

DEEP LEARNING with Python

François Chollet

MEAP



MANNING





**MEAP Edition
Manning Early Access Program
Deep Learning with Python
Version 6**

Copyright 2017 Manning Publications

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

PART 1: THE FUNDAMENTALS OF DEEP LEARNING

- 1 *What is Deep Learning?*
- 2 *Before we start: the mathematical building blocks of neural networks*
- 3 *Getting started with neural networks*
- 4 *Fundamentals of machine learning*

PART 2: DEEP LEARNING IN PRACTICE

- 5 *Deep learning for computer vision*
- 6 *Deep learning for text and sequences*
- 7 *Advanced deep learning best practices*
- 8 *Generative deep learning*
- 9 *Conclusions*

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.

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?

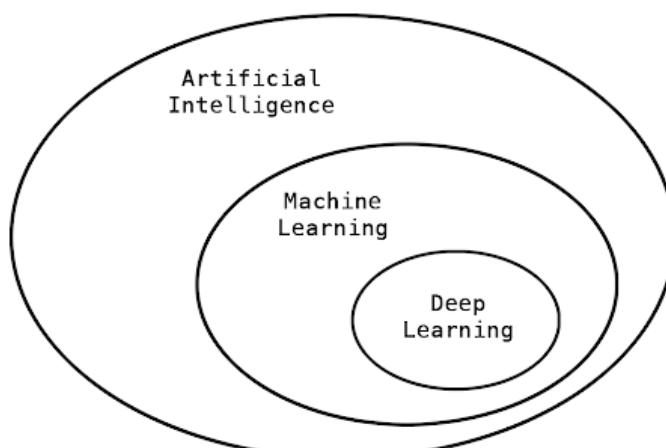


Figure 1.1 Artificial Intelligence, Machine Learning and Deep Learning

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

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 handcraft a sufficiently large set of explicit rules for manipulating knowledge. This approach is known as "symbolic AI", and it was the dominant paradigm in AI from the 1950s to the late 1980s. It reached its peak popularity during the "expert systems" boom of the 1980s.

Although symbolic AI proved suitable to solve well-defined, logical problems, such as playing chess, it turned out to be intractable to figure out explicit rules for solving more complex, fuzzy problems, such as image classification, speech recognition, 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". 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 be 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.

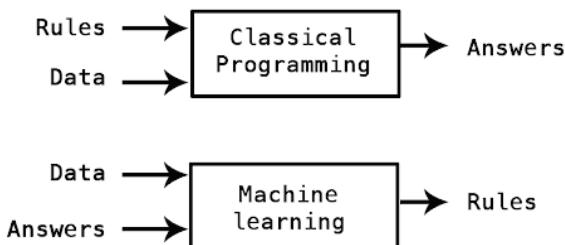


Figure 1.2 Machine learning: a new programming paradigm

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 to execute a data-processing task, given examples of what is expected. So, to do machine learning, we need three things:

- Input data points. For instance, if the task is speech recognition, these data points could be sound files of people speaking. If the task is image tagging, they could be picture files.
- Examples of the expected output. In a speech recognition task, these could be

human-generated transcripts of our sound files. In an image task, expected outputs could tags such as "dog", "cat", and so on.

- A way to measure if the algorithm is doing a good job, to measure the distance between its current output and its expected output. This is used as a feedback signal to adjust the way the algorithm works. This adjustment step is what we call "learning".

A machine learning model transforms its input data into a meaningful output, a process which is "learned" from exposure to known examples of inputs and outputs. Therefore, the central problem in machine learning and deep learning is to *meaningfully transform data*, or in other words, to learn useful "representations" of the input data at hand, representations that get us closer to the expected output. Before we go any further: what's a representation? At its core, it's a different way to look at 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 RGB format, while "make the image less saturated" is simpler in the HSV format. Machine learning models are all about finding appropriate representations for their input data, transformations of the data that make it more amenable to the task at hand, such as a classification task.

Let's make this concrete. 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.

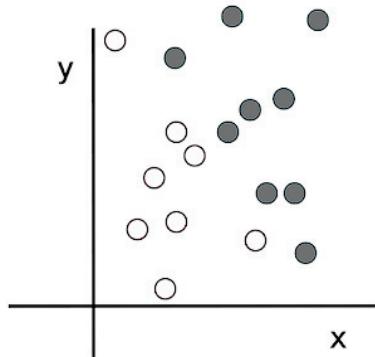


Figure 1.3 Some sample data

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.

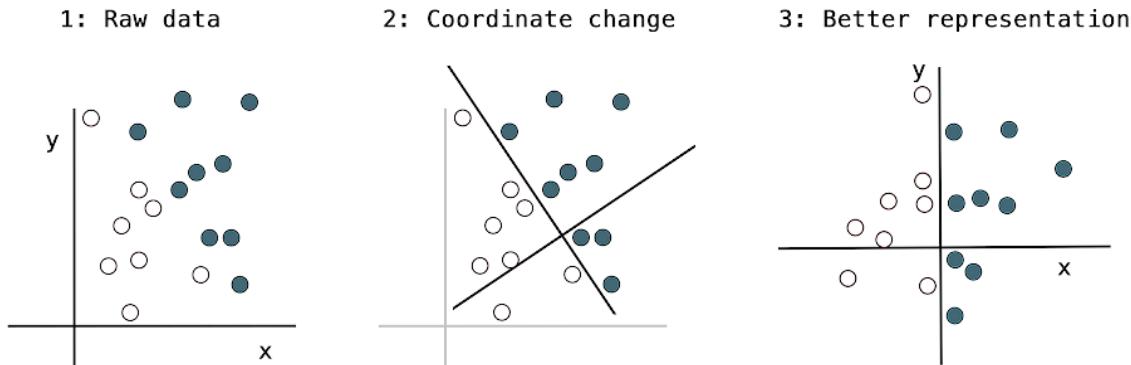


Figure 1.4 Coordinate change

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 $x \geq 0$ 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. "Learning", in the context of machine learning, describes an automatic search process for better representations.

All machine learning algorithms consist of automatically finding such transformations that turn data into more useful representations for a given task. These operations 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 $x \geq 0$), 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.

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

1.1.4 The "deep" in deep learning

Deep learning is a specific subfield of machine learning, a new take on learning representations from data 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.

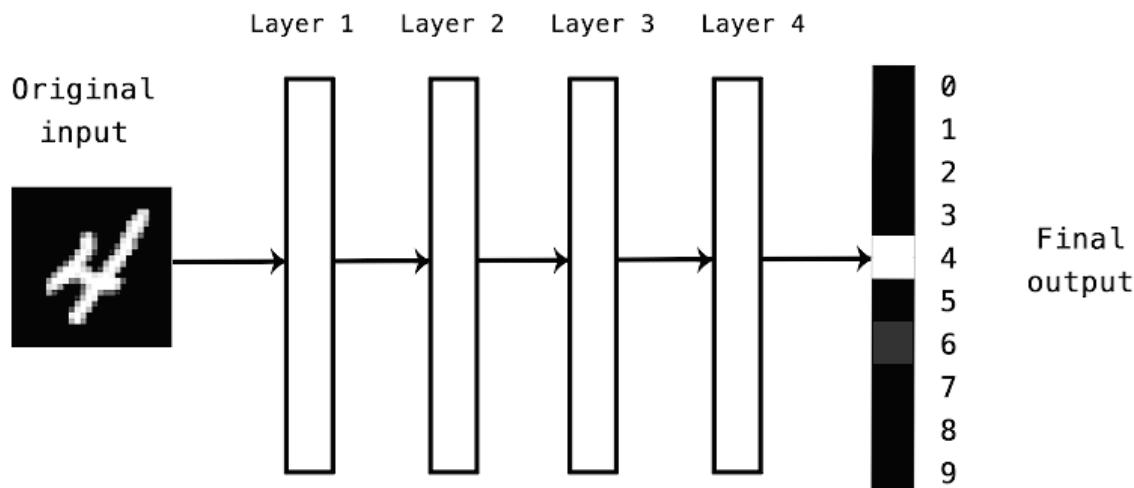


Figure 1.5 A deep neural network for digit classification

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:

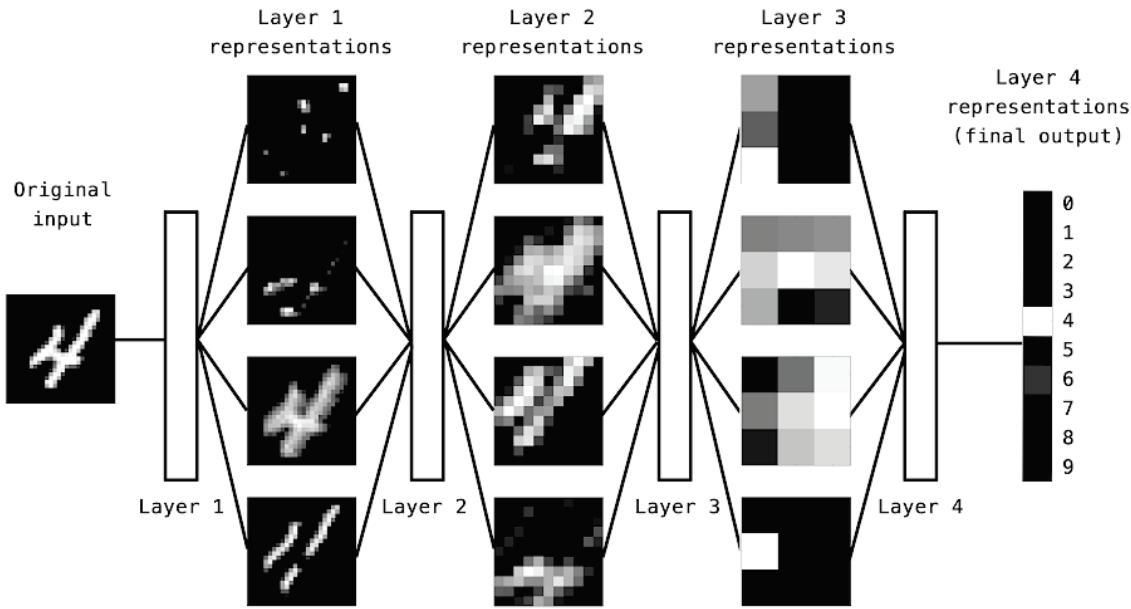


Figure 1.6 Deep representations learned by a digit classification model

As you can see, the network transforms the digit image into representations that are increasingly different from the original image, and increasingly informative about 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!

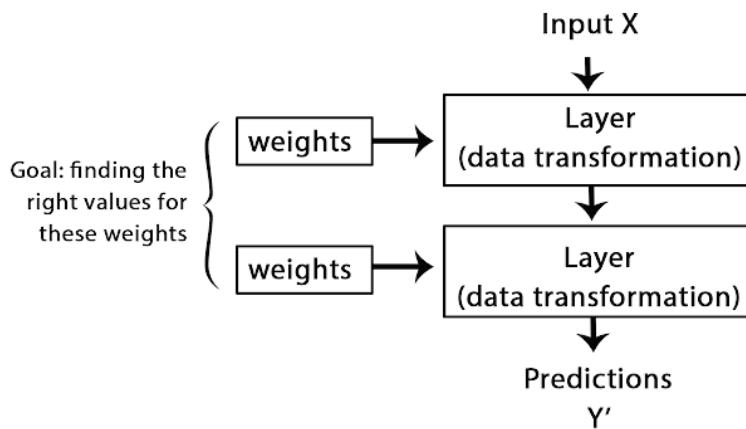


Figure 1.7 A neural network is parametrized by its weights

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.

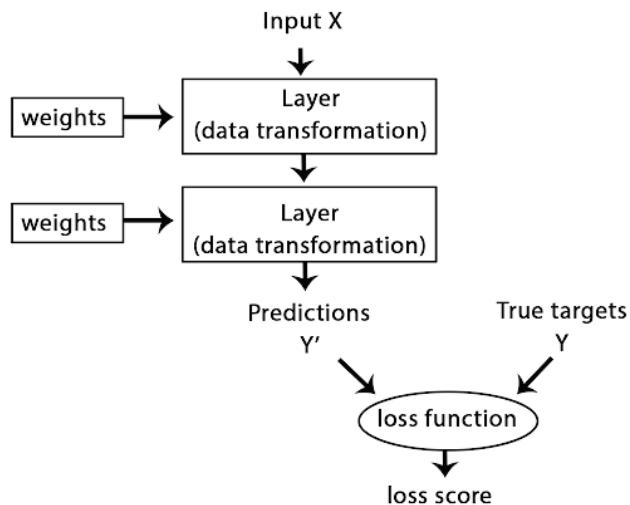


Figure 1.8 A loss function measures the quality of the network's output

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 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.

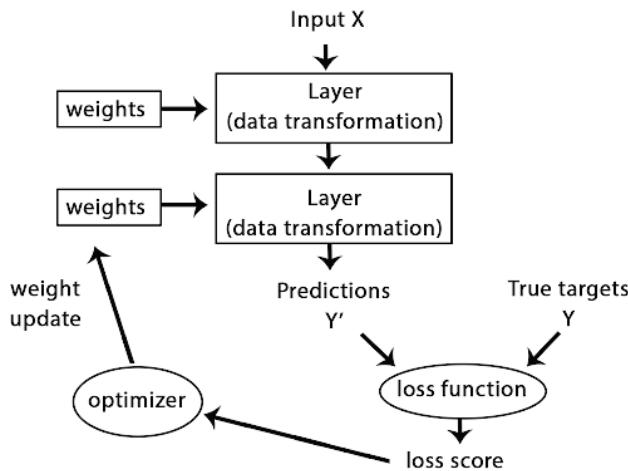


Figure 1.9 The loss score is used as a feedback signal to adjust the weights

Initially, the weights of the network are assigned random values, so the network merely implements a series of random transformations --naturally its output is very far from what 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 over thousands of examples), yields weight values that minimize the loss function. A network with a minimal loss is one for which the outputs are as close as they can be to the targets: a trained network.

Once again: a very simple mechanism, 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.
- Improved text-to-speech conversion.
- 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.
- Superhuman 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 believed 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 yet use AI, your accountant doesn't yet 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, even 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 enough 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 for this hammer. The only way not to fall into this trap is to be familiar with 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 place deep learning in the broader context of machine learning, and better understand where deep learning comes from and why it matters.

1.2.1 Probabilistic modeling

Probabilistic modeling is the application of the principles of statistics to data analysis. It was one of the earliest forms of machine learning, 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 is helpful to be aware of how deep learning originated. Although the core ideas of neural networks were investigated in toy forms as early as the 1950s, the approach took decades to 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" (Figure 1.10) between two sets of points belonging to two different categories. A "decision boundary" can be thought of as a line or surface separating your training data into two spaces corresponding to two categories. To classify new data points, you just need to check which side of the decision boundary they fall on.

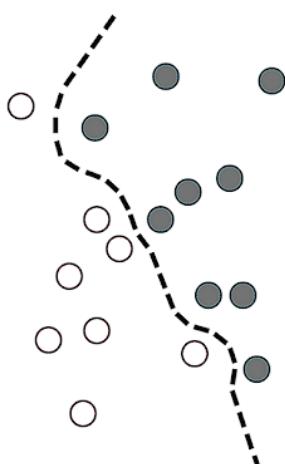


Figure 1.10 A decision boundary

SVMs proceed to find these boundaries in two steps:

- 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, an "hyperplane" would simply be a straight line).
- Then a good decision boundary (a separation hyperplane) is computed by trying to maximize the distance between the hyperplane and the closest data points from each class, a step called "maximizing the margin". This allows the boundary to generalize well to new samples outside of the training dataset.

The technique of mapping data to a high-dimensional representation where a classification problem becomes simpler may look good on paper, but in practice it 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 bypassing 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 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

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. Decisions trees learned from data started getting significant research interest in the 2000s, and by 2010 they were often preferred to kernel methods.

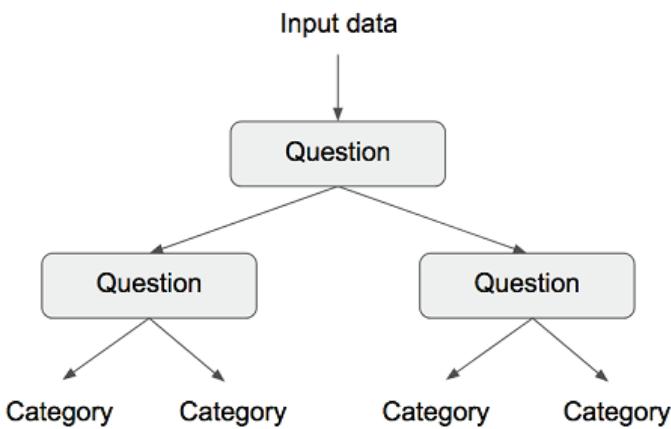


Figure 1.11 A decision tree: the parameters that are learned are the questions about the data. A question could be, for instance, "is coefficient 2 in the data higher than 3.5?".

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. It leverages "gradient boosting", a way to improve any machine learning model by iteratively training new models that specialize in addressing the weak points of the previous models. Applied to decision trees, the use of the "gradient boosting" technique results in models that strictly outperform Random Forests most of the time, while having 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 1.4 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 applications in many other types of problems, such as natural language processing. It has come to completely replace SVMs and decision trees in a wide range of applications. For instance, for several years, the European Organization for Nuclear Research, CERN, used decision tree-based methods for analysis of particle data from the ATLAS detector at the Large Hadron Collider (LHC), but 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 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 yourself. This has greatly simplified machine learning workflows, often replacing very sophisticated 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 is transformative about deep learning is that it allows a model to learn all layers of representation *jointly*, at the same time, rather than in succession ("greedily", as it is called). With joint feature learning, whenever the model adjusts one of its internal features, all other features that depend on it will automatically adapt to the change, without requiring human

intervention. Everything is supervised by a single feedback signal: every change in the model serves the end goal. This is much more powerful than greedily stacking shallow models, as it allows for very complex and abstract representations to be learned by breaking them down into long series of intermediate spaces (layers), each space only a simple transformation away from the previous one.

These are the two essential characteristics of how deep learning learns from data: the *incremental, layer-by-layer way in which increasingly complex representations are developed*, and the fact *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. Together, these two properties have made deep learning vastly more successful than previous approaches to machine learning.

1.2.7 The modern machine learning landscape

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

In 2016, 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.

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 (for shallow learning problems), and deep learning (for perceptual problems). 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.

- Datasets and benchmarks.
- Algorithmic advances.

Because the field is guided by experimental findings rather than by theory, algorithmic advances only become possible when appropriate data and hardware is available to try new ideas (or just scale up old ideas, as is often the case). 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 is what happened during that time: the Internet took, and high-performance graphics chips were developed for the needs of the gaming market.

1.3.1 Hardware

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

However, typical deep learning models used in computer vision or speech recognition require orders of magnitude more computational power than what your laptop can deliver. Throughout the 2000s, companies like NVIDIA and AMD have been investing billions of dollars 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 of 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 among them.

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 trillion 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 couple of days to train an ImageNet model of the sort that would have won the competition a few years ago. Meanwhile, large companies train deep learning models on clusters of hundreds of GPUs of a type developed specifically for the needs of deep learning, such as the NVIDIA 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 "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. When it comes to data, 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 were 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 1.4 million images hand-annotated with 1000 images categories (one category per image). But what makes ImageNet special is not just its large size, but also the yearly competition associated with it. As Kaggle.com has been demonstrating since 2010, public competitions are an excellent way to motivate researchers and engineers to push the envelope. Having common benchmarks that researchers compete to beat has greatly helped the recent rise of deep learning.

1.3.3 Algorithms

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" for neural layers.
- 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, 2015 and 2016, even more advanced ways to help gradient propagation were discovered, such as batch normalization, residual connections, and depthwise separable convolutions. Today we can train from scratch models that are thousands of layers deep.

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. The deep learning hardware startup Nervana Systems was acquired by Intel in 2016 for over \$400.

In fact, machine learning and in particular deep learning have become central to the product strategy of these tech giants. In late 2015, 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 libraries such as Keras, which makes deep learning as easy as manipulating Lego bricks. 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?

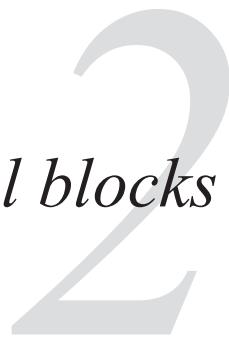
The short answer is yes—deep learning does have several properties that justify its status as an AI revolution, and it is here to stay. We may not still be using neural networks two decades from now, but whatever we use will directly inherit from modern deep learning and its core concepts.

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 prior machine learning approaches, deep learning models can be trained on additional data without restarting from scratch, making them viable for continuous online learning, an important property for very large production models. Furthermore, trained deep learning models are repurposable and thus reusable: for instance it 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. This also makes deep learning applicable to fairly small datasets.

Deep learning has only been in the spotlight for a few years, and we haven't yet established the full scope of what it can do. Every passing 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 2017, 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.

Before we start: the mathematical blocks of neural networks



Understanding deep learning requires familiarity with many simple mathematical concepts: tensors, tensor operations, differentiation, gradient descent... Our goal in this chapter will be to build intuition about these notions without getting overly technical. In particular, we will steer away from mathematical notation, which can be off-putting for those without any mathematics background, and isn't strictly necessary to explain things well.

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

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 models.
- 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—which 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 the Python library 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. You probably haven't even installed Keras yet. Don't worry, that is perfectly fine. In the next chapter, we will review each element in our example and explain them in detail. So don't worry if some steps seem arbitrary or look like magic to you! We've got to start somewhere.

The problem we 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 on classes and labels**

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



Figure 2.1 MNIST sample digits

You don't need to try to reproduce this example on your machine just now. If you wish to, you will first need to set up Keras, which is covered in section 3.3.

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  
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

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  
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

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, remember that you aren't supposed to understand everything about this example just yet.

Listing 2.4 The network architecture

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

The core building block of neural networks is the "layer", a data-processing module 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 of chaining together simple layers which will implement a form of progressive "data distillation". A deep learning model is like a sieve for data processing, made of a succession of increasingly refined data filters—the "layers".

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

To make our network ready for training, we need to pick three more things, as part of "compilation" step:

- A loss function: this 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 into 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 categorically encode the labels, a step which we explain in chapter 3:

Listing 2.7 Preparing the labels

```
from keras.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: we "fit" the model to its training data.

Listing 2.8 Training the network

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

Two quantities are 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 gap between training accuracy and test accuracy is an example of "overfitting", the fact that machine learning models tend to perform worse on new data than on their training data. Overfitting will be a central topic in chapter 3.

This concludes our very first example—you just saw how we could build and train a neural network to classify handwritten digits, in less than 20 lines of Python 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. You will learn about "tensors", the data-storing objects going into the network, about tensor operations, which layers are made of, and about gradient descent, which allows our network to learn from its training examples.

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 use 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 0-dimensional tensor, or 0D tensor). In Numpy, a `float32` or `float64` number is a scalar tensor (or scalar array). You can display the number of axes of a Numpy tensor via the `ndim` attribute; a scalar tensor has 0 axes (`ndim == 0`). The number of axes of a tensor is also called its *rank*.

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])
>>> x.ndim
1
```

Here, this vector has 5 entries, and so will be called a "5-dimensional vector". Do not confuse a 5D 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). "Dimensionality" can either denote the number of entries along a specific axis (e.g. in the case of our 5D vector), or the number of axes in a tensor (e.g. a 5D tensor), which can be quite confusing at times. In the latter case, it is technically more correct to talk about "a tensor of rank 5" (the rank of a tensor being the number of axes), but the ambiguous notation "5D tensor" is very common regardless.

2.2.3 Matrices (2D tensors)

An array of vectors is a matrix, or 2D tensor. A matrix has two axes (often denoted "rows" 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 0D to 4D, although you may go up to 5D if you process video data.

