DEEP LEARNING with Python

François Chollet

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Deep Learning with Python
Version 1

For more information on this and other Manning titles go to
www.manning.com

Welcome

Thank you for purchasing the MEAP for *Deep Learning with Python*. If you are looking for a resource to learn about deep learning from scratch and to quickly become able to use this knowledge to solve real-world problems, you have found the right book. *Deep Learning with Python* is meant for engineers and students with a reasonable amount of Python experience, but no significant knowledge of machine learning and deep learning. It will take you all the way from basic theory to advanced practical applications. However, if you already have experience with deep learning, you should still be able to find value in the latter chapters of this book.

Deep learning is an immensely rich subfield of machine learning, with powerful applications ranging from machine perception to natural language processing, all the way up to creative AI. Yet, its core concepts are in fact very simple. Deep learning is often presented as shrouded in a certain mystique, with references to algorithms that "work like the brain", that "think" or "understand". Reality is however quite far from this science-fiction dream, and I will do my best in these pages to dispel these illusions. I believe that there are no difficult ideas in deep learning, and that's why I started this book, based on premise that all of the important concepts and applications in this field could be taught to anyone, with very few prerequisites.

This book is structured around a series of practical code examples, demonstrating on real-world problems every the notions that gets introduced. I strongly believe in the value of teaching using concrete examples, anchoring theoretical ideas into actual results and tangible code patterns. These examples all rely on Keras, the Python deep learning library. When I released the initial version of Keras almost two years ago, little did I know that it would quickly skyrocket to become one of the most widely used deep learning frameworks. A big part of that success is that Keras has always put ease of use and accessibility front and center. This same reason is what makes Keras a great library to get started with deep learning, and thus a great fit for this book. By the time you reach the end of this book, you will have become a Keras expert.

I hope that you will this book valuable —deep learning will definitely open up new intellectual perspectives for you, and in fact it even has the potential to transform your career, being the most in-demand scientific specialization these days. I am looking forward to your reviews and comments. Your feedback is essential in order to write the best possible book, that will benefit the greatest number of people.

— François Chollet

brief contents

1 What is Deep Learning?

1.1 Artificial intelligence, machine learning and deep learning

In the past few years, Artificial Intelligence (AI) has been a subject of intense media hype. Machine learning, deep learning, and AI come up in countless articles, often outside of technology-minded publications. We are being promised a future of intelligent chatbots, self-driving cars, and virtual assistants --a future sometimes painted in a grim light, and sometimes as an utopia, where human jobs would be scarce and most economic activity would be handled by robots or AI agents. What to make of it?

As a future or current practitioner of machine learning, it is important to be able to recognize the signal in the noise, to tell apart world-changing developments from what are merely over-hyped press releases. What is at stake is our future, and it is a future in which you have an active role to play: after reading this book, you will be part of those who develop the AIs. So let's tackle these questions --what has deep learning really achieved so far? How significant is it? Where are we headed next? Should you believe the hype?

First of all, we need to define clearly what we are talking about when we talk about AI What is artificial intelligence, machine learning, and deep learning? How do they relate to each other?

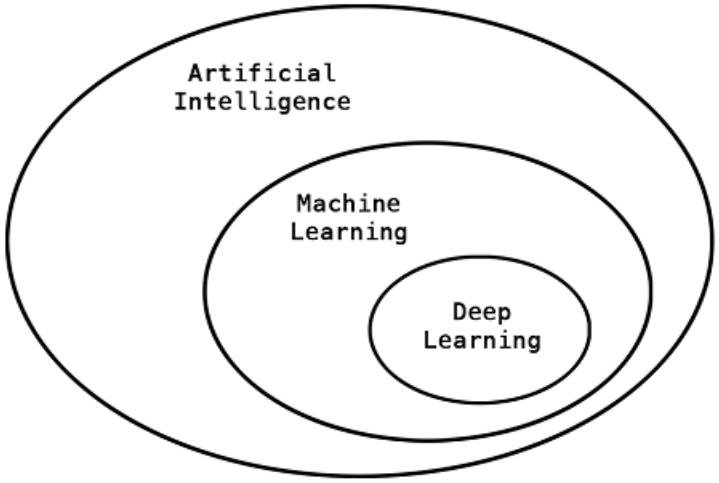

Figure 1.1 Artificial Intelligence, Machine Learning and Deep Learning

1.1.1 Artificial intelligence

Artificial intelligence was born in the 1950s, as a handful of pioneers from the nascent field of computer science started asking if computers could be made to "think" --a question whose ramifications we are still exploring today. A concise definition of the field would be: *the effort to automate intellectual tasks normally performed by humans*. As such, AI is a very general field which encompasses machine learning and deep learning, but also includes many more approaches that do not involve any learning. Early chess programs, for instance, only involved hard-coded rules crafted by programmers, and did not qualify as "machine learning". In fact, for a fairly long time many experts believed that human-level artificial intelligence could be achieved simply by having programmers handcrafting 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, or language translation. A new approach to AI arose to take its 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 design of a general-purpose computer --a mechanical computer. Although visionary and far ahead of its time, the Analytical Engine wasn't actually meant as a general-purpose computer when it was designed in the 1830s and 1840s, since 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"-- and it just "happened" to implement general-purpose computation, a property which we only came to realize during the birth of computer science in the 1940s. 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 are 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", 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 very question: could a computer go beyond "what we know how to order it to perform", and actually "learn" on its own how to perform a specified task? Could a computer surprise us? Rather than crafting data-processing rules by hand, could it possible to automatically learn these rules by looking at data?

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

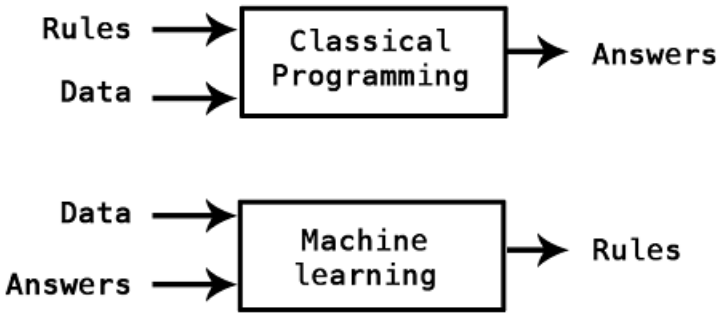

Figure 1.2 Machine learning: a new programming paradigm

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

Although machine learning only started to flourish in the 1990s, it has quickly become the most popular and most successful subfield of AI, a trend driven by the availability of faster hardware and larger datasets. Machine learning is tightly related to mathematical statistics, but it differs from statistics in several important ways. Unlike statistics, machine learning tends to deal with large, complex datasets (e.g. a dataset of millions of images, each consisting of tens of thousands of pixels) for which "classical" statistical analysis such as bayesian analysis would simply be too impractical to be possible. As a result, machine learning, and especially deep learning, exhibits comparatively little mathematical theory --maybe too little-- and is very engineering-oriented. It is a hands-on discipline where ideas get proven empirically much 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 to get some idea of what machine learning algorithms really *do*. We just stated that machine learning discovers rules from "examples" of a data-processing task. 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.
- examples of the expected output. With the speech recognition task above, these would be human-generated transcripts of our sound files.
- a way to measure if the algorithm is doing a good job. This is used as a feedback signal to adjust the way the algorithm works. This adjustment step is what we call "learning".

The central problem in machine learning and deep learning is that of learning 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 your data --to "represent", or "encode" your 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 RBG format, while "make the image less saturated" is simpler in the HSV format.

Let’s make this concrete. Let’s consider an x axis, and y axis, and some points represented by their coordinates in the (x, y) system: our data, as illustrated in figure 1.3.

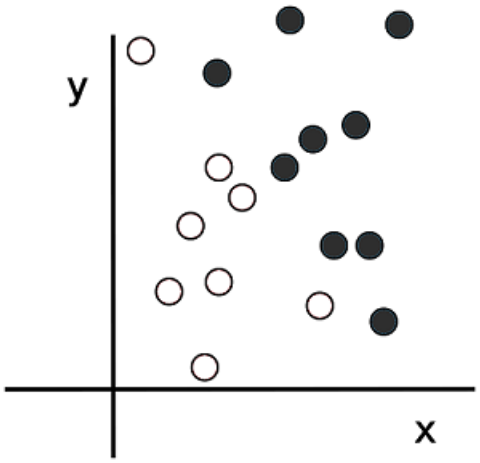

Figure 1.3 Some sample data

As you can see we have a few white points and a few black points. Let’s say we want to develop an algorithm that could take the coordinates `(x, y)` of a point, and output whether the point considered 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 if our algorithm is doing a good job could be, for instance, the percentage of points that are being correctly classified.

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

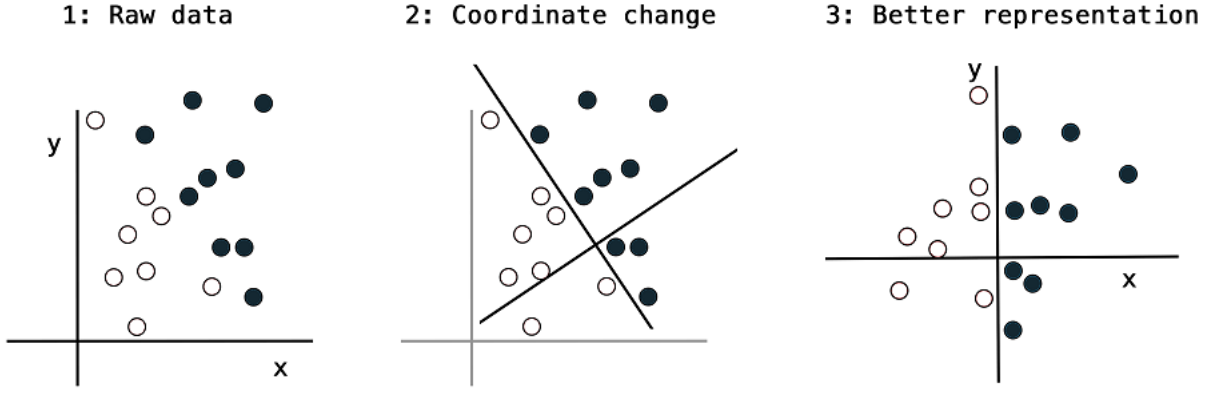

Figure 1.4 Coordinate change

In this new coordinate system, the coordinates of our points can be said to be a new "representation" of our data. And it’s a good one! With this representation, the

black/white classification problem can be expressed as a simple rule: black points are such that x 0 or "white points are such that x < 0". Our new representation basically solves the classification problem.

In this case, we defined our 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.

All machine learning consists in automatically finding such transformations that turn data into more useful representations for a given task. These operations could sometimes be coordinate changes, as we just saw, or could be linear projections (which may destroy information), translations, non-linear operations (such as select all points such that x 0), etc. Machine learning algorithms are not usually very creative in finding these transformations, they are merely searching through a predefined set of operations, called an "hypothesis space".

So that’s what machine learning is, technically: searching for useful representations of some input data, within a pre-defined space of possibilities, using guidance from some feedback signal. This simple idea allows for solving a remarkably broad range of intellectual tasks, from speech recognition to autonomous car driving.

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 which puts an emphasis on learning successive "layers" of increasingly meaningful representations. The "deep" in "deep learning" is not a reference to any kind of "deeper" understanding achieved by the approach, rather, it simply stands for this idea of successive layers of representations --how many layers contribute to a model of the data is called the "depth" of the model. Other appropriate names for the field could have been "layered representations learning" or "hierarchical representations learning". Modern deep learning often involves tens or even hundreds of successive layers of representation --and they are 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 representation of the data. Hence they are sometimes called "shallow learning".

In deep learning, these layered representations are (almost always) learned via models called "neural networks", structured in literal layers stacked one after the 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 is no evidence that the brain implements anything like the learning mechanisms in use in modern deep learning models. One might sometimes come across pop-science articles

proclaiming that deep learning works "like the brain", or was "modeled after the brain", but that is simply not the case. In fact, it would be confusing and counter-productive for new-comers to the field to think of deep learning as being in any way related to the neurobiology. You don't need that shroud of "just like our minds" mystique and mystery. So you might as well forget anything you may have read so far about hypothetical links between deep learning and biology. For our purposes, deep learning is merely a mathematical framework for learning representations from data.

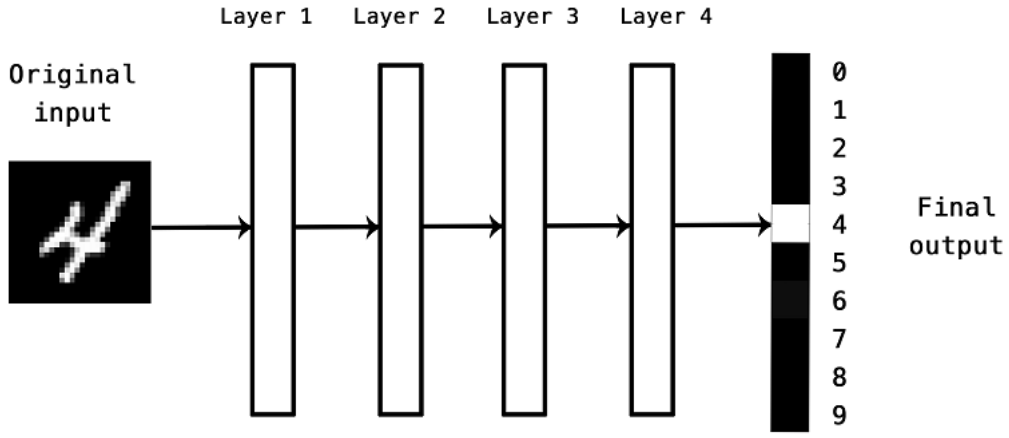

Figure 1.5 A deep neural network for digit classification

What do the representations learned by a deep learning algorithm look like? Let's look at how a 3-layer deep network transforms an image of a digit in order to recognize what digit it is:

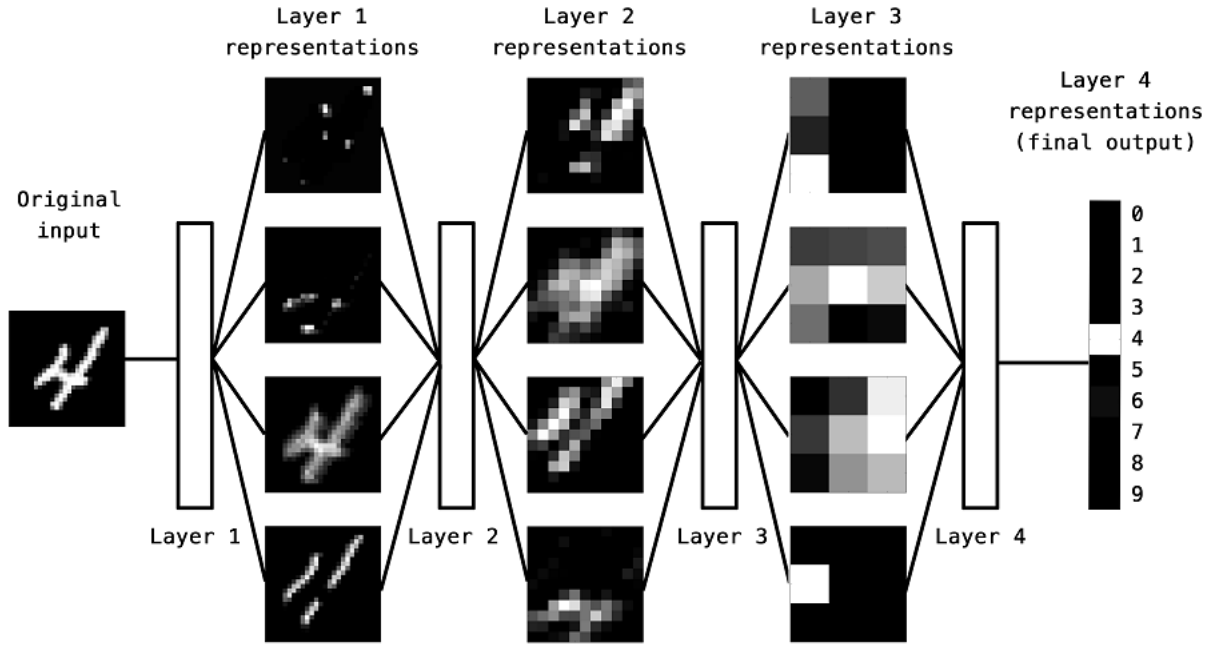

Figure 1.6 Deep representations learned by a digit classification model

As you can see, the network transforms the digit image into representations that are

increasingly different from the original image, and increasingly close to the final result. You can think of a deep network as a multi-stage information distillation operation, where information goes through successive filters and comes out increasingly "purified" (i.e. useful with regard to some task).

