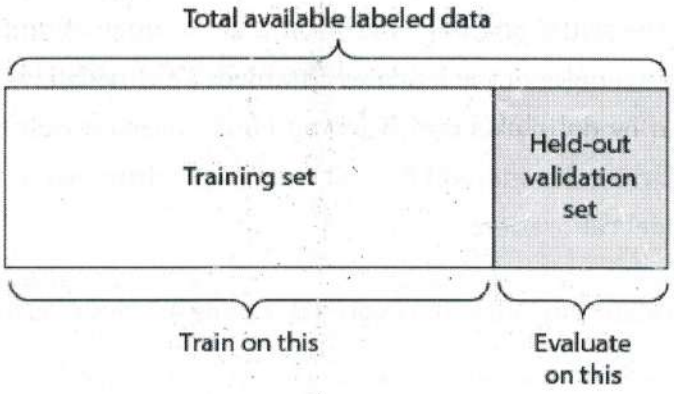
Splitting your data into training, validation, and test sets may seem straightforward, but there are a few advanced ways to do it that can come in handy when little data is available. Let's review three classic evaluation recipes: simple hold-out validation, K-fold validation, and iterated K-fold validation with shuffling.

Simple hold-out validation

Set apart some fraction of your data as your test set. Train on the remaining data, and evaluate on the test set. As you saw in the previous sections, 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, hold-out validation looks like figure 4.1. Listing 4.1 shows a simple implementation.

Figure 4.1. Simple hold-out validation split



Listing 4.1. Hold-out validation

```
indices <- sample(1:nrow(data), size = 0.80 * nrow(data))
evaluation_data  <- data[-indices, ]
training_data <- data[indices, ]
```

```
model <- get_model()                                                    4
model %>% train(training_data)                                          4
validation_score <- model %>% evaluate(validation_data)                 4

model <- get_model()                                                    5
model %>% train(data)                                                   5
test_score <- model %>% evaluate(test_data)                             5
```

- ***1* Shuffling the data is usually appropriate.**
- ***2* Defines the validation set**
- ***3* Defines the training set**
- ***4* Trains a model on the training data, and evaluates it on the validation data**
- ***5* Once you've tuned your hyperparameters, it's common to train your final model from scratch on all non-test data available.**

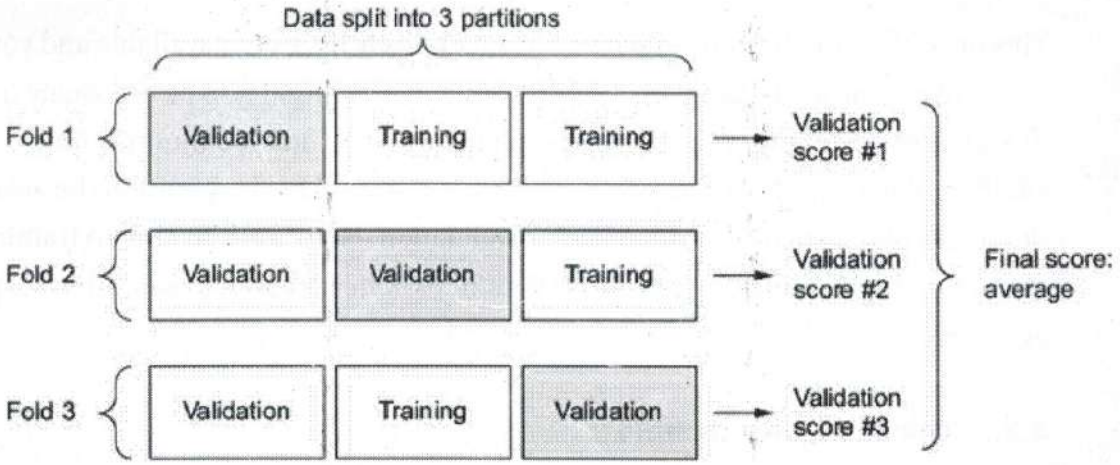
This is the simplest evaluation protocol, and it suffers from one flaw: if little data is available, then your validation and test sets may contain too few samples to be statistically representative of the data at hand. This is easy to recognize: if different random shuffling rounds of the data before splitting end up yielding very different measures of model performance, then you're having this issue. K-fold validation and iterated K-fold validation are two ways to address this, as discussed next.

K-fold validation

With this approach, you split your data into K partitions of equal size. For each partition `i`, train a model on the remaining $K - 1$ partitions, and evaluate it on partition `i`. Your final score is then the averages of the K scores obtained. This method is helpful when the performance of your model shows significant variance based on your train-test split. Like hold-out validation, this method doesn't exempt you from using a distinct validation set for model calibration.

Schematically, K-fold cross-validation looks like figure 4.2. Listing 4.2 shows a simple R pseudocode implementation.