2.2.5 Key attributes

A tensor is defined by 3 key attributes:

- The number of axes it has, its *rank*. 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 such as Numpy.
- 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 Displaying the 4th digit

```
digit = train_images[4]

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

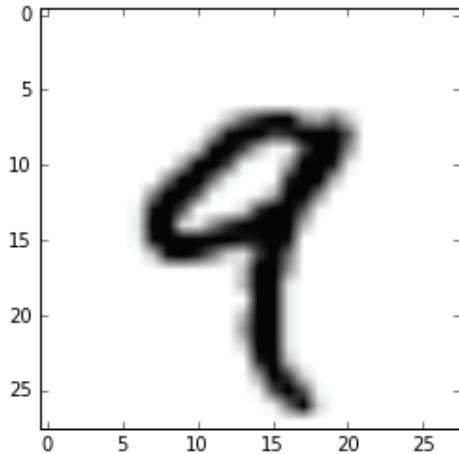


Figure 2.2 The 4th sample in our dataset

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

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.

The 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 one 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. In this setting, each character can be encoded as a binary vector of size 128 (an all-zeros vector except for a 1 entry at the index corresponding to the character). Then each tweet can be encoded as a 2D tensor of shape `(140, 128)`, and a dataset of 1M tweets can be stored in a tensor of shape `(1000000, 140, 128)`.

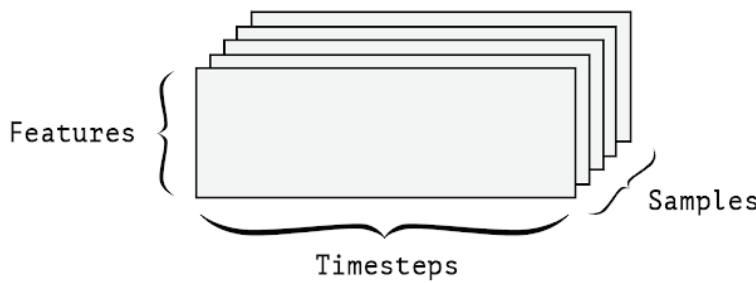


Figure 2.3 A 3D timeseries data tensor.

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.

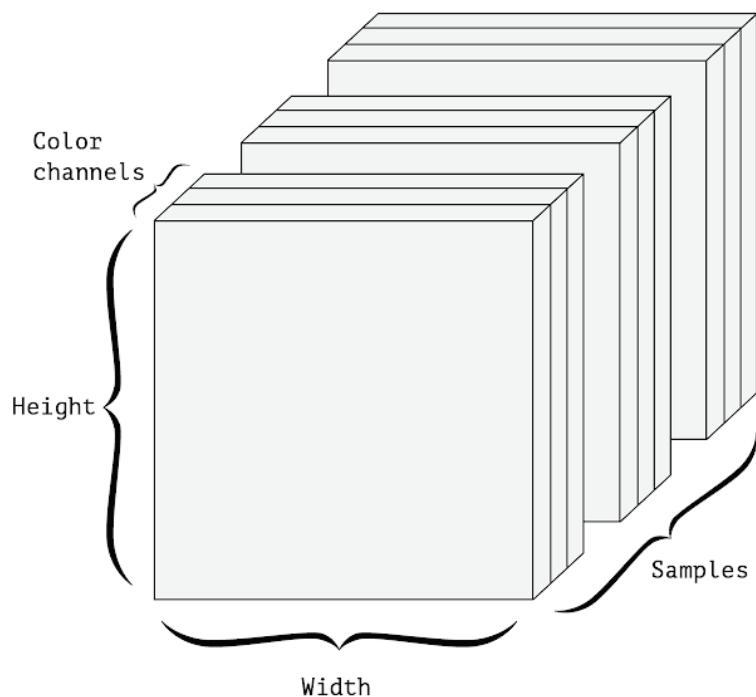


Figure 2.4 A 4D image data tensor.

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

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

This layer can be interpreted as a function, which takes as input a 2D tensor and returns another 2D tensor—a new representation for the input tensor. Specifically, the following 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

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

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

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

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

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

In practice, when dealing with Numpy arrays, these operations are available as well-optimized built-in Numpy functions, which themselves delegate the heavy lifting to a 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 `i` in `range(0, 32)`. At this 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]

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

```
# x is a random tensor with shape (64, 3, 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 z 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)
```

In mathematical notation, you would note the operation with a dot \cdot :

$$z = x \cdot y$$

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 2.31 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 2.32 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(x.shape[0])
for i in range(x.shape[0]):
    for j in range(x.shape[1]):
        z[i] += x[i, j] * y[j]
return z

```

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

Listing 2.33 Alternative naive implementation of matrix-vector dot

```

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

```

Note that as soon as one of the two tensors has 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[0], y.shape[1])`, where coefficients are the vector products between the rows of `x` and the columns of `y`. Here's the naive implementation:

Listing 2.34 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[0], y.shape[1]))
    # We iterate over the rows of x
    for i in range(x.shape[0]):
        # And over the columns of y
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z

```

To understand dot product shape compatibility, it helps to visualize the input and

output tensors by aligning them in the following way:

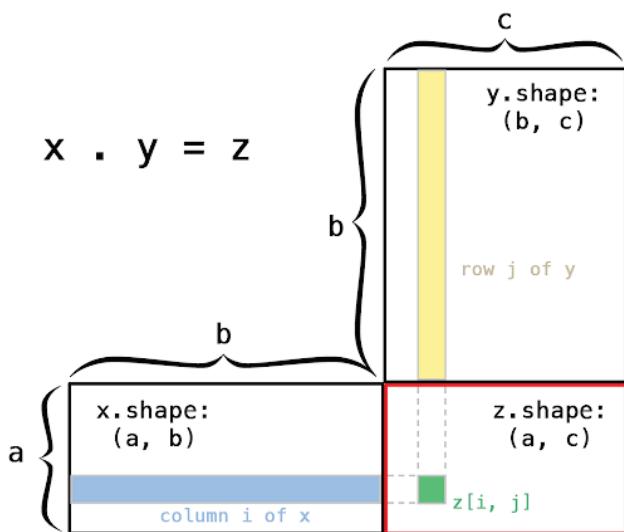


Figure 2.5 Matrix dot product box diagram

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

$$(a, b, c, d) \cdot (d,) \quad (a, b, c)$$

$$(a, b, c, d) \cdot (d, e) \quad (a, b, c, e)$$

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 2.35 MNIST image tensor reshaping

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

Reshaping a tensor means 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 2.36 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 2.37 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:

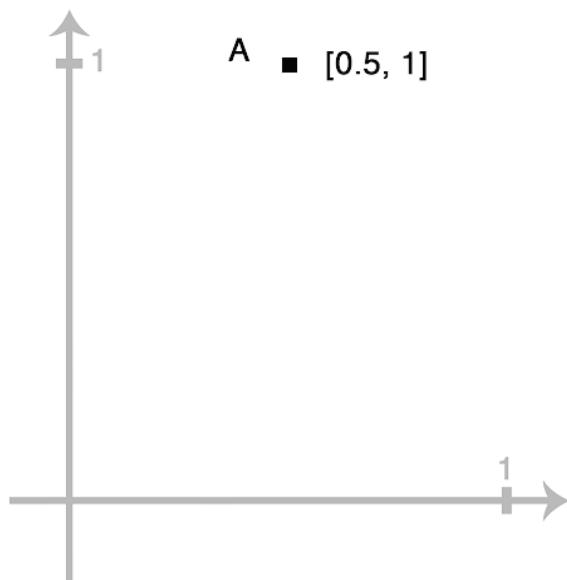


Figure 2.6 A point in a 2D space

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

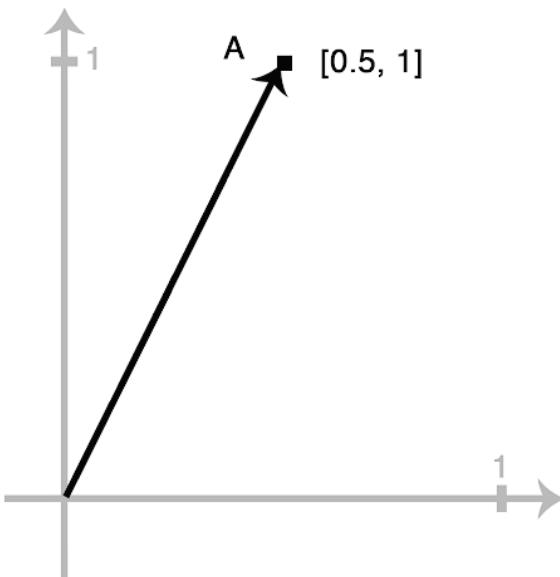


Figure 2.7 A point in a 2D space pictured as an arrow

Let's consider a new point, $B = [1, 0.25]$, which we 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:

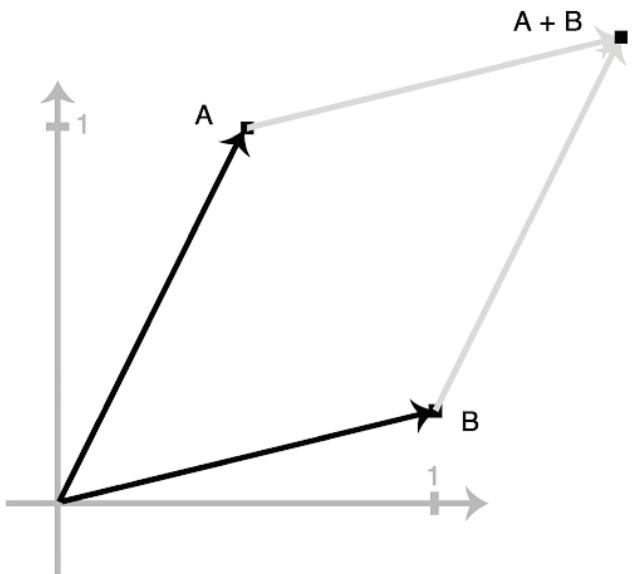


Figure 2.8 Geometric interpretation of the sum of two vectors

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 2×2 matrix $R = [u, v]$ where u and v are both vectors of the plane: $u = [\cos(\theta), \sin(\theta)]$ and $v = [-\sin(\theta), \cos(\theta)]$.

2.3.6 A geometric interpretation of deep learning

You just learned that neural networks consist entirely 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 in 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 (the `kernel` and `bias` attributes, respectively). These weights contain the information learned by the network from exposure to training data.

Initially, these weight matrices are filled with small random values (a step called *random initialization*). Of course, there is no reason to expect that `relu(dot(w, input) + b)`, when `W` and `b` are random, would yield any useful representations. The resulting representations are meaningless—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:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

- 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 slightly reduces the loss on this batch.

We eventually end up with a network that has a very low loss on its training data, i.e. a very low mismatch between predictions y_{pred} and expected targets y : 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 I/O code. Steps 2 and 3 are merely the application of a handful of tensor operations, so you could implement these steps purely from what you have learned in the previous section. The difficult part here is how to do step 4, the update of the weights of the network. 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, usually thousands and sometimes up to millions). 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 then 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 a small epsilon_y change to y .

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

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 $a * epsilon_x$:

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

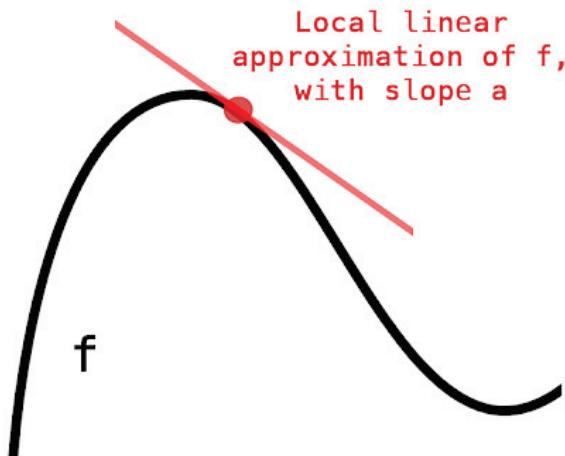


Figure 2.9 Derivative of f in p

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 inputs x and y are frozen, then this can be interpreted as a function mapping values of w to loss values:

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

Let's say that the current value of w is w_0 . Then the derivative of f in the point w_0 , is a tensor $\text{gradient}(f)(w_0)$ with the same shape as w , where each coefficient $\text{gradient}(f)(w_0)[i, j]$ indicates the direction and magnitude of the change in loss_value you would observe when modifying $w_0[i, j]$. That tensor $\text{gradient}(f)(w_0)$ is the *gradient* of the function $f(w) = \text{loss_value}$ in w_0 .

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, $\text{gradient}(f)(w_0)$ can be interpreted as the tensor describing the *curvature* of $f(w)$ around w_0 .

For this reason, in much the same way that, for a function $f(x)$, you 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. $w_1 = w_0 - \text{step} * \text{gradient}(f)(w_0)$ (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 $\text{gradient}(f)(w_0)$ only approximates the curvature when you are close to w_0 , so you don't want to get too far away from w_0 .

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 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: $\text{gradient}(f)(w) = 0$ for w . This is a polynomial equation of N variables, where N is the number of coefficients in the network. While it would be possible to solve such an equation 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:

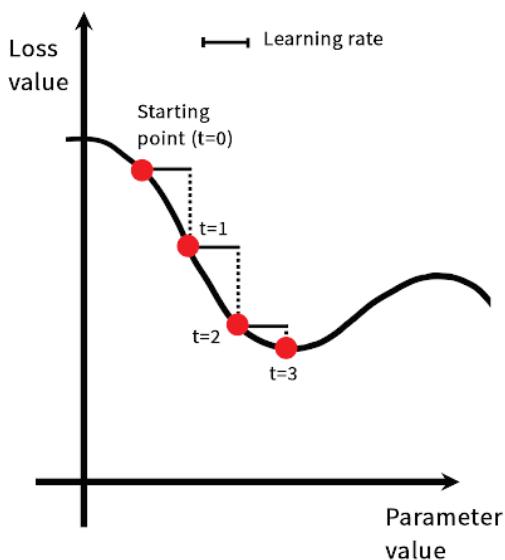


Figure 2.10 SGD down a 1D loss curve (1 learnable parameter).

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 you 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 (as opposed to "mini-batch" 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 far more expensive. The efficient compromise between these two extremes is simply to use mini-batches of reasonable size.

Albeit the figure above illustrates gradient descent in a 1D parameter space, in practice we operate gradient descent in highly-dimensional spaces: every single weight coefficient in a neural network is a free dimension in the space, and there may be tens of thousands or even millions of them. To help you build intuition about loss surfaces, you

could also visualize gradient descent along a 2D loss surface, as in Figure 2.11. But we cannot possibly visualize what the actual process of training a neural network looks like—we cannot represent a 1,000,000-dimensional space in a way that makes sense to humans. As such, it is good to keep in mind that the intuitions we develop through these low-dimensional representations may not always be accurate in practice. This has historically been a source of issues in the world of deep learning research.

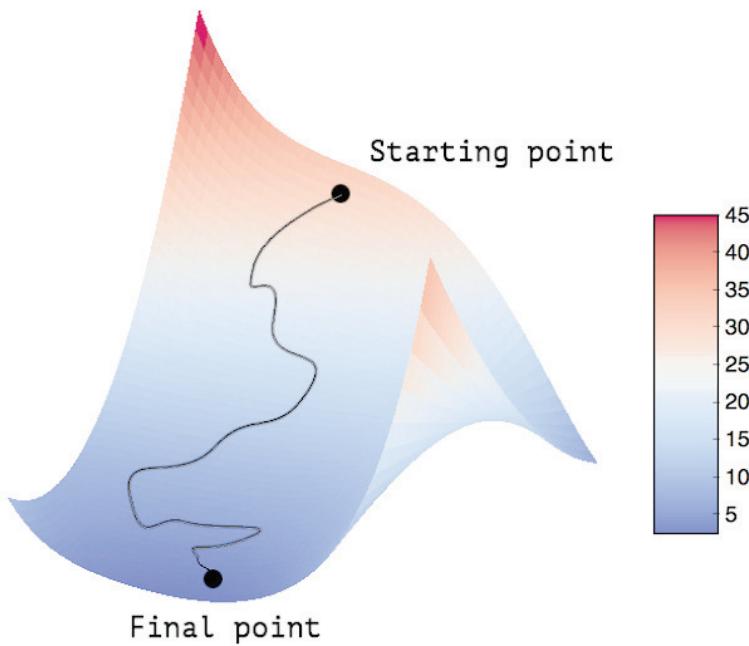


Figure 2.11 Gradient descent down a 2D loss surface (2 learnable parameters).

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 in 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:

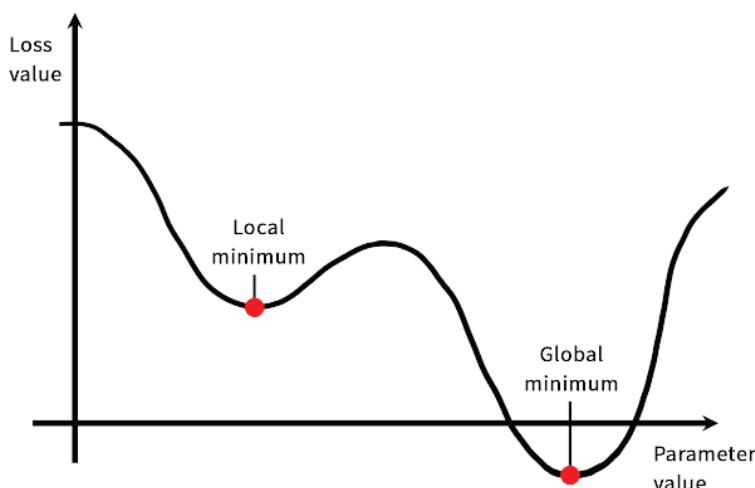


Figure 2.12 A local minimum and a global minimum

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 2.38 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 a , b , and c , with weight matrices w_1 , w_2 and w_3 :

$$f(w_1, w_2, w_3) = a(w_1, b(w_2, c(w_3)))$$

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. All you need is to have a good intuition for how gradient-based optimization works.

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: for instance, it could be the "RMSprop" optimizer, "SGD with momentum", and so on.

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 2.39 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 2.40 Our network

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

Now you understand that this network consists of a chain of two Dense layers, that each layer 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 2.41 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 2.42 The training loop

```
network.fit(train_images, train_labels, 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.

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, networks, objective functions and optimizers.
- Get a quick introduction to Keras, the Python deep learning library which we will use throughout the book.
- Set up a workstation for deep learning, with TensorFlow, Theano, Keras, and GPU support.
- 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* (or *model*).
- 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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

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.

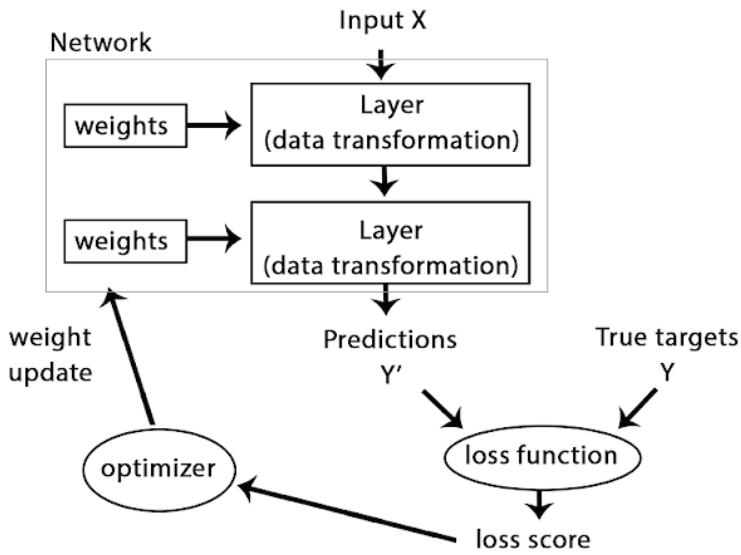


Figure 3.1 Relationship between network, layers, loss function and optimizer

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

3.1.1 Layers: the Lego bricks of deep learning

The fundamental data structure in neural networks is the "layer", to which you have already been introduced 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 2D convolution layers (`Conv2D`).

You can think of layers as the Lego bricks 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 import layers

# A dense layer with 32 output units
layer = layers.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 import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, input_shape=(784,)))
model.add(layers.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 choosing 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, we still have to pick two more 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 can follow to choose 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, all of our code examples 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 generative adversarial 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 3.6 (as of mid-2017). Its documentation is available at keras.io.

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

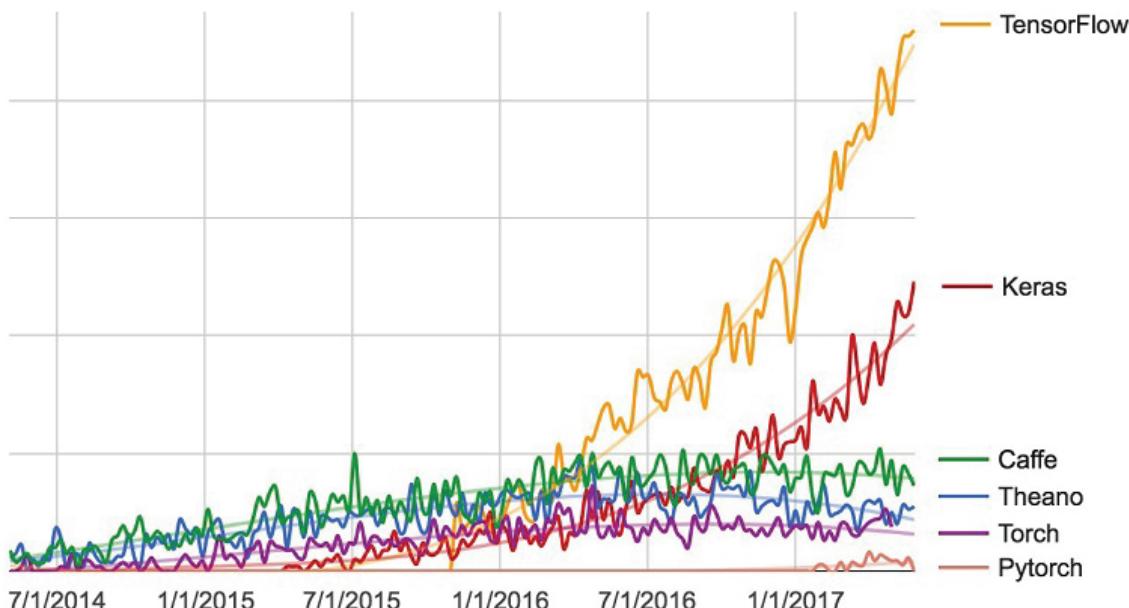


Figure 3.2 Google web search interest for different deep learning frameworks over time

3.2.1 Keras, TensorFlow, Theano, and CNTK

Keras is a model-level library, providing high-level building blocks for developing deep learning models. It 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 three existing backend implementations are the *TensorFlow* backend, the *Theano* backend, and the *CNTK* backend. In the future, it is likely that Keras will be extended to work with even more deep learning execution engines.

TensorFlow, CNTK, and Theano are some of the main platforms for deep learning today. Theano is developed by the MILA lab at *Université de Montréal*, while TensorFlow is developed by Google, and CNTK is developed by Microsoft. Any piece of code that you write with Keras can be run with any of these backends without having to change anything to the code: you can seamlessly switch between the two during development, which often proves useful, for instance if one of these backends proves to be faster for a specific task. By default, I would recommend using the TensorFlow backend for most of your deep learning needs.

Via TensorFlow (or Theano, or CNTK), 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.

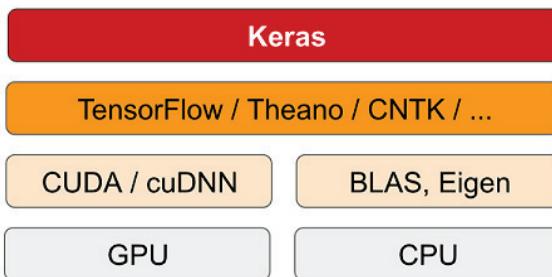


Figure 3.3 The deep learning software and hardware stack.

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 just like our example:

- Define your training data: input tensors and target tensors.
- Define a network of layers (or *model*) that maps 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 by calling the `fit` method of your model.

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 import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.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 7. Until Chapter 6, we will only be using the `Sequential` class in our code examples.

Listing 3.5 A network definition using the functional API

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

model = models.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 import optimizers

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

Lastly, the learning process itself consists of 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, 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 and sequence processing with recurrent neural 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 10x 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, or on Google Cloud Platform. But note that that cloud GPU instances can get quite expensive over time.

Also, whether running locally or in the cloud, it is better for you to be using a Unix workstation. While it is technically possible to use Keras on Windows (all three Keras backends support Windows), we don't recommend it. In the installation instructions we provide as an appendix to this book, we will consider an Ubuntu machine. If you are a Windows user, the simplest solution to get everything running is to set up an Ubuntu dual boot on your machine. It may seem like a hassle, but using Ubuntu will save you a lot of time and a lot of trouble in the long run.

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

3.3.2 Jupyter notebooks: the prefered way to run deep learning experiments

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

You can find more information about Jupyter at jupyter.org.

We recommend using Jupyter notebooks to get started with Keras, albeit that is not a requirement: you could also be running standalone Python scripts, or you could be running code from within an IDE such as PyCharm. We are making all code examples in this book available as open-source Notebooks, downloadable online.

3.3.3 Getting Keras running: two options

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

- Use the official EC2 "Deep Learning" AMI and run Keras experiments as Jupyter notebooks on EC2.
 - Do this preferably if you do not already have a GPU on your local machine.
 - We provide a step by step guide in the appendix "Running Jupyter notebooks on a EC2 GPU instance".
- Install everything from scratch on a local Unix workstation. You can then either run local Jupyter Notebooks or a regular Python codebase.
 - Do this if you already have a high-end NVIDIA GPU.
 - We provide a step by step guide in the appendix "Installing Keras and its dependencies on Unix".

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

3.3.4 Running deep learning jobs in the cloud: pros and cons

If you don't already have a GPU that you can use for deep learning (a recent, high-end NVIDIA GPU), then running deep learning experiments in the cloud is a simple, low-cost way for you to get started without having to buy any additional hardware. If you are using Jupyter notebooks, the experience of running in the cloud is no different from running locally. As of mid-2017, the cloud offering that makes it easiest to get started with deep learning is definitely AWS EC2. In appendix, we provide a step-by-step guide to start running Jupyter notebooks on a EC2 GPU instance.

However, if you are a heavy user of deep learning, this setup is not sustainable in the long term—or even past a few weeks. EC2 instances are expensive: the instance type we recommend in our appendix, the p2.xlarge instance, which will not provide you with much power, already costs \$0.90 per hour (as of mid-2017). Meanwhile, a solid

consumer-class GPU will cost you somewhere between \$1,000 and \$1,500—a price that has been fairly stable over time, even as the specs of these GPUs keep improving. If you are serious about deep learning, you should set up a local workstation with one or more GPUs.

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

3.3.5 What is the best GPU for deep learning?

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

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

From this section onwards, we will assume that you have access to a machine with Keras and its dependencies installed—preferably with GPU support. Make sure you get this step done before you go any further. Go through our step-by-step guides provided in appendix, and look online if you need further help. There is no shortage of tutorials on how to install Keras and common deep learning dependencies.