So that is what deep learning is, technically: a multi-stage way to learn data representations. 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 (e.g. images) to targets (e.g. 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 (called "layers"), and that these data transformations are learned by exposure to examples. Now let’s take a 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, you would say that the transformation implemented by a layer is "parametrized" by its weights. In fact, weights are also sometimes called the "parameters" of a layer. In this context, "learning" will mean finding a set of values for the weights of all layers in a network, such that the network will correctly map your 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 since modifying the value of one parameter will affect the behavior of all others!

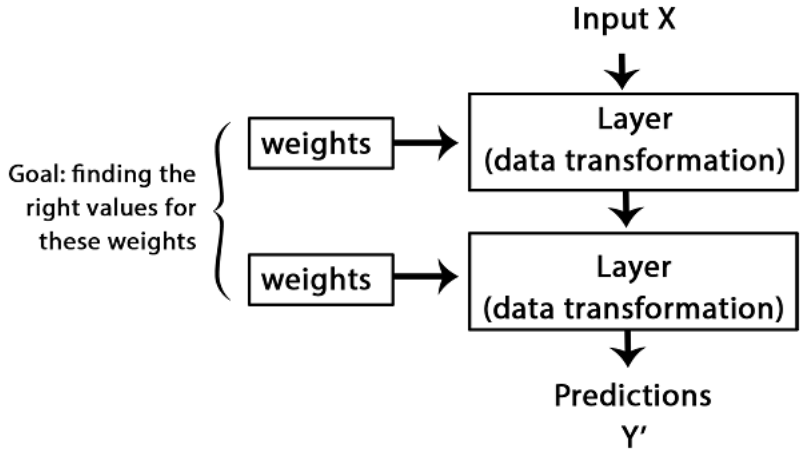

Figure 1.7 A neural network is parametrized by its weights

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 "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.

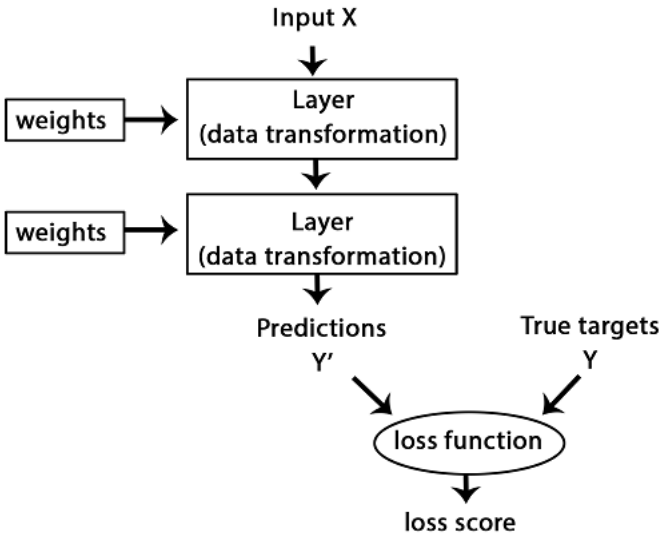

Figure 1.8 A loss function measures the quality of the network's output

The fundamental trick in deep learning is to use this score as a feedback signal to adjust the value of the weights by a little bit, in a way direction that would lower the loss score for the current example. This adjustment is the job of the "optimizer", which implements what is called the "backpropagation" algorithm, the central algorithm in deep learning. In the next chapter we will explain in more detail how backpropagation works.

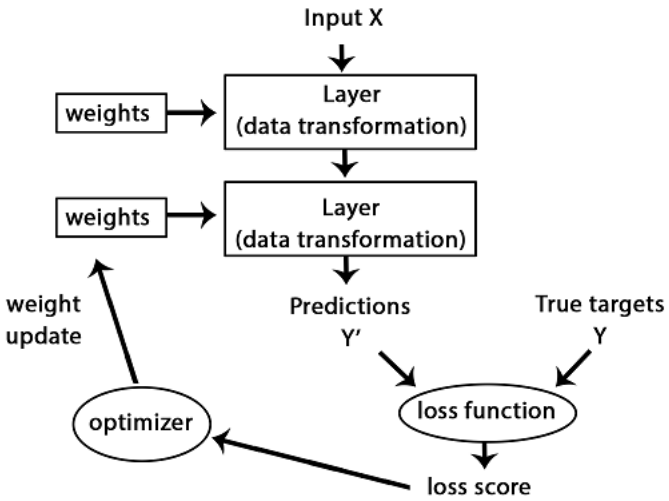

Figure 1.9 The loss score is used as a feedback signal to adjust the weights

Initially, the weights of the network are assigned random values, so the network merely implements a series of random transformation --naturally its output is very far from it should ideally be, and the loss score is accordingly very high. But with every example that the network processes, the weights get adjusted just a little in the right

direction, and the loss score decreases. This is the "training loop", which, repeated a sufficient number of times (typically tens of iterations overs thousands of examples), yields weights 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: a very simple mechanisms, which 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 all *perceptual* problems, such as "seeing" and "hearing" --problems which involve skills that seem very 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.
- Digital assistants such as Google Now or Amazon Alexa.
- Near-human level autonomous driving.
- Improved ad targeting, as used by Google, Baidu, and Bing.
- Improved search results on the web.
- Answering natural language questions.
- Master-level Go playing.

In fact, we are still just exploring the full extent of what deep learning can do. We have started applying it to an even wider variety of problems outside of machine perception and natural language understanding, such as formal reasoning. If successful, this might herald an age where deep learning assists humans in doing science, developing software, 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 actually turn out to be possible. While 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" should not 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 down 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, and a dearth of funding as a result. It started with symbolic AI in the 1960s. In these 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 also 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 in fact that we have no way to predict how long it will take, but in the 1960s and early 1970s, several experts believe it to be right around the corner (and so 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, as 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 a billion dollar a 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.

It might be that we are currently witnessing the third cycle of AI hype and disappointment --and we are still in the phase of intense optimism. The best attitude to adopt is to moderate our expectations for the short term, and make sure that people less familiar with the technical side of the field still have a clear idea of what deep learning can and cannot deliver.

1.1.8 The promise of AI

Although we might have unrealistic short-term expectations for AI, the long-term picture is looking bright. We are only just getting started in applying deep learning to many important problems in which it could prove transformative, from medical diagnoses to digital assistants. While AI research has been moving forward amazingly fast in the past five years, in large part due to a wave of funding never seen before in the short history of A.I, so far relatively little of this progress has made its way into the products and processes that make up our world. Most of the research findings of deep learning are not yet applied, or at least not applied to the full range of problems that they can solve across all industries. Your doctor doesn't use AI, your accountant doesn't use AI. Yourself, you probably don't use AI technologies in your day-to-day life. Of course, you can ask simple questions to your smartphone and get reasonable answers. You can get fairly useful product recommendations on Amazon.com. 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 just accessory to our daily lives. AI has yet to transition to become 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 at this point AI is not yet widely deployed --much like it would have been difficult to believe in the future impact of the Internet back in 1995. Back then most people did not see how the Internet was relevant to them, 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, your friend; it will answer your questions, it will help educate your kids, and it will watch over your health. It will deliver your groceries to your door and it will drive you from point A to point B. It will be your interface to an increasingly complex and increasingly information-intensive world. And even more importantly, AI will help humanity as a whole move forwards, by assisting human scientists in new breakthrough discoveries across all scientific fields, from genomics to mathematics.

On the road to get there, we might face a few setbacks, and maybe a new AI winter --in much the same way that the Internet industry got overhyped in 1998-1999 and suffered from a crash that dried up investment throughout the early 2000s. But we will 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 today.

Don't believe the short-term hype, but do believe in the long-term vision. It may take a while for AI to get 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 seen before in the history of AI, but it isn't the first successful form of machine learning. In fact, it's a safe bet to say that most of the machine learning algorithms in use in the industry today are still not deep learning algorithms. Deep learning isn't always the right tool for the job --sometimes there just isn't even data for deep learning to be applicable, and sometimes the problem is simply 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 looking like a nail. The only way not to fall into this trap is to be familiar of other approaches and practice them when appropriate.

A detailed exposure of classical machine learning approaches is outside of the scope of this book, but we will briefly go over them and describe the historical context in which they were developed. This will allow us to replace 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 one of the earliest forms of machine learning, yet it is 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 the 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 actually predates computers, and was applied by hand decades before its first computer implementation (most likely dating back to the 1950s). The Bayes Theorem and the foundations of statistics themselves date back to the 18th 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 in fact a classification algorithm rather than a regression algorithm. Much like Naive Bayes, logreg predates computing by a long time, yet it is still very useful to this day, thanks to its simple and versatile nature. It is 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 that we cover in these pages; however it helps 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 really get started. For a long time, the missing piece was a lack of an efficient way to train large neural networks. This changed in the mid-1980s, as multiple people independently rediscovered the "backpropagation" algorithm, a way to train chains of parametric operations using gradient descent optimization (later in the book we will go on to 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 together the earlier ideas of convolutional neural networks and backpropagation, and applied them to the problem of handwritten digits classification. The resulting network, dubbed "LeNet", was used by the US Post Office in the 1990s to automate the reading of ZIP codes on mail envelopes.

1.2.3 Kernel methods

As neural networks started gaining 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 SVM was developed by Vapnik and Cortes in the early 1990s at Bell Labs and published in 1995, although an older linear formulation was published by Vapnik and Chervonenkis as early as 1963.

SVM aims at solving classification problems by finding good "decision boundaries" (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.

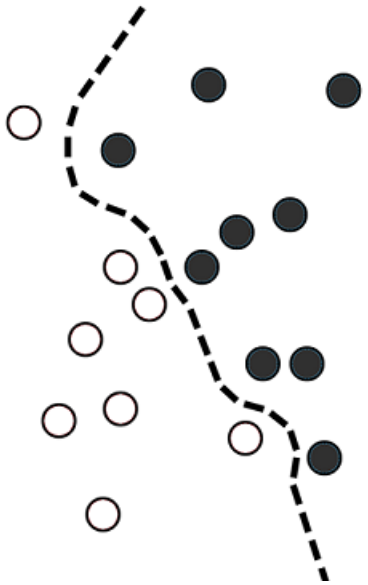

Figure 1.10 A decision boundary

SVMs proceed to find these boundaries in two steps:

- First, the data is mapped to a new high-dimensional representation where the decision boundary can be expressed as an hyperplane (if the data is two-dimensional like in our example, it would simply be a straight line).
- Then a good 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 it practice it is 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: for finding good decision hyperplanes in the new representation space, you don’t have to explicitly compute the coordinates of your points in the new space, you just need to compute the distance between pairs of points in that space, which can be done very efficiently using what is called 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 by-passing the explicit computation of the new representation. Kernel functions are typically crafted by hand rather than learned from data --in the case of 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 it well-understood and easily interpretable. Because of these useful properties, it became extremely popular in the field for a long period of time.

However, SVM proved hard to scale to large datasets and did not provide very good results for "perceptual" problems such as image classification. Since SVM is a "shallow" method, applying 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

Decisions trees learned from data started getting significant research interest in the 2000s, and by 2010 they were often preferred to kernel methods. Decision trees are flowchart-like structures that can allow to classify input data points or predict output values given inputs. They are easy to visualize and interpret.

In particular, the "Random Forest" algorithm introduced a robust and practical take on decision tree learning that involves building a large number of specialized decision trees then ensembling their outputs. Random Forests are applicable to a very wide range of problems --you could say that they are almost always the second-best algorithm for any shallow machine learning task. When the popular machine learning competition website Kaggle.com got started in 2010, Random Forests quickly became a favorite on the platform --until 2014, when Gradient Boosting Machines took over. Gradient Boosting Machines, much like Random Forests, is a machine learning technique based on ensembling weak prediction models, generally decision trees. Its use of the "boosting" technique allows it to strictly outperform Random Forests most of the time, while having very similar properties. It may be one of the best, if not the best, algorithm for dealing with non-perceptual data today. Alongside deep learning, it is one of the most commonly used technique in Kaggle competitions.

1.2.5 Back to neural networks

Around 2010, while neural networks were almost completely shunned by the scientific community at large, a number of people still working on neural networks started making 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 started winning 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 in classifying high-resolution color images into 1000 different categories after training on 2 million images. In 2011, the top-5 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-5 accuracy of 83.6% --a significant

breakthrough. The competition has been dominated by deep convolutional neural networks every year since. By 2015, we had 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, and generally all perceptual tasks. At major computer vision conferences in 2015 or 2016, it had become nearly impossible to find presentations that did not involve convnets in some form. At the same time, deep learning has also found many applications in many other types of problems, such as natural language processing. It has come to replace SVMs and decision trees in a wide range of applications. For instance, for several years CERN used decision tree-based methods for analysis of particle data from the ATLAS detector at the Large Hadron Collider (LHC), but they 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 main reason why deep learning took off so quickly is primarily that it offered better performance on many problems. But that’s not the only reason. Deep learning is also making 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 very simple transformations such as high-dimensional non-linear projections (SVM) or decision trees. But the refined representations required by complex problems generally cannot be attained by such techniques. As such, humans had to go to great length to make the initial input data more amenable to processing by these methods, i.e. they had to manually engineer good layers of representations for their data. This is what 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. This has greatly simplified machine learning workflows, often replacing very complicated multi-stage 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 representation, could shallow methods be applied repeatedly to emulate the effects of deep learning? In practice, there are fast-diminishing returns to successive application of shallow learning methods, because *the optimal first representation layer in a 3-layer model is not the optimal first layer in a 1-layer or 2-layer model*. What’s transformative about deep learning is that it allows to learn all layers of representation *jointly*, at the same time, rather than in succession ("greedily", as it is called). This is much more powerful, as it allows for very complex and abstract representations to be learned by breaking them down into long series of intermediate spaces, each space only a simple

transformation away from the previous one. These are the two essential characteristics of how deep learning lerans from data: the *incremental*, layer-by-layer way in which increasingly complex representations are developed, and the fact these intermediate incremental representations are learned *jointly*, each layer being updated both to follow the representational needs of the layer above and the needs of the layer below.

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 Kaggle competitions. Due to its highly competitive environment (some contests have thousands of entrants) and to the wide variety of machine learning problems covered, Kaggle offers a realistic way to assess what works and what doesn't. What kind of algorithm is reliably winning competitions? What tools do top entrants use?

Today, Kaggle is dominated by two approaches: gradient boosting machines, and deep learning. Specifically, gradient boosting is used for problems where structured data is available, while deep learning is used for perceptual problems such as image classification. Practitioners of the former almost always use the excellent XGB library, which offers support for the two most popular languages of data science: Python and R. Meanwhile, most of the Kaggle entrants leveraging deep learning use the Keras library, due to its easy of use, flexibility and support of Python.

So these are the two techniques that you should be the most familiar with in order to be successful in applied machine learning today: gradient boosting machines, and deep learning. In technical terms, this means that you will need to be familiar with XGB and Keras --the two libraries that are currently dominating Kaggle competitions. With this book in hand, you are already one big step closer.

1.3 Why deep learning, why now?

The two key ideas of deep learning for computer vision, namely convolutional neural networks and backpropagation, were already well-understood in 1989. The LSTM algorithm, fundamental to deep learning for time series, 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, there are three technical forces that are driving advances in machine learning:

- Hardware.
- Dataset 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 is available to try new ideas (or just scale up old ideas). Machine learning is not

mathematics or physics, where major advances can be done with a pen and a piece of paper. It is an engineering science.