Figure 4.2. Three-fold validation



Listing 4.2. K-fold cross-validation

```
k <- 4
indices <- sample(1:nrow(data))
folds <- cut(indices, breaks = k, labels = FALSE)

validation_scores <- c()
for (i in 1:k) {

  validation_indices <- which(folds == i, arr.ind = TRUE)
  validation_data <- data[validation_indices,]                    1
  training_data <- data[-validation_indices,]                     2

  model <- get_model()                                            3
  model %>% train(training_data)
  results <- model %>% evaluate(validation_data)
  validation_scores <- c(validation_scores, results$accuracy)
}

validation_score <- mean(validation_scores)                       4

model <- get_model()                                              5
model %>% train(data)                                             5
results <- model %>% evaluate(test_data)                          5
```

- *1* **Selects the validation-data partition**
- *2* **Uses the remainder of the data as training data**
- *3* **Creates a brand-new instance of the model (untrained)**
- *4* **Validation score: average of the validation scores of the K-folds**
- *5* **Trains the final model on all non-test data available**

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. We'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 $P \times K$ models (where P is the number of iterations you use), which can very expensive.

4.2.2. Things to keep in mind

Keep an eye out for the following when you're choosing an evaluation protocol:

- *Data representativeness*—You want both your training set and test set to be representative of the data at hand. For instance, if you're trying to classify images of digits, and you're starting from an array of samples where the samples are ordered by their class, taking the first 80% of the array as your training set and the remaining 20% as your test set will result in your training set containing only classes 0–7, whereas your test set contains only classes 8–9. This seems like a ridiculous mistake, but it's surprisingly common. For this reason, you usually should *randomly shuffle* your data before splitting it into training and test sets.
- *The arrow of time*—If you're trying to predict the future given the past (for example, tomorrow's weather, stock movements, and so on), you should *not* randomly shuffle your data before splitting it, because doing so will create a *temporal leak*: your model will effectively be trained on data from the future. In such situations, you should always make sure all data in your test set is *posterior* to the data in the training set.
- *Redundancy in your data*—If some data points in your data appear twice (fairly common with real-world data), then shuffling the data and splitting it into a training set and a validation set will result in redundancy between the training and validation sets. In effect, you'll be testing on part of your training data, which is the worst thing you can do! Make sure your training set and validation set are disjoint.

4.3. DATA PREPROCESSING, FEATURE ENGINEERING, AND FEATURE LEARNING

In addition to model evaluation, an important question we must tackle before we dive deeper into model development is the following: how do you prepare the input data and targets before feeding them into a neural network? Many data-preprocessing and feature-engineering 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.

4.3.1. Data preprocessing for neural networks

Data preprocessing aims at making the raw data at hand more amenable to neural networks. This includes vectorization, normalization, handling missing values, and feature extraction.

Vectorization

All inputs and targets in a neural network must be tensors of floating-point data (or, in specific cases, tensors of integers). 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, we started from text represented as lists of integers (standing for sequences of words), and we used one-hot encoding to turn them into a tensor of floating-point data. In the examples of classifying digits and predicting house prices, the data already came in vectorized form, so you were able to skip this step.

Value normalization

In the digit-classification example, you started from image data encoded as integers in the 0–255 range, encoding grayscale values. Before you fed this data into your network, you had to divide by 255 so you'd end up with floating-point values in the 0–1 range. Similarly, when predicting house prices, you started from features that took a variety of ranges—some features had small floating-point values, and others had fairly large integer values. Before you fed this data into your network, you had to normalize each feature independently so that it had a standard deviation of 1 and a mean of 0.

In general, it isn't safe to feed into a neural network data that takes relatively large values (for example, multidigit 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*—That is, all features should take values in roughly the same range.

Additionally, the following stricter normalization practice is common and can help,

although it isn't always necessary (for example, you didn't do this in the digit-classification example).

- Normalize each feature independently to have a mean of 0.
- Normalize each feature independently to have a standard deviation of 1.

This is easy to do with R using the `scale()` function:

```
x <- scale(x)                                    1
```

- ***1* Assuming x is a 2D matrix of shape (samples, features)**

Typically, you'll normalize features in both training and test data. In this case, you want to compute the mean and standard deviation on the training data only and then apply them to both the training and test data. This is what we did in chapter 3 when normalizing features in the Boston housing dataset:

```
mean <- apply(train_data, 2, mean)                                    1
std <- apply(train_data, 2, sd)
train_data <- scale(train_data, center = mean, scale = std)           2
test_data <- scale(test_data, center = mean, scale = std)
```

- ***1* Calculates the mean and standard deviation on the training data**
- ***2* Scales the training and test data using the mean and standard deviation from the training data**

The `caret` and `recipes` R packages both include many more high-level functions for data preprocessing and normalization.

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.

In general, with neural networks, it's safe to input missing values as 0, with the condition that 0 isn't already a meaningful value. The network will learn from exposure to the data that the value 0 means *missing data* and will start ignoring the value.

