

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?

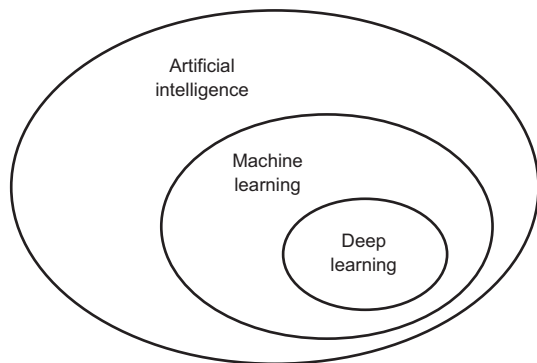


Figure 1.1 Artificial intelligence, machine learning, and deep learning

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

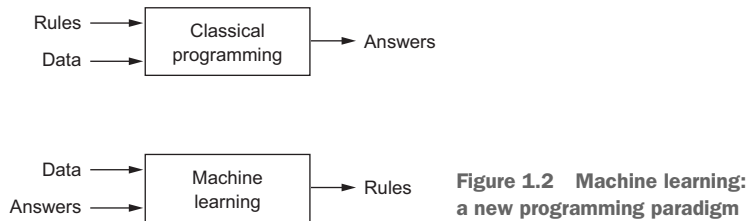
In Victorian England, Lady Ada Lovelace was a friend and collaborator of Charles Babbage, the inventor of the *Analytical Engine*: the first-known general-purpose, mechanical computer. Although visionary and far ahead of its time, the Analytical

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.

Machine learning arises from this question: could a computer go beyond “what 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.



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.

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

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*. I 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 RG 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.

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.

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

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

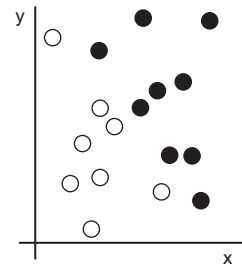


Figure 1.3
Some sample data

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.

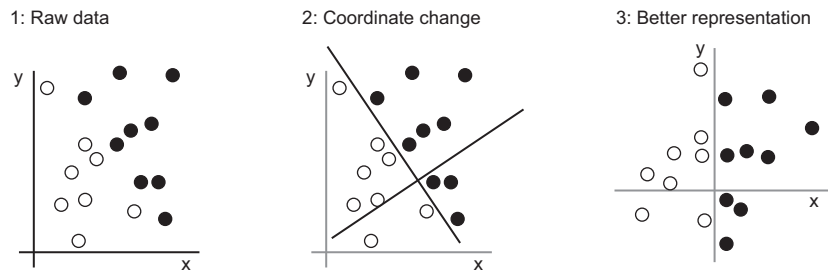


Figure 1.4 Coordinate change

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.

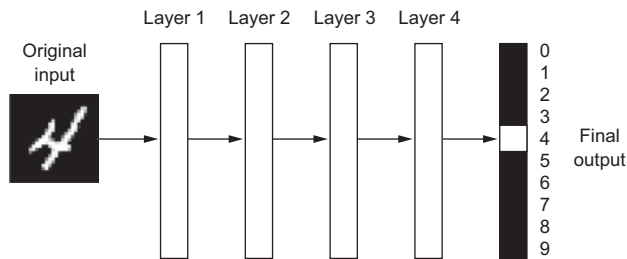


Figure 1.5 A deep neural network for digit classification

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).

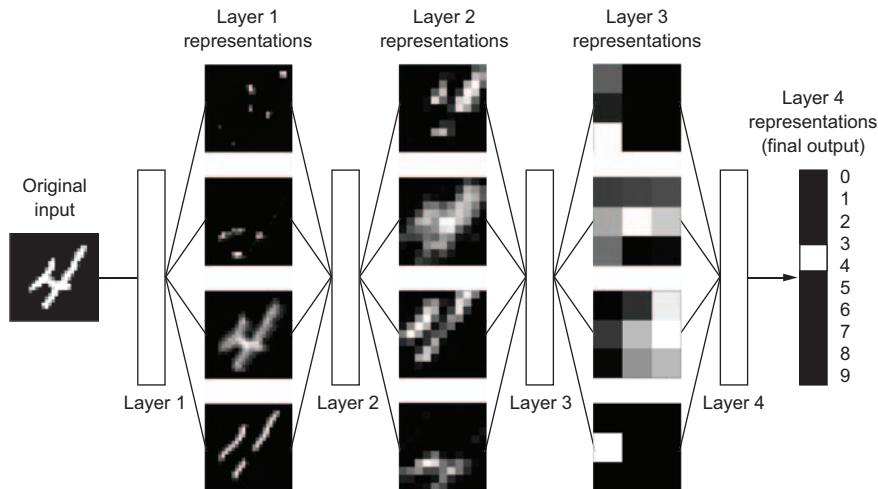


Figure 1.6 Deep representations learned by a digit-classification model

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 input and targets. You also know that deep neural networks do this input-to-target