So the real bottleneck throughout the 1990s and 2000s was data and hardware. But here's what happened during that time: the Internet, and gaming GPUs.

1.3.1 Hardware

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

However, typical deep learning models used in computer vision or speech recognition require many 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 into developing fast, massively parallel chips (graphical processing units, GPUs) for powering 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, a programming interface for its line of GPUs. A small number of GPUs started replacing massive clusters of CPUs in a number various highly-parallelizable applications, starting with physics modeling. Deep neural networks, consisting mostly of many small matrix multiplications, are also highly parallelizable, and around 2011, some researchers started writing CUDA implementations of neural nets --Dan Ciresan and Alex Krizhevsky were some of the first to do it.

So what happened is that the gaming market has subsidized supercomputing for the next generation of artificial intelligence applications. Sometimes, big things start as games. Today, the Nvidia Titan X, a gaming GPU that cost $1000 at the end of 2015, can deliver a peak of 6.6 TLOPS in single-precision, i.e. 6.6 trillions of 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 only takes a few days to train an ImageNet model of the sort that would have won the competition a couple 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 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 even starting to go beyond GPUs, and is investing into increasingly specialized and efficient chips for deep learning. In 2016, at its annual I/O convention, Google revealed its mysterious "TPU" project (tensor processing unit), a new chip design developed from the ground-up to run deep neural networks, reportedly 10x faster and far more energy-efficient than top-of-line GPUs.

1.3.2 Data

Artificial Intelligence 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. Besides the exponential progress in storage hardware over the past twenty 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 could not 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 is one dataset that has been a catalyst for the rise of deep learning, it is the ImageNet dataset, consisting in 2 million images hand-annotated with 1000 images categories (one category per image). But what makes ImageNet special is not just its large size, it is the yearly competition associated with. As Kaggle as been demonstrating since 2010, public competitions are a excellent way to motivate researches 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

Besides hardware and data, up until the late 2000s, we were still missing a reliable way to train very deep neural networks. As a result, neural networks were still fairly shallow, leveraging only one or two layers of representations, and so they were not able to shine against more refined shallow methods such as SVMs or 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 development of several simple but important algorithmic improvements that allowed for better gradient propagation:

- Better "activation functions", such as "rectified linear units".
- Better "weight initialization" schemes. It started with layer-wise pre-training, which was quickly abandoned.
- Better optimization schemes, such as RMSprop and Adam.

It is only when these improvements started allowing for training models with ten or more layers that deep learning really started to shine.

Finally, in 2014 and 2015, even more advanced ways to help gradient propagation were discovered, such as batch normalization and residual connections. 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 started taking the spotlight, the total venture capital investment in AI was around $19M, going almost entirely to practical applications of shallow machine learning approaches. By 2014, it had risen to a staggering $394M. Dozens of startups launched in these 3 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 $500M --the largest acquisition of an AI company in history. In 2014, Baidu started a deep learning research center in Silicon Valley, investing $300M in the project.

In fact, machine learning and in particular deep learning have become central to the product strategy of these tech giants. In 2016, Sundar Pichai, Google CEO, stated:

"Machine learning is a core, transformative way by which we're rethinking how we're doing everything. We are thoughtfully applying it across all our products, be it search, ads, YouTube, or Play. And we're in early days, but you will see us—in a systematic way—apply machine learning in all these areas."

As a result of this wave of investment, the number of people working on deep learning went in just 5 years from a few hundreds, to tens of thousands, and research progress has reached a frenetic pace. There are currently no signs that this trend is going to slow anytime soon.

1.3.5 The democratization of deep learning

One 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 auto-differentiation, greatly simplifying the implementation of new models, and by the rise of user-friendly tools such as Keras, a library that makes deep learning as easy as manipulating Lego(r) blocks. After its release early 2015, Keras has quickly become the go-to deep learning solution for large numbers of new startups, grad students, and for many 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 fashion that might not last? Will we still be using deep neural networks in 20 years?

The short answer is yes --deep learning does have several properties that justify its status as an AI revolution, and it is here to stay. We may not still be using neural networks two decades from now, but whatever we use will directly inherit from deep learning and its core concepts.

These important properties can be broadly sorted into 3 categories:

- Simplicity. Deep learning removes the need for feature engineering, replacing complex, brittle and engineering-heavy pipelines with simple end-to-end trainable models typically built using only 5 or 6 different tensor operations.
- Scalability. Deep learning is highly amenable to parallelization on GPUs or TPUs, making it capable of taking full advantage of Moore's law. Besides, 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 being the amount of parallel computational power available, which thanks to Moore's law is a fast-moving barrier).
- Versatility and reusability. Contrarily to many 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 repurposeable and thus resuable: for instance it is 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.

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. Every month we still come up with new use cases, or with engineering improvements lifting previously known limitations. Following a scientific revolution, progress generally follows a sigmoid curve: it starts with a period of fast progress than gradually stabilizes, as researchers start hitting against hard limitations and further improvements become more incremental. With deep learning in 2016, it seems that we are still in the first half of that sigmoid, and there is a lot more progress yet to come in the next few years.

2 *Before we start: the mathematical blocks of neural networks*

In this chapter, you will:

- take a look at your first working example of a neural network
- learn about tensors, the data format underlying all deep learning
- learn about tensor operations, the mathematical building blocks of neural networks
- understand the way neural networks learn from data: via gradient descent optimization

After reading this chapter, you will have an intuitive understanding of how neural networks work, and you will be able to move on to practical applications. Practical applications will start with the next chapter.

2.1 A first look at a neural network

We will now take a look at a first concrete example of a neural network, which makes use of Keras to learn to classify hand-written digits. Unless you already have experience with Keras or similar libraries, you will not understand everything about this first example right away. That is perfectly fine. In the next chapter, we will review each element in our example and explain them in detail. So do not worry if some steps seem arbitrary or look like magic! We've got to start somewhere.

The problem we are trying to solve here is to classify grayscale images of handwritten digits (28 pixels by 28 pixels), into their 10 categories (0 to 9). The dataset we will use is the MNIST dataset, a classic dataset in the machine learning community, which has been around for almost as long as the field itself and has been very 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 will see MNIST come up over and over again, in scientific papers, blog posts, and so on. You can take a look at some MNIST samples in figure 2.1.

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

Figure 2.1 MNIST sample digits

You don’t need to try to reproduce this example on your machine just now. If you wish to, you will first need to set up Keras, which is covered in section 4.2.

The MNIST dataset comes pre-loaded 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`. Our images are encoded as Numpy arrays, and the labels are simply an array of digits, ranging from 0 to 9. There is a one-to-one correspondence between the images and the labels.

Let’s have a look at the training data:

Listing 2.2 The training data

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

Let’s have a look at the test data:

Listing 2.3 The test data

```
>>> test_images.shape
```

```
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
[7 2 1 ..., 4 5 6]
```

Our workflow will be as follow: first we will present our neural network with the training data, `train_images` and `train_labels`. The network will then learn to associate images and labels. Finally, we will ask the network to produce predictions for `test_images`, and we will verify if these predictions match the labels from `test_labels`.

Let's build our network --again, you aren't supposed to understand everything about this example just yet.

Listing 2.4 The network architecture

```
from keras.models import Sequential
from keras.layers import Dense

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

The core building block of neural networks is the "layer", a data-processing module which you can conceive as a "filter" for data. Some data comes in, and comes out in a more useful form. Precisely, 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 really consists in chaining together simple layers which will implement a form of progressive "data distillation". Deep learning layers are like Lego(r) blocks for data processing.

Here our network consists in a sequence of two `Dense` layers, 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 our network ready for training, we need to pick three more things, as part of "compilation" step:

- A loss function: the is how the network will be able to measure how good a job it is doing on its training data, and thus how it will be able to steer itself in the right direction.
- An optimizer: this is 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 will 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 2.5 The compilation step

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

Before training, we will preprocess our data by reshaping it the shape that 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 2.6 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 one-hot encoded the labels, a step which we explain in chapter 3:

Listing 2.7 Preparing the labels

```
from keras.utils.np_utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

We are now ready to train our network, which in Keras is done via a call to the `fit` method of the network:

Listing 2.8 Training the network

```
>>> network.fit(train_images, train_labels, nb_epoch=5, batch_size=128)
Epoch 1/10
60000/60000 [==============================] - 9s - loss: 0.2524 - acc: 0.9273
Epoch 2/10
51328/60000 [========================>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

Two quantities are being 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 (i.e. 98.9%) on the training data. Now let’s check that our model performs well on the test set too:

Listing 2.9 Evaluating the network

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

Our test set accuracy turns out to be 97.8% --that's quite a bit lower than the training set accuracy. This is an example of "overfitting", which will be a central topic in chapter 3.

We just saw how we could build and a train a neural network to classify handwritten digits --in less than 20 lines of code. In the next chapter, we will go in detail over every moving piece we just previewed, and clarify what is really going on behind the scenes.

2.2 Data representations for neural networks

In our previous example, we started from data stored in multi-dimensional Numpy arrays (also called "tensors"). In general, all machine learning systems in our time uses tensors as their basic data structure. Tensors are fundamental to the field --so fundamental in fact, 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, a container for numbers. You may be already familiar with matrices, which are 2D tensors: tensors are merely a generalization of matrices to an arbitrary number of dimensions (note that in the context of tensors, "dimension" is often called "axis").

2.2.1 Scalars (0D tensors)

A tensor that contains only one number is called a "scalar" (or "scalar tensor", or "0D tensor"). In Numpy, a `float32` or `float64` 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`).

Listing 2.10 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 will be said to have exactly one "axis":

Listing 2.11 A Numpy vector

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

Here, this vector has 5 entries, and so will be a called a "5-dimensional vector". Do not confuse a 5-dimensional vector with a 5D tensor! A 5D vector has only one axis and has 5 dimensions along its axis, while a 5D tensor has 5 axes (and may have any number of dimensions along each axis).

2.2.3 Matrices (2D tensors)

An array of vectors is a matrix, or 2D tensor. A matrix has two axes (often denoted "lines" and "columns"). You can visually interpret a matrix as a rectangular grid of numbers:

Listing 2.12 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 our example above, `[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:

Listing 2.13 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 will generally manipulate tensors that are 0-4D, although you may go up to 5D if you process video data.

2.2.5 Key attributes

A tensor is defined by 3 key attributes:

- The number of axes it has. For instance, a 3D tensor has 3 axes, and a matrix has 2 axes. This is also called the tensor's `ndim`, throughout Python libraries.
- Its shape. This is a tuple of integers that describes how many dimensions the tensor has along each axis. For instance, our matrix example above has shape `(3, 5)`, and our 3D tensor example had shape `(3, 3, 5)`. A vector will have a shape with a single element, such as `(5,)`, while a scalar will have an empty shape, `()`.
- Its data type (usually called `dtype` throughout Python libraries). This is the type of the data contained inside the tensor; for instance a tensor's type could be `float32`, `uint8`, `float64`... In rare occasions you may witness a `char` tensor. Note that string tensors don't exist in Numpy (nor in most other libraries), since tensors live in pre-allocated contiguous memory segments, and strings, being variable-length, would preclude the use of this implementation.

To make this more concrete, let's take a look back at the data we processed in our MNIST example:

Listing 2.14 Let's load the MNIST dataset

```
from keras.datasets import mnist

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

Listing 2.15 Let's display the number of axes of the tensor `train_images`: the `ndim` attribute

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

Listing 2.16 Let's display its shape

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

Listing 2.17 Let's display 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 is an array of

60,000 matrices of 28x28 integers. Each such matrix is a grayscale image, with coefficients between 0 and 255.

Let’s display the 4th digit in this 3D tensor, using the library Matplotlib (part of the standard scientific Python suite):

Listing 2.18 Dislaying the 4th digit

```
digit = train_images[4]

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

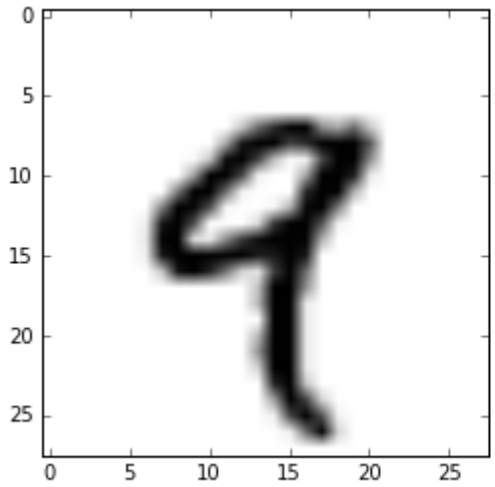

Figure 2.2 The 4th sample in our dataset

2.2.6 Manipulating tensors in Numpy

In the example above, 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 take a look at the tensor slicing operations that you can do on Numpy arrays.

The following selects digits #10 to #100 and puts them in an array of shape `(90, 28, 28)`:

Listing 2.19 Slicing a tensor

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

It is equivalent to this more detailed notation, where one specifies a start index and

stop index for the slice along each tensor axis. Note that : will simply be equivalent to selecting the entire axis.

Listing 2.20 Advanced tensor slicing

```
>>> my_slice = train_images[10:100, :, :]  # equivalent to the above example
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]  # also equivalent to the above example
>>> my_slice.shape
(90, 28, 28)
```

In general, one may select between any two indices along each tensor axis. For instance, in order to select 14x14 pixels in the bottom right corner of all images, one would do:

Listing 2.21 Advanced tensor slicing (continued)

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

It is 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 our images to patches of 14x14 pixels centered in the middle, one would do:

Listing 2.22 Advanced tensor slicing (continued)

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

2.2.7 The notion of data batch

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

Besides, deep learning models do not process an entire dataset at once, rather they break down the data into small batches. Concretely, here’s one batch of our MNIST digits, with batch size of 128:

Listing 2.23 Slicing a tensor into batches

```
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 will frequently encounter when using Keras or other deep learning libraries.

2.2.8 Real-world examples of data tensors

Let’s make data tensors more concrete still with a few examples similar to what you will encounter later on.

The data you will manipulate will almost always fall into one of the following categories:

- Vector data: 2D tensors
- Timeseries data or sequence data: 3D tensors
- Images: 4D tensors
- Video: 5D tensors

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 (i.e. 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 a few concrete examples:

- An actuarial dataset of people, where we consider for each person their age, zipcode, 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 our 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.

Time axis will always be the second axis (axis of index 1), by convention. Let’s have a 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 a day worth of data.

- A dataset of tweets, where we encode each tweet as a sequence of 140 characters out of an alphabet of 128 unique characters. Thus each tweet can be encoded as a 2D tensor of shape `(140, 128)`, and a dataset of 1M tweets can be stored in a tensor of shape `(1000000, 140, 128)`.

2.2.11 Image data

Images typically have 3 dimensions: width, height, and color depth. Although grayscale images (like our MNIST digits) only have a single color channel and could thus be stored in 2D tensors, by convention image tensors are always 3D, with a 1-dimensional color channel for grayscale images.

A batch of 128 grayscale images of size 256x256 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)`.

There are two conventions for shapes of images tensors: the TensorFlow convention and the Theano convention.

The TensorFlow machine learning framework, from Google, places the color depth axis at the end, as we just saw: `(samples, width, height, color_depth)`. Meanwhile, Theano places the color depth axis right after the batch axis: `(samples, color_depth, width, height)`. With the Theano convention, our examples above 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 will need 5D tensors. A video can be understood as a sequence of frames, each frame being a color image. Since each frame can be stored in a 3D tensor `(width, height, color_depth)`, then a sequence of frames can be stored in 4D tensor `(frames, width, height, color_depth)`, and thus a batch of different videos can be stored in a 5D tensor of shape `(samples, frames, width, height, color_depth)`.

For instance, a 60-second, 256x144 YouTube video clip sampled at 4 frames per second would have 240 frames. A batch of 4 such video clips would be stored in a tensor of shape `(4, 240, 256, 144, 3)`. That's a total of 106,168,320 values! If the `dtype` of the tensor is `float32`, then each value is stored in `32 bits`, so the tensor would represent 425MB. Heavy! Videos you encounter in real life are much lighter because they are not stored in `float32` and they are typically compressed by a large factor (e.g. in the MPEG format).