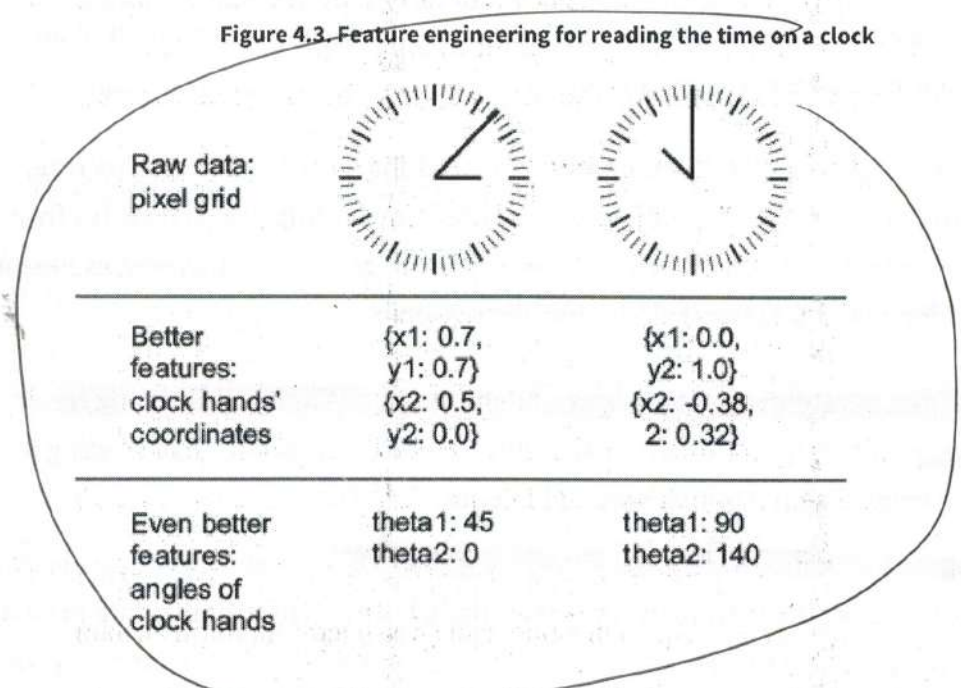
Note that if you're expecting missing values in the test data, but the network was trained on data without any missing values, the network won't have learned to ignore missing values! In this situation, you should artificially generate training samples with missing entries: copy some training samples several times, and drop some of the features that you expect are likely to be missing in the test data.

4.3.2. 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 (nonlearned) 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 4.3).

Figure 4.3. Feature engineering for reading the time on a clock



If you choose to use the raw pixels of the image as input data, then you have a difficult machine-learning problem on your hands. You'll need a convolutional neural network to solve it, and you'll have to expend quite a bit of computational resources to train the network.

But if you already understand the problem at a high level (you understand how humans read time on a clock face), then you can come up with much better input features for a machine-learning algorithm: for instance, it's easy to write a short R 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. It usually requires understanding the problem in depth.

Before deep learning, feature engineering used to be critical, 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 essential to its success. For instance, before convolutional neural networks became successful on the MNIST digit-classification problem, solutions were typically based on hardcoded features such as the number of loops in a digit image, the height of each digit in an image, a histogram of pixel values, and so on.

Fortunately, modern deep learning removes the need for most feature engineering, because neural networks are capable of automatically extracting useful features from raw data. Does this mean you don't have to worry about feature engineering as long as you're using deep neural networks? No, for two reasons:

- Good features still allow you to solve problems more elegantly while using fewer resources. For instance, it would be ridiculous to solve the problem of reading a clock face using a convolutional neural network.
- Good features let you solve a problem with far less data. The ability of deep-learning models to learn features on their own relies on having lots of training data available; if you have only a few samples, then the information value in their features becomes critical.

4.4. OVERFITTING AND UNDERFITTING

In all three examples in the previous chapter—predicting movie reviews, topic classification, and house-price regression—the performance of the model on the held-out validation data always peaked after a few epochs and then began to degrade: the model quickly started to *overfit* to the training data. Overfitting happens in every machine-learning problem. Learning how to deal with overfitting is essential to

mastering machine learning.

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 adjust the model based on its training data.

At the beginning of training, optimization and generalization are correlated: the lower the loss on training data, the lower the loss on test data. While this is happening, your model is said to be *underfit*: there is still progress to be made; the network hasn't yet modeled all relevant patterns in the training data. But after a certain number of iterations on the training data, generalization stops improving, and validation metrics stall and then begin to degrade: the model is starting to overfit. 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.

To prevent a model from learning misleading or irrelevant patterns found in the training data, *the best solution is to get more training data.* A model trained on more data will naturally generalize better. When that 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 what information it's allowed to store. If a network can only afford to memorize a small number of 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*. Let's review some of the most common regularization techniques and apply them in practice to improve the movie-classification model from section 3.4.

4.4.1. Reducing the network's size

The simplest way to prevent overfitting is to reduce the size of the model: the number of learnable parameters in the model (which is determined by the number of layers and the number of units per layer). In deep learning, the number of learnable parameters in a model is often referred to as the model's *capacity*. Intuitively, a model with more parameters has more *memorization capacity* and therefore can easily learn a perfect dictionary-like mapping between training samples and their targets—a mapping without any generalization power. For instance, a model with 500,000 binary parameters could easily be made to learn the class of every digit in the MNIST training set: we'd need only 10 binary parameters for each of the 50,000 digits. But such a

model would be useless for classifying new digit samples. Always keep this in mind: deep-learning models tend to be good at fitting to the training data, but the real challenge is generalization, not fitting.