We can now start diving into practical Keras examples.

3.4 Classifying movie reviews: a binary classification example

Two-class classification, or binary classification, may be the most widely applied kind of machine learning problem. In this example, 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 3.8 Loading the IMDB dataset

```
from keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
```

The argument `num_words=10000` means 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 3.9 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 3.10 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 3.11 Decoding the integer sequences back into sentences

```
# word_index is a dictionary mapping words to an integer index
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 3.12 Encoding the integer sequences into a binary matrix

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((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 3.13 An encoded sample

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

We should also vectorize our labels, which is straightforward:

Listing 3.14 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 be 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 a `relu` activation 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 you 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.

In the next chapter, you will learn formal principles to guide you in making these choices. 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). A `relu` (rectified linear unit) is a function meant to zero-out negative values (see Figure 3.4) while a sigmoid "squashes" arbitrary values into the `[0, 1]` interval, (see Figure 3.5), thus outputting something that can be interpreted as a probability.

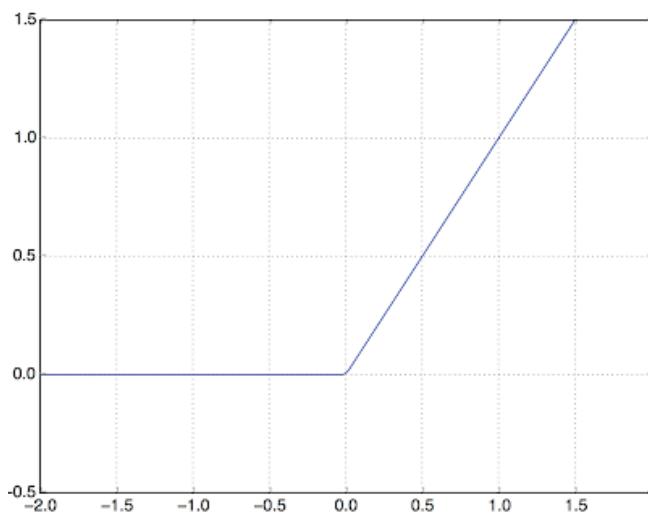


Figure 3.4 The Rectified Linear Unit function

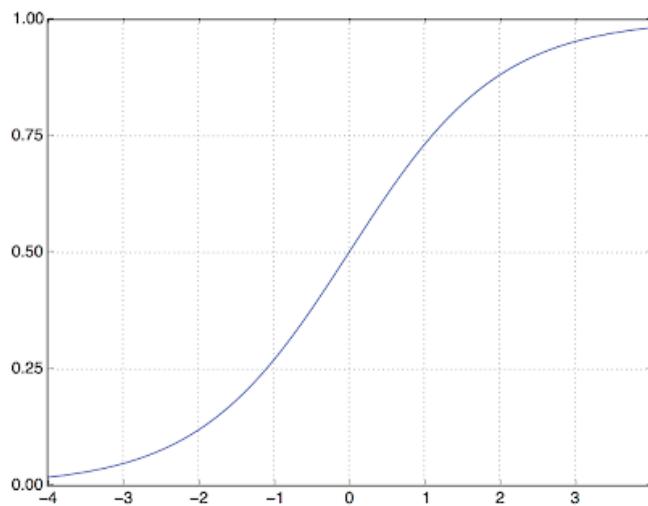


Figure 3.5 The sigmoid function

Here's what our network looks like:

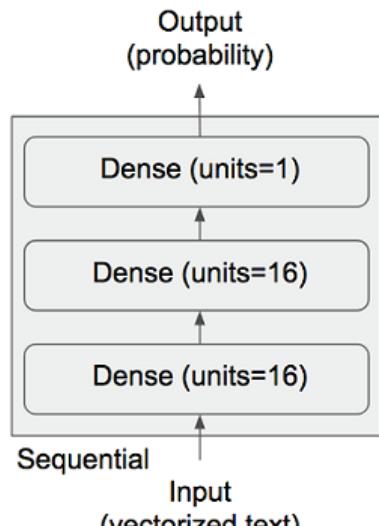


Figure 3.6 Our 3-layer network

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

Listing 3.15 Our model definition

```
from keras import models
from keras import layers

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

NOTE**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 of two linear operations, a dot product and an addition.

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

So the layer could only learn *linear transformations* (affine transformations) of the input data, 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 Theory, 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 3.16 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 3.17 Configuring the optimizer

```
from keras import optimizers

model.compile(optimizer=optimizers.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 3.18 Using custom losses and metrics

```
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss=losses.binary_crossentropy,
              metrics=[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 a "validation set" by setting apart 10,000 samples from the original training data:

Listing 3.19 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 3.20 Training our model

```
history = model.fit(partial_x_train,
                     partial_y_train,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

```
epochs=20,  
batch_size=512,  
validation_data=(x_val, y_val))
```

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 3.21 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 3.22 Plotting the training and validation loss

```
import matplotlib.pyplot as plt  
  
acc = history.history['acc']  
val_acc = history.history['val_acc']  
loss = history.history['loss']  
val_loss = history.history['val_loss']  
  
epochs = range(1, len(acc) + 1)  
  
# "bo" is for "blue dot"  
plt.plot(epochs, loss, 'bo', label='Training loss')  
# b is for "solid blue line"  
plt.plot(epochs, val_loss, 'b', label='Validation loss')  
plt.title('Training and validation loss')  
plt.xlabel('Epochs')  
plt.ylabel('Loss')  
plt.legend()  
  
plt.show()
```

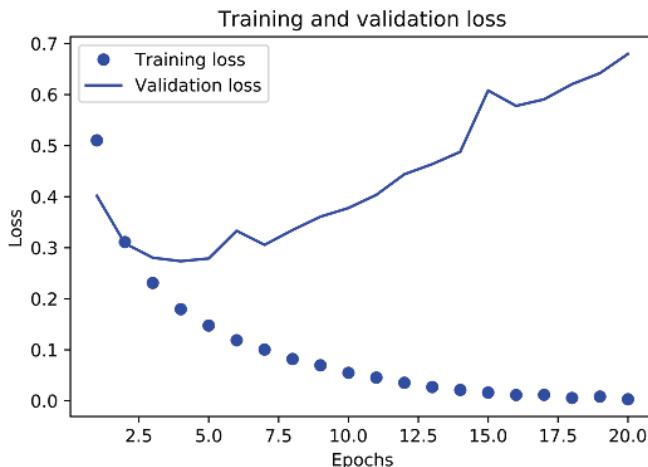


Figure 3.7 Training and validation loss

Listing 3.23 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, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

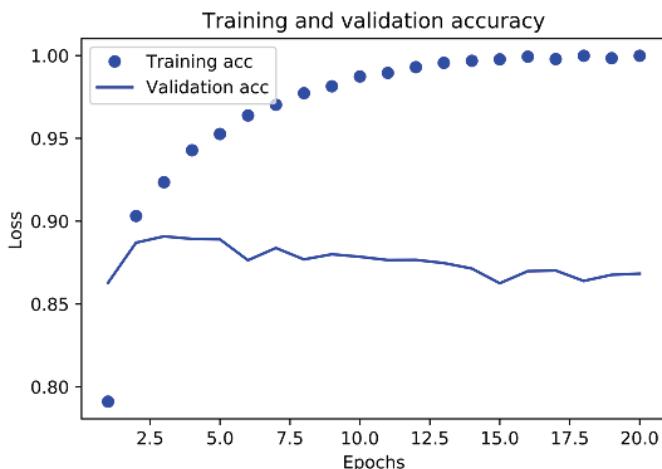


Figure 3.8 Training and validation accuracy

The dots are the training loss and accuracy, while the solid lines 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 range 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 3.24 Re-training a model from scratch

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

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

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

Listing 3.25 Our final results

```
>>> results
[0.2929924130630493, 0.8832799999999995]
```

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 3.26 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 confident 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 Wrapping up

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 sequences 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 you will likely use them frequently.
- In a binary classification problem (two output 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 increasingly 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 densely-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". If each data point could have belonged to multiple categories (in our case, topics) then we would be facing a "multi-label, multi-class classification" problem.

3.5.1 The Reuters dataset

We will 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

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

Listing 3.27 Loading the Reuters dataset

```
from keras.datasets import reuters
(train_data, train_labels), (test_data, test_labels) = reuters.load_data(num_words=10000)
```

Like with the IMDB dataset, the argument `num_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 3.28 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 3.29 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 3.30 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 3.31 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 our previous example:

Listing 3.32 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 just 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". For a more detailed explanation of one-hot encoding, you can refer to Chapter 6, Section 1. In our case, one-hot encoding of our labels consists in embedding each label as an all-zero vector with a 1 in the place of the label index, e.g.:

Listing 3.33 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, which you have already seen in action in our MNIST example:

Listing 3.34 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 too limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information.

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

Listing 3.35 Our model definition

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.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 3.36 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 3.37 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 3.38 Training our model

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

Let's display its loss and accuracy curves:

Listing 3.39 Plotting the training and validation loss

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

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

plt.show()
```

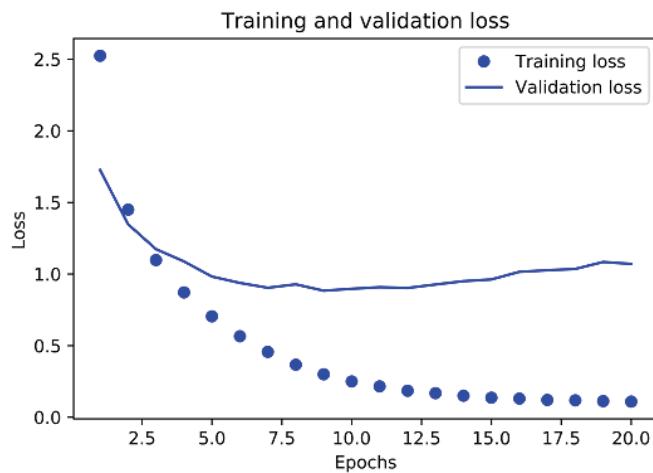


Figure 3.9 Training and validation accuracy

Listing 3.40 Plotting the training and validation accuracy

```
plt.clf()    # clear figure

acc = history.history['acc']
val_acc = history.history['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

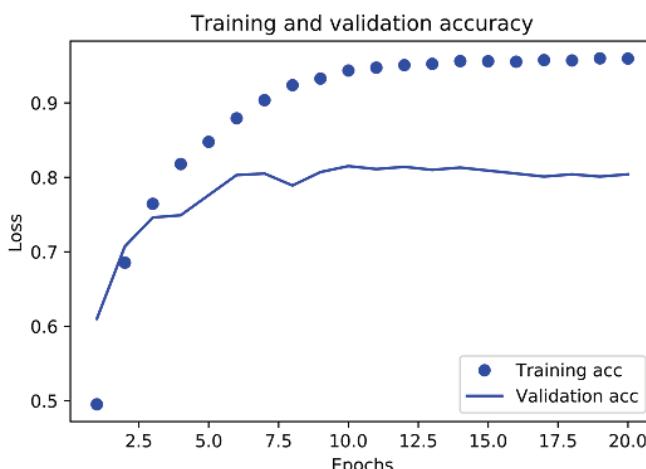


Figure 3.10 Training and validation accuracy

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 3.41 Re-training a model from scratch

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

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=9,
          batch_size=512,
          validation_data=(x_val, y_val))
results = model.evaluate(x_test, one_hot_test_labels)
```

Listing 3.42 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 3.43 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 3.44 Generating predictions for new data

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

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

Listing 3.45 Taking a look at our predictions

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

The coefficients in this vector sum to 1:

Listing 3.46 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 3.47 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 3.48 Encoding the labels as integer arrays

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

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 3.49 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. 4-dimensional.

Listing 3.50 A model with an information bottleneck

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

Our network now seems to peak at ~71% validation accuracy, a 8% 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 a single hidden layer, or three hidden layers.

3.5.9 Wrapping up

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

- If you 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` function.
- If you need to classify data into a large number of 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 discrete label of an input data point. Another common type of machine learning problem is "regression", which consists of predicting a continuous value instead of a discrete label. For instance, predicting the temperature tomorrow, given meteorological data, or predicting the time that a software project will take to complete, given its specifications.

Do not mix up "regression" with the algorithm "logistic regression": confusingly, "logistic regression" is not a regression algorithm, it is 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 has another interesting difference from our two previous examples: it has very few data points, only 506 in total, split between 404 training samples and 102 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 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 3.51 Loading the Boston housing dataset

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

Listing 3.52 Taking a look at the data

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

As you can see, we have 404 training samples and 102 test samples. 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. $1000 * (\text{Bk} - 0.63)^2$ where Bk is the proportion of Black people by town.
13. % lower status of the population.

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

Listing 3.53 Taking a look at the targets

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

The prices are typically between \$10,000 and \$50,000. If that sounds cheap, remember 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 heterogeneous data, but it would definitely make learning more difficult. A widespread best practice to deal with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), 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 3.54 Normalizing the data

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

test_data -= mean
test_data /= std
```

Note that the quantities 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 data normalization.

3.6.3 Building our network

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

Listing 3.55 Our model definition

```
from keras import models
from keras import layers

def build_model():
    # Because we will need to instantiate
    # the same model multiple times,
    # we use a function to construct it.
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                          input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

Our network ends with a single unit, and no activation (i.e. it will be linear layer). 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 difference 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 evaluating our model.

The best practice in such situations is to use K-fold cross-validation. It consists of 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.

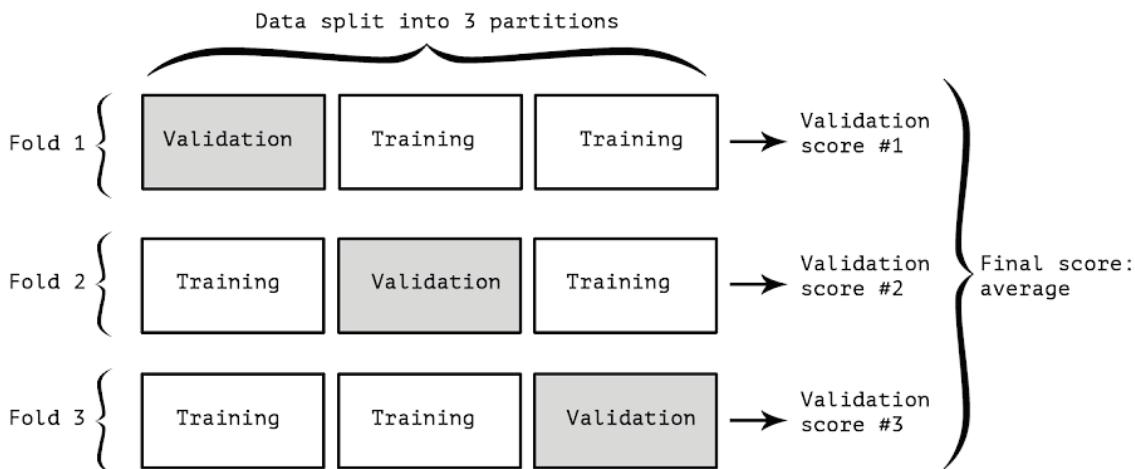


Figure 3.11 3-fold cross-validation

In terms of code, this is straightforward:

Listing 3.56 K-fold validation

```
import numpy as np

k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
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,
              epochs=num_epochs, 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 `num_epochs = 100` yields the following results:

Listing 3.57 Validation MAE scores for successive "folds"

```
>>> all_scores
[2.588258957792037, 3.1289568449719116, 3.1856116051248984, 3.0763342615401386]
>>> np.mean(all_scores)
2.9947904173572462
```

As you can notice, the different runs do indeed show rather different validation scores. This is because each fold uses a different subset of the data for validation, and the validation set is not necessarily representative of the full dataset.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

scores, from 2.6 to 3.2. Their average (3.0) is a much more 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 \$3,000 on average, which is still significant considering that the prices range from \$10,000 to \$50,000.

Let’s try training the network for a bit longer: 500 epochs. 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 3.58 Saving the validation logs at each fold

```
num_epochs = 500
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),
                         epochs=num_epochs, 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 3.59 Building the history of successive mean K-fold validation scores

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

Let’s plot this:

Listing 3.60 Plotting validation scores

```
import matplotlib.pyplot as plt

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

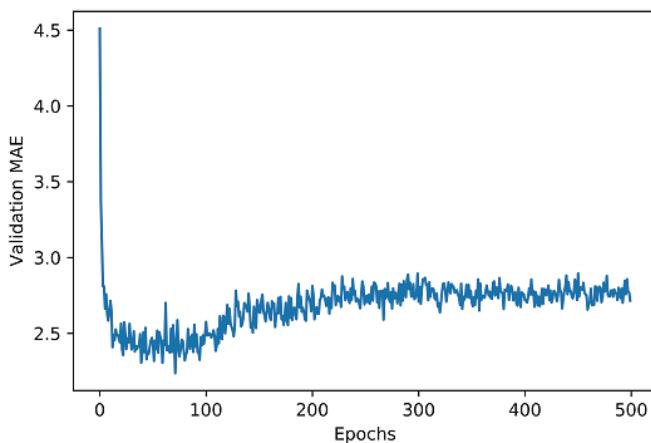


Figure 3.12 Validation MAE by epoch

It may be a bit hard to see the plot due to scaling issues and relatively high variance. Let's:

- Omit the first 10 data points, which are on a different scale from the rest of the curve.
- Replace each point with an exponential moving average of the previous points, to obtain a smooth curve.

Listing 3.61 Plotting validation scores - excluding the first 10 data points

```
def smooth_curve(points, factor=0.9):
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
    return smoothed_points

smooth_mae_history = smooth_curve(average_mae_history[10:])

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

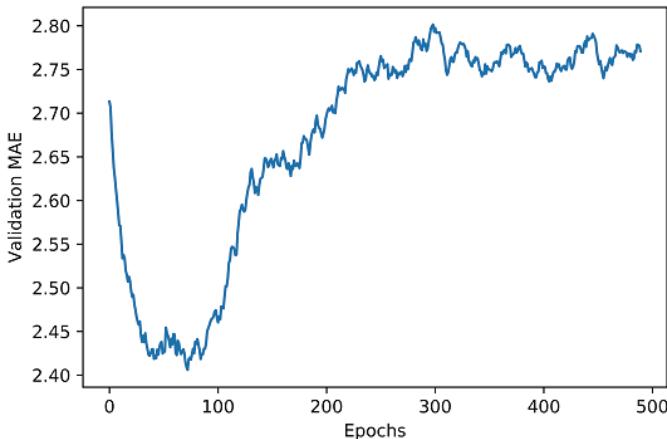


Figure 3.13 Validation MAE by epoch - excluding the first 10 data points

According to this plot, it seems that validation MAE stops improving significantly after about 80 epochs. Past that point, we start overfitting.

Once we are done tuning other parameters of our model (besides the number of epochs, we could also adjust the size of the hidden layers), 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 3.62 Training the final model

```
# Get a fresh, compiled model.
model = build_model()
# Train it on the entirety of the data.
model.fit(train_data, train_targets,
          epochs=80, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

Listing 3.63 Our final result

```
>>> test_mae_score
2.5532484335057877
```

We are still off by about \$2,550.

3.6.5 Wrapping up

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 or two), in order to avoid severe overfitting.

This example concludes our series of three introductory practical examples. You are now able to handle common types of problems with vector data input:

- Binary (2-class) classification.
- Multi-class, single-label classification.
- Scalar regression.

In the next chapter, you will acquire a more formal understanding of some of the concepts you have encountered in these first examples, such as data preprocessing, model evaluation, and overfitting.

Fundamentals of machine learning



After three practical examples, you are starting to get familiar with how to approach classification and regression problems using neural networks, and you have witnessed the central problem of machine learning: overfitting. This chapter will formalize some of the intuition you are starting to form into a solid conceptual framework for attacking and solving deep learning problems.

In this chapter, you will:

- Learn about more forms of machine learning, beyond classification and regression.
- Learn about formal evaluation procedures for machine learning models, a simple version of which you have already seen in action a few times.
- Learn how to prepare data for deep learning, and what is "feature engineering".
- Learn ways to tackle the central problem of machine learning: overfitting, which we faced in all of our three previous examples.

Finally, we will consolidate all these concepts—model evaluation, data preprocessing and feature engineering, tackling overfitting—into a detailed 7-step workflow for tackling any machine learning problem.

4.1 Four different brands of machine learning

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

Supervised learning is just the tip of the iceberg. Machine learning is a vast field with a complex subfield taxonomy. Machine learning algorithms generally fall into four broad categories:

4.1.1 Supervised learning.

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

4.1.2 Unsupervised learning.

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

4.1.3 Self-supervised learning.

This is actually a specific instance of supervised learning, but it different enough that it deserves its own category. Self-supervised learning is supervised learning without human-annotated labels. There are still labels involved (since the learning has to be supervised by something), but they are generated from the input data itself, typically using a heuristic algorithm. You can think of it as supervised learning without any humans in the loop. For instance, "autoencoders" are a well-known instance of self-supervised learning, where the generated targets are... the input themselves, unmodified. In the same way, trying to predict the next frame in a video given past frames, or the next word in a text given previous words, would be another instance of self-supervised learning (temporally supervised learning, in this case: supervision comes from future input data). Note that the distinction between supervised, self-supervised and unsupervised learning can be blurry sometimes—these categories are more of continuum without solid frontiers. Self-supervised learning can be reinterpreted as either supervised or unsupervised learning depending on whether you pay attention to the learning mechanism or to the context of its application.

4.1.4 Reinforcement learning.

Long overlooked, this branch of machine learning has recently started getting a lot of attention, after Google DeepMind successfully applied it to learning to play Atari games (and later, to learning to play Go at the highest level). In reinforcement learning, an "agent" receives information about its environment and learns to pick actions that will maximize some reward. For instance, a neural network that "looks" at a video game screen and outputs game actions in order to maximize its score can be trained via reinforcement learning. Currently, reinforcement learning is mostly a research area and has not yet had significant practical successes beyond games. In time, however, I would expect to see reinforcement learning take over an increasingly large range of real-world applications—self-driving, robotics, resource management, education... It is an idea whose time has come, or will come soon.

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

Although supervised learning mostly consists of classification and regression, there are more exotic variants as well:

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

4.1.5 Classification and regression glossary

Classification and regression involve many specialized terms. You have already come across some of them in our first examples, and you will see more of them come up in the following chapters. They have precise, machine-learning specific definitions, and you should be familiar with them.

Sample, or input: one data points that goes into your model.

Prediction, or output: what goes out of your model.

Target: the truth. What your model should ideally have predicted, according to an external source of data.

Prediction error, or loss value: a measure of the distance between your model's prediction and the target.

Classes: set of possible labels to choose from in a classification problem, e.g. when

classifying cat and dog pictures, "dog" and "cat" are the two classes.

Label: specific instance of a class annotation in a classification problem. For instance, if picture #1234 is annotated as containing the class "dog", then "dog" is a label of picture #1234.

Ground-truth, or annotations: all targets for a dataset, typically collected by humans.

Binary classification: classification task where each input sample should be categorized into two exclusive categories.

Multi-class classification: classification task where each input sample should be categorized into more than two categories: for instance, classifying handwritten digits is a multi-class classification task.

Multi-label classification: classification task where each input sample can be assigned multiple labels. For instance, a given image may contain both a cat and a dog, and should be annotated both with the "cat" label and the "dog" label. The number of labels per image is usually variable.

Scalar regression: task where the target is a continuous scalar value. House price prediction is a good example: the different target prices form a continuous space.

Vector regression: task where the target is a set of continuous values, e.g. a continuous vector. If you are doing regression against multiple values (e.g. the coordinates of a bounding box in an image) then your are doing vector regression.

Mini-batch or *batch*: a small set of samples that are being processed at once by the model (typically between 8 and 128 samples). It is often a power of 2 in order to facilitate memory allocation on GPU. When training, a mini-batch is used to compute a *single* gradient descent update applied to the weights of the model.

4.2 Evaluating machine learning models

In the three examples we covered in the previous chapters, we split our data into a training set, a validation set, and a test set. The reason why we did not evaluate our models on the same data as they were trained on quickly became evident: after just a few epochs, all three models started to *overfit*, which is to say that their performance on never-seen-before data started stalling (or even worsening) compared to their performance on the training data—which always go up as training progresses.

In machine learning, our goal is to achieve models that *generalize*, i.e. that perform well on never-seen-before data, and overfitting is the central obstacle. We can only control that which we can observe, so it is crucial to be able to reliably measure the generalization power of our model. In the next sections, we will take a look at strategies for mitigating overfitting and maximizing generalization. In the present section, we will focus on how we can measure generalization, i.e. how to evaluate machine learning models.

4.2.1 Training, validation, and test sets

Evaluating a model always boils down to splitting your available data into three sets: training, validation, and test set. You train on the training data, and evaluate your model on the validation data. Once your model is ready for prime time, you test it one final time on the test data.

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

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

Central to this phenomenon is the notion of "information leak". Every time you are tuning a hyperparameter of your model based on the model's performance on the validation set, some information about the validation data is leaking into your model. If you only do this once, for one parameter, then very few bits of information would be leaking and your validation set would remain a reliable way to evaluate your model. But if you repeat this many times, running one experiment, evaluating on the validation set, modifying your model as a result, then you are leaking an increasingly significant amount of information about the validation set into your model.

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

Splitting your data into a training, validation, and test sets may seem straightforward, but there are a few advanced ways to do it which can come in handy when very few data is available. Let's review three classic evaluation recipes.

4.2.2 Simple hold-out validation

Set apart some fraction of your data as your test set. Train on remaining data, evaluate on the test set. As you saw in the previous sections, in order to prevent information leaks, you should not tune your model based on the test set, and therefore you should *also* reserve a validation set.

Schematically, hold-out validation looks like this:



Figure 4.1 Simple hold-out validation split

Here's a simple implementation:

Listing 4.1 Hold-out validation