2.3 The gears of neural networks: tensor operations

Much like any computer program can be ultimately reduced to a small set of binary operations on binary inputs (such as AND, OR, NOR, etc.), 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 is possible to add tensors, multiply tensors, and so on.

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

Listing 2.24 A Keras layer

```
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, this function (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 simply `max(x, 0)`.

Although this section deals entirely with linear algebra expressions, you won’t find any mathematical notation here. We’ve found that mathematical concepts could be more readily mastered by programmers with no mathematical background if they were expressed as short Python snippets instead of mathematical equations. So we will use Numpy code all along.

2.3.1 Element-wise operations

The "relu" operation and the addition are element-wise operations, i.e. operations that are applied independently to each entry in the tensors considered. This means that these operations are highly amenable to massively parallel implementations (so-called "vectorized" implementations, a term which come from the "vector processor" supercomputer architecture from the 1970-1990 period). If you wanted to write a naive Python implementation of an element-wise operation, you would use a `for` loop:

Listing 2.25 A naive implementation of an element-wise "relu" operation

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

    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
```

```
            x[i, j] = max(x[i, j], 0)
    return x
```

Same for addition:

Listing 2.26 A naive implementation of element-wise addition

```
def naive_add(x, y):
    # x and y are 2D Numpy tensors
    assert len(x.shape) == 2
    assert x.shape == y.shape

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

On the same principle, you can 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 BLAS implementation (Basic Linear Algebra Subprograms) if you have one installed, which you should. BLAS are low-level, highly-parallel, efficient tensor manipulation routines typically implemented in Fortran or C.

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

Listing 2.27 Native element-wise operation in Numpy

```
import numpy as np

# element-wise addition
z = x + y

# element-wise relu
z = np.maximum(z, 0.)
```

2.3.2 Broadcasting

In our naive implementation of `naive_add` above, we only support the addition of 2D tensors with identical shapes. But in the `Dense` layer introduced earlier, we were adding a 2D tensor with a vector. What happens with addition when the shape of the two tensors being added differ?

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

1) axes are added to the smaller tensor to match the `ndim` of the larger tensor (called broadcast axes). 2) the smaller tensor is then 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 `0  i < 32`. At his point we can proceed to add `x` and `Y`, since they have the same shape.

In terms of implementation, no new 2D tensor would actually be created since that would be terribly inefficient, so the repetition operation would be entirely virtual, i.e. it would be happening 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:

Listing 2.28 A naive implementation of matrix-vector addition

```
def naive_add_matrix_and_vector(x, y):
    # x is a 2D Numpy tensor
    # y is a Numpy vector
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]

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

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

You can thus do:

Listing 2.29 Applying the element-wise `maximum` operation to two tensors of different shapes via broadcasting

```
import numpy as np

# x is a random tensor with shape (64, 32, 10)
x = np.random.random((64, 3, 32, 10))
# y is a random tensor with shape (32, 10)
y = np.random.random((32, 10))

# the output has shape (64, 3, 32, 10) like x
z = np.maximum(x, y)
```

2.3.3 Tensor dot

The dot operation, also called "tensor product" (not to be confused with element-wise product) is the most common, most useful of tensor operations. Contrarily to element-wise operations, it combines together entries in the input tensors.

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 is done using the standard `dot` operator:

Listing 2.30 Numpy `dot` operations between two tensors

```
import numpy as np

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

or:

Listing 2.31 Keras `dot` operations between two tensors

```
from keras import backend as K

z = K.dot(x, y)
```

In mathematical notation, you would not 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 is computed as such:

Listing 2.32 A naive implementation of `dot`

```
def naive_vector_dot(x, y):
    # x and y are Numpy vectors
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]

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

You will 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 dot product.

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

Listing 2.33 A naive implementation of matrix-vector `dot`

```
import numpy as np

def naive_matrix_vector_dot(x, y):
    # x is a Numpy matrix
    # y is a Numpy vector
    assert len(x.shape) == 2
```

```
    assert len(y.shape) == 1
    # the 1st dimension of x must be
    # the same as the 0th dimension of y!
    assert x.shape[1] == y.shape[0]

    # this operation returns a vector of 0s
    # with the same shape as y
    z = np.zeros_like(y)
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[j] += x[i, j] + y[j]
    return z
```

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

Listing 2.34 Alternative naive implementation of matrix-vector `dot`

```
def naive_matrix_vector_dot(x, y):
    z = np.zeros_like(y)
    for j in range(x.shape[1]):
        z[j] = naive_vector_dot(x[:, j], y)
    return z
```

Note that as soon as one of the two tensors has a `ndim` higher than 1, `dot` is no longer symmetric, which is to say that `dot(x, y)` is not the same as `dot(y, x)`.

Of course, dot product generalizes to tensors with 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[1], y.shape[0])`, where coefficients are the vector products between the rows of x and the columns of y. Here's the naive implementation:

Listing 2.35 A naive implementation of matrix-matrix `dot`

```
def naive_matrix_dot(x, y):
    # x and y are Numpy matrices
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    # the 1st dimension of x must be
    # the same as the 0th dimension of y!
    assert x.shape[1] == y.shape[0]

    # this operation returns a matrix of 0s
    # with a specific shape
    z = np.zeros((x.shape[1], y.shape[0]))
    # we iterate over the rows of x
    for i in range(x.shape[1]):
        # and over the columns of y
        for j in range(y.shape[0]):
            row_x = x[:, j]
            column_y = y[i, :]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

To understand dot product shape compatibility, it helps to visualize the input and output tensors by aligning them in the following way:

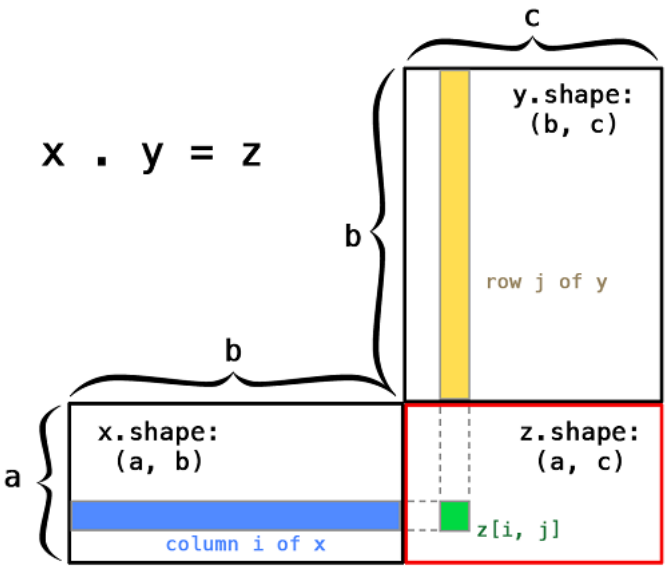

Figure 2.3 Matrix dot product box diagram

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 will likely be drawing such diagrams a lot.

More generally, you can take the dot product between higher-dimensional tensors, following the same rules for shape compatibility as outlined above 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 is essential to understand is tensor reshaping. Although not used in the `Dense` layers in our first neural network example, we used it when we pre-processed the digits data before feeding them into our network:

Listing 2.36 MNIST image tensor reshaping

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

Reshaping a tensor means that re-arranging its rows and columns so as to match a target shape. Naturally the reshaped tensor will have the same total number of coefficients as the initial tensor. Reshaping is best understood via simple examples:

Listing 2.37 Tensor reshaping examples

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

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

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

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

Listing 2.38 Matrix transposition

```
>>> x = np.zeros((300, 20))  # creates an all-zeros matrix of shape (300, 20)
>>> x = np.transpose(x)
>>> print(x.shape)
(20, 300)
```

2.3.5 Geometric interpretation of tensor operations

Because the contents of the tensors being manipulated by tensor operations can be interpreted as being coordinates of points in some geometric space, all tensor operations have a geometric interpretation.

For instance, let’s consider addition. We will start from the following vector:

`A = [0.5, 1.0]`

It is a point in a 2D space:

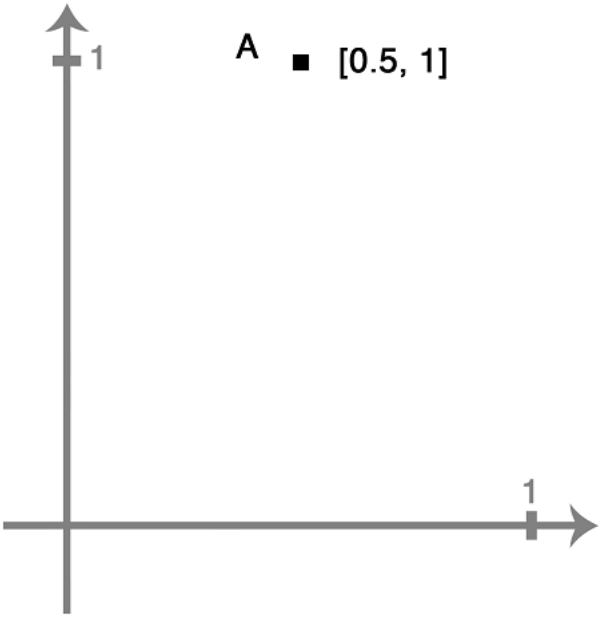

Figure 2.4 A point in a 2D space

It is common to picture a vector as an arrow linking the origin to the point:

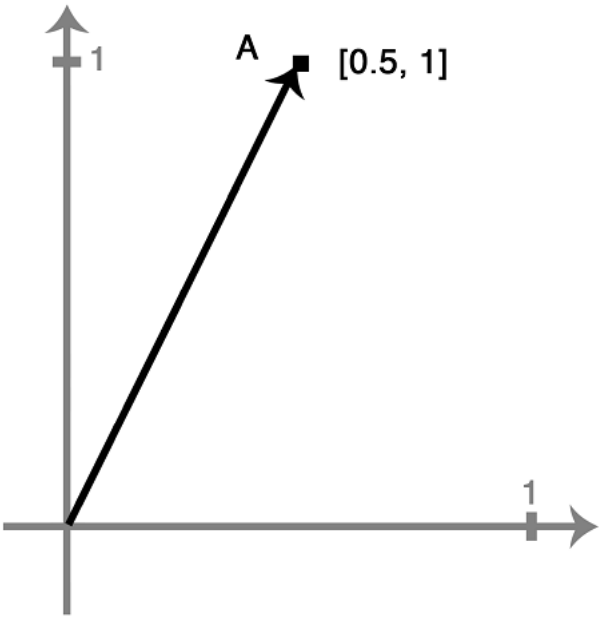

Figure 2.5 A point in a 2D space pictured as an arrow

Let's consider a new point, `B = [1, 0.25]`, which we will add to the previous one. This is done geometrically by simply chaining together the vector arrows, with the resulting location being the vector representing the sum of the previous two vectors:

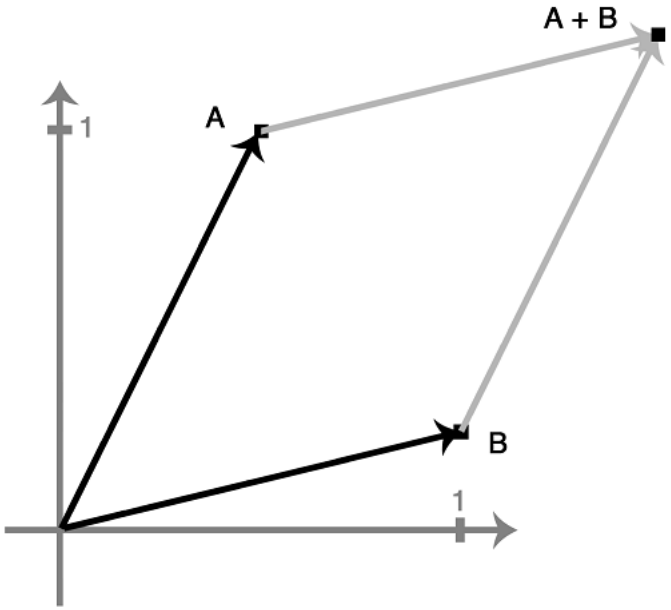

Figure 2.6 Geometric interpretation of the sum of two vectors

In general, elementary geometric operations such as affine transformations, rotations, scaling, etc. can be expressed as tensor operations. For instance, a rotation of a 2D vector by an angle theta can be achieved via dot product with a 2x2 matrix `R = [u, v]` where `u` and `v` and 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 in chains of tensors operations, and that all these tensor operations are really 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, a red one and a blue one. Superpose them. Now crumple them together into a small paper 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 to 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.

Uncrumpling paper balls is what all machine learning is about: finding neat representations for complex, highly folded data manifolds. At this point, you should already have a pretty good intuition as to why deep learning excels at it: it takes the

approach of incrementally decomposing a very 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 disentangle the data a little bit --and a deep stack of layers makes tractable an extremely complicated disentanglement process.

2.4 The engine of neural networks: gradient-based optimization

As we saw the previous section, each neural layer from our first network example transforms its input data as:

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

In this expression, W and b are tensors which are attributes of the layer. They are called the "weights", or "trainable parameters" of the layer. 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 are is reason to expect that `relu(dot(W, input) + b)`, when W and b are random, would yield any useful representations. The resulting representations are terrible --but they are 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 is called a *training loop*, which schematically looks like this:

```
Repeat as long as needed:
    1) draw a batch of training samples x and corresponding targets y
    2) run the network on x (this is called "forward pass"),
        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
        would have slightly reduce the loss on this batch.
```

You eventually end up with a network that has a very low mismatch between predictions y_pred and expected targets y, i.e. a network that 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 really simple.

Step 1 sounds easy enough --just pure Python 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've learned in the previous section. The difficult part here is how to do step 4. Given an individual weight coefficient in the network, how can we 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 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 re-run the forward pass, the loss increases to `0.6`. But if you lower the coefficient to `0.25`, the loss gets down to `0.4`. In this case it seems like updating the coefficient by `-0.05` would contribute to minimizing the loss. This would have to be repeated for all coefficients in the network.

However, such an approach would be horribly inefficient, since you would need to compute two forward passes (which are expensive) for every individual coefficient (and there are many). A much better approach is to leverage 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. We can them move the coefficients in the direction opposite to the gradient, thus decreasing the loss.

If you already know what "differentiable" means and what a "gradient" is, you can skip to the section "Stochastic gradient descent". Otherwise, the two sections below 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 an small `epsilon_y` change to `y`.

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

Besides, since our function is "smooth" (i.e. its curve doesn't have any abrupt angles), when `epsilon_x` is "small enough", then around a certain point `p`, it is possible to approximate `f` as a linear function of slope `a`, so that `epsilon_y` becomes `a * epsilon_x`:

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

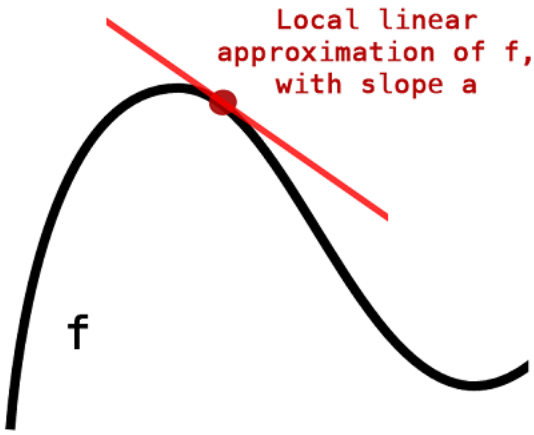

Figure 2.7 Derivative of f in p

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

The slope a is called the "derivative" of `f` in `p`. If `a` is negative, it means that a small change of `x` around `p` would result in a decrease of `f(x)` (like in our figure), and if `a` is positive, then a small change in `x` would result in an increase of `f(x)`. Further, the absolute value of `a` (the "magnitude" of the derivative) tells us how "fast" this increase or decrease would happen.

For every differentiable function `f(x)` ("differentiable" just means "can be derived", e.g. smooth continuous functions can be derived), there exists a derivative function `f'(x)` which 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`, etc.

If you are 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 lower the value of `f(x)`, you just need to move `x` by a little bit in the direction opposite to the derivative.

2.4.2 Derivative of a tensor operation: the gradient

A "gradient" is the derivative of a tensor operation. It is the generalization of the concept of derivative to functions of multi-dimensional inputs, i.e. to functions that take tensors as inputs.

Consider an input vector `x`, a matrix `W`, a target `y` and a loss function `loss`. We use `W` to compute a target candidate `y_pred`, and we 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 x and y is frozen, then this can be interpreted as a function mapping values of `W` to loss values:

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

Let’s say that the current value of `W` is `W0`. Then the derivative of `f` in the point `W0`, is a tensor `gradient(f)(W0)` with the same shape as `W`, where each coefficient `gradient(f)(W0)[i, j]` indicates the direction and magnitude of the change in `loss_value` you would observe when modifying `W0[i, j]`. That tensor `gradient(f)(W0)` is the *gradient* of `f(W) = loss_value` in `W0`.

We saw earlier that the derivative of a function `f(x)` of a single coefficient could be interpreted as the slope of the curve of `f`. Likewise, `gradient(f)(W0)` can be interpreted as the tensor describing the *curvature* of `f(W)` around `W0`.

For this reason, in much the same way that, for a function `f(x)`, you could lower the value of `f(x)` by moving `x` by a little bit in the direction opposite to the derivative, with a function `f(W)` of a tensor, you can lower `f(W)` by moving `W` in the direction opposite to the gradient, e.g. `W1 = W0 - step * gradient(f)(W0)` (where `step` is a small scaling factor). That simply means "going opposite to the curvature", which intuitively should get you lower on the curve. Note that the scaling factor `step` is needed because `gradient(f)(W0)` only approximates the curvature when you are close to `W0`, so you don't want to get too far away from `W0`.

2.4.3 Stochastic gradient descent

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

Applied to a neural network, that would mean finding analytically the combination of weights values that yields the smallest possible loss function. This would be done by solving the equation: `gradient(f)(W) = 0` for `W`. This is a polynomial equation of N variables, where N is the number of coefficients in the network. While it would be possible to solve such an equation for for N = 2 or N = 3, it is intractable for real neural networks, where the number of parameters is never below a few thousands and can often get to several tens of millions.

So instead, we use the four-step algorithm outlined at the beginning of this section: we modify the parameters little by little based on the current loss value on a random batch of data. Since we are dealing with a differentiable function, we can compute its gradient, which gives us an efficient way to implement step 4: if we update the weights in the direction opposite to the gradient, the loss will get a little lower every time.