On the other hand, if the network has limited memorization resources, it won't be able to learn this mapping as easily; 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) in order to find the correct model size for your data. The general workflow to find an appropriate model size is to start with relatively few layers and parameters, and increase the size of the layers or add new layers until you see diminishing returns with regard to validation loss.

Let's try this on the movie-review classification network. The original network is shown next.

Listing 4.3. Original model

```
library(keras)

model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Now let's try to replace it with this smaller network.

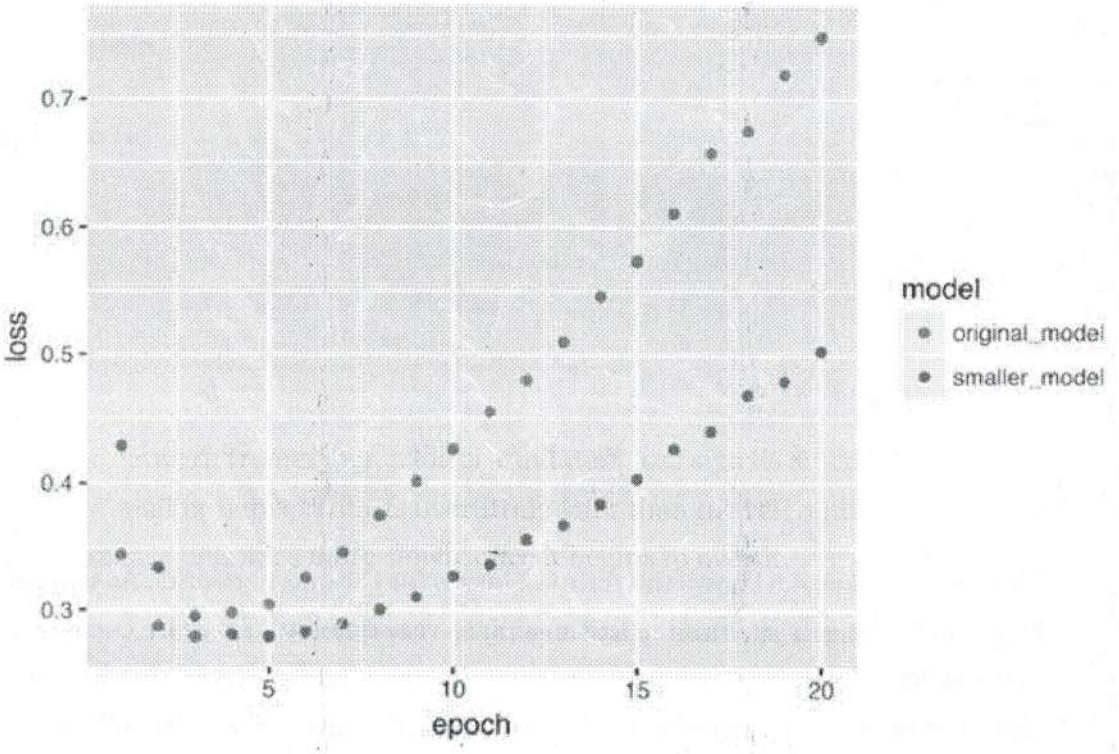
Listing 4.4. Version of the model with lower capacity

```
model <- keras_model_sequential() %>%
  layer_dense(units = 4, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 4, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Figure 4.4 shows a comparison of the validation losses of the original network and the

smaller network (remember, a lower validation loss signals a better model). As you can see, the smaller network starts overfitting later than the reference network, and its performance degrades more slowly once it begins to overfit.