```
num_validation_samples = 10000

# Shuffling the data is usually appropriate
np.random.shuffle(data)

# Define the validation set
validation_data = data[:num_validation_samples]
data = [num_validation_samples:]

# Define the training set
training_data = data[:]

# Train a model on the training data
# and evaluate it on the validation data
model = get_model()
model.train(training_data)
validation_score = model.evaluate(validation_data)

# At this point you can tune your model,
# retrain it, evaluate it, tune it again...

# Once you have tuned your hyperparameters,
# is it common to train your final model from scratch
# on all non-test data available.
model = get_model()
model.train(np.concatenate([training_data,
                           validation_data]))
test_score = model.evaluate(test_data)
```

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

4.2.3 K-fold validation

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

Schematically, K-fold cross-validation looks like this:

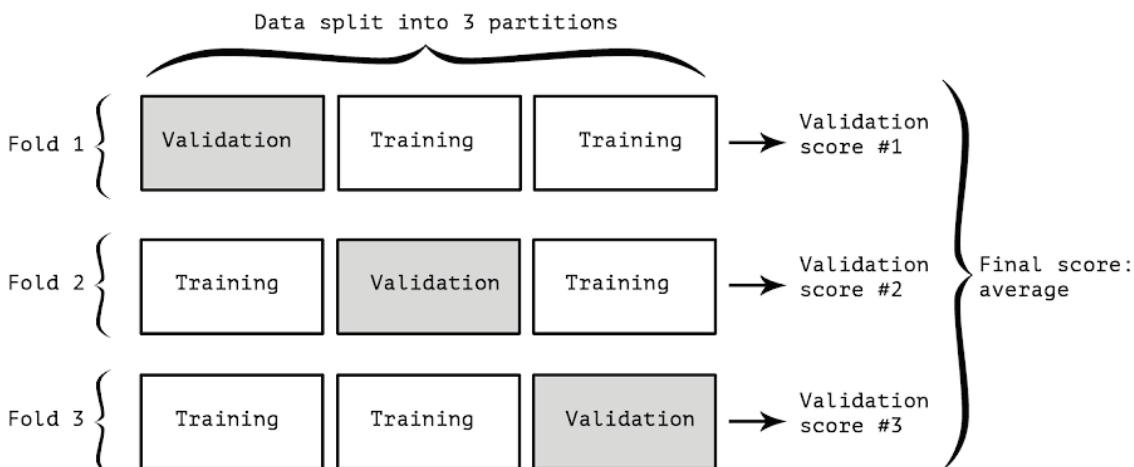


Figure 4.2 3-fold validation

Here's a simple implementation:

Listing 4.2 K-fold cross-validation

```

k = 4
num_validation_samples = len(data) // k

np.random.shuffle(data)

validation_scores = []
for fold in range(k):
    # Select the validation data partition
    validation_data = data[num_validation_samples * fold: num_validation_samples * (fold + 1)]
    # The remainder of the data is used as training data.
    # Note that the "+" operator below is list concatenation, not summation
    training_data = data[:num_validation_samples * fold] + data[num_validation_samples * (fold + 1):]

    # Create a brand new instance of our model (untrained)
    model = get_model()
    model.train(training_data)
    validation_score = model.evaluate(validation_data)
    validation_scores.append(validation_score)

# This is our validation score:
# the average of the validation scores of our k folds
validation_score = np.average(validation_scores)

# We train our final model on all non-test data available
model = get_model()
model.train(data)
test_score = model.evaluate(test_data)
  
```

4.2.4 Iterated K-fold validation with shuffling

This one is for situations in which you have relatively little data available and you need to evaluate your model as precisely as possible. I have found it to be extremely helpful in Kaggle competitions. It consists of applying K-fold validation multiple times, shuffling the data every time before splitting it K-ways. Your final score would be the average of the scores obtained at each run of K-fold validation. Note that you end up training and evaluating $P * K$ models (where P is the number of iterations you use), which can be very expensive.

4.2.5 Keep in mind...

There are a few things to keep an eye out for when picking an evaluation protocol:

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

4.3 Data preprocessing, feature engineering and feature learning

Besides model evaluation, an important question we must tackle before we dive deeper into model development is the following: how to prepare the input data and targets before feeding them into a neural network? Many data preprocessing and feature engineering techniques are domain-specific (e.g. specific to text data or image data), and we will cover those in the next chapters as we encounter them in practical examples. For now, we will review the basics, common to all data domains.

4.3.1 Data preprocessing for neural networks

VECTORIZATION

All inputs and targets in a neural network must be tensors of floating point data (or in specific cases, tensors of integers). Whatever data you need to process—sound, images, text—you must first turn it into tensors, a step called "data vectorization". For instance, in our two previous text classification examples, we started from text represented as lists of integers (standing for sequences of words), and we used "one-hot encoding" to turn them into a tensor of `float32` data. In the digits classification example and house price prediction example, the data already came in vectorized form, so we could skip this step.

VALUE NORMALIZATION

In our digits classification example, we started from image data encoded as integers in the 0-255 range, encoding grayscale values. Before we fed this data into our network, we had to cast it to `float32` and divide by 255, so we would end up with floating point values in the 0-1 range. Similarly, in our house price prediction example, we started from features that took a variety of ranges—some features had small floating point values, others had fairly large integer values. Before we fed this data into our network, we had to normalize each feature independently so that each feature would have a standard deviation of 1 and a mean of 0.

In general, it isn't safe to feed into a neural network data that takes relatively "large" values (e.g. multi-digit integers, which is much larger than the initial values taken by the weights of a network), or data that is "heterogeneous", e.g. data where one feature would be in the 0-1 range and another in the 100-200 range. It can trigger large gradient updates which will prevent your network from converging. To make learning easier for your network, your data should:

- Take "small" values: typically most values should be in the 0-1 range.
- Be homogenous, i.e. all features should take values roughly in the same range.

Additionally, the following stricter normalization practice is common and can definitely help, although it isn't always necessary (e.g. we did not do this in our digits classification example) :

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

This is easy to do with Numpy arrays:

Listing 4.3 Feature-wise normalization of 2D Numpy data

```
# Assuming x is a 2D data matrix of shape (samples, features)
x -= x.mean(axis=0)
x /= x.std(axis=0)
```

HANDLING MISSING VALUES

You may sometimes have missing values in your data. For instance, in our house price prediction example, the first feature (the column of index 0 in the data) was "per capita crime rate". What if this feature was not available for all samples? We would then have missing values in our training or test data.

In general, with neural networks, it is safe to input missing values as 0, under the condition that 0 is not already a meaningful value. The network will learn from exposure to the data that the value 0 simply means "missing data" and will start ignoring the value. However, note that if you are expecting missing values in the test data but the network was trained on data without any missing values, then the network will not have learned to ignore missing values! In this situation, then you should artificially generate training samples with missing entries: simply copy some training samples several times and drop some of the features that you expect are susceptible to go missing in the test data.

4.3.2 Feature engineering

Feature engineering is the process of using your own knowledge about the data and about the machine learning algorithm at hand (in our case a neural network) to make the algorithm work better by applying hard-coded (non-learned) transformations to the data before it goes into the model. In many cases, it isn't reasonable to expect a machine learning model to be able to learn from completely arbitrary data. The data needs to be presented to the model in a way that will make the job of the model easier. One intuitive example of this is the following: suppose that we are trying to develop a model that can take as input an image of a clock, and can output the time of the day.

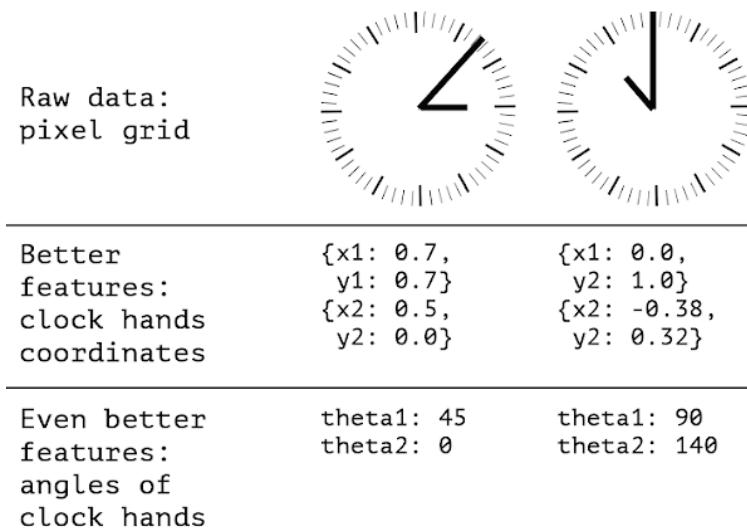


Figure 4.3 Feature engineering for reading time on a clock

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

However, if you already understand the problem at a high-level (you understand how humans read time on a clock face), then you can come up with much better input features for a ML algorithm: for instance, it is easy to write a 5-line Python script to follow the black pixels of the clock hands and output the (x, y) coordinates of the tip of each hand. Then a very simple ML algorithm can learn to associate these coordinates with the appropriate time of the day.

You can go even further: you can do a coordinate change, and express the (x, y) coordinates as polar coordinates with regard to the center of the image. Your input would simply become... the angle `theta` of each clock hand. At this point your features are making the problem so easy that no machine learning is required anymore; a simple rounding operation and dictionary lookup are enough to recover the approximate time of day.

That's the essence of feature engineering: making a problem easier by expressing it in a simpler way. It usually requires understanding the problem in-depth.

Before deep learning, feature engineering used to be critical, because classical "shallow" algorithms did not have hypothesis spaces rich enough to learn useful features by themselves. The way you would present the data to the algorithm would be essential to its success. For instance, before convolutional neural networks started becoming successful on the MNIST digits classification problem, solutions were typically based on hard-coded features such as the number of loops in a digit image, the height of each digit in an image, an histogram of pixel values, and so on.

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

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

4.4 Overfitting and underfitting

In all the examples we saw in the previous chapter—movie review sentiment prediction, topic classification, and house price regression—we could notice that the performance of our model on the held-out validation data would always peak after a few epochs and would then start degrading, i.e. our model would quickly start to *overfit* to the training data. Overfitting happens in every single machine learning problem. Learning how to deal with overfitting is essential to mastering machine learning.

The fundamental issue in machine learning is the tension between optimization and generalization. "Optimization" refers to the process of adjusting a model to get the best

performance possible on the training data (the "learning" in "machine learning"), while "generalization" refers to how well the trained model would perform on data it has never seen before. The goal of the game is to get good generalization, of course, but you do not control generalization; you can only adjust the model based on its training data.

At the beginning of training, optimization and generalization are correlated: the lower your loss on training data, the lower your loss on test data. While this is happening, your model is said to be *under-fit*: there is still progress to be made; the network hasn't yet modeled all relevant patterns in the training data. But after a certain number of iterations on the training data, generalization stops improving, validation metrics stall then start degrading: the model is then starting to over-fit, i.e. is it starting to learn patterns that are specific to the training data but that are misleading or irrelevant when it comes to new data.

To prevent a model from learning misleading or irrelevant patterns found in the training data, *the best solution is of course to get more training data*. A model trained on more data will naturally generalize better. When that is no longer possible, the next best solution is to modulate the quantity of information that your model is allowed to store, or to add constraints on what information it is allowed to store. If a network can only afford to memorize a small number of patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well.

The processing of fighting overfitting in this way is called *regularization*. Let's review some of the most common regularization techniques, and let's apply them in practice to improve our movie classification model from the previous chapter.

4.4.1 Fighting overfitting

REDUCING THE NETWORK'S SIZE

The simplest way to prevent overfitting is to reduce the size of the model, i.e. the number of learnable parameters in the model (which is determined by the number of layers and the number of units per layer). In deep learning, the number of learnable parameters in a model is often referred to as the model's "capacity". Intuitively, a model with more parameters will have more "memorization capacity" and therefore will be able to easily learn a perfect dictionary-like mapping between training samples and their targets, a mapping without any generalization power. For instance, a model with 500,000 binary parameters could easily be made to learn the class of every digits in the MNIST training set: we would only need 10 binary parameters for each of the 50,000 digits. Such a model would be useless for classifying new digit samples. Always keep this in mind: deep learning models tend to be good at fitting to the training data, but the real challenge is generalization, not fitting.

On the other hand, if the network has limited memorization resources, it will not be able to learn this mapping as easily, and thus, in order to minimize its loss, it will have to resort to learning compressed representations that have predictive power regarding the targets—precisely the type of representations that we are interested in. At the same time, keep in mind that you should be using models that have enough parameters that they

won't be underfitting: your model shouldn't be starved for memorization resources. There is a compromise to be found between "too much capacity" and "not enough capacity".

Unfortunately, there is no magical formula to determine what the right number of layers is, or what the right size for each layer is. You will have to evaluate an array of different architectures (on your validation set, not on your test set, of course) in order to find the right model size for your data. The general workflow to find an appropriate model size is to start with relatively few layers and parameters, and start increasing the size of the layers or adding new layers until you see diminishing returns with regard to the validation loss.

Let's try this on our movie review classification network. Our original network was as such:

Listing 4.4 Our original model

```
from keras import models
from keras import layers

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

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

Listing 4.5 A version of our model with lower capacity

```
model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Here's a comparison of the validation losses of the original network and the smaller network. The dots are the validation loss values of the smaller network, and the crosses are the initial network (remember: a lower validation loss signals a better model).

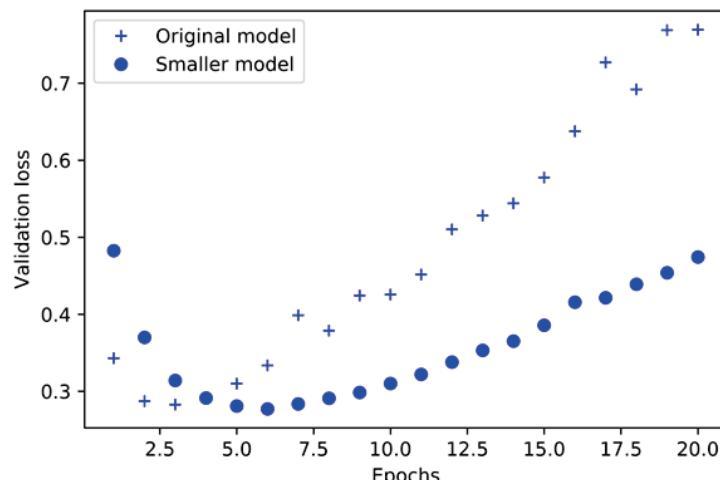


Figure 4.4 Effect of model capacity on validation loss: trying a smaller model

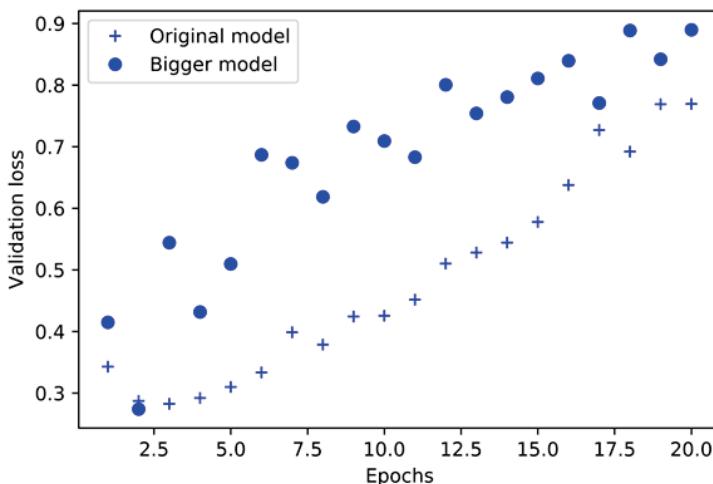
As you can see, the smaller network starts overfitting later than the reference one (after 6 epochs rather than 4) and its performance degrades much more slowly once it starts overfitting.

Now, for kicks, let's add to this benchmark a network that has much more capacity, far more than the problem would warrant:

Listing 4.6 A version of our model with higher capacity

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Here's how the bigger network fares compared to the reference one. The dots are the validation loss values of the bigger network, and the crosses are the initial network.

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

The bigger network starts overfitting almost right away, after just one epoch, and overfits much more severely. Its validation loss is also more noisy.

Meanwhile, here are the training losses for our two networks:

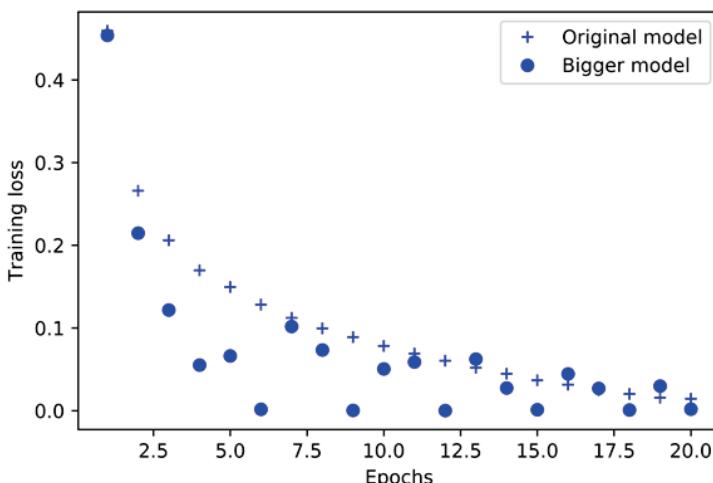


Figure 4.6 Effect of model capacity on training loss: trying a bigger model

As you can see, the bigger network gets its training loss near zero very quickly. The more capacity the network has, the quicker it will be able to model the training data (resulting in a low training loss), but the more susceptible it is to overfitting (resulting in a large difference between the training and validation loss).

ADDING WEIGHT REGULARIZATION

You may be familiar with *Occam's Razor* principle: given two explanations for something, the explanation most likely to be correct is the "simplest" one, the one that makes the least amount of assumptions. This also applies to the models learned by neural networks: given some training data and a network architecture, there are multiple sets of weights values (multiple *models*) that could explain the data, and simpler models are less likely to overfit than complex ones.

A "simple model" in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters altogether, as we saw in the section above). Thus a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to only take small values, which makes the distribution of weight values more "regular". This is called "weight regularization", and it is done by adding to the loss function of the network a *cost* associated with having large weights. This cost comes in two flavors:

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

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

Listing 4.7 Adding L2 weight regularization to our model

```
from keras import regularizers

model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                      activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                      activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

`l2(0.001)` means that every coefficient in the weight matrix of the layer will add $0.001 * \text{weight_coefficient_value}$ to the total loss of the network. Note that because this penalty is *only added at training time*, the loss for this network will be much higher at training than at test time.

Here's the impact of our L2 regularization penalty:

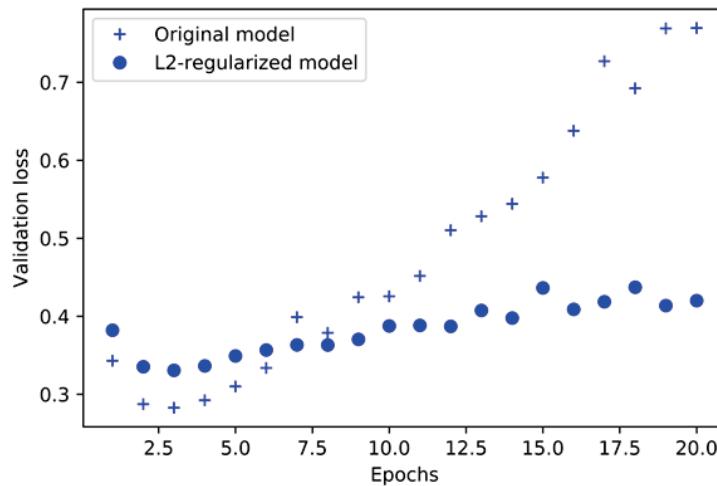


Figure 4.7 Effect of L2 weight regularization on validation loss

As you can see, the model with L2 regularization (dots) has become much more resistant to overfitting than the reference model (crosses), even though both models have the same number of parameters.

As alternatives to L2 regularization, you could use one of the following Keras weight regularizers:

Listing 4.8 Different weight regularizers available in Keras

```
from keras import regularizers

# L1 regularization
regularizers.l1(0.001)

# L1 and L2 regularization at the same time
regularizers.l1_l2(l1=0.001, l2=0.001)
```

ADDING DROPOUT

Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Hinton and his students at the University of Toronto. Dropout, applied to a layer, consists of randomly "dropping out" (i.e. setting to zero) a number of output features of the layer during training. Let's say a given layer would normally have returned a vector [0.2, 0.5, 1.3, 0.8, 1.1] for a given input sample during training; after applying dropout, this vector will have a few zero entries distributed at random, e.g. [0, 0.5, 1.3, 0, 1.1]. The "dropout rate" is the fraction of the features that are being zeroed-out; it is usually set between 0.2 and 0.5. At test time, no units are dropped out, and instead the layer's output values are scaled down by a factor equal to the dropout rate, so as to balance for the fact that more units are active than at training time.

Consider a Numpy matrix containing the output of a layer, `layer_output`, of shape `(batch_size, features)`. At training time, we would be zero-ing out at random a fraction of the values in the matrix:

Listing 4.9 Dropout implementation: dropping out units at training time

```
# At training time: we drop out 50% of the units in the output
layer_output *= np.randint(0, high=2, size=layer_output.shape)
```

At test time, we would be scaling the output down by the dropout rate. Here we scale by 0.5 (because we were previous dropping half the units):

Listing 4.10 Dropout implementation: test-time rescaling

```
# At test time:
layer_output *= 0.5
```

Note that this process can be implemented by doing both operations at training time and leaving the output unchanged at test time, which is often the way it is implemented in practice:

Listing 4.11 Alternative implementation of Dropout

```
# At training time:
layer_output *= np.randint(0, high=2, size=layer_output.shape)
# Note that we are scaling *up* rather scaling *down* in this case
layer_output /= 0.5
```

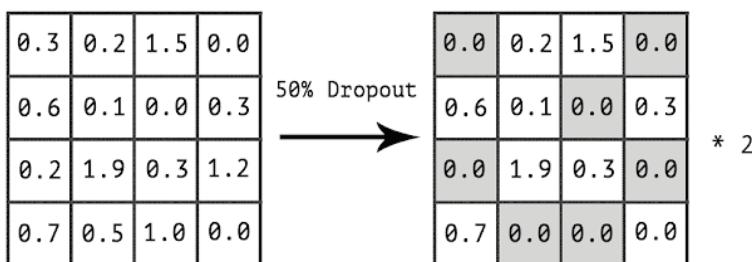


Figure 4.8 Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time, the activation matrix would be unchanged.

This technique may seem strange and arbitrary. Why would this help reduce overfitting? Geoff Hinton has said that he was inspired, among other things, by a fraud prevention mechanism used by banks—in his own words: *"I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting"*.

The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that are not significant (what Hinton refers to as "conspiracies"), which the network would start memorizing if no noise was present.

In Keras you can introduce dropout in a network via the `Dropout` layer, which gets applied to the output of layer right before it, e.g.:

Listing 4.12 Using Dropout in Keras

```
model.add(layers.Dropout(0.5))
```

Let's add two `Dropout` layers in our IMDB network to see how well they do at reducing overfitting:

Listing 4.13 Adding Dropout to our IMDB network

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

Let's plot the results:

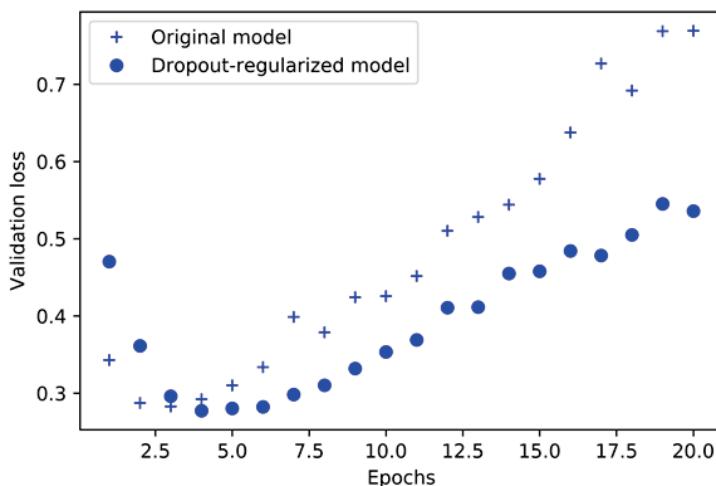


Figure 4.9 Effect of dropout on validation loss

Again, a clear improvement over the reference network.

To recap: here the most common ways to prevent overfitting in neural networks:

- Getting more training data.
- Reducing the capacity of the network.
- Adding weight regularization.
- Adding dropout.

4.5 The universal workflow of machine learning

What we present here is a universal blueprint you can use to attack and solve any machine learning problem, tying together the different concepts you learned about in this chapter: problem definition, evaluation, feature engineering, and fighting overfitting.