```
Repeat as long as needed:
    1) draw a batch of training samples x and corresponding targets y
    2) run the network on x (this is called "forward pass"),
        obtain predictions y_pred
    3) compute the "loss" of the network on the batch,
        a measure of the mismatch between y_pred and y
    4.1) compute the gradient of the loss with regard to
        the parameters of the network (this is called "backward pass")
    4.2) move the parameters a little in the direction opposite to
        the gradient, e.g. `W -= step * gradient`,
        thus lowering the loss on the batch by a bit.
```

Easy enough! What we have just described is called "mini-batch Stochastic Gradient Descent" (minibatch SGD). The term "stochastic" refers to the fact that each batch of

data is drawn at random ("stochastic" is a scientific synonym of "random"). Let's visualize what happens in 1D, when our network has only one parameter and we only have one training sample:

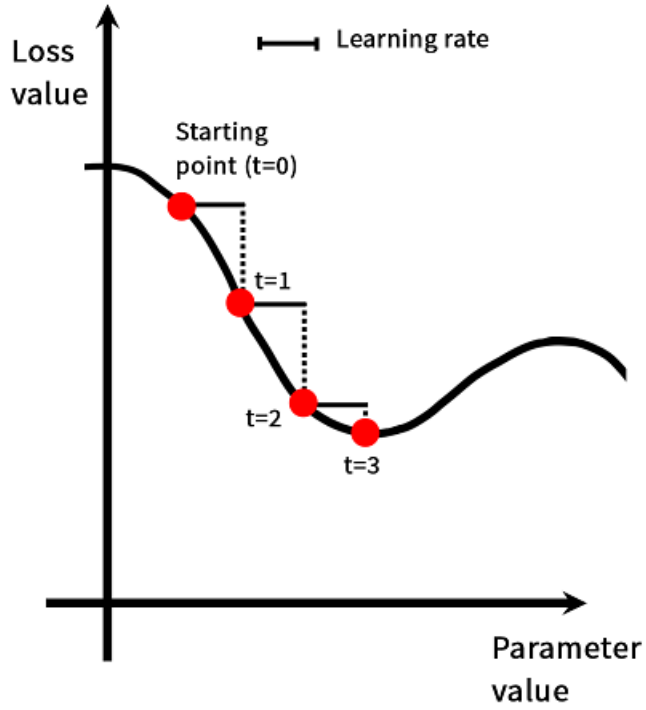

Figure 2.8 SGD in 1D space

As you can see from this figure, intuitively it is 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 besides, it could get stuck in a local minimum. If `step` is too large, your updates may end up getting your to completely random locations on the curve.

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

Additionally, there exists 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", but also "Adagrad", "RMSprop", and several others. Such variants are known as "optimization methods" or "optimizers". In particular, the concept of *momentum*, which is used is many of these variants, deserves your attention. Momentum addresses two issues with SGD: convergence speed, and local minima. Consider the following curve of a loss as a function of a network parameter:

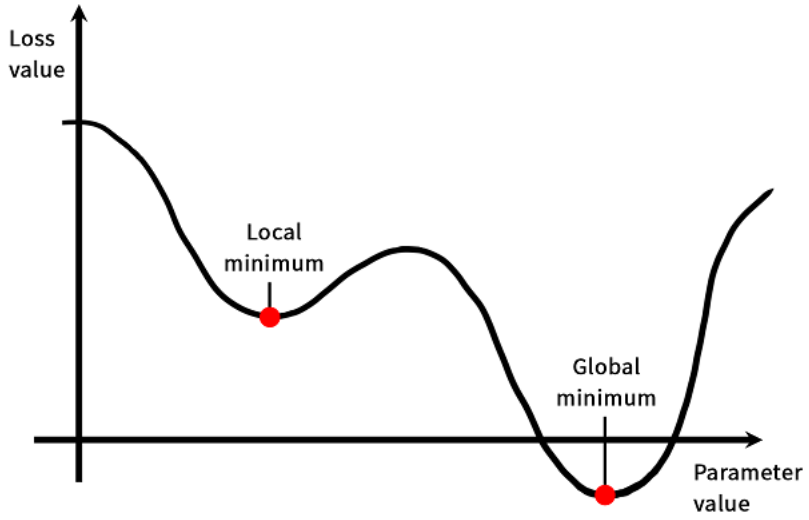

Figure 2.9 A local minimum and a global minimum

As you can see, around a certain parameter value, there is a "local minimum": around that point, going left would result in the loss increasing, but so would going right. If the parameter considered was 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.

A way to avoid such issues is to use "momentum", which draws inspiration from physics. A useful mental image here would be to imagine the optimization process as a small ball rolling down the loss curve. If it has enough "momentum", the ball would not get stuck in a ravine and would end up at the global minimum. Momentum is implemented by moving the ball at each based not only on the current slope value (i.e. current acceleration) but also based 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 based on the previous parameter update, such as in this naive implementation:

Listing 2.39 A naive implementation of gradient descent with momentum

```
past_velocity = 0.
momentum = 0.1  # a constant momentum factor
while loss > 0.01:  # optimization loop
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum + learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

2.4.4 Chaining derivatives: the backpropagation algorithm

In the above algorithm we just casually assumed that since our function was differentiable, we could explicitly compute its derivative. In practice, a neural network function consists of many tensor operations chained together, each of them having a simple, known derivative: for instance, this would be a network `f` composed of three tensor operations 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 backwards 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 implement their networks in modern frameworks which are capable of "symbolic differentiation", such as TensorFlow. It 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) which 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 will never have to implement the backpropagation algorithm by hand. For this reason, we won't waste your time and focus on deriving the exact formulation of the backpropagation algorithm in these pages.

2.4.5 In summary: training neural networks using gradient descent

At this point, you know everything there is to know about how neural networks "learn". "Learning" simply means a finding a combination of model parameters that minimizes a loss function for a given set of training data samples and their corresponding targets. This is done by drawing random batches of data samples and their targets, and computing the *gradient* of the network parameters with respect to the loss on the batch. The network parameters are then moved "a bit" (the magnitude of the move is defined by the *learning rate*) in the direction opposite to the gradient. The whole process is made possible by the fact that neural networks are chains of differentiable tensor operations, and thus it is possible to apply the *chain rule* of derivation to find the gradient function mapping the current parameters and current batch of data to a gradient value.

Two key concepts that you will see come up a lot in the future chapters are that of "loss" and "optimizer". These are the two things you need to define before you start feeding data into a network. The "loss" is the quantity that you will attempt to minimize

during training, so it should represent a measure of success on the task you are trying to solve. The "optimizer" specifies the exact way in which the gradient of the loss will be used to update parameters: it could be the "RMSprop" optimizer, "SGD with momentum", etc.

2.5 Looking back on our first example

You've reached the end of this chapter, and you should already have a general understanding of what is going on behind the scenes in a neural network. Let's go back to our first example and review each piece of it in the light of what you've learned in the previous three sections.

This was our input data:

Listing 2.40 MNIST 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 our 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:

Listing 2.41 Our network

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

Now you understand that this network consist in a chain of two `Dense` layers, that each layer just applies a few simple tensor operations to the input data, and that these operations involve weight tensors. Weights tensors, which are attributes of the layers, are where the "knowledge" of the network persists.

This was the network compilation step:

Listing 2.42 The compilation step

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

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

Finally, this was the training loop:

Listing 2.43 The training loop

```
network.fit(train_images, train_labels, nb_epoch=5, batch_size=128)
```

Now you understand what is going on when you call `fit`: the network will start iterating on the training data in mini-batches of 128 samples, 5 times over (each iteration over all of 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 in total (469 per epoch), and the loss of the network will be sufficiently low, so that your 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.

3 *Getting started with neural networks*

This chapter is designed to get you started with using neural networks to solve real problems. You will consolidate the knowledge you gained from our very first practical example, and you apply what you have learned to three new problems covering the three most common use cases of neural networks: binary classification, multi-class classification, and scalar regression.

In this chapter, you will:

- Take a closer look at the core components of neural networks we introduced in our first example: layers, network, objective function and optimizer
- Follow a step-by-step guide to setting up a Ubuntu workstation for deep learning, with TensorFlow, Theano, Keras and GPU support
- Get a quick introduction to Keras, the Python deep learning library which we will use throughout the book
- Dive into three introductory examples of how to use neural networks to solve real problems:
    - classifying movie reviews into positive and negative ones (binary classification)
    - classifying news wires by their topic (multi-class classification)
    - estimating the price of a house given real estate data (regression)

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

3.1 Anatomy of a neural network

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

- *Layers*, which are combined into a *network*
- The *Input data* and corresponding *targets*
- The *loss function*, which defines the feedback signal which is used for learning
- The *optimizer*, which determines how the learning proceeds.

You can visualize their interaction in the following way: the *network*, composed of *layers* chained together, maps the *input data* into *predictions*. The *loss function* then compares these predictions to the *targets*, producing a loss value, a measure how well the predictions of the network match what was expected. The *optimizer* uses this loss value to update the weights of the network.

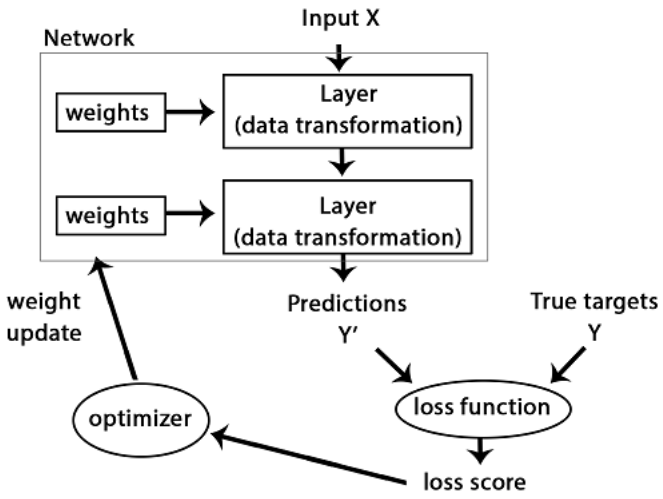

Figure 3.1 Relationship between network, layers, loss function and optimizer

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

3.1.1 Layers: the Lego(r) blocks of deep learning

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

Different layers are appropriate for different tensor formats and different types of data processing. For instance, simple vector data, stored in 2D tensors of shape `(samples, features)`, is often processed by "fully-connected" layers, also called

"densely-connected" or "dense" layers (the `Dense` class in Keras). Sequence data, stored in 3D tensors of shape `(samples, timesteps, features)`, is typically processed by "recurrent" layers such as a `LSTM` layer. Image data, stored in 4D tensors, is usually processed by `Convolution2D` layers.

You can think of layers as the Lego(r) blocks of deep learning, a metaphor which is made explicit by frameworks like Keras. Building deep learning models in Keras is done by clipping together compatible layers to form useful data transformation pipelines. The notion of "layer compatibility" here refers specifically to the fact that every layer will only accept input tensors of a certain shape, and will return output tensors of a certain shape. Consider the following example:

Listing 3.1 A layer

```
from keras.layers import Dense

layer = Dense(32, input_shape=(784,))
```

We are creating a layer that will only accept as input 2D tensors where the first dimension is `784` (the zero-th dimension, the batch dimension, is unspecified and thus any value would be accepted). And this layer will return a tensor where the first dimension has been transformed to be `32`:

Listing 3.2 Our layer's output shape

```
>>> layer.output_shape
(None, 32)
```

Thus this layer can only be connected to an upstream that expects 32-dimensional vectors as its input. When using Keras you don't have to worry about compatibility, because the layers that you add to your models are dynamically built to match the shape of the incoming layer. For instance, if you write the following:

Listing 3.3 Automatic shape inference in action

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(32, input_shape=(784,)))
model.add(Dense(32))
```

The second layer did not receive an input shape argument --instead it automatically inferred its input shape as being the output shape of the layer that came before.

3.1.2 Models: networks of layers

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

However, as you move forward, you will be exposed to a much broader variety of network topologies. Some common ones include:

- Two-branch networks
- Multi-head networks
- Inception blocks

The topology of a network defines an *hypothesis space*. You may remember that in chapter one, we defined machine learning as "searching for useful representations of some input data, within a pre-defined space of possibilities, using guidance from some feedback signal". By chosen a network topology, you have constrained your "space of possibilities" (hypothesis space) to a specific series of tensor operations, mapping input data to output data. What you will then be "searching" for, is a good set of values for the weight tensors involved in these tensor operations.

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

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

Once the network architecture is defined, it remains to pick two things:

- The loss function (or objective function), the quantity that will be minimized during training. It represents a measure of success on the task at hand.
- The optimizer, which determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent.

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

Picking the right objective function for the right problem is extremely important: your network will take any shortcut it can to minimize it, so if the objective doesn’t fully correlate with actual success on the task at hand, your network will end up doing things you may not have wanted. Imagine a stupid omnipotent AI trained via stochastic gradient descent, with the poorly-chosen objective function of "maximizing the average

well-being of all humans alive". To make its job easier, this AI might choose to kill all humans except a few, and focus on the well-being on the remaining ones --since average well-being is not affected by how many humans are left. That might not be what you intended! Just remember that all neural networks you build will be just as ruthless in lowering their loss function --so choose the objective wisely.

Thankfully, when it comes to common problems such as classification, regression, or sequence predictions, there are simple guidelines that you follow to chose the right loss: for instance, you will use binary crossentropy for a two-class classification problem, categorical crossentropy for a many-class classification problem, mean squared error for a regression problem, CTC for a sequence learning problem… only when you are working on truly new research problems will you have to develop your own objective functions.

In the next few chapters, we will detail explicitly which loss functions to pick, for a wide range of common tasks.

3.2 Introduction to Keras

Throughout this book, our code examples will use Keras. Keras is a deep learning framework for Python which provides a convenient way to define and train almost any kind of deep learning model. Keras was initially developed for researchers, aiming at enabling fast experimentation.

Keras has the following key features:

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

Keras is distributed under the permissive MIT license, which means that it can be freely used in commercial projects. It is compatible with any version of Python from 2.7 to 3.5. Its documentation is available at keras.io.

Keras has over 50,000 users, ranging from academic researchers and engineers at both startups and large companies, to graduate students and hobbyists. Keras is used at Google, Netflix, Yelp, CERN, at dozens of startups working on a wide range of problems (even a self-driving startup: Comma.ai)…

3.2.1 Keras, TensorFlow, and Theano

Keras is a model-level library, providing high-level building blocks for developing deep learning models. It does not handle itself low-level operations such as tensor manipulation and differentiation. Instead, it relies on a specialized, well-optimized tensor library to do so, serving as the "backend engine" of Keras. Rather than picking one single tensor library and making the implementation of Keras tied to that library, Keras handles the problem in a modular way, and several different backend engines can be plugged seamlessly into Keras. Currently, the two existing backend implementations are the *TensorFlow* backend and the *Theano* backend. In the future, it is possible that Keras will be extended to work with even more engines, if new ones come out that offer advantages over TensorFlow and Theano.

TensorFlow and Theano are two of the fundamental platforms for deep learning today. Theano is developed by the MILA lab at Universite de Montreal, while TensorFlow is developed by Google. Any piece of code that you write with Keras can be run with TensorFlow or with Theano without having to change anything: you can seamlessly switch between the two during development, which often proves useful, for instance if of the two engines proves to be faster for a specific task.

Via TensorFlow (or Theano), Keras is able to run on both CPU and GPU seamlessly. When running on CPU, TensorFlow is itself wrapping a low-level library for tensor operations called Eigen. On GPU, TensorFlow wraps a library of well-optimized deep learning operations called cuDNN, developed by NVIDIA.

3.2.2 Developing with Keras: a quick overview

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

- Define your training data: input tensors and target tensors.
- Define a network of layers (a "model") that will map your inputs to your targets.
- Configure the learning process by picking a loss function, an optimizer, and some metrics to monitor.
- Iterate on your training data.

There are two ways to define a model: using the `Sequential` class (only for linear stacks of layers, which is the most common network architecture by far), and the "functional API" (for directed acyclic graphs of layers, allowing to build completely arbitrary architectures).

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

Listing 3.4 A network definition using the Sequential model

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(32, activation='relu', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

And here's the same model defined using the functional API. With this API, you are manipulating the data tensor that the model processes, and applying layers to this tensor as if they were functions. A detailed guide to what you can with the functional API can be found in chapter 12.

Listing 3.5 A network definition using the function API

```
from keras.models import Model
from keras.layers import Dense, Input

input_tensor = Input(shape=(784,))
x = Dense(32, activation='relu')(input_tensor)
output_tensor = Dense(10, activation='softmax')(x)

model = Model(input=input_tensor, output=output_tensor)
```

Once your model architecture is defined, it doesn't matter whether you used a `Sequential` model or the functional API: all following steps are the same.

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

Listing 3.6 Defining a loss function and an optimizer

```
from keras.optimizers import RMSprop

model.compile(optimizer=RMSprop(lr=0.001), loss='mse', metrics=['accuracy'])
```

Lastly, the learning process itself consists in passing Numpy arrays of input data (and the corresponding target data) to the model via the `fit()` method, similarly to what you would do in Scikit-Learn or several other machine learning libraries:

Listing 3.7 Training a model

```
model.fit(input_tensor, target_tensor, batch_size=128, nb_epochs=10)
```

Over the next few chapters, you will build a solid intuition as to what type of network

architectures work for different kinds of problems, how to pick the right learning configuration, and how to tweak a model until it gives you the results you want to see. We'll start with three basic examples in the next three sections: a two-class classification example, a many-class classification example, and a regression example.

3.3 Setting up a deep learning workstation

3.3.1 Preliminary considerations

Before you can get started developing deep learning applications, you need to set up your workstation. It is highly recommended, though not strictly necessary, to run deep learning code on a modern NVIDIA GPU. Some applications, in particular image processing with convolutional networks, will be excruciatingly slow on CPU, even with a fast multi-core CPU. And even for applications that can realistically be run on CPU, you would generally observe a 5x to 20x speedup by using a modern GPU. If you don't want to install a GPU on your machine, you could alternatively consider running your experiments on a AWS EC2 GPU instance. But note that that GPUs on EC2 GPU instances are older models and may not prove cost-effective.

Also, it is better for you to be using a Unix workstation. While it is technically possible to use Keras on Windows (using the Theano backend, since Theano has experimental Windows support whereas TensorFlow does not), we don't recommend it. In the installation instructions that follow, we will consider a Ubuntu machine. If you are a Windows user, the simplest solution to get everything running is to set up a Ubuntu dual boot on your machine. It may seem like a hassle, but using Ubuntu will save you a lot of time and a lot of trouble in the long run.

Note that in order to use Keras, you need to install Theano *or* TensorFlow (or both, if you want to be able to switch back and forth between the two backends). Here we will take the time to install both.

Lastly, note that if you are a Docker user, you can by-pass a lot of manual installation steps by simply using the Docker image that we provide for Keras at `github.com/fchollet/keras/tree/master/docker`.

3.3.2 What is the best GPU for deep learning?

This question gets asked a lot. If you are going to buy a GPU, which one should you choose? The first thing to note it that it would have to be a Nvidia GPU. Nvidia is the only graphics computing company to have invested into deep learning so far, and modern deep learning frameworks can only run on Nvidia cards.

In general, I would recommend the Titan X (Pascal architecture) as the best card on the market for deep learning. For lower budgets, you might want to consider the GTX 1060.

3.3.3 Overview of the installation process

The process of setting up a deep learning workstation is fairly involved, and consists in the following steps, which we will cover in detail:

- Installing the Python scientific suite: Numpy and Scipy, and making sure that you have a BLAS installed so that your models run fast on CPU.
- Installing two extras packages that come in handy when using Keras: HDF5 (for saving large neural network files) and Graphviz (for visualizing neural network architectures).
- Making sure that your GPU can run deep learning code, by installing CUDA drivers and cuDNN.
- Installing a backend for Keras: TensorFlow, or Theano.
- Installing Keras itself.

It may seem like a daunting process. In fact, the only difficult part is to set up GPU support --otherwise, the whole process can be done in a few commands and only takes a couple of minutes.

We will start by assuming a fresh installation of Ubuntu, with a NVIDIA GPU available. For instance, you can follow these steps on an AWS EC2 GPU instance (g2.2xlarge). Before we start, let's make sure that you have `pip` install and that your package manager is up-to-date:

Listing 3.8 Preliminary steps

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ apt-get install python-pip python-dev
```

3.3.4 Installing the Python scientific suite

If you are on Mac we recommend that you install the Python scientific suite via Anaconda, which you can get at: `www.continuum.io/downloads`. Note that this will note include HDF5 and Graphviz, which you would have to install manually. Below we list the steps for a manual installation of the Python scientific suite on Ubuntu.

1) Install a BLAS (OpenBLAS in this case): this ensures that you can run fast tensor operations on CPU.

Listing 3.9 Installing OpenBLAS

```
$ sudo apt-get install build-essential cmake git unzip pkg-config
$ sudo apt-get install libopenblas-dev liblapack-dev
```

2) Install the Python scientific suite: Numpy, Scipy and MatPlotLib. This is necessary for doing any kind of machine learning or scientific computing in Python, whether you

are doing deep learning or not.

Listing 3.10 Installing the Python scientific suite

```
$ sudo apt-get install python-numpy python-scipy python-matplotlib
```

3) Install HDF5: this is a library for storing large files of numeric data in an efficient binary format, originally developed by NASA. It will allow you to save your Keras models to disk quickly and efficiently.

Listing 3.11 Installing HDF5

```
$ sudo apt-get install libhdf5-serial-dev
```

4) Install Graphviz and PyDot-ng: these two packages will allow you to visualize Keras models. They are not necessary to run Keras itself, so you could also skip this step and only install these packages when you need them.

Listing 3.12 Installing Graphviz and PyDot-ng

```
$ sudo apt-get install graphviz
$ sudo pip install pydot-ng
```

3.3.5 Setting up GPU support

Again, using a GPU is not strictly necessary, but it is strongly recommended. All code examples found in this book can be run a laptop CPU, but you may sometimes have to wait for several hours for a model to train, versus mere minutes on a good GPU. If you don’t have a modern NVIDIA GPU, you can skip this step and go directly to "Installing TensorFlow without GPU support".

In order to be able to use your NVIDIA GPU for deep learning, you need to installing two things:

- CUDA: this is a set of drivers for your GPU that allows it to run a low-level programming language for parallel computing.
- cuDNN: this is a library of highly optimized primitives for deep learning. When using cuDNN, you can typically increase the training speed of your models by 50%-100% when running on GPU.

1) Download CUDA:

Listing 3.13 Downloading CUDA

```
$ wget http://developer.download.nvidia.com/
```

```
compute/cuda/7.5/Prod/local_installers/cuda_7.5.18_linux.run
```

2) Make it executable:

Listing 3.14 Making the CUDA installer archive executable

```
$ chmod +x cuda_7.5.18_linux.run
```

3) Extract the installers to a `cuda_installers` directory:

Listing 3.15 Extracting the CUDA installers

```
$ mkdir cuda_installers
$ sudo ./cuda_7.5.18_linux.run -extract=./cuda_installers
```

4) Run the CUDA installers in order:

Listing 3.16 Installing CUDA

```
$ cd cuda_installers
$ sudo ./NVIDIA-Linux-x86_64-352.39.run
$ modprobe nvidia
$ sudo ./cuda-linux64-rel-7.5.18-19867135.run
$ sudo ./cuda-samples-linux-7.5.18-19867135.run
```

5) Configure your environment variables:

a) first, make sure that you see a `cuda-7.5` folder in `/usr/local` b) add the following lines in your `.bashrc` file:

Listing 3.17 Configuring environment variables for CUDA

```
export CUDA_HOME=/usr/local/cuda-7.5
export LD_LIBRARY_PATH=${CUDA_HOME}/lib64:$LD_LIBRARY_PATH
export PATH=${CUDA_HOME}/bin:${PATH}
```

`.bashrc` is a Bash configuration file that can be opened via: `$ nano ~/.bashrc`.

After you are done, refresh your Bash configuration via:

Listing 3.18 Refreshing the Bash configuration

```
$ source ~/.bashrc
```

6) Install cuDNN:

a) Register for a free NVIDIA developer account (unfortunately this is necessary in order to gain access to the cuDNN download) and download cuDNN at

developer.nvidia.com/cudnn (you should select cuDNN v5). Note that if you are working with a EC2 install, you won't be able to download the cuDNN archive directly to your instance: instead, you should download it to your local machine and then upload it to your EC2 instance (via `scp`).

b) To install cuDNN from the archive, you just need to extract the archive's content then copy two files to their appropriate location:

Listing 3.19 Installing cuDNN

```
$ tar -zxf cudnn-7.5-linux-x64-v5.0-ga.tgz
$ cd cuda
$ sudo cp lib64/* /usr/local/cuda/lib64/
$ sudo cp include/* /usr/local/cuda/include/
```

3.3.6 Installing TensorFlow without GPU support

If you intend on using a GPU, then you should skip this step and go to "Installing TensorFlow with GPU support".

TensorFlow without GPU support can be installed from PyPI using Pip:

Listing 3.20 Installing TensorFlow CPU-only version

```
sudo pip install tensorflow
```

3.3.7 Installing TensorFlow with GPU support

On the TensorFlow GitHub page (github.com/tensorflow/tensorflow), you will find the exact url of the binary you need to install, such as:

Listing 3.21 Installing TensorFlow, GPU version

```
sudo pip install
--upgrade TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/
gpu/tensorflow-0.9.0-cp27-none-linux_x86_64.whl
```

3.3.8 Installing Theano

Since you have already installed TensorFlow, you don't have to install Theano in order to run Keras code. However, it can sometimes be useful to switch back and forth from TensorFlow to Theano when building Keras models.

Theano can be installed from PyPI:

Listing 3.22 Installing Theano

```
sudo pip install theano --no-deps
```

If you are using a GPU, then you should configure Theano to use your GPU. You edit create a Theano configuration file with:

Listing 3.23 Editing the Theano configuration file

```
nano ~/.theanorc
```

Then fill in your file with the following configuration:

Listing 3.24 Configuring Theano

```
[global]
floatX = float32
device = gpu0

[nvcc]
fastmath = True
```

3.3.9 Installing Keras

You can install Keras from PyPI:

Listing 3.25 Installing Keras from PyPI

```
sudo pip install keras --no-deps
```

Alternatively, you can install Keras from GitHub, which will allow you to access to `keras/examples` folder, containing many examples scripts for you to learn from:

Listing 3.26 Installing Keras from GitHub

```
$ git clone https://github.com/fchollet/keras
$ cd keras
$ sudo python setup.py install
```

You can now try to run a Keras script, such as this MNIST example:

Listing 3.27 Testing Keras

```
python examples/mnist_cnn.py
```

After you have run Keras at least once, the Keras configuration file can be found at: `~/.keras/keras.json`. You can edit it to select the backend that Keras runs on: `tensorflow` or `theano`. Your configuration file should like this:

Listing 3.28 The Keras configuration file

```
{
    "image_dim_ordering": "tf",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

While the Keras script `examples/mnist_cnn.py` is running, you can monitor GPU utilization in a different shell window via:

Listing 3.29 Monitoring GPU utilization

```
$ watch -n 5 nvidia-smi -a --display=utilization
```

You're all set up! Congratulations. You can now start building deep learning applications.

3.4 Classifying movie reviews: a binary classification example

Two-class classification, or binary classification, may be the most widely applied kind of machine learning problem. In this example we will learn to classify movie reviews into "positive" reviews and "negative" reviews, just based on the text content of the reviews.

3.4.1 The IMDB dataset

We'll be working with "IMDB dataset", a set of 50,000 highly-polarized reviews from the Internet Movie Database. They are split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting in 50% negative and 50% positive reviews.

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

Just like the MNIST dataset, the IMDB dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

The following code will load the dataset (when you run it for the first time, about

80MB of data will be downloaded to your machine):

Listing 3.30 Loading the IMDB dataset

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(nb_words=10000)
```

The argument `nb_words=10000` means that we will only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows us to work with vector data of manageable size.