Figure 4.4. Effect of model capacity on validation loss: trying a smaller model



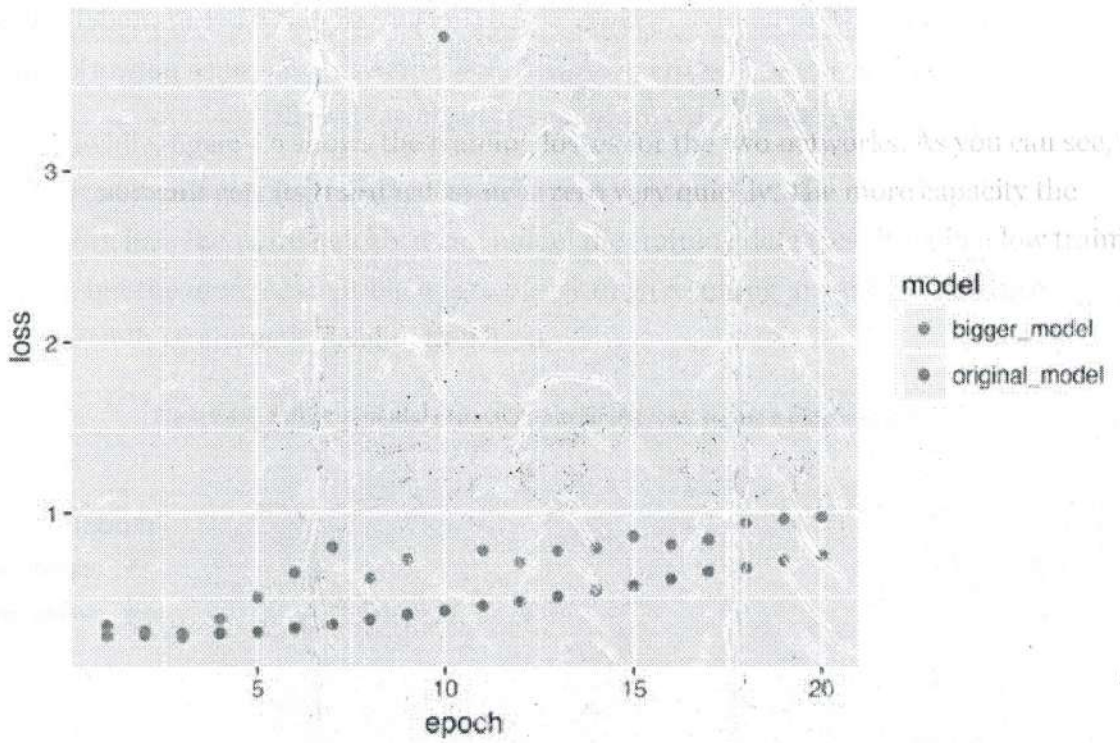
Now, for kicks, let's add to this benchmark a network that has much more capacity—far more than the problem warrants.