mapping via a deep sequence of simple data transformations (layers) and that these data transformations are learned by exposure to examples. Now let's look at how this learning happens, concretely.

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

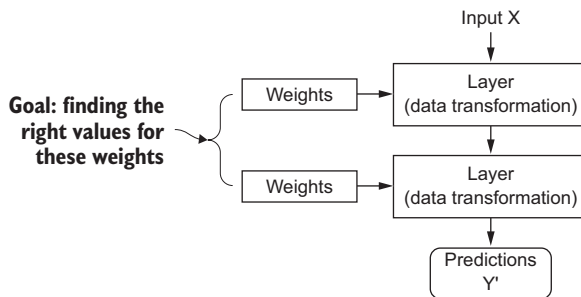


Figure 1.7 A neural network is parameterized by its weights.

To control something, first you need to be able to observe it. To control the output of a neural network, you need to be able to measure how far this output is from what you expected. This is the job of the *loss function* of the network, also 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).

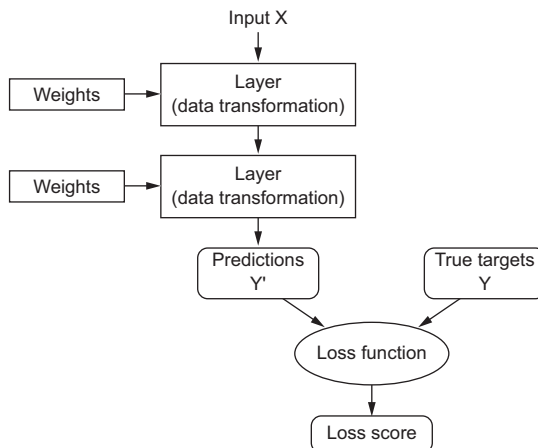


Figure 1.8 A loss function measures the quality of the network's output.

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.

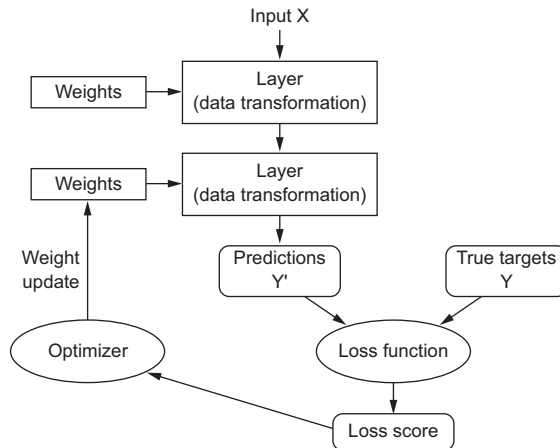


Figure 1.9 The loss score is used as a feedback signal to adjust the weights.

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 likely be possible. Although some world-changing applications like autonomous cars are already within reach, many more are likely to remain elusive for a long time, such as believable dialogue systems, human-level machine translation across arbitrary languages, and human-level natural-language understanding. In particular, talk of *human-level general intelligence* shouldn't be taken too seriously. The risk with high expectations for the short term is that, as technology fails to deliver, research investment will dry up, slowing progress for a long time.

This has happened before. Twice in the past, AI went through a cycle of intense optimism followed by disappointment and skepticism, with a dearth of funding as a result. It started with symbolic AI in the 1960s. In those early days, projections about AI were flying high. One of the best-known pioneers and proponents of the symbolic AI approach was Marvin Minsky, who claimed in 1967, "Within a generation ... the problem of creating 'artificial intelligence' will substantially be solved." Three years later, in 1970, he made a more precisely quantified prediction: "In from three to eight years we will have a machine with the general intelligence of an average human being." In 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 large impact on our world, because it isn’t yet widely deployed—much as, back in 1995, it would have been difficult to believe in the future impact of the internet. Back then, most people didn’t see how the internet was relevant to them and how it was going to change their lives. The same is true for deep learning and AI today. But make no mistake: AI is coming. In a not-so-distant future, AI will be your assistant, even your friend; it will answer your questions, help educate your kids, and watch over your health. It will deliver your groceries to your door and drive you from point A to point B. It will be your interface to an increasingly complex and information-intensive world. And, even more important, AI will help humanity as a whole move forward, by assisting human scientists in new breakthrough discoveries across all scientific fields, from genomics to mathematics.