The variables `train_data` and `test_data` are lists of reviews, each review being a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists of 0s and 1s, where 0 stands for "negative" and 1 stands for "positive":

Listing 3.31 A look at the training data and labels

```
>> train_data[0]
[1, 14, 22, 16, ... 178, 32]

>> train_labels[0]
1
```

Since we restricted ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:

Listing 3.32 A look at the training data

```
>> max([max(sequence) for sequence in train_data])
9999
```

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

Listing 3.33 Decoding the integer sequences back into sentences

```
# word_index is a dictionary mapping words to an integer indice
word_index = imdb.get_word_index()
# we reverse it, mapping integer indices to words
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
# we decode the review; note that our indices were offset by 3
# because 0, 1 and 2 are reserved indices for "padding", "start of sequence", and "unknown".
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

3.4.2 Preparing the data

We cannot feed lists of integers into a neural network. We have to turn our lists into tensors. There are two ways we could do that:

- we could pad our lists so that they all have the same length, and turn them into an integer tensor of shape `(samples, word_indices)`, then use as first layer in our network a layer capable of handling such integer tensors (the `Embedding` layer, which we will cover in detail later in the book)
- we could one-hot-encode our lists to turn them into vectors of 0s and 1s. Concretely, this would mean for instance turning the sequence `[3, 5]` into a 10,000-dimensional vector that would be all-zeros except for indices 3 and 5, which would be ones. Then we could use as first layer in our network a `Dense` layer, capable of handling floating point vector data.

We will go with the latter solution. Let's vectorize our data, which we will do manually for maximum clarity:

Listing 3.34 Encoding the integer sequences into a binary matrix

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    # create an all-zero matrix of shape (len(sequences), dimension)
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # set specific indices of results[i] to 1s
    return results

# our vectorized training data
x_train = vectorize_sequences(train_data)
# our vectorized test data
x_test = vectorize_sequences(test_data)
```

Here's what our samples look like now:

Listing 3.35 An encoded sample

```
>> x_train[0]
array([ 0.,  1.,  1., ...,  0.,  0.,  0.])
```

We should also vectorize our labels, which is straightforward:

Listing 3.36 Encoding the labels

```
# our vectorized labels:
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

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

3.4.3 Building our network

Our input data is simply vectors, and our labels are scalars (1s and 0s): this is the easiest setup you will ever encounter. A type of network that performs well on such a problem would a simple stack of fully-connected (`Dense`) layers with `relu` activations: `Dense(16, activation='relu')`

The argument being passed to each `Dense` layer (16) is the number of "hidden units" of the layer. What's a hidden unit? It's a dimension in the representation space of the layer. You may remember from the previous chapter that each such `Dense` layer with `relu` implements the following chain of tensor operations:

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

Having 16 hidden units means that the weight matrix W will have shape `(input_dimension, 16)`, i.e. the dot product with W will project the input data onto a 16-dimensional representation space (and then we would add the bias vector `b` and apply the `relu` operation). You can intuitively understand the dimensionality of your representation space as "how much freedom your are allowing the network to have when learning internal representations". Having more hidden units (a higher-dimensional representation space) allows your network to learn more complex representations, but it makes your network more computationally expensive and may lead to learning unwanted patterns (patterns that will improve performance on the training data but not on the test data).

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

- how many layers to use
- how many "hidden units" to chose for each layer

Soon you will learn principles to guide you in making these choices: the "three principles of neural network architecture". For the time being, you will have to trust us with the following architecture choice: two intermediate layers with 16 hidden units each, and a third layer which will output the scalar prediction regarding the sentiment of the current review. The intermediate layers will use `relu` as their "activation function", and the final layer will use a sigmoid activation so as to output a probability (a score between 0 and 1, indicating how likely the sample is to have the target "1", i.e. how likely the review is to be positive).

Here's what it looks like:

/// [figure: 3 layer network]

And here's the Keras implementation, very similar to the MNIST example you saw previously:

Listing 3.37 Our model definition

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(16, activation='relu', input_dim=10000))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

NOTE **What are activation functions and why are they necessary?**

Without an activation function like `relu` (also called a *non-linearity*), our `Dense` layer would consist in two linear operations, a dot product and an addition.

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

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

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

Lastly, we need to pick a loss function and an optimizer. Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it best to use the `binary_crossentropy` loss. It isn't the only viable choice: you could use, for instance, `mean_squared_error`. But crossentropy is usually the best choice when you are dealing with models that output probabilities. Crossentropy is a quantity from the field of information, that measures the "distance" between probability distributions, or in our case, between the ground-truth distribution and our predictions.

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

Listing 3.38 Compiling our model

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
```

We are passing our optimizer, loss function and metrics as strings, which is possible because `rmsprop`, `binary_crossentropy` and `accuracy` are packaged as part of Keras. Sometimes you may want to configure the parameters of your optimizer, or pass a custom loss function or metric function. This former can be done by passing an optimizer class instance as the `optimizer` argument:

Listing 3.39 Configuring the optimizer

```
from keras.optimizers import RMSprop

model.compile(optimizer=RMSprop(lr=0.001), loss='binary_crossentropy', metrics=['accuracy'])
```

The latter can be done by passing function objects as the `loss` or `metrics` arguments:

Listing 3.40 Using custom losses and metrics

```
from keras.objectives import binary_crossentropy
from keras.metrics import binary_accuracy

model.compile(optimizer=RMSprop(lr=0.001), loss=binary_crossentropy, metrics=[binary_accuracy])
```

3.4.4 Validating our approach

In order to monitor during training the accuracy of the model on data that it has never seen before, we will create "validation set" by setting apart 10,000 samples from the original training data:

Listing 3.41 Setting aside a validation set

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

We will now train our model for 20 epochs (20 iterations over all samples in the `x_train` and `y_train` tensors), in mini-batches of 512 samples. At this same time we will monitor loss and accuracy on the 10,000 samples that we set apart. This is done by passing the validation data as the `validation_data` argument:

Listing 3.42 Training our model

```
history = model.fit(partial_x_train, partial_y_train, nb_epoch=20, batch_size=512, validation_data=(x_va
```

On CPU, this will take less than two seconds per epoch --training is over in 20

seconds. At the end of every epoch, there is a slight pause as the model computes its loss and accuracy on the 10,000 samples of the validation data.

Note that the call to `model.fit()` returns a `History` object. This object has a member `history`, which is a dictionary containing data about everything that happened during training. Let's take a look at it:

Listing 3.43 The `history` dictionary

```
>> history_dict = history.history
>> history_dict.keys()
[u'acc', u'loss', u'val_acc', u'val_loss']
```

It contains 4 entries: one per metric that was being monitored, during training and during validation. Let's use MatPlotLib to plot the training and validation loss side by side, as well as the training and validation accuracy:

Listing 3.44 Plotting the training and validation loss

```
import matplotlib.pyplot as plt

loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values) + 1)

# "bo" is for "blue dot"
plt.plot(epochs, loss_values, 'bo')
# b+ is for "blue crosses"
plt.plot(epochs, val_loss_values, 'b+')
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.show()
```

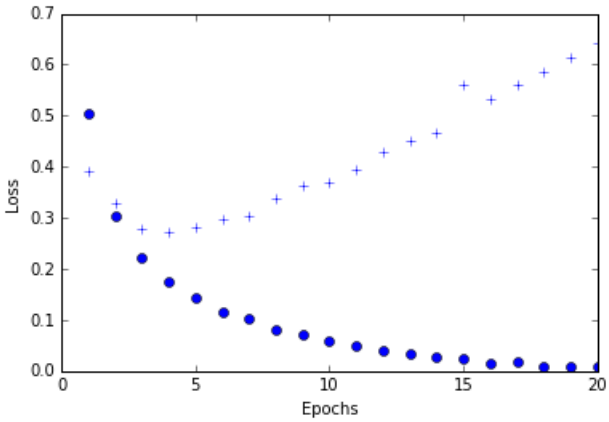

Figure 3.2 Training and validation loss

Listing 3.45 Plotting the training and validation accuracy

```
plt.clf()   # clear figure
```

```
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc_values, 'bo')
plt.plot(epochs, val_acc_values, 'b+')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.show()
```

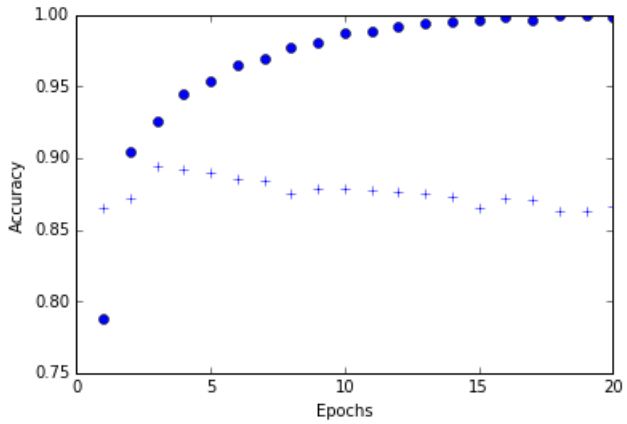

Figure 3.3 Training and validation accuracy

The dots are the training loss and accuracy, while the crosses are the validation loss and accuracy. Note that your own results may vary slightly due to a different random initialization of your network.

As you can see, the training loss decreases with every epoch and the training accuracy increases with every epoch. That's what you would expect when running gradient descent optimization --the quantity you are trying to minimize should get lower with every iteration. But that isn't the case for the validation loss and accuracy: they seem to peak at the fourth epoch. This is an example of what we were warning against earlier: a model that performs better on the training data isn't necessarily a model that will do better on data it has never seen before. In precise terms, what you are seeing is "overfitting": after the second epoch, we are over-optimizing on the training data, and we ended up learning representations that are specific to the training data and do not generalize to data outside of the training set.

In this case, to prevent overfitting, we could simply stop training after three epochs. In general, there is a never of techniques you can leverage to mitigate overfitting, which we will cover in the next chapter.

Let's train a new network from scratch for four epochs, then evaluate it on our test data:

Listing 3.46 Re-training a model from scratch

```
model = Sequential()
```

```
model.add(Dense(16, activation='relu', input_dim=10000))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

model.fit(x_train, y_train, nb_epoch=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

Listing 3.47 Our final results

```
>>> results
[0.2929924130630493, 0.88327999999999995]
```

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

3.4.5 Using a trained network to generate predictions on new data

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

Listing 3.48 Generating predictions for new data

```
>> model.predict(x_test)
[[ 0.98006207]
 [ 0.99758697]
 [ 0.99975556]
 ...,
 [ 0.82167041]
 [ 0.02885115]
 [ 0.65371346]]
```

As you can see, the network is very confident for some samples (0.99 or more, or 0.01 or less) but less confidents for others (0.6, 0.4).

3.4.6 Further experiments

- We were using 2 hidden layers. Try to use 1 or 3 hidden layers and see how it affects validation and test accuracy.
- Try to use layers with more hidden units or less hidden units: 32 units, 64 units…
- Try to use the `mse` loss function instead of `binary_crossentropy`
- Try to use the `tanh` activation (an activation that was popular in the early days of neural networks) instead of `relu`

These experiments will help convince you that the architecture choices we have made are all fairly reasonable, although they can still be improved!

3.4.7 Conclusions

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

- There's usually quite a bit of preprocessing you need to do on your raw data in order to be able to feed it --as tensors-- into a neural network. In the case of sequence of words, they can be encoded as binary vectors --but there are other encoding options too.
- Stacks of `Dense` layers with `relu` activations can solve a wide range of problems (including sentiment classification) and will likely use them frequently.
- In a binary classification problem (two classes), your network should end with a `Dense` layer with 1 unit and a `sigmoid` activation, i.e. the output of your network should be a scalar between 0 and 1, encoding a probability.
- With such a scalar sigmoid output, on a binary classification problem, the loss function you should use is `binary_crossentropy`.
- The `rmsprop` optimizer is generally a good enough choice of optimizer, whatever your problem. That's one less thing for you to worry about.
- As they get better on their training data, neural networks eventually start *overfitting* and end up obtaining worse results on data never-seen-before. Make sure to always monitor performance on data that is outside of the training set.

3.5 Classifying newswires: a multi-class classification example

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

In this section, we will build a network to classify Reuters newswires into 46 different mutually-exclusive topics. Since we have many classes, this problem is an instance of "multi-class classification", and since each data point should be classified into only one category, the problem is more specifically an instance of "single-label, multi-class classification". Each each data points could be belong to multiple categories (in our case, topics) then we would be facing "multi-label, multi-class classification".

3.5.1 The Reuters dataset

We'll be working with the "Reuters dataset", a set of short newswires and their topics, published by Reuters in 1986. It's a very simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set.

Like IMDB and MNIST, the Reuters dataset comes packaged as part of Keras. Let's take a look right away:

Listing 3.49 Loading the Reuters dataset

```
from keras.datasets import reuters

(train_data, train_labels), (test_data, test_labels) = reuters.load_data(nb_words=10000)
```

Like with the IMDB dataset, the argument `nb_words=10000` restricts the data to the 10,000 most frequently occurring words found in the data.

We have 8,982 training examples and 2,246 test examples:

Listing 3.50 Taking a look at the data

```
>>> len(train_data)
8982
>>> len(test_data)
2246
```

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

Listing 3.51 Taking a look at the data

```
>>> train_data[10]
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979,
3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]
```

Here's how you can decode it back to words, in case you are curious:

Listing 3.52 Decoding a newswires back to text

```
word_index = reuters.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
# note that our indices were offset by 3
# because 0, 1 and 2 are reserved indices for "padding", "start of sequence", and "unknown".
decoded_newswire = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

The label associated with an example is an integer between 0 and 45: a topic index.

Listing 3.53 Taking a look at the labels

```
>>> train_labels[10]
3
```

3.5.2 Preparing the data

We can vectorize the data with the exact same code as in the previous example:

Listing 3.54 Encoding the data

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
```

```
# our vectorized training data
x_train = vectorize_sequences(train_data)
# our vectorized test data
x_test = vectorize_sequences(test_data)
```

To vectorize the labels, there are two possibilities: we could use cast the label list as an integer tensor, or we could use a "one-hot" encoding. One-hot encoding is a widely used format for categorical data, also called "categorical encoding". Here it consist in embedding each label as an all-zero vector with a 1 in the place of the label index, e.g.:

Listing 3.55 One-hot encoding the labels

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

# our vectorized training labels
one_hot_train_labels = to_one_hot(train_labels)
# our vectorized test labels
one_hot_test_labels = to_one_hot(test_labels)
```

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

Listing 3.56 One-hot encoding the labels, the Keras way

```
from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

3.5.3 Building our network

This topic classification problem looks very similar to our previous movie review classification problem: in both cases, we are trying to classify short snippets of text. There is however a new constraint here: the number of output classes has gone from 2 to 46, i.e. the dimensionality of the output space is much larger.

In a stack of `Dense` layers like what we were using, each layer can only access information present in the output of the previous layer. If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers: each layer can potentially become an "information bottleneck". In our previous example, we were using 16-dimensional intermediate layers, but a 16-dimensional space may be to limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information.

For this reason we will use larger layers. Let’s go with 64:

Listing 3.57 Our model definition

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=10000))
model.add(Dense(64, activation='relu'))
model.add(Dense(46, activation='softmax'))
```

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

- We are ending the network with a `Dense` layer of size 46. This means that for each input sample, our network will output a 46-dimensional vector. Each entry in this vector (each dimension) will encode a different output class.
- The last layer uses a `softmax` activation. You have already seen this pattern in the MNIST example. It means that the network will output a *probability distribution* over the 46 different output classes, i.e. for every input sample, the network will produce a 46-dimensional output vector where `output[i]` is the probability that the sample belongs to class `i`. The 46 scores will sum to 1.

The best loss function to use in this case is `categorical_crossentropy`. It measures the distance between two probability distributions: in our case, between the probability distribution output by our network, and the true distribution of the labels. By minimizing the distance between these two distributions, we train our network to output something as close as possible to the true labels.