Listing 4.5. Version of the model with higher capacity

```
model <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(10000)) %>
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

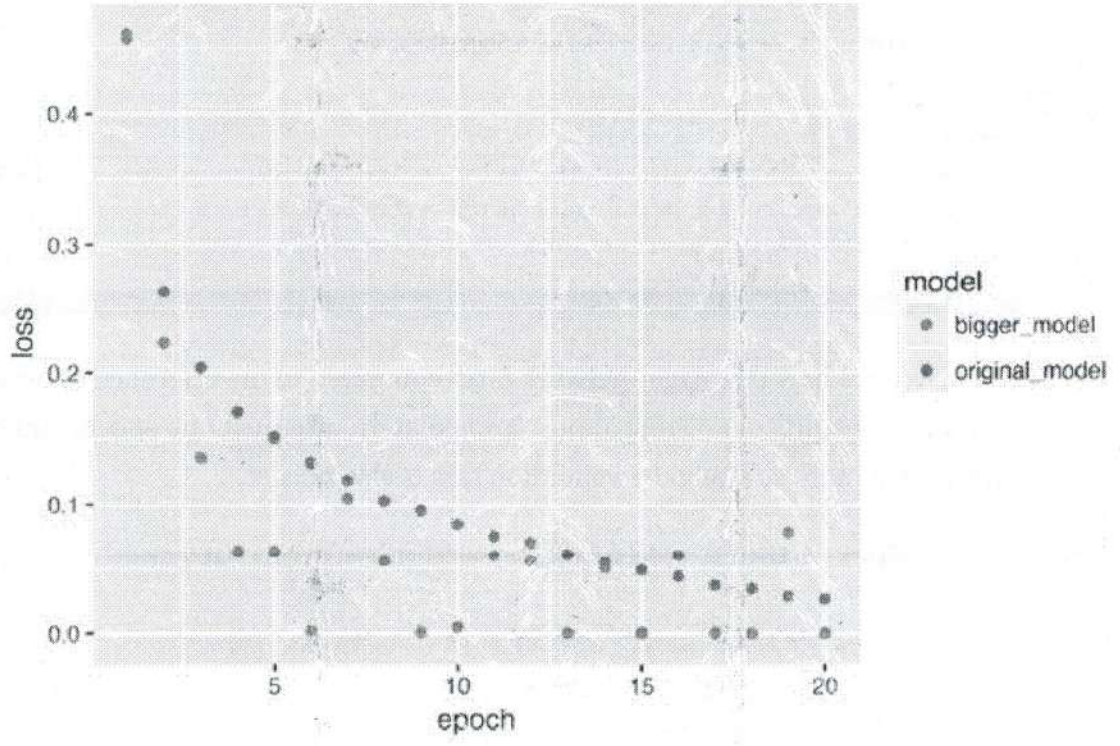
Figure 4.5 shows how the bigger network fares compared to the reference network. The bigger network starts overfitting almost immediately, after just one epoch, and it overfits much more severely. Its validation loss is also noisier.

Figure 4.5. Effect of model capacity on validation loss: trying a bigger model



Meanwhile, figure 4.6 shows the training losses for the two networks. As you can see, the bigger network gets its training loss near zero very quickly. The more capacity the network has, the more quickly it can model the training data (resulting in a low training loss), but the more susceptible it is to overfitting (resulting in a large difference between the training and validation loss).

Figure 4.6. Effect of model capacity on training loss: trying a bigger model



4.4.2. 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 network 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 network a *cost* associated with having large weights. This cost comes in two flavors:

- *L1 regularization*—The cost added is proportional to the *absolute value of the weight coefficients* (the *L1 norm* of the weights).
- *L2 regularization*—The cost added is proportional to the *square of the value of the weight coefficients* (the *L2 norm* of the weights). L2 regularization is also called *weight decay* in the context of neural networks. Don't let the different name confuse you: weight decay is mathematically the same as L2 regularization.

In Keras, weight regularization is added by passing *weight regularizer instances* to layers as keyword arguments. Let's add L2 weight regularization to the movie-review classification network.

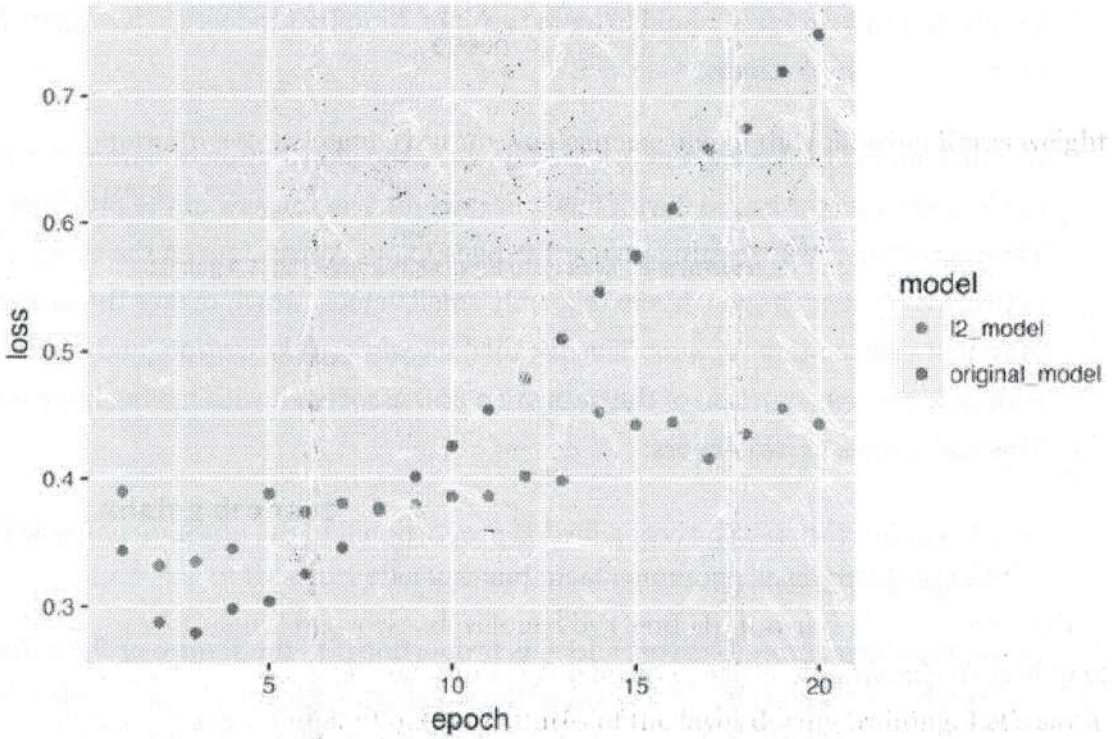
Listing 4.6. Adding L2 weight regularization to the model

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.001),
              activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.001),
              activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

`regularizer_l2(0.001)` means every coefficient in the weight matrix of the layer will add `0.001 * weight_coefficient_value` to the total loss of the network. Note that because this penalty is *only added at training time*, the loss for this network will be much higher at training time than at test time.

Figure 4.7 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 4.7. 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 4.7. Different weight regularizers available in Keras

```
regularizer_l1(0.001)
regularizer_l1_l2(l1 = 0.001, l2 = 0.001)
```

4.4.3. Adding dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks, 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 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:

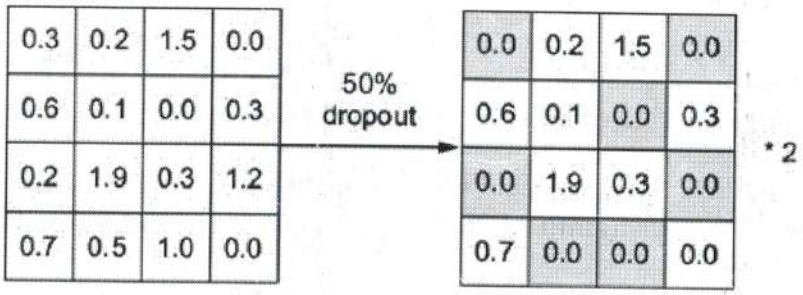
```
layer_output <- layer_output * sample(0:1, length(layer_output),
                                      replace = TRUE)
```

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 <- layer_output * 0.5
```

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 4.8):

Figure 4.8. Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time, the activation matrix is unchanged.




```
layer_output <- layer_output * sample(0:1, length(layer_output),      1
                                      replace = TRUE)
layer_output <- layer_output / 0.5                                    2
```

- **1 At training time**
- **2 Note that we're scaling up rather scaling down in this case.**

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."[1] 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 network will start memorizing if no noise is present.