On the way, we may face a few setbacks and maybe 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.

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

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.

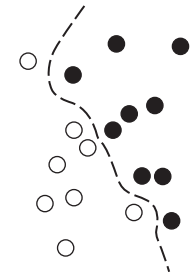


Figure 1.10
A decision boundary

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

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).

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. Decision trees learned from data began to receive significant research interest in the 2000s, and by 2010 they were often preferred to kernel methods.

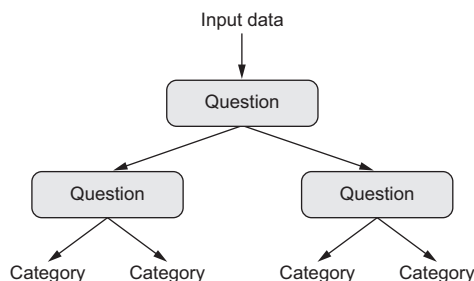


Figure 1.11 A decision tree: the parameters that are learned are the questions about the data. A question could be, for instance, “Is coefficient 2 in the data greater than 3.5?”

In particular, the *Random Forest* algorithm introduced a robust, practical take on decision-tree learning that involves building a large number of specialized decision trees and then ensembling their outputs. Random forests are applicable to a wide range of problems—you could say that they're almost always the second-best algorithm for any shallow machine-learning task. When the popular machine-learning competition website Kaggle (<http://kaggle.com>) got started in 2010, random forests quickly became a favorite on the platform—until 2014, when *gradient boosting machines* took over. A gradient boosting machine, much like a random forest, is a machine-learning technique based on ensembling weak prediction models, generally decision trees. It

uses *gradient boosting*, a way to improve any machine-learning model by iteratively training new models that specialize in addressing the weak points of the previous models. Applied to decision trees, the use of the gradient boosting technique results in models that strictly outperform random forests most of the time, while having similar properties. It may be one of the best, if not *the* best, algorithm for dealing with nonperceptual data today. Alongside deep learning, it's one of the most commonly used techniques in Kaggle competitions.

1.2.5 Back to neural networks

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

In 2011, Dan Ciresan from IDSIA began to win academic image-classification competitions with GPU-trained deep neural networks—the first practical success of modern deep learning. But the watershed moment came in 2012, with the entry of Hinton's group in the yearly large-scale image-classification challenge ImageNet. 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, which offers support for the two most popular languages of data science: Python and R. Meanwhile, most of the Kaggle entrants using deep learning use the Keras library, due to its ease of use, flexibility, and support of 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 ideas 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 Ciresan⁴ and Alex Krizhevsky⁵ 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 top-of-the-line 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

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,

⁶ The ImageNet Large Scale Visual Recognition Challenge (ILSVRC), www.image-net.org/challenges/LSVRC.

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 depth-wise separable convolutions. Today we can train from scratch models that are thousands of layers deep.

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.

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

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 went 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 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 for Python that support autodifferentiation, greatly simplifying the implementation of new models—and by the rise of user-friendly libraries such as Keras, which makes deep learning as easy as manipulating LEGO bricks. After its release in early 2015, Keras quickly became the go-to deep-learning solution for large numbers of new startups, graduate students, and researchers pivoting into the field.

1.3.6 Will it last?


Is there anything special about deep neural networks that makes them the “right” approach for companies to be investing in and for researchers to flock to? Or is deep learning just a fad that may not last? Will we still be using deep neural networks in 20 years?