Listing 3.58 Compiling our model

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
              metrics=['accuracy'])
```

3.5.4 Validating our approach

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

Listing 3.59 Setting aside a validation set

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

Now let's train our network for 20 epochs:

Listing 3.60 Training our model

```
history = model.fit(partial_x_train, partial_y_train, nb_epoch=20, batch_size=512,
                    validation_data=(x_val, y_val))
```

Let's display its loss and accuracy curves:

Listing 3.61 Plotting the training and validation loss

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, loss_values, 'bo')
plt.plot(epochs, val_loss_values, 'b+')
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.show()
```

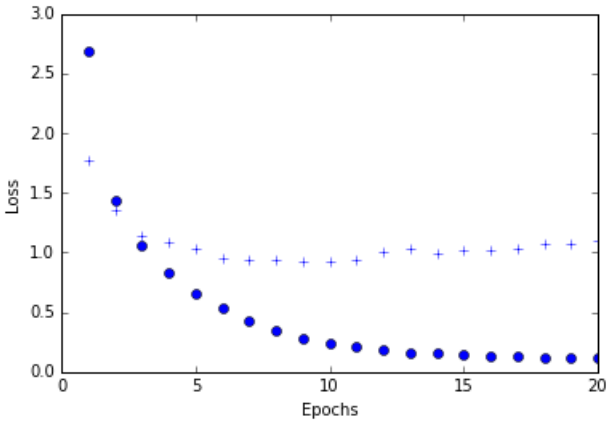

Figure 3.4 Training and validation accuracy

Listing 3.62 Plotting the training and validation accuracy

```
plt.clf()   # clear figure
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc_values, 'bo')
plt.plot(epochs, val_acc_values, 'b+')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.show()
```

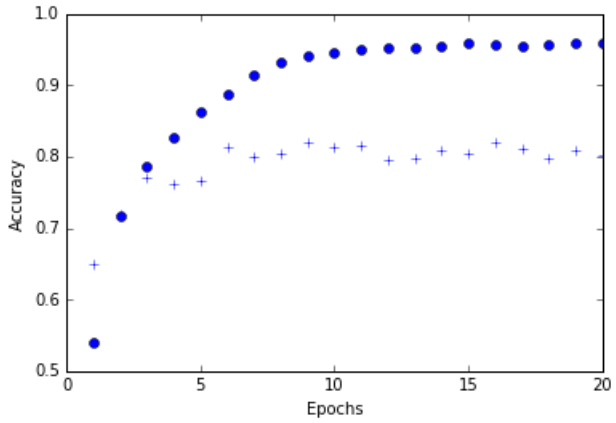

Figure 3.5 Training and validation accuracy

It seems that the network starts overfitting after 9 epochs. Let's train a new network from scratch for 9 epochs, then let's evaluate it on the test set:

Listing 3.63 Re-training a model from scratch

```
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=10000))
model.add(Dense(64, activation='relu'))
model.add(Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, one_hot_train_labels, nb_epoch=9, batch_size=512,
          validation_data=(x_test, one_hot_test_labels))
results = model.evaluate(x_test, one_hot_test_labels)
```

Listing 3.64 Our final results

```
>>> results
[0.9565213431445807, 0.79697239536954589]
```

Our approach reaches an accuracy of ~80%. With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%, but in our case it is closer to 19%, so our results seem pretty good, at least when compared to a random baseline:

Listing 3.65 Accuracy of a random baseline

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> float(np.sum(np.array(test_labels) == np.array(test_labels_copy))) / len(test_labels)
0.18655387355298308
```

3.5.5 Generating predictions on new data

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

Listing 3.66 Generating predictions for new data

```
predictions = model.predict(x_test)
```

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

Listing 3.67 Taking a look at our predictions

```
>>> predictions[0].shape
(46,)
```

The coefficients in this vector sum to 1:

Listing 3.68 Taking a look at our predictions

```
>>> np.sum(predictions[0])
1.0
```

The largest entry is the predicted class, i.e. the class with the highest probability:

Listing 3.69 Taking a look at our predictions

```
>>> np.argmax(predictions[0])
4
```

3.5.6 A different way to handle the labels and the loss

We mentioned earlier that another way to encode the labels would be to cast them as an integer tensor, like such:

Listing 3.70 Encoding the labels as integer arrays

```
y_train = np.array(labels_train)
y_test = np.array(labels_test)
```

The only thing it would change is the choice of the loss function. Our previous loss, `categorical_crossentropy`, expects the labels to follow a categorical encoding. With integer labels, we should use `sparse_categorical_crossentropy`:

Listing 3.71 Using the `sparse_categorical_crossentropy` loss

```
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
```

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

3.5.7 On the importance of having sufficiently large intermediate layers

We mentioned earlier that since our final outputs were 46-dimensional, we should avoid intermediate layers with much less than 46 hidden units. Now let's try to see what happens when we introduce an information bottleneck by having intermediate layers significantly less than 46-dimensional, e.g. 8-dimensional.

Listing 3.72 A model with an information bottleneck

```
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=10000))
model.add(Dense(8, activation='relu'))
model.add(Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, one_hot_train_labels, nb_epoch=10, batch_size=128,
          validation_data=(x_test, one_hot_test_labels))
```

Our network now seems to peak at ~73% test accuracy, a 7% absolute drop. This drop is mostly due to the fact that we are now trying to compress a lot of information (enough information to recover the separation hyperplanes of 46 classes) into an intermediate space that is too low-dimensional. The network is able to cram *most* of the necessary information into these 8-dimensional representations, but not all of it.

3.5.8 Further experiments

- Try using larger or smaller layers: 32 units, 128 units…
- We were using two hidden layers. Now try to use one hidden layer, or three hidden layers.

3.5.9 Conclusions

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

- If you are trying to classify data points between N classes, your network should end with a `Dense` layer of size N.
- In a single-label, multi-class classification problem, your network should end with a `softmax` activation, so that it will output a probability distribution over the N output classes.

- *Categorical crossentropy* is almost always the loss function you should use for such problems. It minimizes the distance between the probability distributions output by the network, and the true distribution of the targets.
- There are two ways to handle labels in multi-class classification:
  - encoding the labels via "categorical encoding" (also known as "one-hot encoding") and using `categorical_crossentropy` as your loss function.
  - encoding the labels as integers and using the `sparse_categorical_crossentropy` loss function.
- If you need to classify data into N categories, then you should avoid creating information bottlenecks in your network by having intermediate layers that are too small.

3.6 Predicting house prices: a regression example

In our two previous examples, we were considering classification problems, where the goal was to predict a single discreet label of an input data point. Another common type of machine learning problem is "regression", which consists in predicting a continuous value instead instead of a discreet label. For instance, predicting the temperature tomorrow given meteorological data, or predicting the time that a software project will take given its specifications.

Do not confuse "regression" with the algorithm "logistic regression": confusingly, "logistic regression" is not a regression algorithm, it's a classification algorithm.

3.6.1 The Boston Housing Price dataset

We will be attempting to predict the median price of homes in a given Boston suburb in the mid-1970s, given a few data points about the suburb at the time, such as the crime rate, the local property tax rate, etc.

The dataset we will be using another interesting difference from our two previous examples: it has very few data points, only 506 in total, split between 406 training samples and 100 test samples, and each "feature" in the input data (e.g. the crime rate is a feature) has a different scale. For instance some values are proportions, which take a values between 0 and 1, others take values between 1 and 12, others between 0 and 100…

Let's take a look at the data:

Listing 3.73 Loading the Boston housing dataset

```
from keras.datasets import boston_housing

(train_data, train_targets), (test_data, test_targets) =  boston_housing.load_data()
```

Listing 3.74 Taking a look at the data

```
>>> train_data.shape
```

```
(406, 13)
>>> test_data.shape
(100, 13)
```

As you can see we have 406 training samples and 100 test samples. The data comprises 13 features. The 13 features in the input data are as follow:

1. Per capita crime rate.
2. Proportion of residential land zoned for lots over 25,000 square feet.
3. Proportion of non-retail business acres per town.
4. Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).
5. Nitric oxides concentration (parts per 10 million).
6. Average number of rooms per dwelling.
7. Proportion of owner-occupied units built prior to 1940.
8. Weighted distances to five Boston employment centres.
9. Index of accessibility to radial highways.
10. Full-value property-tax rate per $10,000.
11. Pupil-teacher ratio by town.
12. 1000 * (Bk - 0.63) ** 2 where Bk is the proportion of blacks by town.
13. % lower status of the population.

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

Listing 3.75 Taking a look at the targets

```
>>> targets
[4.98   9.14   4.03   ...  4.21   3.57   6.19]
```

The prices are typically between $3,000 and $30,000. If that sounds cheap, remember this was the mid-1970s, and these prices are not inflation-adjusted.

3.6.2 Preparing the data

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

Listing 3.76 Normalizing the data

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

```
test_data -= mean
test_data /= std
```

Note the quantities that we use for normalizing the test data have been computed using the training data. We should never use in our workflow any quantity computed on the test data, even for something as simple as preprocessing.

3.6.3 Building our network

Before so few samples are available (just 406 after reserving 100 of them for testing), we will be using a very small network with only one hidden layer, and 64 hidden units in this layer. In general, the less training data you have, the worse overfitting will be, and using a small network is one way to mitigate overfitting.

Listing 3.77 Our model definition

```
def build_model():
    model = Sequential()
    model.add(Dense(64, activation='relu', input_dim=train_data.shape[1]))
    model.add(Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

Our network ends with a single unit, and not activation. This is a typical setup for scalar regression (i.e. regression where we are trying to predict a single continuous value). Applying an activation function would constrain the range that the output can take; for instance if we applied a `sigmoid` activation function to our last layer, the network could only learn to predict values between 0 and 1. Here, because the last layer is purely linear, the network is free to learn to predict values in any range.

Note that we are compiling the network with the `mse` loss function --Mean Squared Error, the square of the different between the predictions and the targets, a widely used loss function for regression problems.

We are also monitoring a new metric during training: `mae`. This stands for Mean Absolute Error. It is simply the absolute value of the difference between the predictions and the targets. For instance, a MAE of 0.5 on this problem would mean that our predictions are off by $500 on average.

3.6.4 Validating our approach using k-fold validation

To evaluate our network while we keep adjusting its parameters (such as the number of epochs used for training), we could simply split the data into a training set and a validation set as we were doing in our previous examples. However, because we have so few data points, the validation set would end up being very small (e.g. about 100 examples). A consequence is that our validation scores may change a lot depending on *which* data points we choose to use for validation and which we choose for training, i.e. the validation scores may have a high variance with regard to the validation split. This would prevent us from reliably evaluate our model.

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

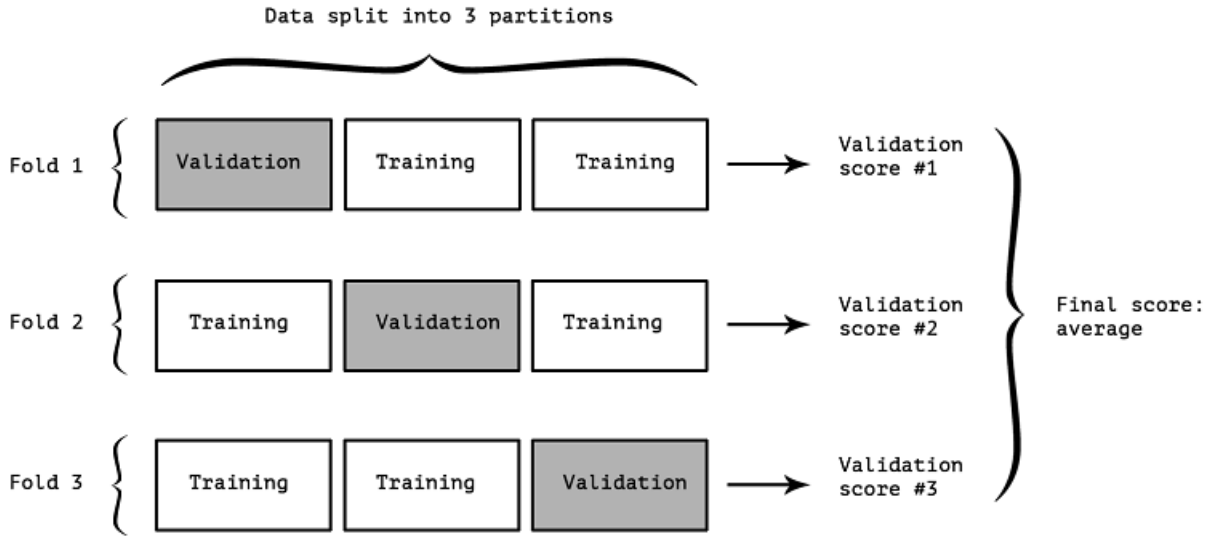

Figure 3.6 3-fold cross-validation

In terms of code, this is straightforward:

Listing 3.78 K-fold validation

```
k = 4
num_val_samples = len(train_data) // k
all_scores = []
for i in range(k):
    print('processing fold #', i)
    # prepare the validation data: data from partition # k
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    # prepare the training data: data from all other partitions
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
```

```
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    # build the Keras model (already compiled)
    model = build_model()
    # train the model (in silent mode, verbose=0)
    model.fit(partial_train_data, partial_train_targets,
              nb_epoch=100, batch_size=1, verbose=0)
    # evaluate the model on the validation data
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

Running the above snippet with `nb_epoch=100` yields the following results:

Listing 3.79 Validation scores for successive "folds"

```
>>> all_scores
[1.1636574666688937, 1.3182317475871284, 0.28683523110824055, 0.33867029609656568]
>>> np.mean(all_scores)
0.77684868536520701
```

As you can notice, the different runs do indeed show widly different validation scores, from 0.3 to 1.3. Their average is a much reliable metric than any single of these scores --that's the entire point of K-fold cross-validation. In this case we are off by $780 on average, which is still significant considering that the prices range from 3,000 to 30,000.

Let's try training the network for a bit longer. To keep a record of how well the model did at each epoch, we will modify our training loop to save the per-epoch validation score log:

Listing 3.80 Saving the validation logs at each fold

```
all_mae_histories = []
for i in range(k):
    print('processing fold #', i)
    # prepare the validation data: data from partition # k
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    # prepare the training data: data from all other partitions
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    # build the Keras model (already compiled)
    model = build_model()
    # train the model (in silent mode, verbose=0)
    history = model.fit(partial_train_data, partial_train_targets,
```

```
                      validation_data=(val_data, val_targets),
                      nb_epoch=500, batch_size=1, verbose=0)
    mae_history = history.history['val_mean_absolute_error']
    all_mae_histories.append(mae_history)
```

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

Listing 3.81 Building the history of successive mean K-fold validation scores

```
average_mae_history = [np.mean([x[i] for x in all_mae_histories]) for i in range(500)]
```

Let's plot this:

Listing 3.82 Plotting validation scores

```
import matplotlib.pyplot as plt

plt.plot(range(500), average_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
```

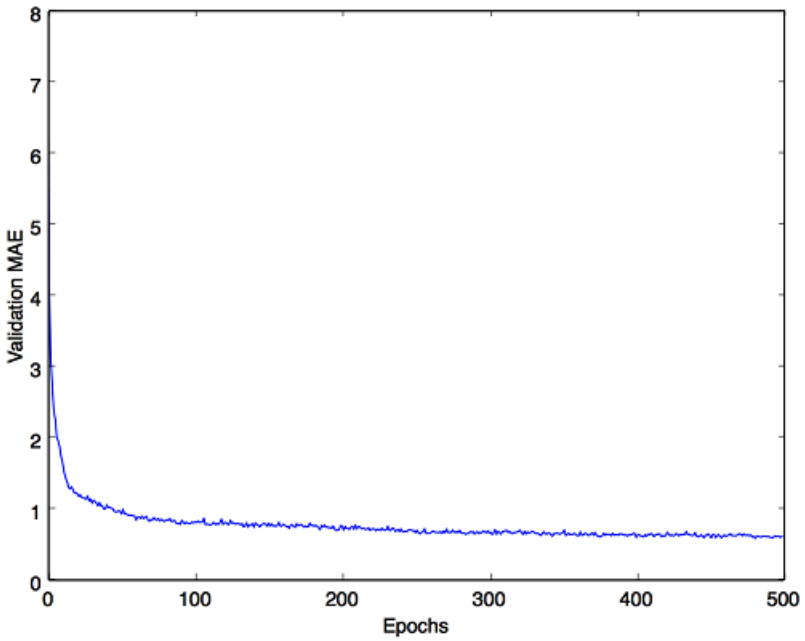

Figure 3.7 Validation MAE by epoch

According to this plot, it seems that validation MAE stops improving significantly after after 300 epochs.

Once we are done tuning other parameters of our model (besides the number of epochs, we could also adjust the size of the hidden layer), we can train a final "production" model on all of the training data, with the best parameters, then look at its performance on the test data:

Listing 3.83 Training the final model

```
model.fit(train_data, train_targets,
          nb_epoch=300, batch_size=1, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

Listing 3.84 Our final result

```
>>> test_mae_score
0.070807943344116206
```

We are still off by about $710.

3.6.5 Conclusions

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

- Regression is done using different loss functions from classification; Mean Squared Error (MSE) is a commonly used loss function for regression.
- Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally the concept of "accuracy" does not apply for regression. A common regression metric is Mean Absolute Error (MAE).
- When features in the input data have values in different ranges, each feature should be scaled independently as a preprocessing step.
- When there is little data available, using K-Fold validation is a great way to reliably evaluate a model.
- When little training data is available, it is preferable to use a small network with very few hidden layers (typically only one).