[1] See the Reddit thread "AMA: We are the Google Brain team. We'd love to answer your questions about machine learning," http://mng.bz/XrsS.

In Keras, you can introduce dropout in a network via `layer_dropout`, which is applied to the output of the layer immediately before it:

```
layer_dropout(rate = 0.5)
```

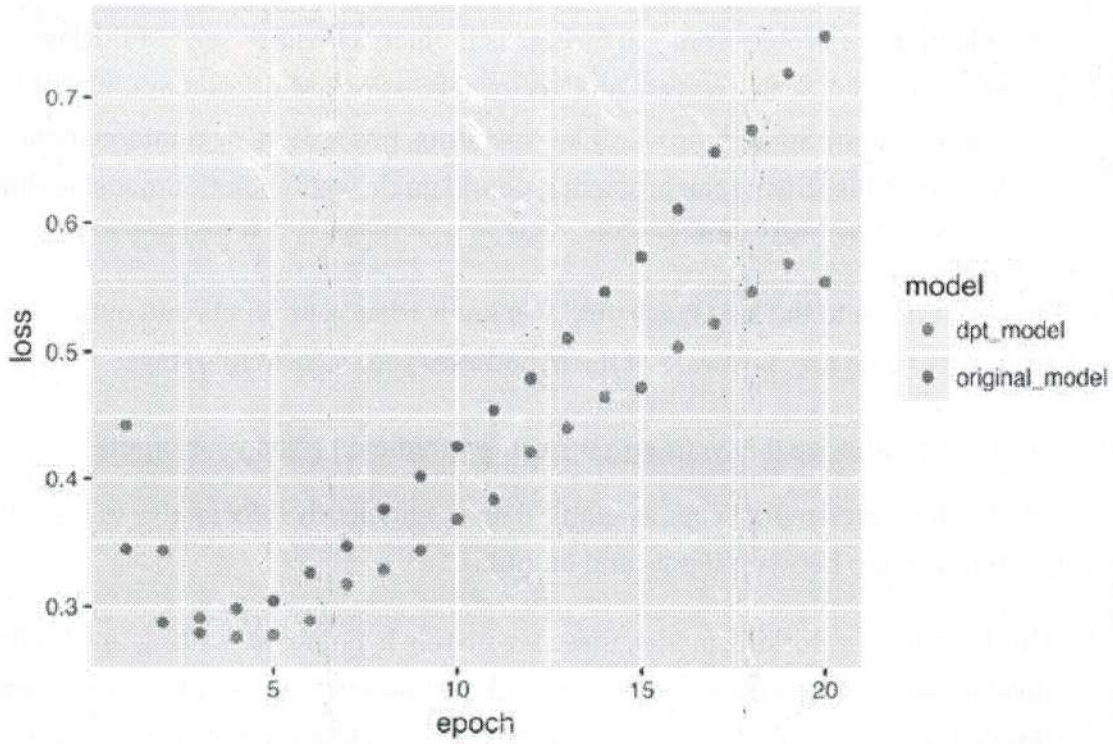
Let's add two dropout layers in the IMDB network to see how well they do at reducing overfitting.

Listing 4.8. Adding dropout to the IMDB network

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Figure 4.9 shows a plot of the results. Again, this is a clear improvement over the reference network.

Figure 4.9. Effect of dropout on validation loss



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

- Get more training data.
- Reduce the capacity of the network.
- Add weight regularization.
- Add dropout.

4.5. THE UNIVERSAL WORKFLOW OF MACHINE LEARNING

In this section, we'll present a universal blueprint that you can use to attack and solve any machine-learning problem. The blueprint ties together the concepts you've learned about in this chapter: problem definition, evaluation, feature engineering, and fighting overfitting.

4.5.1. Defining the problem and assembling a dataset

First, you must define the problem at hand:

- What will your input data be? What are you trying to predict? You can only learn to predict something if you have available training data: for example, you can only learn to classify the sentiment of movie reviews if you have both movie reviews and sentiment annotations available. As such, data availability is usually the limiting factor at this stage (unless you have the means to pay people to collect data for you).

- What type of problem are you facing? Is it binary classification? Multiclass classification? Scalar regression? Vector regression? Multiclass, multilabel classification? Something else, like clustering, generation, or reinforcement learning? Identifying the problem type will guide your choice of model architecture, loss function, and so on.

You can't move to the next stage until you know what your inputs and outputs are, and what data you'll use. Be aware of the hypotheses you make at this stage:

- You hypothesize that your outputs can be predicted given your inputs.
- You hypothesize that your available data is sufficiently informative to learn the relationship between inputs and outputs.

Until you have a working model, these are merely hypotheses, waiting to be validated or invalidated. Not all problems can be solved; 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.