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

- *Simplicity*—Deep learning removes the need for feature engineering, replacing complex, brittle, engineering-heavy pipelines with simple, end-to-end trainable models that are typically built using only five or six different tensor operations.
- *Scalability*—Deep learning is highly amenable to parallelization on GPUs or TPUs, so it can take full advantage of Moore’s law. In addition, deep-learning models are trained by iterating over small batches of data, allowing them to be trained on datasets of arbitrary size. (The only bottleneck is the amount of parallel computational power available, which, thanks to Moore’s law, is a fast-moving barrier.)
- *Versatility and reusability*—Unlike many prior machine-learning approaches, deep-learning models can be trained on additional data without restarting from 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. Following a scientific revolution, progress generally follows a sigmoid curve: it starts with a period of fast progress, which gradually stabilizes as researchers hit hard limitations, and then further improvements become incremental. 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.



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 Python library Keras to learn to classify handwritten digits. Unless you already have experience with Keras or similar libraries, you won't understand everything about this first example right away. 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.

Note on 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*.



Figure 2.1 MNIST sample digits

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 a set of four Numpy arrays.

Listing 2.1 Loading the MNIST dataset in Keras

```
from keras.datasets import mnist

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

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

The images are encoded as Numpy arrays, and the labels are an array of digits, ranging from 0 to 9. The images and labels have a one-to-one correspondence.

Let's look at the training data:

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

And here's the test data:

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

The workflow will be as follows: First, we'll feed the neural network the training data, `train_images` and `train_labels`. The network will then learn to associate images and labels. Finally, we'll ask the network to produce predictions for `test_images`, and we'll verify whether these predictions match the labels from `test_labels`.

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

Listing 2.2 The network architecture

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

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 Dense 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=['accuracy'])
```

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 `uint8` with values in the $[0, 255]$ interval. We transform it into a `float32` array of shape $(60000, 28 * 28)$ with values between 0 and 1.

Listing 2.4 Preparing the image data

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

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

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

Listing 2.5 Preparing the labels

```
from keras.utils import to_categorical

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: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [=====>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

Two quantities are displayed during training: the loss of the 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:

```
>>> test_loss, test_acc = network.evaluate(test_images, test_labels)
>>> print('test_acc:', test_acc)
test_acc: 0.9785
```

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

This concludes our first example—you just saw how you can build and train a neural network to classify handwritten digits in less than 20 lines of Python code. In the next chapter, I'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 Numpy arrays, also called *tensors*. In general, all current machine-learning systems use tensors as their basic data structure. Tensors are fundamental to the field—so fundamental that Google’s TensorFlow was named after them. So what’s a tensor?

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

2.2.1 Scalars (0D tensors)

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

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

2.2.2 Vectors (1D tensors)

An array of numbers is called a *vector*, or 1D tensor. A 1D tensor is said to have exactly one axis. Following is a Numpy vector:

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

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

2.2.3 Matrices (2D tensors)

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

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

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

2.2.4 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. Following is a Numpy 3D tensor:

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

By packing 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. This is also called the tensor's `ndim` in Python libraries such as Numpy.
- *Shape*—This is a tuple of integers that describes how many dimensions the tensor has along each axis. For instance, the previous matrix example has shape `(3, 5)`, and the 3D tensor example has shape `(3, 3, 5)`. A vector has a shape with a single element, such as `(5,)`, whereas a scalar has an empty shape, `()`.
- *Data type* (usually called `dtype` in Python libraries)—This is the type of the data contained in the tensor; for instance, a tensor's type could be `float32`, `uint8`, `float64`, and so on. On rare occasions, you may see a `char` tensor. Note that string tensors don't exist in Numpy (or in most other libraries), because tensors live in preallocated, contiguous memory segments: and strings, being variable length, would preclude the use of this implementation.

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

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

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

```
>>> print(train_images.ndim)
3
```

Here's its shape:

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

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

```
>>> print(train_images.dtype)
uint8
```

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

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

Listing 2.6 Displaying the fourth digit

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

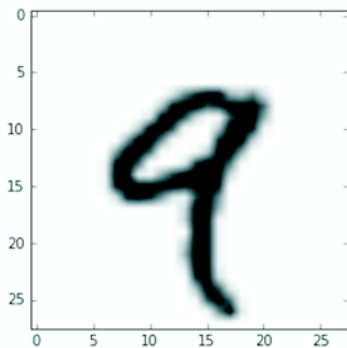


Figure 2.2 The fourth sample in our dataset

2.2.6 Manipulating tensors in Numpy

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

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

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

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

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

Equivalent to the previous example

Also equivalent to the previous example

In general, you may select 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[:, 14:, 14:]
```

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

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

2.2.7 The notion of data batches

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

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

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

And here's the next batch:

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

And the n th batch:

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

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

2.2.8 Real-world examples of data tensors

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

- *Vector data*—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).