4.5.1 Define the problem and assemble a dataset

First, you must define the problem at hand:

- What will your input data will be? What will you be trying to predict? You can only learn to predict something if you have available training data, e.g. you can only learn to classify the sentiment of movie reviews if you have both movie reviews and sentiment annotations available. As such, data availability is usually the limiting factor at this stage (unless you have the means to pay people to collect data for you).
- What type of problem are you facing—is it binary classification? Multi-class classification? Scalar regression? Vector regression? Multi-class, multi-label classification? Something else, like clustering, generation or reinforcement learning? Identifying the problem type will guide your choice of model architecture, loss function, and so on.

You cannot move to the next stage until you know what your inputs and outputs are, and what data you will be using. Be aware of the hypotheses that you are making at this stage:

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

Until you have a working model, then these are merely hypotheses, waiting to be validated or invalidated. Not all problems can be solved; just because you have assembled examples of inputs X and targets Y doesn't mean that X contains enough information to predict Y. For instance, if you are trying to predict the movements of a stock on the stock market given its recent price history, you are unlikely to succeed, since price history simply doesn't contain much predictive information.

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

Keep it in mind: machine learning can only be used to memorize patterns which are present in your training data. You can only recognize what you have seen before. Using machine learning trained on past data to predict the future is making the assumption that the future will behave like the past. That is often not the case.

4.5.2 Pick a measure of success

To control something, you need to be able to observe it. To achieve success, you must define what you mean by success—accuracy? Precision-Recall? Customer retention rate? Your metric for success will guide the choice of your loss function, i.e. the choice of what your model will optimize. It should directly align with your higher-level goals, such as the success of your business.

For balanced classification problems, where every class is equally likely, accuracy and ROC-AUC are common metrics. For class-imbalanced problems, one may use Precision-Recall. For ranking problems or multi-label classification, one may use Mean Average Precision. And it isn't uncommon to have to define your own custom metric by which you will measure success. To get a sense of the diversity of machine learning success metrics and how they relate to different problem domains, it is helpful to browse data science competitions on [Kaggle.com](https://kaggle.com), as they showcase a wide range of different problems and evaluation metrics.

4.5.3 Decide on an evaluation protocol

Once you know what you are aiming for, you must establish how you will measure your current progress. We have previously reviewed three common evaluation protocols:

- Maintaining a hold-out validation set; this is the way to go when you have plenty of data.
- Doing K-fold cross-validation; this is the way to go when you have too few samples for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

hold-out validation to be reliable.

- Doing iterated K-fold validation; this is for performing highly accurate model evaluation when little is available.

Just pick one of these; in most cases the first one will work well enough.

4.5.4 Prepare your data

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

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

Once your tensors of input data and target data are ready, you can start training models.

4.5.5 Develop a model that does better than a baseline

Your goal at this stage is to achieve "statistical power", i.e. develop a small model that is capable of beating a dumb baseline. In our MNIST digits classification example, anything that gets an accuracy higher than 0.1 can be said to have statistical power; in our IMDB example it would be anything with an accuracy higher than 0.5.

Note that it is not always possible to achieve statistical power. If you cannot beat a random baseline after trying multiple reasonable architectures, it may be that the answer to the question you are asking isn't actually present in the input data. Remember that you are making two hypotheses:

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

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

Assuming that things go well—there are three keys choices you need to make in order to build your first working model:

- Choice of the last-layer activation. This establishes useful constraints on the network's output: for instance in our IMDB classification example we used `sigmoid` in the last layer, in the regression example we didn't use any last-layer activation, etc.
- Choice of loss function. It should match the type of problem you are trying to solve: for instance in our IMDB classification example we used `binary_crossentropy`, in the regression example we used `mse`, etc.
- Choice of optimization configuration: what optimizer will you use? What will its learning

rate be? In most cases it is safe to go with `rmsprop` and its default learning rate.

Regarding the choice of a loss function: note that it isn't always possible to directly optimize for the metric that measures success on a problem. Sometimes there is no easy way to turn a metric into a loss function; loss functions, after all, need to be computable given only a mini-batch of data (ideally, a loss function should be computable for as few as a single data point) and need to be differentiable (otherwise you cannot use backpropagation to train your network). For instance, the widely used classification metric ROC-AUC (Receiver Operating Characteristic Area Under the Curve) cannot be directly optimized. Hence in classification tasks it is common to optimize for a proxy metric of ROC-AUC, such as crossentropy. In general, one can hope that the lower the crossentropy gets, the higher the ROC-AUC will be.

Here is a table to help you pick a last-layer activation and a loss function for a few common problem types:

Problem type	Last-layer activation	Loss function
Binary classification	sigmoid	<code>binary_crossentropy</code>
Multi-class, single-label classification	softmax	<code>categorical_crossentropy</code>
Multi-class, multi-label classification	sigmoid	<code>binary_crossentropy</code>
Regression to arbitrary values	None	<code>mse</code>
Regression to values between 0 and 1	sigmoid	<code>mse</code> or <code>binary_crossentropy</code>

4.5.6 Scale up: develop a model that overfits

Once you have obtained a model that has statistical power, the question becomes: is your model powerful enough? Does it have enough layers and parameters to properly model the problem at hand? For instance, a network with a single hidden layer with 2 units would have statistical power on MNIST, but would not be sufficient to solve the problem well. Remember that the universal tension in machine learning is between optimization and generalization; the ideal model is one that stands right at the border between under-fitting and over-fitting; between under-capacity and over-capacity. To figure out where this border lies, first you must cross it.

To figure out how big a model you will need, you must develop a model that overfits. This is fairly easy:

- Add layers.
- Make your layers bigger.
- Train for more epochs.

Always monitor the training loss and validation loss, as well as the training and validation values for any metrics you care about. When you see that the performance of the model on the validation data starts degrading, you have achieved overfitting.

The next stage is to start regularizing and tuning your model, in order to get as close as possible to the ideal model, that is neither underfitting nor overfitting.

4.5.7 Regularize your model and tune your hyperparameters

This is the part that will take you the most time: you will repeatedly modify your model, train it, evaluate on your validation data (not your test data at this point), modify it again... until your model is as good as it can get.

These are some of things you should be trying:

- Add dropout.
- Try different architectures, add or remove layers.
- Add L1 / L2 regularization.
- Try different hyperparameters (such as the number of units per layer, the learning rate of the optimizer) to find the optimal configuration.
- Optionally iterate on feature engineering: add new features, remove features that do not seem to be informative.

Be mindful of the following: every time you are using feedback from your validation process in order to tune your model, you are leaking information about your validation process into your model. Repeated just a few times, this is innocuous, but done systematically over many iterations will eventually cause your model to overfit to the validation process (even though no model is directly trained on any of the validation data). This makes your evaluation process less reliable, so keep it in mind.

Once you have developed a seemingly good enough model configuration, you can train your final production model on all data available (training and validation) and evaluate it one last time on the test set. If it turns out that the performance on the test set is significantly worse than the performance measured on the validation data, this could mean either that your validation procedure wasn't that reliable after all, or alternatively it could mean that have started overfitting to the validation data while tuning the parameters of the model. In this case you may want to switch to a more reliable evaluation protocol (e.g. iterated K-fold validation).

In summary, this is the universal workflow of machine learning:

- 1) Define the problem at hand and the data you will be training on; collect this data or annotate it with labels if need be.
- 2) Choose how you will measure success on your problem. Which metrics will you be monitoring on your validation data?
- 3) Determine your evaluation protocol: hold-out validation? K-fold validation? Which portion of the data should you use for validation?
- 4) Develop a first model that does better than a basic baseline: a model that has "statistical power".
- 5) Develop a model that overfits.
- 6) Regularize your model and tune its hyperparameters, based on performance on the validation data.

A lot of machine learning research tends to focus only on the last step—but keep in mind the big picture.

5

Deep learning for computer vision

In this chapter, you will learn about convolutional neural networks (or "convnets"), a type of deep learning model almost universally used in computer vision applications. You will learn to apply them to image classification problems, in particular those involving small training datasets, the most common use case if you are not a large tech company.

We will start with an introduction to the theory behind convnets, specifically:

- What is convolution and max-pooling?
- What are convnets?
- What do convnets learn?

Then we will cover image classification with small datasets:

- Training your own small convnets from scratch.
- Using data augmentation to mitigate overfitting.
- Using a pre-trained convnet to do feature extraction.
- Fine-tuning a pre-trained convnet.

Finally, we will cover a few techniques for visualizing what convnets learn and how they make classification decisions.

5.1 Introduction to convnets

We are about to dive into the theory of what convnets are and why they have been so successful at computer vision tasks. But first, let's take a practical look at a very simple convnet example. We will use our convnet to classify MNIST digits, a task that you've already been through in Chapter 2, using a densely-connected network (our test accuracy then was 97.8%). Even though our convnet will be very basic, its accuracy will still blow out of the water that of the densely-connected model from Chapter 2.

The 6 lines of code below show you what a basic convnet looks like. It's a stack of Conv2D and MaxPooling2D layers. We'll see in a minute what they do concretely. Importantly, a convnet takes as input tensors of shape (`image_height, image_width, image_channels`) (not including the batch dimension). In our case, we will configure

our convnet to process inputs of size $(28, 28, 1)$, which is the format of MNIST images. We do this via passing the argument `input_shape=(28, 28, 1)` to our first layer.

Listing 5.1 Instantiating a small convnet

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Let's display the architecture of our convnet so far:

Listing 5.2 Displaying a summary of the model so far

```
>>> model.summary()

Layer (type)                 Output Shape              Param #
=====
conv2d_1 (Conv2D)            (None, 26, 26, 32)      320
maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32)      0
conv2d_2 (Conv2D)            (None, 11, 11, 64)      18496
maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64)        0
conv2d_3 (Conv2D)            (None, 3, 3, 64)        36928
=====
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

You can see above that the output of every `Conv2D` and `MaxPooling2D` layer is a 3D tensor of shape `(height, width, channels)`. The width and height dimensions tend to shrink as we go deeper in the network. The number of channels is controlled by the first argument passed to the `Conv2D` layers (e.g. 32 or 64).

The next step would be to feed our last output tensor (of shape $(3, 3, 64)$) into a densely-connected classifier network like those you are already familiar with: a stack of `Dense` layers. These classifiers process vectors, which are 1D, whereas our current output is a 3D tensor. So first, we will have to flatten our 3D outputs to 1D, and then add a few `Dense` layers on top:

Listing 5.3 Adding a classifier on top of the convnet

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

We are going to do 10-way classification, so we use a final layer with 10 outputs and a softmax activation. Now here's what our network looks like:

Listing 5.4 Displaying a summary of the full model

```
>>> model.summary()

Layer (type)                 Output Shape              Param #
=====                      =====
conv2d_1 (Conv2D)            (None, 26, 26, 32)      320
maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32)      0
conv2d_2 (Conv2D)            (None, 11, 11, 64)      18496
maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64)        0
conv2d_3 (Conv2D)            (None, 3, 3, 64)        36928
flatten_1 (Flatten)          (None, 576)             0
dense_1 (Dense)              (None, 64)              36928
dense_2 (Dense)              (None, 10)              650
=====
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

As you can see, our (3, 3, 64) outputs were flattened into vectors of shape (576,), before going through two Dense layers.

Now, let's train our convnet on the MNIST digits. We will reuse a lot of the code we have already covered in the MNIST example from Chapter 2.

Listing 5.5 Training our convnet on MNIST images

```
from keras.datasets import mnist
from keras.utils import to_categorical

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

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

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

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

Let's evaluate the model on the test data:

Listing 5.6 Evaluating the trained model

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> test_acc
```

0.9908000000000001

While our densely-connected network from Chapter 2 had a test accuracy of 97.8%, our basic convnet has a test accuracy of 99.3%: we decreased our error rate by 68% (relative). Not bad!

But why does this simple convnet work so well compared to a densely-connected model? To answer this, let's dive into what these `Conv2D` and `MaxPooling2D` layers actually do.

5.1.1 The convolution operation

The fundamental difference between a densely-connected layer and a convolution layer is this: dense layers learn global patterns in their input feature space (e.g. for a MNIST digit, patterns involving all pixels), while convolution layers learn local patterns (see Figure 5.1), i.e. in the case of images, patterns found in small 2D windows of the inputs. In our example above, these windows were all 3x3.

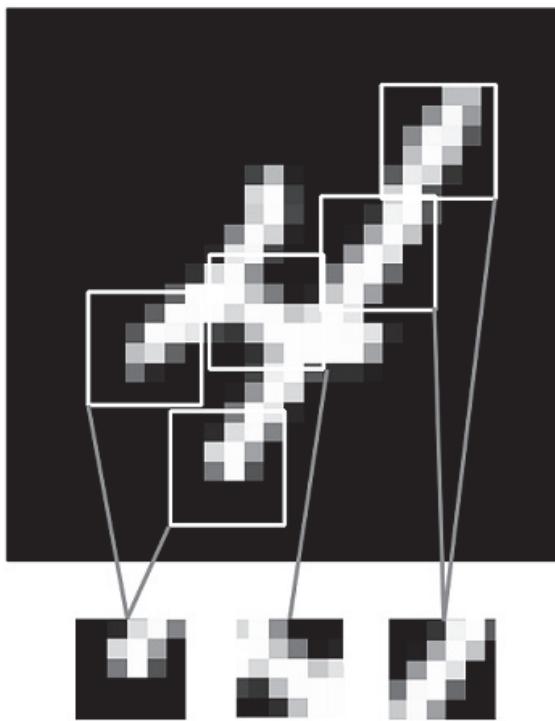


Figure 5.1 Images can be broken down into local patterns such as edges, textures, etc.

This key characteristic gives convnets two interesting properties:

- The patterns they learn are *translation-invariant*, i.e. after learning a certain pattern in the bottom right corner of a picture, a convnet is able to recognize it anywhere, e.g. in the top left corner. A densely-connected network would have to learn the pattern anew if it appeared at a new location. This makes convnets very data-efficient when processing images (since *the visual world is fundamentally translation-invariant*): they need less training samples to learn representations that have generalization power.
- They can learn *spatial hierarchies of patterns* (figure 5.2). A first convolution layer will learn small local patterns such as edges, but a second convolution layer will learn larger patterns made of the features of the first layers. And so on. This allows convnets to

efficiently learn increasingly complex and abstract visual concepts (since *the visual world is fundamentally spatially hierarchical*).

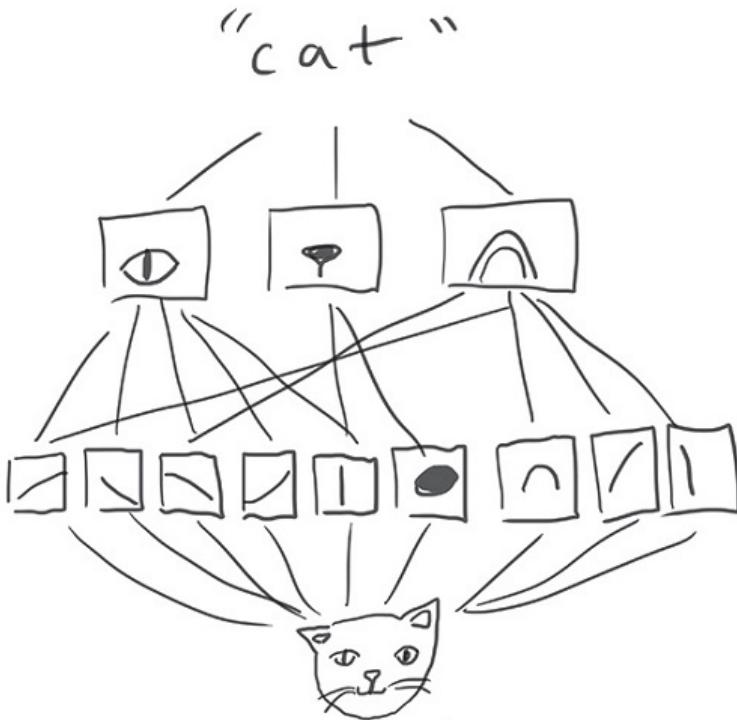


Figure 5.2 The visual world forms a spatial hierarchy of visual modules: hyperlocal edges combine into local objects such as eyes or ears, which combine into high-level concepts such as "cat"

Convolutions operate over 3D tensors, called "feature maps", with two spatial axes ("height" and "width") as well as a "depth" axis (also called the "channels" axis). For a RGB image, the dimension of the "depth" axis would be 3, since the image has 3 color channels, red, green, and blue. For a black and white picture, like our MNIST digits, the depth is just 1 (levels of gray). The convolution operation extracts patches from its input feature map, and applies a same transformation to all of these patches, producing an *output feature map*. This output feature map is still a 3D tensor: it still has a width and a height. Its depth can be arbitrary, since the output depth is a parameter of the layer, and the different channels in that depth axis no longer stand for specific colors like in an RGB input, rather they stand for what we call *filters*. Filters encode specific aspects of the input data: at a high level, a single filter could be encoding the concept "presence of a face in the input", for instance.

In our MNIST example, the very first convolution layer takes a feature map of size $(28, 28, 1)$ and outputs a feature map of size $(26, 26, 32)$, i.e. it computes 32 "filters" over its input. Each of these 32 output channels contains a 26×26 grid of values, which is a "response map" of the filter over the input, indicating the response of that filter pattern at different locations in the input (figure 5.3). That is what the term "feature map" really means: every dimension in depth axis is a feature (or filter), and the 2D tensor output $[:, :, n]$ is the 2D spatial "map" of the response of this filter over the input.

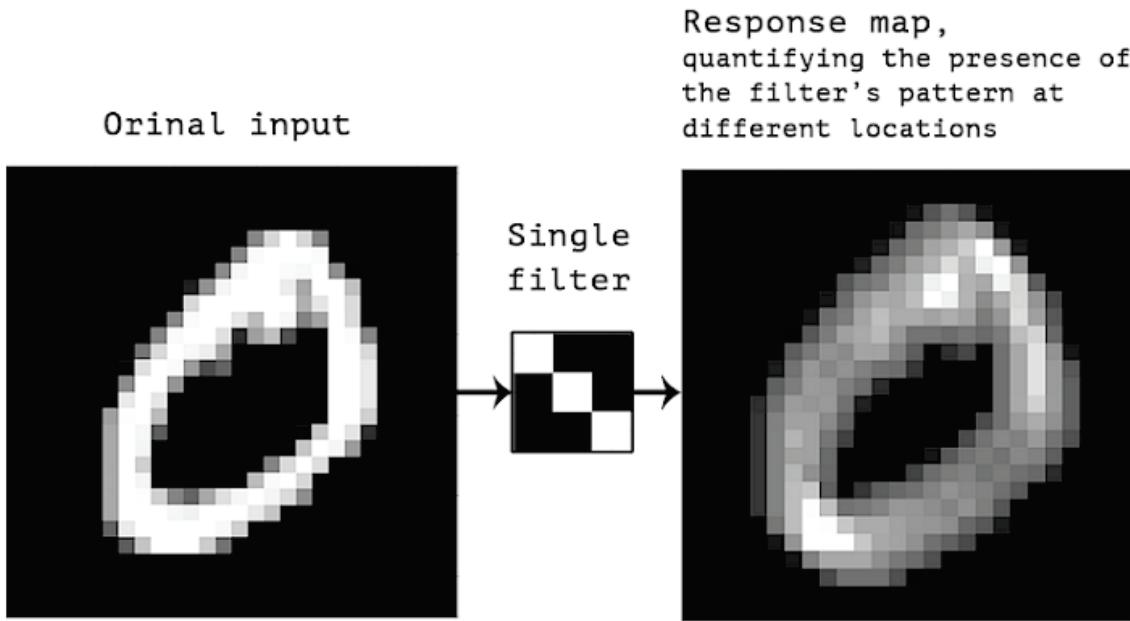


Figure 5.3 The concept of response map: a response map is a 2D map of presence of a pattern at different locations in an input

CNNs are defined by two key parameters:

- The size of the patches that are extracted from the inputs (typically 3x3 or 5x5). In our example it was always 3x3, which is a very common choice.
- The depth of the output feature map, i.e. the number of filters computed by the convolution. In our example, we started with a depth of 32 and ended with a depth of 64.

In Keras `Conv2D` layers, these parameters are the first arguments passed to the layer: `Conv2D(output_depth, (window_height, window_width))`.

A convolution works by "sliding" these windows of size 3x3 or 5x5 over the 3D input feature map, stopping at every possible location, and extracting the 3D patch of surrounding features (`(window_height, window_width, input_depth)`). Each such 3D patch is then transformed (via a tensor product with a same learned weight matrix, called "convolution kernel") into a 1D vector of shape `(output_depth,)`. All these vectors are then spatially reassembled into a 3D output map of shape `(height, width, output_depth)`. Every spatial location in the output feature maps corresponds to the same location in the input feature map (e.g. the bottom right corner of the output contains information about the bottom right corner of the input). For instance, with 3x3 windows, the vector `output[i, j, :]` comes from the 3D patch `input[i-1:i+1, j-1:j+1, :]`. The full process is detailed in figure 5.4.

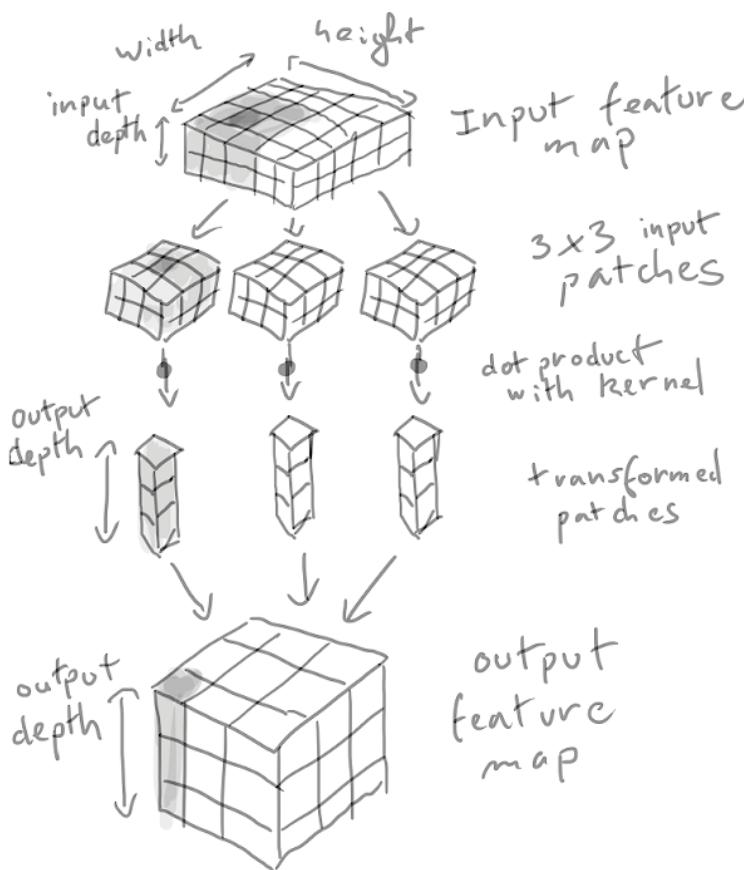


Figure 5.4 How convolution works

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

- Border effects, which can be countered by padding the input feature map.
- The use of "strides", which we will define in a second.

Let's take a deeper look at these notions.

UNDERSTANDING BORDER EFFECTS AND PADDING

Consider a 5x5 feature map (25 tiles in total). There are only 9 different tiles around which you can center a 3x3 window (see figure 5.5 below), forming a 3x3 grid. Hence the output feature map will be 3x3: it gets shrunk a little bit, by exactly two tiles alongside each dimension in this case. You can see this "border effect" in action in our example above: we start with 28x28 inputs, which become 26x26 after the first convolution layer.

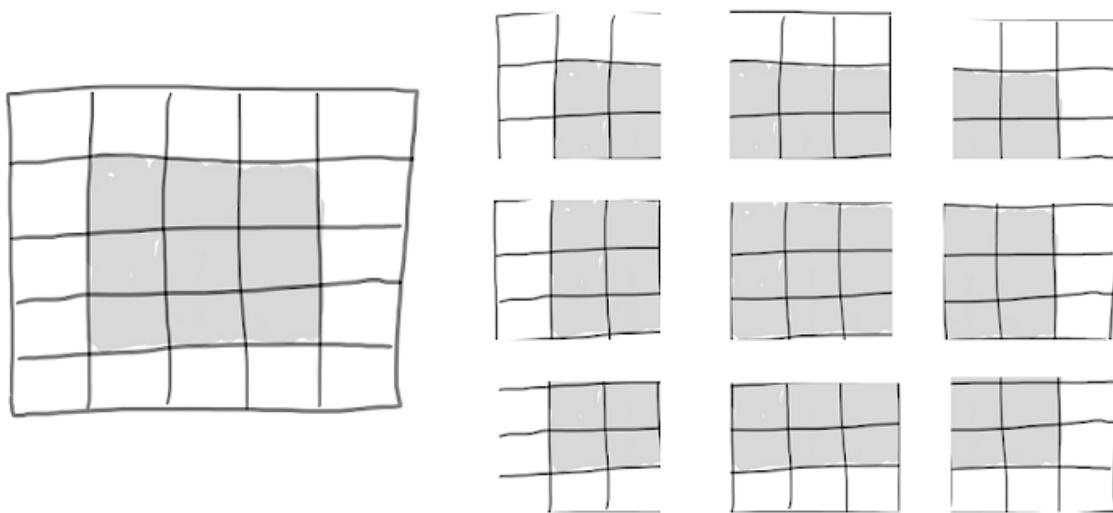


Figure 5.5 Valid locations of 3x3 patches in a 5x5 input feature map

If you want to get an output feature map with the same spatial dimensions as the input, you can use *padding*. Padding consists in adding an appropriate number of rows and columns on each side of the input feature map so as to make it possible to fit center convolution windows around every input tile. For a 3x3 window, one would add one column on the right, one column on the left, one row at the top, one row at the bottom. For a 5x5 window, it would be two rows (see figure 5.6).