One class of unsolvable problems you should be aware of is *nonstationary problems*. Suppose you're trying to build a recommendation engine for clothing, you're training it on one month of data (August), and you want to start generating recommendations in the winter. One big issue is that the kinds of clothes people buy change from season to season: clothes buying is a nonstationary phenomenon over the scale of a few months. What you're trying to model changes over time. In this case, the right move is to constantly retrain your model on data from the recent past, or gather data at a timescale where the problem is stationary. For a cyclical problem like clothes buying, a few years' worth of data will suffice to capture seasonal variation—but remember to make the time of the year an input of your model!

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.

4.5.2. Choosing a measure of success

To control something, you need to be able to observe it. To achieve success, you must define what you mean by success—accuracy? Precision and recall? Customer-retention rate? Your metric for success will guide the choice of a loss function: what your model

will optimize. It should directly align with your higher-level goals, such as the success of your business.

For balanced-classification problems, where every class is equally likely, accuracy and *area under the receiver operating characteristic curve* (ROC AUC) are common metrics. For class-imbalanced problems, you can use precision and recall. For ranking problems or multilabel classification, you can use mean average precision. 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.

4.5.3. Deciding on an evaluation protocol

Once you know what you're aiming for, you must establish how you'll measure your current progress. We've previously reviewed three common evaluation protocols:

- *Maintaining a hold-out validation set*—The way to go when you have plenty of data
- *Doing K-fold cross-validation*—The right choice when you have too few samples for hold-out validation to be reliable
- *Doing iterated K-fold validation*—For performing highly accurate model evaluation when little data is available

Just pick one of these. In most cases, the first will work well enough.

4.5.4. Preparing your data

Once you know what you're training on, what you're optimizing for, and how to evaluate your approach, you're almost ready to begin training models. But first, you should format your data in a way that can be fed into a machine-learning model—here, we'll assume a deep neural network:

- As you saw previously, your data should be formatted as tensors.
- The values taken by these tensors should usually be scaled to small values: for example, in the [-1, 1] range or [0, 1] range.
- If different features take values in different ranges (heterogeneous data), then the data should be normalized.
- You may want to do some feature engineering, especially for small-data problems.

Once your tensors of input data and target data are ready, you can begin to train models.

4.5.5. Developing a model that does better than a baseline

Your goal at this stage is to achieve *statistical power*: that is, to develop a small model that is capable of beating a dumb baseline. In the MNIST digit-classification example, anything that achieves an accuracy greater than 0.1 can be said to have statistical power; in the IMDB example, it's anything with an accuracy greater than 0.5.

Note that it's not always possible to achieve statistical power. If you can't beat a random 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 make 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 relationships between inputs and outputs.

It may well be that these hypotheses are false, in which case you must go back to the drawing board.

Assuming that things go well, you need to make three key choices to build your first working model:

- *Last-layer activation*—This establishes useful constraints on the network's output. For instance, the IMDB classification example used `sigmoid` in the last layer; the regression example didn't use any last-layer activation; and so on.
- *Loss function*—This should match the type of problem you're trying to solve. For instance, the IMDB example used `binary_crossentropy`, the regression example used `mse`, and so on.
- *Optimization configuration*—What optimizer will you use? What will its learning rate be? In most cases, it's safe to go with `rmsprop` and its default learning rate.

Regarding the choice of a loss function, note that it isn't always possible to directly optimize for the metric that measures success on a problem. Sometimes there is no easy way to turn a metric into a loss function; loss functions, after all, need to be computable given only a mini-batch of data (ideally, a loss function should be computable for as little as a single data point) and must be differentiable (otherwise, you can't use backpropagation to train your network). For instance, the widely used classification

metric ROC AUC can't be directly optimized. Hence, in classification tasks, it's common to optimize for a proxy metric of ROC AUC, such as crossentropy. In general, you can hope that the lower the crossentropy gets, the higher the ROC AUC will be.

Table 4.1 can help you choose a last-layer activation and a loss function for a few common problem types.

Table 4.1. 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
Regression to arbitrary values	None	mse
Regression to values between 0 and 1	sigmoid	mse or binary_crossentropy

4.5.6. Scaling up: developing a model that overfits

Once you've obtained a model that has statistical power, the question becomes, is it sufficiently powerful? Does it have enough layers and parameters to properly model the problem at hand? For instance, a network with a single hidden layer with two units would have 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:

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.

The next stage is to start regularizing and tuning the model, to get as close as possible to the ideal model that neither underfits nor overfits.

4.5.7. Regularizing your model and tuning your hyperparameters

This step 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. These are some things you should try:

- Add dropout.
- Try different architectures: add or remove layers.
- Add L1 and/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 feature engineering: add new features, or remove features that don't seem to be informative.

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; but 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 the 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).

4.6. SUMMARY

- Define the problem at hand and the data on which you'll train. Collect this data, or annotate it with labels if need be.

- Choose how you'll measure success on your problem. Which metrics will you monitor on your validation data?
- Determine your evaluation protocol: Hold-out validation? K-fold validation? Which portion of the data should you use for validation?
- Develop a first model that does better than a basic baseline: a model with statistical power.
- Develop a model that overfits.
- 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.