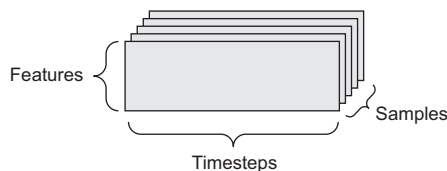


Figure 2.3 A 3D timeseries data tensor

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

- A dataset of stock prices. Every minute, we store the current price of the stock, the highest price in the past minute, and the lowest price in the past minute. Thus every minute is encoded as a 3D vector, an entire day of trading is encoded as a 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 280 characters out of an alphabet of 128 unique characters. In this setting, each character can be encoded as a binary vector of size 128 (an all-zeros vector except for a 1 entry at the index corresponding to the character). Then each tweet can be encoded as a 2D tensor of shape $(280, 128)$, and a dataset of 1 million tweets can be stored in a tensor of shape $(1000000, 280, 128)$.

2.2.11 Image data

Images typically have three dimensions: height, width, and color depth. Although grayscale images (like our MNIST digits) have only a single color channel and could thus be stored in 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).

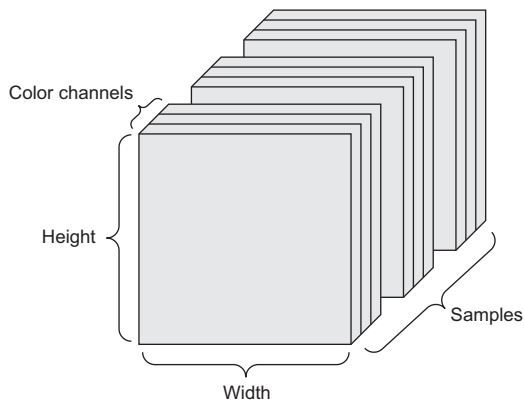


Figure 2.4 A 4D image data tensor (channels-first convention)

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.

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 dtype of the tensor was float32, then each value would be stored in 32 bits, so the tensor would represent 405 MB. Heavy! Videos you encounter in real life are much lighter, because they aren't stored in float32, and they're typically compressed by a large factor (such as in the MPEG format).

2.3 The gears of neural networks: tensor operations

Much as any computer program can be ultimately reduced to a small set of binary operations on binary inputs (AND, OR, NOR, and so on), all transformations learned by deep neural networks can be reduced to a handful of *tensor operations* 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 Keras layer instance looks like this:

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

This layer can be interpreted as a function, which takes as input a 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. I've found that mathematical concepts can be more readily mastered by programmers with no mathematical background if they're expressed as short Python snippets instead of mathematical equations. So we'll use Numpy code throughout.

2.3.1 Element-wise operations

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

```
def naive_relu(x):
    assert len(x.shape) == 2  # ← x is a 2D Numpy tensor.

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

You do the same for addition:

```
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape

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

← **x and y are 2D Numpy tensors.**

← **Avoid overwriting the input tensor.**

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

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

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

```
import numpy as np

z = x + y
z = np.maximum(z, 0.)
```

← **Element-wise addition**

← **Element-wise relu**

2.3.2 Broadcasting

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?

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

- 1 Axes (called *broadcast axes*) are added to the smaller tensor to match the `ndim` of the larger tensor.
- 2 The smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor.

Let's look at a concrete example. Consider `x` with shape `(32, 10)` and `y` with shape `(10,)`. First, we add an empty first axis to `y`, whose shape becomes `(1, 10)`. Then, we repeat `y` 32 times alongside this new axis, so that we end up with a tensor `Y` with shape `(32, 10)`, where `Y[i, :] == y` for `i in range(0, 32)`. At this point, we can proceed to add `x` and `Y`, because they have the same shape.