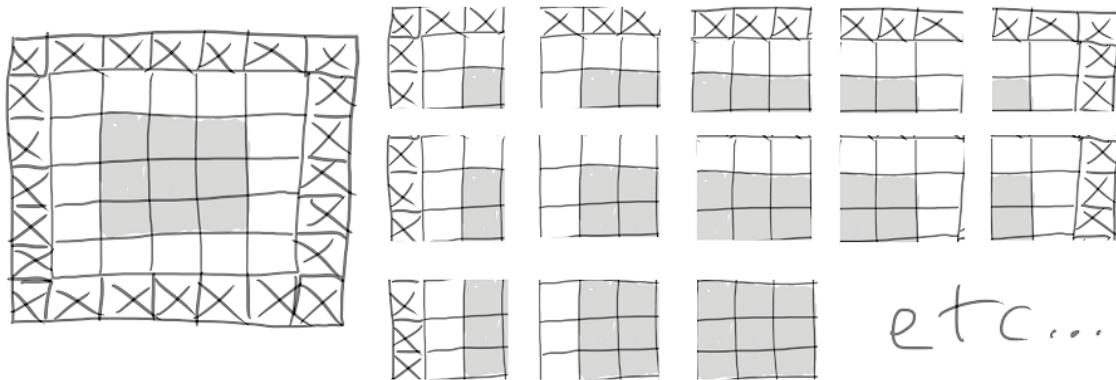


Figure 5.6 Padding a 5x5 input in order to be able to extract 25 3x3 patches

In `Conv2D` layers, padding is configurable via the `padding` argument, which takes two values: "valid", which means no padding (only "valid" window locations will be used), and "same", which means "pad in such a way as to have an output with the same width and height as the input". The `padding` argument defaults to "valid".

UNDERSTANDING CONVOLUTION STRIDES

The other factor that can influence output size is the notion of "stride". In our description of convolution so far, we have been assuming that the center tile of the convolution windows were all contiguous. However, the distance between two successive windows is actually a parameter of the convolution, called its "stride", which defaults to one. It is possible to have "strided convolutions", i.e. convolutions with a non-unit stride. In figure 5.7 you can see the patches extracted by a convolution with stride 2 over a 5x5 input (without padding):

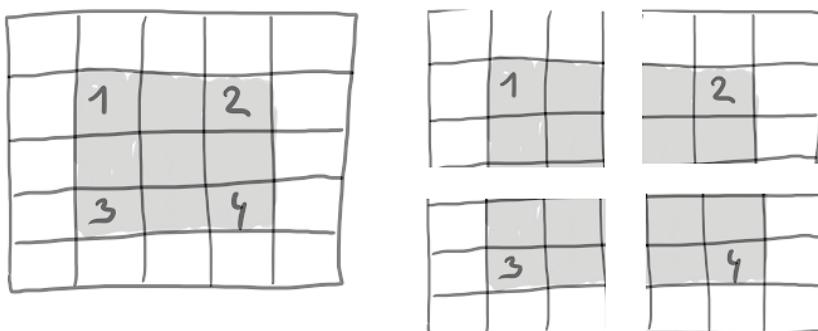


Figure 5.7 3x3 convolution patches with 2x2 strides

Using stride 2 means that the width and height of the feature map get downsampled by a factor 2 (besides any changes induced by border effects). Strided convolutions are rarely used in practice, although they can come in handy for some types of models, and it is generally good to be familiar with the concept.

To downsample feature maps, instead of strides, we tend to use the "max pooling" operation, which you saw in action in our first convnet example. Let's take a look at that one.

5.1.2 The max pooling operation

In our convnet example, you may have noticed that the size of the feature maps gets halved after every `MaxPooling2D` layer. For instance, before the first `MaxPooling2D` layers, the feature map is 26x26, but the max pooling operation halves it to 13x13. That's the role of max pooling: to aggressively downsample feature maps, much like strided convolutions.

Max pooling consists in extracting windows from the input feature maps and outputting the max value of each channel. It's conceptually similar to convolution, except that instead of transforming local patches via a learned linear transformation (the convolution kernel), they are transformed via a hard-coded `max` tensor operation. A big difference with convolution, though, is the fact max pooling is usually done with 2x2 windows and stride 2, so as to downsample the feature maps by a factor 2. On the other hand, convolution is most typically done with 3x3 windows and no stride (stride 1).

Why do we downsample feature maps in such a way? Why not remove the max pooling layers and keep fairly large feature maps all the way up? Let's take a look at this

option. The convolutional base of our model would then look like this:

Listing 5.7 A convnet without pooling layers

```
model_no_max_pool = models.Sequential()
model_no_max_pool.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Listing 5.8 Displaying a summary of the model

```
>>> model_no_max_pool.summary()

Layer (type)                 Output Shape              Param #
=====                      =====
conv2d_4 (Conv2D)           (None, 26, 26, 32)      320
=====
conv2d_5 (Conv2D)           (None, 24, 24, 64)     18496
=====
conv2d_6 (Conv2D)           (None, 22, 22, 64)     36928
=====
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

What's wrong with this setup? Two things.

- It isn't conducive to learning a spatial hierarchy of features. The 3x3 windows in the 3rd layers would only contain information coming from 7x7 windows in the initial input. The high-level patterns learned by our convnet would still be very small with regard to the initial input, which may not be enough to learn to classify digits (try recognizing a digit by only looking at it through windows of 7x7 pixels!). We need the features from the last convolution layer to contain information about the totality of the input.
- The final feature map has $22 \times 22 \times 64 = 31,000$ total coefficients per sample. This is huge. If we were to flatten it to stick a Dense layer of size 512 on top, that layer would have 15.8 million parameters. This is way too large for such a small model, and would result in intense overfitting.

In short, the reason to use downsampling is simply to reduce the number of feature map coefficients to process, as well as to induce spatial filter hierarchies by making successive convolution layers look at increasingly large windows (in terms of the fraction of the original input they cover).

Note that max pooling is not the only way one can achieve such downsampling. As you already know, you could also use strides in the previous convolution layer. And you could also use average pooling instead of max pooling, where each local input patch is transformed by taking the average value of each channel over the patch, rather than the max. However, max pooling tends to work better than these alternative solutions. In a nutshell, the reason for this is that features tend to encode the spatial "presence" of some pattern or concept over the different tiles of the feature map (hence the term "feature map"), and it is more informative to look at the *maximal presence* of different features than at their *average presence*. So the most reasonable subsampling strategy is to first produce dense maps of features (via unstrided convolutions) and then look at the

maximal activation of the features over small patches, rather than looking at sparser windows of the inputs (via strided convolutions) or averaging input patches, which could cause you to miss feature presence information or dilute it.

At this point, you understand the basics of convnets—feature maps, convolution, max pooling—and you know how to build a small convnet to solve a toy problem such as MNIST digits classification. Now let's move on to more useful practical applications.

5.2 Training a convnet from scratch on a small dataset

Having to train an image classification model using only very little data is a common situation, which you likely encounter yourself in practice if you ever do computer vision in a professional context.

Having "few" samples can mean anywhere from a few hundreds to a few tens of thousands of images. As a practical example, we will focus on classifying images as "dogs" or "cats", in a dataset containing 4000 pictures of cats and dogs (2000 cats, 2000 dogs). We will use 2000 pictures for training, 1000 for validation, and finally 1000 for testing.

In this section, we will review one basic strategy to tackle this problem: training a new model from scratch on what little data we have. We will start by naively training a small convnet on our 2000 training samples, without any regularization, to set a baseline for what can be achieved. This will get us to a classification accuracy of 71%. At that point, our main issue will be overfitting. Then we will introduce *data augmentation*, a powerful technique for mitigating overfitting in computer vision. By leveraging data augmentation, we will improve our network to reach an accuracy of 82%.

In the next section, we will review two more essential techniques for applying deep learning to small datasets: *doing feature extraction with a pre-trained network* (this will get us to an accuracy of 90% to 96%), and *fine-tuning a pre-trained network* (this will get us to our final accuracy of 97%). Together, these three strategies—training a small model from scratch, doing feature extracting using a pre-trained model, and fine-tuning a pre-trained model—will constitute your future toolbox for tackling the problem of doing computer vision with small datasets.

5.2.1 The relevance of deep learning for small-data problems

You will sometimes hear that deep learning only works when lots of data is available. This is in part a valid point: one fundamental characteristic of deep learning is that it is able to find interesting features in the training data on its own, without any need for manual feature engineering, and this can only be achieved when lots of training examples are available. This is especially true for problems where the input samples are very high-dimensional, like images.

However, what constitutes "lots" of samples is relative—relative to the size and depth of the network you are trying to train, for starters. It isn't possible to train a convnet to solve a complex problem with just a few tens of samples, but a few hundreds can potentially suffice if the model is small and well-regularized and if the task is simple.

Because convnets learn local, translation-invariant features, they are very data-efficient on perceptual problems. Training a convnet from scratch on a very small image dataset will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering. You will see this in action in this section.

But what's more, deep learning models are by nature highly repurposable: you can take, say, an image classification or speech-to-text model trained on a large-scale dataset then reuse it on a significantly different problem with only minor changes. Specifically, in the case of computer vision, many pre-trained models (usually trained on the ImageNet dataset) are now publicly available for download and can be used to bootstrap powerful vision models out of very little data. That's what we will do in the next section.

For now, let's get started by getting our hands on the data.

5.2.2 Downloading the data

The cats vs. dogs dataset that we will use isn't packaged with Keras. It was made available by Kaggle.com as part of a computer vision competition in late 2013, back when convnets weren't quite mainstream. You can download the original dataset at: www.kaggle.com/c/dogs-vs-cats/data (you will need to create a Kaggle account if you don't already have one—don't worry, the process is painless).

The pictures are medium-resolution color JPEGs. They look like this:



Figure 5.8 Samples from the cats vs. dogs dataset. Sizes were not modified: the samples are heterogeneous in size, appearance, etc.

Unsurprisingly, the cats vs. dogs Kaggle competition in 2013 was won by entrants who used convnets. The best entries could achieve up to 95% accuracy. In our own example, we will get fairly close to this accuracy (in the next section), even though we will be training our models on less than 10% of the data that was available to the competitors. This original dataset contains 25,000 images of dogs and cats (12,500 from each class) and is 543MB large (compressed). After downloading and uncompressing it,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

we will create a new dataset containing three subsets: a training set with 1000 samples of each class, a validation set with 500 samples of each class, and finally a test set with 500 samples of each class.

Here are a few lines of code to do this:

Listing 5.9 Copying images to train, validation and test directories

```
import os, shutil

# The path to the directory where the original
# dataset was uncompressed
original_dataset_dir = '/Users/fchollet/Downloads/kaggle_original_data'

# The directory where we will
# store our smaller dataset
base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'
os.mkdir(base_dir)

# Directories for our training,
# validation and test splits
train_dir = os.path.join(base_dir, 'train')
os.mkdir(train_dir)
validation_dir = os.path.join(base_dir, 'validation')
os.mkdir(validation_dir)
test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)

# Directory with our training cat pictures
train_cats_dir = os.path.join(train_dir, 'cats')
os.mkdir(train_cats_dir)

# Directory with our training dog pictures
train_dogs_dir = os.path.join(train_dir, 'dogs')
os.mkdir(train_dogs_dir)

# Directory with our validation cat pictures
validation_cats_dir = os.path.join(validation_dir, 'cats')
os.mkdir(validation_cats_dir)

# Directory with our validation dog pictures
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
os.mkdir(validation_dogs_dir)

# Directory with our validation cat pictures
test_cats_dir = os.path.join(test_dir, 'cats')
os.mkdir(test_cats_dir)

# Directory with our validation dog pictures
test_dogs_dir = os.path.join(test_dir, 'dogs')
os.mkdir(test_dogs_dir)

# Copy first 1000 cat images to train_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 cat images to validation_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 cat images to test_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_cats_dir, fname)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

```

shutil.copyfile(src, dst)

# Copy first 1000 dog images to train_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_dogs_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 dog images to validation_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_dogs_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 dog images to test_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_dogs_dir, fname)
    shutil.copyfile(src, dst)

```

As a sanity check, let's count how many pictures we have in each training split (train/validation/test):

Listing 5.10 Counting our images

```

>>> print('total training cat images:', len(os.listdir(train_cats_dir)))
total training cat images: 1000
>>> print('total training dog images:', len(os.listdir(train_dogs_dir)))
total training dog images: 1000
>>> print('total validation cat images:', len(os.listdir(validation_cats_dir)))
total validation cat images: 500
>>> print('total validation dog images:', len(os.listdir(validation_dogs_dir)))
total validation dog images: 500
>>> print('total test cat images:', len(os.listdir(test_cats_dir)))
total test cat images: 500
>>> print('total test dog images:', len(os.listdir(test_dogs_dir)))
total test dog images: 500

```

So we have indeed 2000 training images, and then 1000 validation images and 1000 test images. In each split, there is the same number of samples from each class: this is a balanced binary classification problem, which means that classification accuracy will be an appropriate measure of success.

5.2.3 Building our network

We've already built a small convnet for MNIST in the previous example, so you should be familiar with them. We will reuse the same general structure: our convnet will be a stack of alternated Conv2D (with `relu` activation) and MaxPooling2D layers.

However, since we are dealing with bigger images and a more complex problem, we will make our network accordingly larger: it will have one more Conv2D + MaxPooling2D stage. This serves both to augment the capacity of the network, and to further reduce the size of the feature maps, so that they aren't overly large when we reach the Flatten layer. Here, since we start from inputs of size 150x150 (a somewhat arbitrary choice), we end up with feature maps of size 7x7 right before the Flatten layer.

Note that the depth of the feature maps is progressively increasing in the network (from 32 to 128), while the size of the feature maps is decreasing (from 148x148 to 7x7). This is a pattern that you will see in almost all convnets.

Since we are attacking a binary classification problem, we are ending the network with a single unit (a `Dense` layer of size 1) and a `sigmoid` activation. This unit will encode the probability that the network is looking at one class or the other.

Listing 5.11 Instantiating a small convnet for cats vs. dogs classification

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

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

Listing 5.12 Displaying a summary of the model

```
>>> model.summary()

Layer (type)                  Output Shape                 Param #
=================================================================
conv2d_1 (Conv2D)              (None, 148, 148, 32)      896
maxpooling2d_1 (MaxPooling2D)  (None, 74, 74, 32)        0
conv2d_2 (Conv2D)              (None, 72, 72, 64)       18496
maxpooling2d_2 (MaxPooling2D)  (None, 36, 36, 64)       0
conv2d_3 (Conv2D)              (None, 34, 34, 128)      73856
maxpooling2d_3 (MaxPooling2D)  (None, 17, 17, 128)      0
conv2d_4 (Conv2D)              (None, 15, 15, 128)      147584
maxpooling2d_4 (MaxPooling2D)  (None, 7, 7, 128)        0
flatten_1 (Flatten)            (None, 6272)             0
dense_1 (Dense)               (None, 512)              3211776
dense_2 (Dense)               (None, 1)                513
=====
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

For our compilation step, we'll go with the `RMSprop` optimizer as usual. Since we

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

ended our network with a single sigmoid unit, we will use binary crossentropy as our loss (as a reminder, check out the table in Chapter 4, section 5 for a cheatsheet on what loss function to use in various situations).

Listing 5.13 Configuring our model for training

```
from keras import optimizers  
  
model.compile(loss='binary_crossentropy',  
               optimizer=optimizers.RMSprop(lr=1e-4),  
               metrics=['acc'])
```

5.2.4 Data preprocessing

As you already know by now, data should be formatted into appropriately pre-processed floating point tensors before being fed into our network. Currently, our data sits on a drive as JPEG files, so the steps for getting it into our network are roughly:

- Read the picture files.
- Decode the JPEG content to RBG grids of pixels.
- Convert these into floating point tensors.
- Rescale the pixel values (between 0 and 255) to the [0, 1] interval (as you know, neural networks prefer to deal with small input values).

It may seem a bit daunting, but thankfully Keras has utilities to take care of these steps automatically. Keras has a module with image processing helper tools, located at `keras.preprocessing.image`. In particular, it contains the class `ImageDataGenerator` which allows to quickly set up Python generators that can automatically turn image files on disk into batches of pre-processed tensors. This is what we will use here.

NOTE**Understanding Python generators**

A Python generator is an object that acts as an iterator, i.e. an object you can use with the `for/in` operator. Generators are built using the `yield` operator.

Here is an example of a generator that yields integers:

```
def generator():
    i = 0
    while True:
        i += 1
        yield i

for item in generator():
    print(item)
    if item > 4:
        break
```

It prints:

```
1
2
3
4
5
```

Listing 5.14 Using ImageDataGenerator to read images from directories

```
from keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    # This is the target directory
    train_dir,
    # All images will be resized to 150x150
    target_size=(150, 150),
    batch_size=20,
    # Since we use binary_crossentropy loss, we need binary labels
    class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary')
```

Let's take a look at the output of one of these generators: it yields batches of 150x150 RGB images (shape `(20, 150, 150, 3)`) and binary labels (shape `(20,)`). 20 is the number of samples in each batch (the batch size). Note that the generator yields these batches indefinitely: it just loops endlessly over the images present in the target folder. For this reason, we need to `break` the iteration loop at some point.

Listing 5.15 Displaying the shapes of a batch of data and labels

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

```
>>> for data_batch, labels_batch in train_generator:
>>>     print('data batch shape:', data_batch.shape)
>>>     print('labels batch shape:', labels_batch.shape)
>>>     break
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

Let's fit our model to the data using the generator. We do it using the `fit_generator` method, the equivalent of `fit` for data generators like ours. It expects as first argument a Python generator that will yield batches of inputs and targets indefinitely, like ours does. Because the data is being generated endlessly, the generator needs to know example how many samples to draw from the generator before declaring an epoch over. This is the role of the `steps_per_epoch` argument: after having drawn `steps_per_epoch` batches from the generator, i.e. after having run for `steps_per_epoch` gradient descent steps, the fitting process will go to the next epoch. In our case, batches are 20-sample large, so it will take 100 batches until we see our target of 2000 samples.

When using `fit_generator`, one may pass a `validation_data` argument, much like with the `fit` method. Importantly, this argument is allowed to be a data generator itself, but it could be a tuple of Numpy arrays as well. If you pass a generator as `validation_data`, then this generator is expected to yield batches of validation data endlessly, and thus you should also specify the `validation_steps` argument, which tells the process how many batches to draw from the validation generator for evaluation.

Listing 5.16 Fitting our model using a batch generator

```
history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=30,
    validation_data=validation_generator,
    validation_steps=50)
```

It is good practice to always save your models after training:

Listing 5.17 Saving our model

```
model.save('cats_and_dogs_small_1.h5')
```

Let's plot the loss and accuracy of the model over the training and validation data during training:

Listing 5.18 Displaying curves of loss and accuracy during training

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
```

```

val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()

```

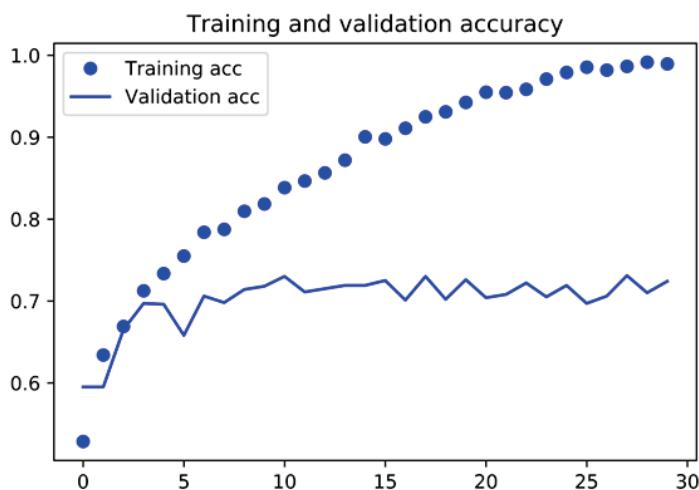


Figure 5.9 Training and validation accuracy (training values as dots, validation values as solid lines)

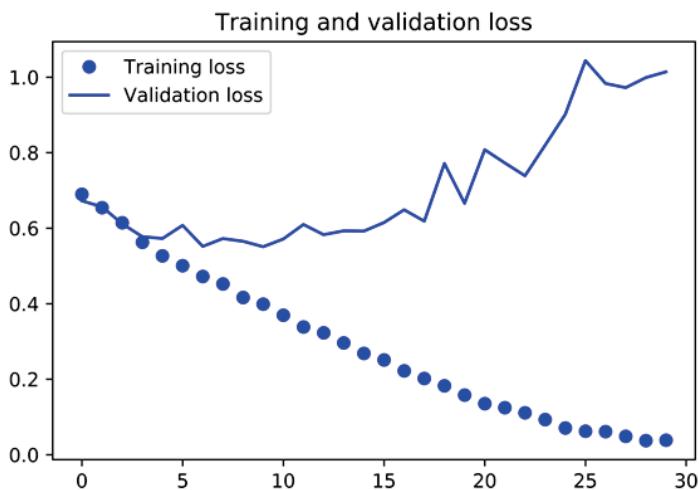


Figure 5.10 Training and validation loss (training values as dots, validation values as solid lines)

These plots are characteristic of overfitting. Our training accuracy increases linearly over time, until it reaches nearly 100%, while our validation accuracy stalls at 70-72%. Our validation loss reaches its minimum after only five epochs then stalls, while the

training loss keeps decreasing linearly until it reaches nearly 0.

Because we only have relatively few training samples (2000), overfitting is going to be our number one concern. You already know about a number of techniques that can help mitigate overfitting, such as dropout and weight decay (L2 regularization). We are now going to introduce a new one, specific to computer vision, and used almost universally when processing images with deep learning models: *data augmentation*.

5.2.5 Using data augmentation

Overfitting is caused by having too few samples to learn from, rendering us unable to train a model able to generalize to new data. Given infinite data, our model would be exposed to every possible aspect of the data distribution at hand: we would never overfit. Data augmentation takes the approach of generating more training data from existing training samples, by "augmenting" the samples via a number of random transformations that yield believable-looking images. The goal is that at training time, our model would never see the exact same picture twice. This helps the model get exposed to more aspects of the data and generalize better.

In Keras, this can be done by configuring a number of random transformations to be performed on the images read by our `ImageDataGenerator` instance. Let's get started with an example:

Listing 5.19 Setting up a data augmentation configuration via `ImageDataGenerator`

```
datagen = ImageDataGenerator(  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest')
```

These are just a few of the options available (for more, see the Keras documentation). Let's quickly go over what we just wrote:

- `rotation_range` is a value in degrees (0-180), a range within which to randomly rotate pictures.
- `width_shift` and `height_shift` are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
- `shear_range` is for randomly applying shearing transformations.
- `zoom_range` is for randomly zooming inside pictures.
- `horizontal_flip` is for randomly flipping half of the images horizontally—relevant when there are no assumptions of horizontal asymmetry (e.g. real-world pictures).
- `fill_mode` is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

Let's take a look at our augmented images:

Listing 5.20 Displaying some randomly augmented training images

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

```
# This is module with image preprocessing utilities
from keras.preprocessing import image

fnames = [os.path.join(train_cats_dir, fname) for fname in os.listdir(train_cats_dir)]

# We pick one image to "augment"
img_path = fnames[3]

# Read the image and resize it
img = image.load_img(img_path, target_size=(150, 150))

# Convert it to a Numpy array with shape (150, 150, 3)
x = image.img_to_array(img)

# Reshape it to (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)

# The .flow() command below generates batches of randomly transformed images.
# It will loop indefinitely, so we need to `break` the loop at some point!
i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```



Figure 5.11 Generation of cat pictures via random data augmentation

If we train a new network using this data augmentation configuration, our network will never see twice the same input. However, the inputs that it sees are still heavily intercorrelated, since they come from a small number of original images—we cannot

produce new information, we can only remix existing information. As such, this might not be quite enough to completely get rid of overfitting. To further fight overfitting, we will also add a Dropout layer to our model, right before the densely-connected classifier:

Listing 5.21 Defining a new convnet that includes dropout

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

Let's train our network using data augmentation and dropout:

Listing 5.22 Training our convnet using data augmentation generators

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,)

# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    # This is the target directory
    train_dir,
    # All images will be resized to 150x150
    target_size=(150, 150),
    batch_size=32,
    # Since we use binary_crossentropy loss, we need binary labels
    class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary')

history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=100,
    validation_data=validation_generator,
    validation_steps=50)
```

Let's save our model—we will be using it in the section on convnet visualization.