In terms of implementation, no new 2D tensor is created, because that would be terribly inefficient. The repetition operation is entirely virtual: it happens at the algorithmic level rather than at the memory level. But thinking of the vector being

repeated 10 times alongside a new axis is a helpful mental model. Here's what a naive implementation would look like:

```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]

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

x is a 2D Numpy tensor.

y is a Numpy vector.

Avoid overwriting the input tensor.

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

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

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

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

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

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

2.3.3 Tensor 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 Numpy, Keras, Theano, and TensorFlow. `dot` uses a different syntax in TensorFlow, but in both Numpy and Keras it's done using the standard `dot` operator:

```
import numpy as np
z = np.dot(x, y)
```

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

```
z = x . y
```

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

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
```

x and y are Numpy vectors.

```

z = 0.
for i in range(x.shape[0]):
    z += x[i] * y[i]
return z

```

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

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

```

import numpy as np

def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]

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

```

x is a Numpy matrix.

y is a Numpy vector.

The first dimension of x must be the same as the 0th dimension of y!

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

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

```

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

```

Note that as soon as one of the two tensors has an `ndim` greater than 1, `dot(x, y)` isn't the same as `dot(y, x)`.

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

```

def naive_matrix_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]

    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z

```

x and y are Numpy matrices.

The first dimension of x must be the same as the 0th dimension of y!

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

Iterates over the rows of x ...

... and over the columns of y.

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

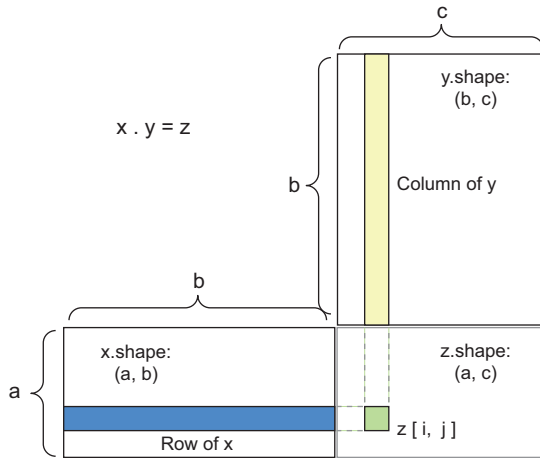


Figure 2.5 Matrix dot-product box diagram

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

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

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

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

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

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

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

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

2.3.5 Geometric interpretation of tensor operations

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

$A = [0.5, 1]$

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

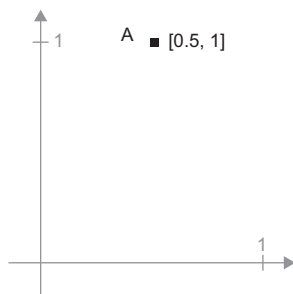


Figure 2.6 A point in a 2D space

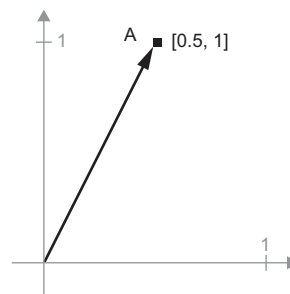


Figure 2.7 A point in a 2D space pictured as an arrow

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

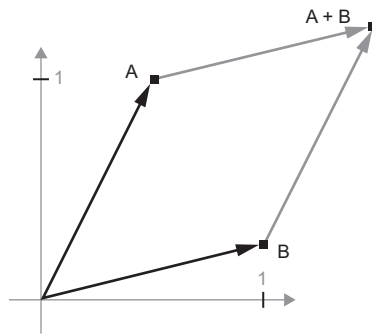


Figure 2.8 Geometric interpretation of the sum of two vectors

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 θ 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.

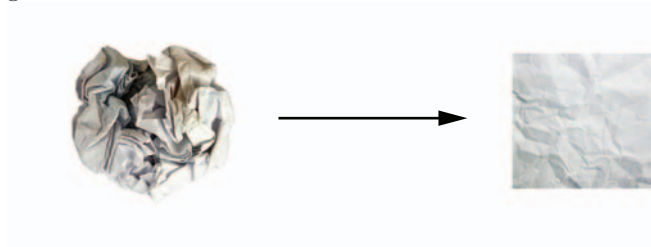


Figure 2.9 Uncrumpling a complicated manifold of data

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 $\text{relu}(\text{dot}(W, \text{input}) + b)$, 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 :

$$f(x + \text{epsilon}_x) = y + \text{epsilon}_y$$

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 $a * \text{epsilon}_x$:

$$f(x + \text{epsilon}_x) = y + a * \text{epsilon}_x$$

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.