Listing 5.23 Saving our model

```
model.save('cats_and_dogs_small_2.h5')
```

Let's plot our results again:

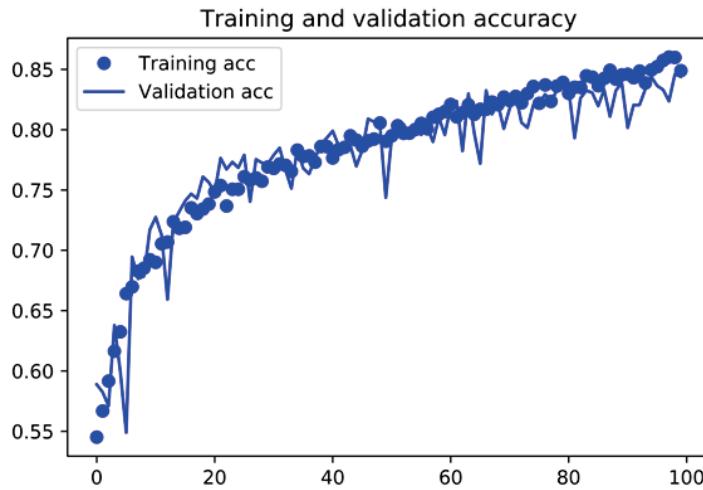


Figure 5.12 Training and validation accuracy (training values as dots, validation values as solid lines)

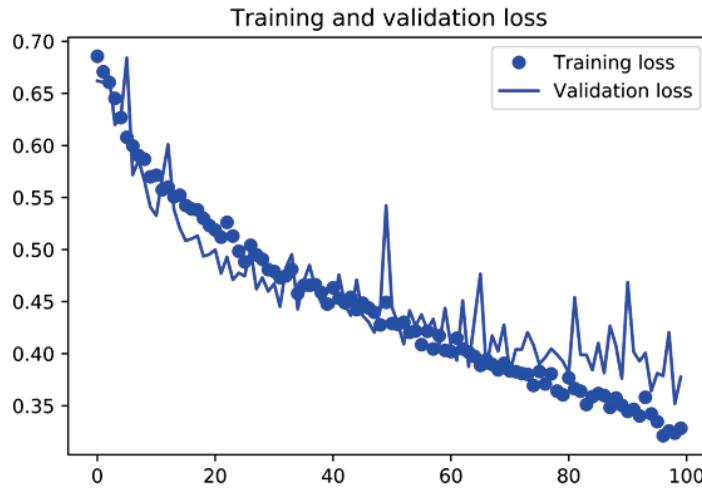


Figure 5.13 Training and validation loss (training values as dots, validation values as solid lines)

Thanks to data augmentation and dropout, we are no longer overfitting: the training curves are rather closely tracking the validation curves. We are now able to reach an accuracy of 82%, a 15% relative improvement over the non-regularized model.

By leveraging regularization techniques even further and by tuning the network's parameters (such as the number of filters per convolution layer, or the number of layers in the network), we may be able to get an even better accuracy, likely up to 86-87%. However, it would prove very difficult to go any higher just by training our own convnet

from scratch, simply because we have so little data to work with. As a next step to improve our accuracy on this problem, we will have to leverage a pre-trained model, which will be the focus of the next two sections.

5.3 Using a pre-trained convnet

A common and highly effective approach to deep learning on small image datasets is to leverage a pre-trained network. A pre-trained network is simply a saved network previously trained on a large dataset, typically on a large-scale image classification task. If this original dataset is large enough and general enough, then the spatial feature hierarchy learned by the pre-trained network can effectively act as a generic model of our visual world, and hence its features can prove useful for many different computer vision problems, even though these new problems might involve completely different classes from those of the original task. For instance, one might train a network on ImageNet (where classes are mostly animals and everyday objects) and then re-purpose this trained network for something as remote as identifying furniture items in images. Such portability of learned features across different problems is a key advantage of deep learning compared to many older shallow learning approaches, and it makes deep learning very effective for small-data problems.

In our case, we will consider a large convnet trained on the ImageNet dataset (1.4 million labeled images and 1000 different classes). ImageNet contains many animal classes, including different species of cats and dogs, and we can thus expect to perform very well on our cat vs. dog classification problem.

We will use the VGG16 architecture, developed by Karen Simonyan and Andrew Zisserman in 2014, a simple and widely used convnet architecture for ImageNet. Although it is a bit of an older model, far from the current state of the art and somewhat heavier than many other recent models, we chose it because its architecture is similar to what you are already familiar with, and easy to understand without introducing any new concepts. This may be your first encounter with one of these cutesie model names—VGG, ResNet, Inception, Inception-ResNet, Xception... you will get used to them, as they will come up frequently if you keep doing deep learning for computer vision.

There are two ways to leverage a pre-trained network: *feature extraction* and *fine-tuning*. We will cover both of them. Let's start with feature extraction.

5.3.1 Feature extraction

Feature extraction consists of using the representations learned by a previous network to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.

As we saw previously, convnets used for image classification comprise two parts: they start with a series of pooling and convolution layers, and they end with a densely-connected classifier. The first part is called the "convolutional base" of the model. In the case of convnets, "feature extraction" will simply consist of taking the

convolutional base of a previously-trained network, running the new data through it, and training a new classifier on top of the output.

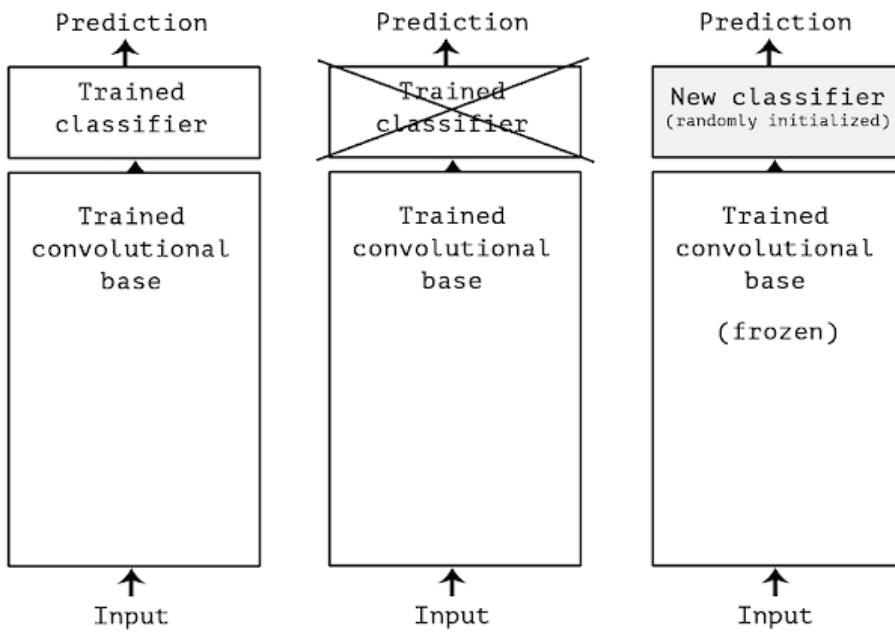


Figure 5.14 Swapping classifiers while keeping the same convolutional base

Why only reuse the convolutional base? Could we reuse the densely-connected classifier as well? In general, it should be avoided. The reason is simply that the representations learned by the convolutional base are likely to be more generic and therefore more reusable: the feature maps of a convnet are presence maps of generic concepts over a picture, which is likely to be useful regardless of the computer vision problem at hand. On the other end, the representations learned by the classifier will necessarily be very specific to the set of classes that the model was trained on—they will only contain information about the presence probability of this or that class in the entire picture. Additionally, representations found in densely-connected layers no longer contain any information about *where* objects are located in the input image: these layers get rid of the notion of space, whereas the object location is still described by convolutional feature maps. For problems where object location matters, densely-connected features would be largely useless.

Note that the level of generality (and therefore reusability) of the representations extracted by specific convolution layers depends on the depth of the layer in the model. Layers that come earlier in the model extract local, highly generic feature maps (such as visual edges, colors, and textures), while layers higher-up extract more abstract concepts (such as "cat ear" or "dog eye"). So if your new dataset differs a lot from the dataset that the original model was trained on, you may be better off using only the first few layers of the model to do feature extraction, rather than using the entire convolutional base.

In our case, since the ImageNet class set did contain multiple dog and cat classes, it is likely that it would be beneficial to reuse the information contained in the densely-connected layers of the original model. However, we will choose not to, in order

to cover the more general case where the class set of the new problem does not overlap with the class set of the original model.

Let's put this in practice by using the convolutional base of the VGG16 network, trained on ImageNet, to extract interesting features from our cat and dog images, and then training a cat vs. dog classifier on top of these features.

The VGG16 model, among others, comes pre-packaged with Keras. You can import it from the `keras.applications` module. Here's the list of image classification models (all pre-trained on the ImageNet dataset) that are available as part of `keras.applications`:

- Xception
- InceptionV3
- ResNet50
- VGG16
- VGG19
- MobileNet

Let's instantiate the VGG16 model:

Listing 5.24 Instantiating the VGG16 convolutional base

```
from keras.applications import VGG16

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(150, 150, 3))
```

We passed three arguments to the constructor:

- `weights`, to specify which weight checkpoint to initialize the model from
- `include_top`, which refers to including or not the densely-connected classifier on top of the network. By default, this densely-connected classifier would correspond to the 1000 classes from ImageNet. Since we intend to use our own densely-connected classifier (with only two classes, cat and dog), we don't need to include it.
- `input_shape`, the shape of the image tensors that we will feed to the network. This argument is purely optional: if we don't pass it, then the network will be able to process inputs of any size.

Here's the detail of the architecture of the VGG16 convolutional base: it's very similar to the simple convnets that you are already familiar with.

Listing 5.25 Displaying a summary of the convolutional base

```
>>> conv_base.summary()

Layer (type)                 Output Shape              Param #
================================================================
input_1 (InputLayer)          (None, 150, 150, 3)    0
block1_conv1 (Convolution2D)  (None, 150, 150, 64)   1792
block1_conv2 (Convolution2D)  (None, 150, 150, 64)   36928
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Convolution2D)	(None, 75, 75, 128)	73856
block2_conv2 (Convolution2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Convolution2D)	(None, 37, 37, 256)	295168
block3_conv2 (Convolution2D)	(None, 37, 37, 256)	590080
block3_conv3 (Convolution2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Convolution2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Convolution2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Convolution2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Convolution2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Convolution2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Convolution2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0
=====		
Total params:	14,714,688	
Trainable params:	14,714,688	
Non-trainable params:	0	

The final feature map has shape `(4, 4, 512)`. That's the feature on top of which we will stick a densely-connected classifier.

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

- Running the convolutional base over our dataset, recording its output to a Numpy array on disk, then using this data as input to a standalone densely-connected classifier similar to those you have seen in the first chapters of this book. This solution is very fast and cheap to run, because it only requires running the convolutional base once for every input image, and the convolutional base is by far the most expensive part of the pipeline. However, for the exact same reason, this technique would not allow us to leverage data augmentation at all.
- Extending the model we have (`conv_base`) by adding `Dense` layers on top, and running the whole thing end-to-end on the input data. This allows us to use data augmentation, because every input image is going through the convolutional base every time it is seen by the model. However, for this same reason, this technique is far more expensive than the first one.

We will cover both techniques. Let's walk through the code required to set-up the first one: recording the output of `conv_base` on our data and using these outputs as inputs to a new model.

We will start by simply running instances of the previously-introduced `ImageDataGenerator` to extract images as Numpy arrays as well as their labels. We will extract features from these images simply by calling the `predict` method of the

conv_base model.

Listing 5.26 Extracting features using the pre-trained convolutional base

```

import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator

base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
        features[i * batch_size : (i + 1) * batch_size] = inputs_batch
        labels[i * batch_size : (i + 1) * batch_size] = labels_batch
        i += 1
    if i * batch_size >= sample_count:
        # Note that since generators yield data indefinitely in a loop,
        # we must `break` after every image has been seen once.
        break
    return features, labels

train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)

```

The extracted features are currently of shape (samples, 4, 4, 512). We will feed them to a densely-connected classifier, so first we must flatten them to (samples, 8192):

```

train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
test_features = np.reshape(test_features, (1000, 4 * 4 * 512))

```

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

```

from keras import models
from keras import layers
from keras import optimizers

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer=optimizers.RMSprop(lr=2e-5),
              loss='binary_crossentropy',
              metrics=['acc'])

```

```
history = model.fit(train_features, train_labels,
                     epochs=30,
                     batch_size=20,
                     validation_data=(validation_features, validation_labels))
```

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

Let's take a look at the loss and accuracy curves during training:

Listing 5.27 Plotting our results

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

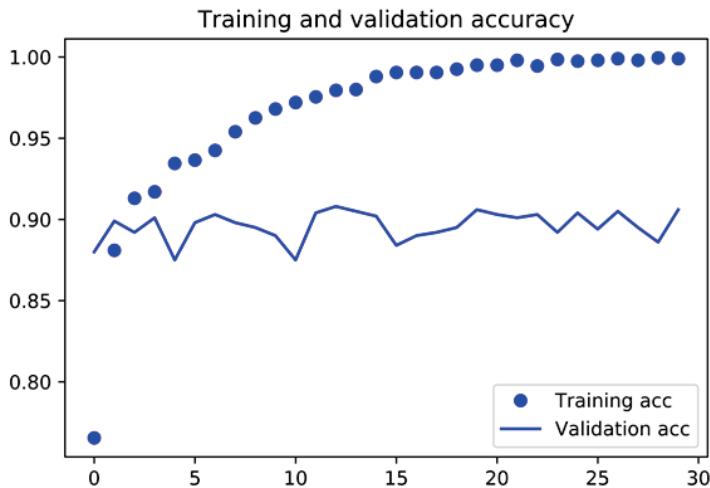


Figure 5.15 Training and validation accuracy for simple feature extraction (training values as dots, validation values as solid lines)

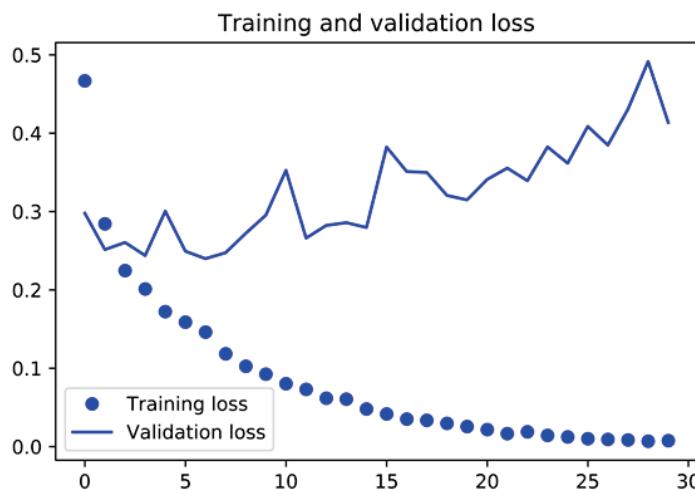


Figure 5.16 Training and validation loss for simple feature extraction (training values as dots, validation values as solid lines)

We reach a validation accuracy of about 90%, much better than what we could achieve in the previous section with our small model trained from scratch. However, our plots also indicate that we are overfitting almost from the start—despite using dropout with a fairly large rate. This is because this technique does not leverage data augmentation, which is essential to preventing overfitting with small image datasets.

Now, let's review the second technique we mentioned for doing feature extraction, which is much slower and more expensive, but which allows us to leverage data augmentation during training: extending the `conv_base` model and running it end-to-end on the inputs. Note that this technique is in fact so expensive that you should only attempt it if you have access to a GPU: it is absolutely intractable on CPU. If you cannot run your code on GPU, then the previous technique is the way to go.

Because models behave just like layers, you can add a model (like our `conv_base`) to a `Sequential` model just like you would add a layer. So you can do the following:

Listing 5.28 Adding a densely-connected classifier on top of the convolutional base

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

This is what our model looks like now:

Listing 5.29 Summary of the extended model

```
>>> model.summary()

Layer (type)                 Output Shape              Param #
=====
```

vgg16 (Model)	(None, 4, 4, 512)	14714688
flatten_1 (Flatten)	(None, 8192)	0
dense_1 (Dense)	(None, 256)	2097408
dense_2 (Dense)	(None, 1)	257
=====		
Total params:	16,812,353	
Trainable params:	16,812,353	
Non-trainable params:	0	

As you can see, the convolutional base of VGG16 has 14,714,688 parameters, which is very large. The classifier we are adding on top has 2 million parameters.

Before we compile and train our model, a very important thing to do is to freeze the convolutional base. "Freezing" a layer or set of layers means preventing their weights from getting updated during training. If we don't do this, then the representations that were previously learned by the convolutional base would get modified during training. Since the Dense layers on top are randomly initialized, very large weight updates would be propagated through the network, effectively destroying the representations previously learned.

In Keras, freezing a network is done by setting its `trainable` attribute to `False`:

Listing 5.30 Freezing the convolutional base

```
>>> print('This is the number of trainable weights '
      'before freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights before freezing the conv base: 30
>>> conv_base.trainable = False
>>> print('This is the number of trainable weights '
      'after freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights after freezing the conv base: 4
```

With this setup, only the weights from the two Dense layers that we added will be trained. That's a total of four weight tensors: two per layer (the main weight matrix and the bias vector). Note that in order for these changes to take effect, we must first compile the model. If you ever modify weight trainability after compilation, you should then re-compile the model, or these changes would be ignored.

Now we can start training our model, with the same data augmentation configuration that we used in our previous example:

Listing 5.31 Training the model end-to-end with a frozen convolutional base

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')
```

```
# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    # This is the target directory
    train_dir,
    # All images will be resized to 150x150
    target_size=(150, 150),
    batch_size=20,
    # Since we use binary_crossentropy loss, we need binary labels
    class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary')

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=2e-5),
              metrics=['acc'])

history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=30,
    validation_data=validation_generator,
    validation_steps=50)
```

Let's plot our results again:

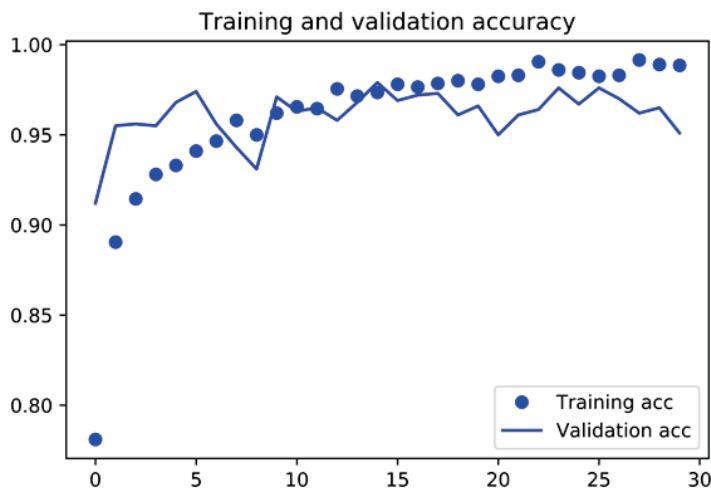


Figure 5.17 Training and validation accuracy for feature extraction with data augmentation (training values as dots, validation values as solid lines)

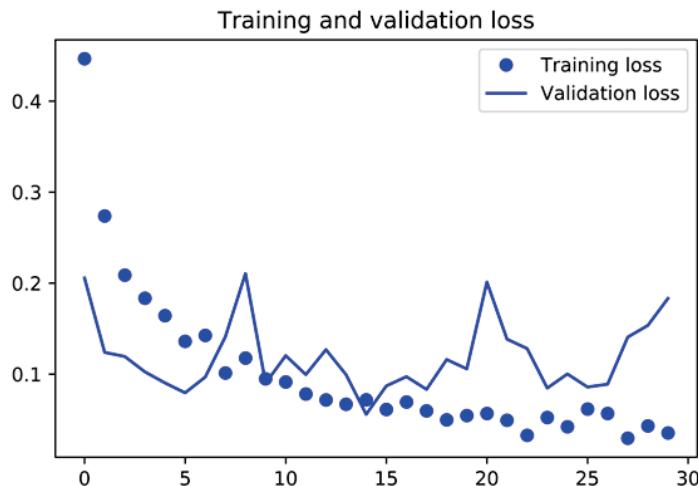


Figure 5.18 Training and validation loss for feature extraction with data augmentation (training values as dots, validation values as solid lines)

As you can see, we reach a validation accuracy of about 96%. This is much better than our small convnet trained from scratch.

5.3.2 Fine-tuning

Another widely used technique for model reuse, complementary to feature extraction, is *fine-tuning*. Fine-tuning consists in unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model (in our case, the fully-connected classifier) and these top layers. This is called "fine-tuning" because it slightly adjusts the more abstract representations of the model being reused, in order to make them more relevant for the problem at hand.

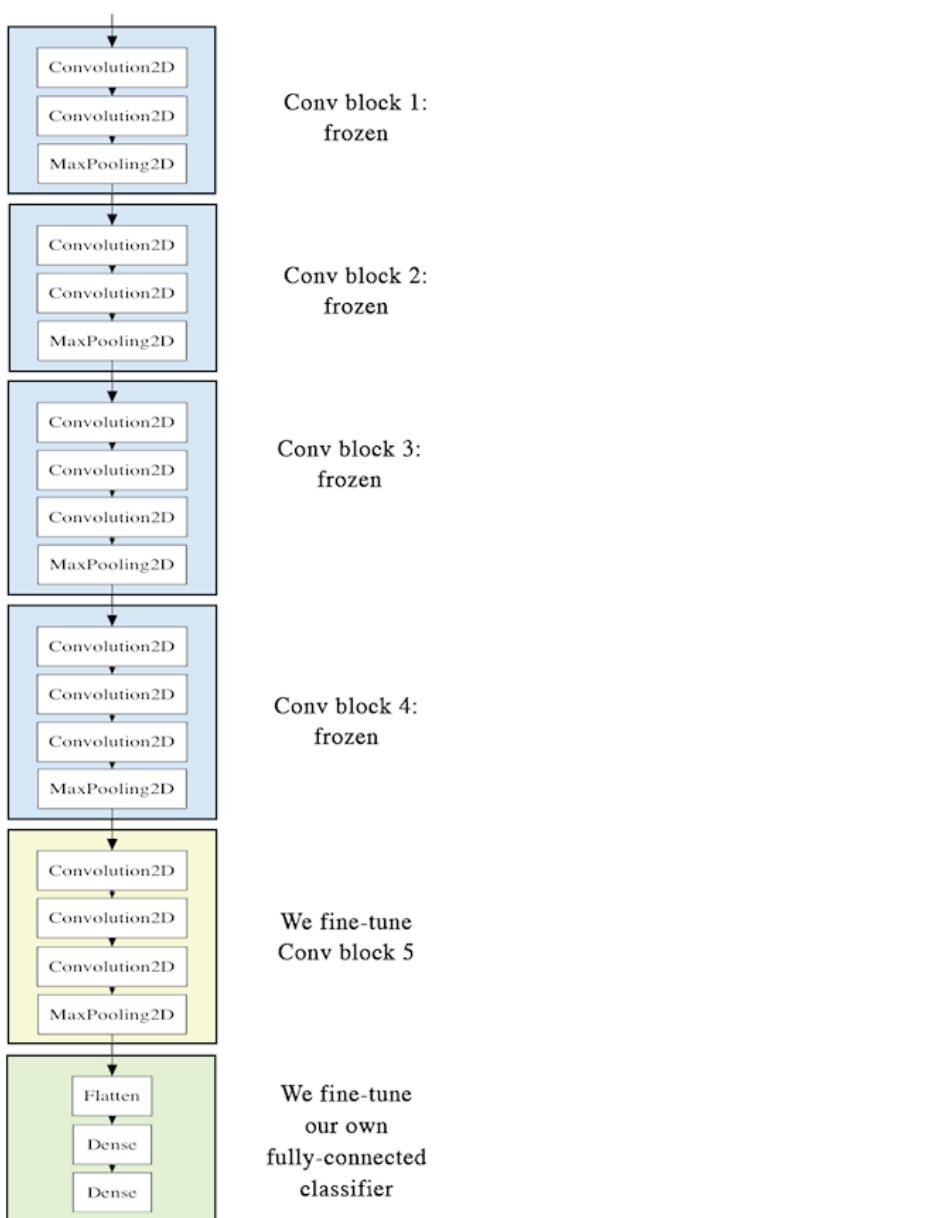


Figure 5.19 Fine-tuning the last convolutional block of the VGG16 network

We have stated before that it was necessary to freeze the convolution base of VGG16 in order to be able to train a randomly initialized classifier on top. For the same reason, it is only possible to fine-tune the top layers of the convolutional base once the classifier on top has already been trained. If the classifier wasn't already trained, then the error signal propagating through the network during training would be too large, and the representations previously learned by the layers being fine-tuned would be destroyed. Thus the steps for fine-tuning a network are as follow:

- 1) Add your custom network on top of an already trained base network.
- 2) Freeze the base network.
- 3) Train the part you added.
- 4) Unfreeze some layers in the base network.
- 5) Jointly train both these layers and the part you added.

We have already completed the first 3 steps when doing feature extraction. Let's proceed with the 4th step: we will unfreeze our `conv_base`, and then freeze individual layers inside of it.

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

Listing 5.32 Displaying a summary of the convolutional base

```
>>> conv_base.summary()

Layer (type)                  Output Shape                 Param #
=====                      =====
input_1 (InputLayer)          (None, 150, 150, 3)      0
block1_conv1 (Convolution2D)   (None, 150, 150, 64)    1792
block1_conv2 (Convolution2D)   (None, 150, 150, 64)    36928
block1_pool (MaxPooling2D)     (None, 75, 75, 64)      0
block2_conv1 (Convolution2D)   (None, 75, 75, 128)    73856
block2_conv2 (Convolution2D)   (None, 75, 75, 128)    147584
block2_pool (MaxPooling2D)     (None, 37, 37, 128)    0
block3_conv1 (Convolution2D)   (None, 37, 37, 256)    295168
block3_conv2 (Convolution2D)   (None, 37, 37, 256)    590080
block3_conv3 (Convolution2D)   (None, 37, 37, 256)    590080
block3_pool (MaxPooling2D)     (None, 18, 18, 256)    0
block4_conv1 (Convolution2D)   (None, 18, 18, 512)   1180160
block4_conv2 (Convolution2D)   (None, 18, 18, 512)   2359808
block4_conv3 (Convolution2D)   (None, 18, 18, 512)   2359808
block4_pool (MaxPooling2D)     (None, 9, 9, 512)      0
block5_conv1 (Convolution2D)   (None, 9, 9, 512)      2359808
block5_conv2 (Convolution2D)   (None, 9, 9, 512)      2359808
block5_conv3 (Convolution2D)   (None, 9, 9, 512)      2359808
block5_pool (MaxPooling2D)     (None, 4, 4, 512)      0
=====
Total params: 14714688
```

We will fine-tune the last 3 convolutional layers, which means that all layers up until `block4_pool` should be frozen, and the layers `block5_conv1`, `block5_conv2` and `block5_conv3` should be trainable.