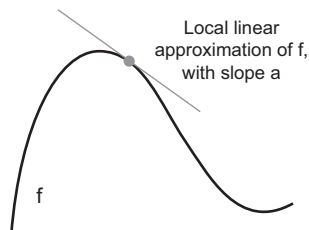


Figure 2.10 Derivative of f in p

For every differentiable function $f(x)$ (*differentiable* means “can be derived”: for example, smooth, continuous functions can be derived), there exists a derivative function $f'(x)$ that maps values of x to the slope of the local linear approximation of f in those

points. For instance, the derivative of $\cos(x)$ is $-\sin(x)$, the derivative of $f(x) = a * x$ is $f'(x) = a$, and so on.

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 W_0 . Then the derivative of f in the point W_0 is a tensor $\text{gradient}(f)(W_0)$ with the same shape as W , where each coefficient $\text{gradient}(f)(W_0)[i, j]$ indicates the direction and magnitude of the change in `loss_value` you observe when modifying $W_0[i, j]$. That tensor $\text{gradient}(f)(W_0)$ is the gradient of the function $f(W) = \text{loss_value}$ in W_0 .

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, $\text{gradient}(f)(W_0)$ can be interpreted as the tensor describing the *curvature* of $f(W)$ around W_0 .

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, $W_1 = W_0 - \text{step} * \text{gradient}(f)(W_0)$ (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 $\text{gradient}(f)(W_0)$ only approximates the curvature when you're close to W_0 , so you don't want to get too far from W_0 .

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 $\text{gradient}(f)(w) = 0$ 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 $w \leftarrow w - \text{step} * \text{gradient}$ —thus reducing the loss on the batch a bit.

Easy enough! What I 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.

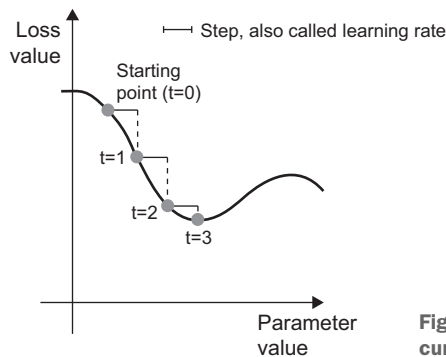
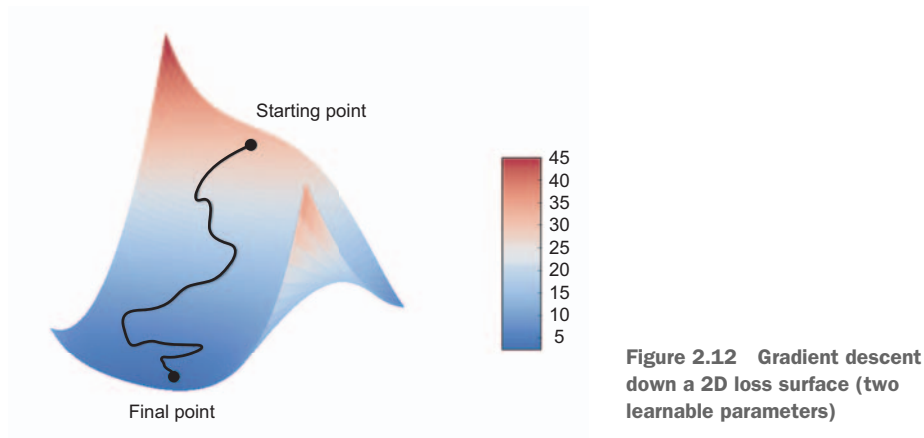


Figure 2.11 SGD down a 1D loss curve (one learnable parameter)

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.



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.

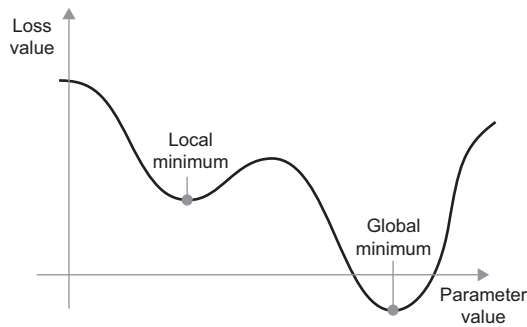


Figure 2.13 A local minimum and a global minimum

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:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum + learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

Constant momentum factor
 Optimization loop

2.4.4 Chaining derivatives: the Backpropagation algorithm

In the 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 derived 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:

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

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

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

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

This was our network:

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(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.

This was the network-compilation step:

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['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.

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.