Why not fine-tune more layers? Why not fine-tune the entire convolutional base? We could. However, we need to consider that:

- Earlier layers in the convolutional base encode more generic, reusable features, while layers higher up encode more specialized features. It is more useful to fine-tune the more specialized features, as these are the ones that need to be repurposed on our new problem. There would be fast-decreasing returns in fine-tuning lower layers.
- The more parameters we are training, the more we are at risk of overfitting. The

convolutional base has 15M parameters, so it would be risky to attempt to train it on our small dataset.

Thus, in our situation, it is a good strategy to only fine-tune the top 2 to 3 layers in the convolutional base.

Let's set this up, starting from where we left off in the previous example:

Listing 5.33 Freezing all layers up to a specific one

```
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

Now we can start fine-tuning our network. We will do this with the RMSprop optimizer, using a very low learning rate. The reason for using a low learning rate is that we want to limit the magnitude of the modifications we make to the representations of the 3 layers that we are fine-tuning. Updates that are too large may harm these representations.

Now let's proceed with fine-tuning:

Listing 5.34 Fine-tuning our model

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=100,
    validation_data=validation_generator,
    validation_steps=50)
```

Let's plot our results using the same plotting code as before:

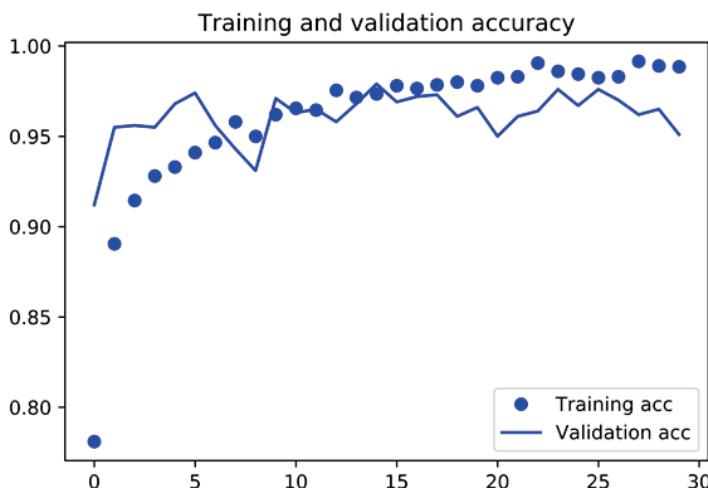


Figure 5.20 Training and validation accuracy for fine-tuning (training values as dots, validation values as solid lines)

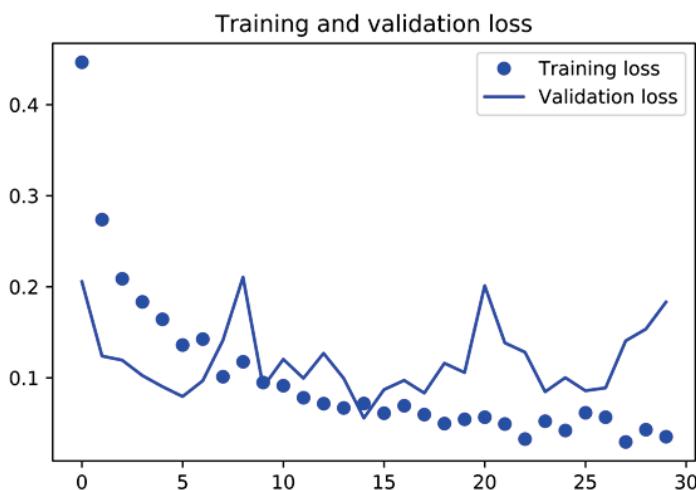


Figure 5.21 Training and validation loss for fine-tuning (training values as dots, validation values as solid lines)

These curves look very noisy. To make them more readable, we can smooth them by replacing every loss and accuracy with exponential moving averages of these quantities. Here's a trivial utility function to do this:

Listing 5.35 Smoothing our plots

```
def smooth_curve(points, factor=0.8):
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
    return smoothed_points

plt.plot(epochs,
         smooth_curve(acc), 'bo', label='Smoothed training acc')
plt.plot(epochs,
         smooth_curve(val_acc), 'b', label='Smoothed validation acc')
plt.title('Training and validation accuracy')
plt.legend()
```

```

plt.figure()

plt.plot(epochs,
         smooth_curve(loss), 'bo', label='Smoothed training loss')
plt.plot(epochs,
         smooth_curve(val_loss), 'b', label='Smoothed validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()

```

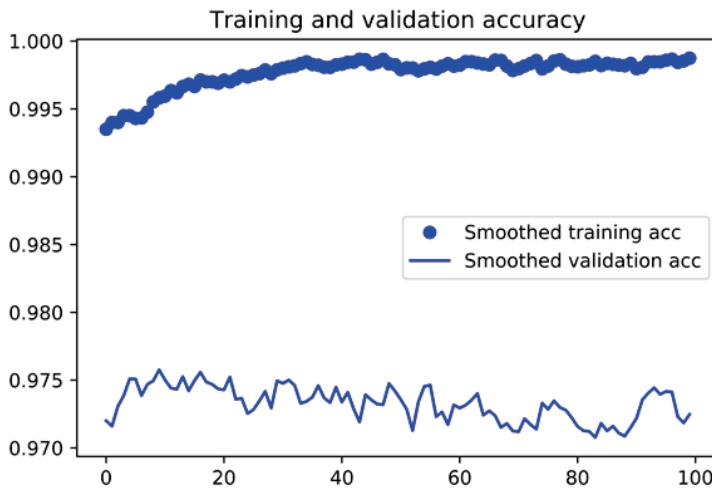


Figure 5.22 Smoothed curves for training and validation accuracy for fine-tuning (training values as dots, validation values as solid lines)

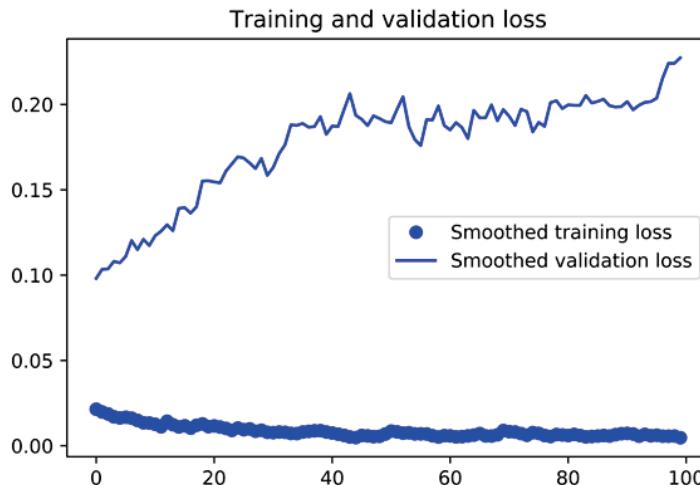


Figure 5.23 Smoothed curves for training and validation loss for fine-tuning (training values as dots, validation values as solid lines)

These curves look much cleaner and more stable. We are seeing a nice 1% absolute improvement.

Note that the loss curve does not show any real improvement (in fact, it is deteriorating). You may wonder, how could accuracy improve if the loss isn't decreasing? The answer is simple: what we display is an average of pointwise loss values, but what actually matters for accuracy is the distribution of the loss values, not their average, since accuracy is the result of a binary thresholding of the class probability

predicted by the model. The model may still be improving even if this isn't reflected in the average loss.

We can now finally evaluate this model on the test data:

```
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary')

test_loss, test_acc = model.evaluate_generator(test_generator, steps=50)
print('test acc:', test_acc)
```

Here we get a test accuracy of 97%. In the original Kaggle competition around this dataset, this would have been one of the top results. However, using modern deep learning techniques, we managed to reach this result using only a very small fraction of the training data available (about 10%). There is a huge difference between being able to train on 20,000 samples compared to 2,000 samples!

5.3.3 Take-aways: using convnets with small datasets

Here's what you should take away from the exercises of these past two sections:

- Convnets are the best type of machine learning models for computer vision tasks. It is possible to train one from scratch even on a very small dataset, with decent results.
- On a small dataset, overfitting will be the main issue. Data augmentation is a powerful way to fight overfitting when working with image data.
- It is easy to reuse an existing convnet on a new dataset, via feature extraction. This is a very valuable technique for working with small image datasets.
- As a complement to feature extraction, one may use fine-tuning, which adapts to a new problem some of the representations previously learned by an existing model. This pushes performance a bit further.

Now you have a solid set of tools for dealing with image classification problems, in particular with small datasets.

5.4 Visualizing what convnets learn

It is often said that deep learning models are "black boxes", learning representations that are difficult to extract and present in a human-readable form. While this is partially true for certain types of deep learning models, it is definitely not true for convnets. The representations learned by convnets are highly amenable to visualization, in large part because they are *representations of visual concepts*. Since 2013, a wide array of techniques have been developed for visualizing and interpreting these representations. We won't survey all of them, but we will cover three of the most accessible and useful ones:

- Visualizing intermediate convnet outputs ("intermediate activations"). This is useful to understand how successive convnet layers transform their input, and to get a first idea of the meaning of individual convnet filters.
- Visualizing convnets filters. This is useful to understand precisely what visual pattern or concept each filter in a convnet is receptive to.

- Visualizing heatmaps of class activation in an image. This is useful to understand which part of an image where identified as belonging to a given class, and thus allows to localize objects in images.

For the first method—activation visualization—we will use the small convnet that we trained from scratch on the cat vs. dog classification problem two sections ago. For the next two methods, we will use the VGG16 model that we introduced in the previous section.

5.4.1 Visualizing intermediate activations

Visualizing intermediate activations consists in displaying the feature maps that are output by various convolution and pooling layers in a network, given a certain input (the output of a layer is often called its "activation", the output of the activation function). This gives a view into how an input is decomposed unto the different filters learned by the network. These feature maps we want to visualize have 3 dimensions: width, height, and depth (channels). Each channel encodes relatively independent features, so the proper way to visualize these feature maps is by independently plotting the contents of every channel, as a 2D image. Let's start by loading the model that we saved in section 5.2:

Listing 5.36 Loading a saved model and printing a summary

```
>>> from keras.models import load_model
>>> model = load_model('cats_and_dogs_small_2.h5')
>>> model.summary() # As a reminder.

Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 148, 148, 32)    896
maxpooling2d_5 (MaxPooling2D) (None, 74, 74, 32)      0
conv2d_6 (Conv2D)            (None, 72, 72, 64)     18496
maxpooling2d_6 (MaxPooling2D) (None, 36, 36, 64)      0
conv2d_7 (Conv2D)            (None, 34, 34, 128)    73856
maxpooling2d_7 (MaxPooling2D) (None, 17, 17, 128)    0
conv2d_8 (Conv2D)            (None, 15, 15, 128)    147584
maxpooling2d_8 (MaxPooling2D) (None, 7, 7, 128)      0
flatten_2 (Flatten)          (None, 6272)             0
dropout_1 (Dropout)          (None, 6272)             0
dense_3 (Dense)              (None, 512)              3211776
dense_4 (Dense)              (None, 1)                513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

This will be the input image we will use—a picture of a cat, not part of images that the network was trained on:

Listing 5.37 Preprocessing a single image

```
img_path = '/Users/fchollet/Downloads/cats_and_dogs_small/test/cats/cat.1700.jpg'

# We preprocess the image into a 4D tensor
from keras.preprocessing import image
import numpy as np

img = image.load_img(img_path, target_size=(150, 150))
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
# Remember that the model was trained on inputs
# that were preprocessed in the following way:
img_tensor /= 255.

# Its shape is (1, 150, 150, 3)
print(img_tensor.shape)
```

Let's display our picture:

Listing 5.38 Displaying the test picture

```
import matplotlib.pyplot as plt

plt.imshow(img_tensor[0])
```

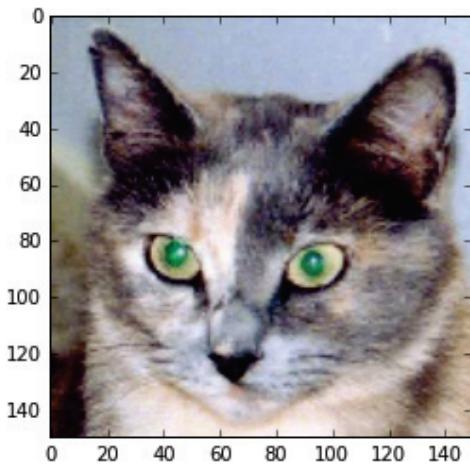


Figure 5.24 Our test cat picture

In order to extract the feature maps we want to look at, we will create a Keras model that takes batches of images as input, and outputs the activations of all convolution and pooling layers. To do this, we will use the Keras class `Model`. A `Model` is instantiated using two arguments: an input tensor (or list of input tensors), and an output tensor (or list of output tensors). The resulting class is a Keras model, just like the `Sequential` models that you are familiar with, mapping the specified inputs to the specified outputs. What sets the `Model` class apart is that it allows for models with multiple outputs, unlike `Sequential`. For more information about the `Model` class, see Chapter 7, Section 1.

Listing 5.39 Instantiating a Model from an input tensor and a list of output tensors

```
from keras import models

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

When fed an image input, this model returns the values of the layer activations in the original model. This is the first time you encounter a multi-output model in this book: until now the models you have seen only had exactly one input and one output. In the general case, a model could have any number of inputs and outputs. This one has one input and 5 outputs, one output per layer activation.

Listing 5.40 Running our model in predict mode

```
# This will return a list of 5 Numpy arrays:
# one array per layer activation
activations = activation_model.predict(img_tensor)
```

For instance, this is the activation of the first convolution layer for our cat image input:

Listing 5.41 First entry in the outputs: the output of the first layer of the original model

```
>>> first_layer_activation = activations[0]
>>> print(first_layer_activation.shape)
(1, 148, 148, 32)
```

It's a 148x148 feature map with 32 channels. Let's try visualizing the 4th channel:

Listing 5.42 Plotting the 4th channel of the activation of the first layer of the original model

```
import matplotlib.pyplot as plt

plt.matshow(first_layer_activation[0, :, :, 4], cmap='viridis')
```

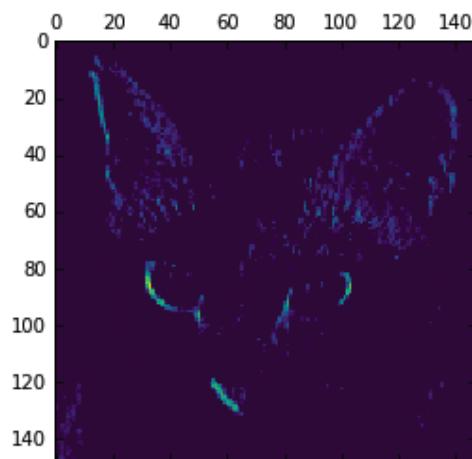


Figure 5.25 4th channel of the activation of the first layer on our test cat picture

This channel appears to encode a diagonal edge detector. Let's try the 7th channel—but note that your own channels may vary, since the specific filters learned by convolution layers are not deterministic.

Listing 5.43 Plotting the 7th channel of the activation of the first layer of the original model

```
plt.matshow(first_layer_activation[0, :, :, 7], cmap='viridis')
```

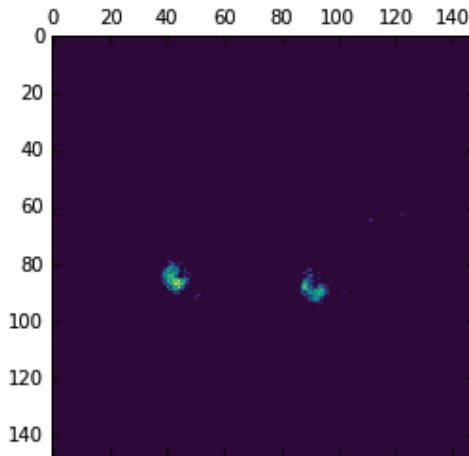


Figure 5.26 7th of the activation of the first layer on our test cat picture

This one looks like a "bright green dot" detector, useful to encode cat eyes. At this point, let's go and plot a complete visualization of all the activations in the network. We'll extract and plot every channel in each of our 5 activation maps, and we will stack the results in one big image tensor, with channels stacked side by side.

Listing 5.44 Visualizing every channel in every intermediate activation

```
# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:8]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))
```

```
# We'll tile each filter into this big horizontal grid
for col in range(n_cols):
    for row in range(images_per_row):
        channel_image = layer_activation[0,
                                         :, :,
                                         col * images_per_row + row]
        # Post-process the feature to make it visually palatable
        channel_image -= channel_image.mean()
        channel_image /= channel_image.std()
        channel_image *= 64
        channel_image += 128
        channel_image = np.clip(channel_image, 0, 255).astype('uint8')
        display_grid[col * size : (col + 1) * size,
                     row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')
```

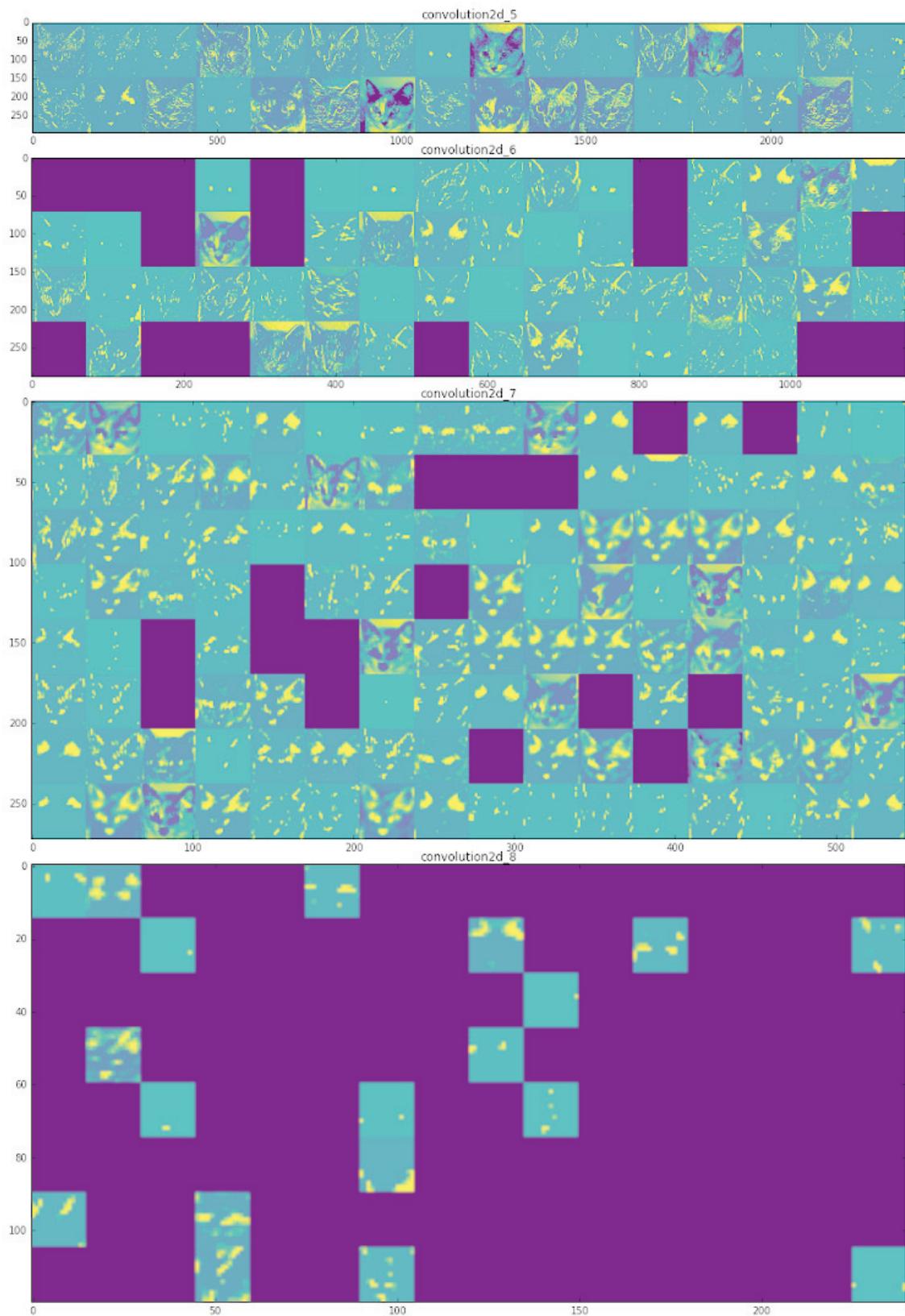


Figure 5.27 Every channel of every layer activation on our test cat picture

A few remarkable things to note here:

- The first layer acts as a collection of various edge detectors. At that stage, the activations

are still retaining almost all of the information present in the initial picture.

- As we go higher-up, the activations become increasingly abstract and less visually interpretable. They start encoding higher-level concepts such as "cat ear" or "cat eye". Higher-up presentations carry increasingly less information about the visual contents of the image, and increasingly more information related to the class of the image.
- The sparsity of the activations is increasing with the depth of the layer: in the first layer, all filters are activated by the input image, but in the following layers more and more filters are blank. This means that the pattern encoded by the filter isn't found in the input image.

We have just evidenced a very important universal characteristic of the representations learned by deep neural networks: the features extracted by a layer get increasingly abstract with the depth of the layer. The activations of layers higher-up carry less and less information about the specific input being seen, and more and more information about the target (in our case, the class of the image: cat or dog). A deep neural network effectively acts as an *information distillation pipeline*, with raw data going in (in our case, RBG pictures), and getting repeatedly transformed so that irrelevant information gets filtered out (e.g. the specific visual appearance of the image) while useful information get magnified and refined (e.g. the class of the image).

This is analogous to the way humans and animals perceive the world: after observing a scene for a few seconds, a human can remember which abstract objects were present in it (e.g. bicycle, tree) but could not remember the specific appearance of these objects. In fact, if you tried to draw a generic bicycle from mind right now, chances are you could not get it even remotely right, even though you have seen thousands of bicycles in your lifetime. Try it right now: this effect is absolutely real. Your brain has learned to completely abstract its visual input, to transform it into high-level visual concepts while completely filtering out irrelevant visual details, making it tremendously difficult to remember how things around us actually look.

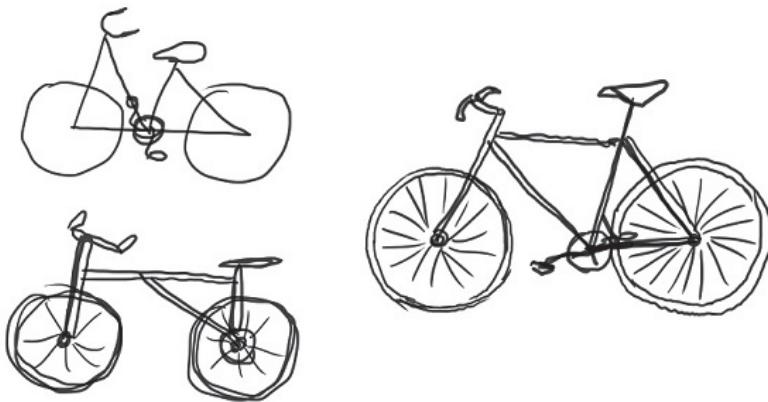


Figure 5.28 Left: attempts to draw a bicycle from memory. Right: what a schematic bicycle should look like.

5.4.2 Visualizing convnet filters

Another easy thing to do to inspect the filters learned by convnets is to display the visual pattern that each filter is meant to respond to. This can be done with *gradient ascent in input space*: applying *gradient descent* to the value of the input image of a convnet so as to maximize the response of a specific filter, starting from a blank input image. The resulting input image would be one that the chosen filter is maximally responsive to.

The process is simple: we will build a loss function that maximizes the value of a given filter in a given convolution layer, then we will use stochastic gradient descent to adjust the values of the input image so as to maximize this activation value. For instance, here's a loss for the activation of filter 0 in the layer "block3_conv1" of the VGG16 network, pre-trained on ImageNet:

Listing 5.45 Defining the loss tensor for filter visualization

```
from keras.applications import VGG16
from keras import backend as K

model = VGG16(weights='imagenet',
               include_top=False)

layer_name = 'block3_conv1'
filter_index = 0

layer_output = model.get_layer(layer_name).output
loss = K.mean(layer_output[:, :, :, filter_index])
```

To implement gradient descent, we will need the gradient of this loss with respect to the model's input. To do this, we will use the `gradients` function packaged with the `backend` module of Keras:

Listing 5.46 Obtaining the gradient of the loss with regard to the input

```
# The call to `gradients` returns a list of tensors (of size 1 in this case)
# hence we only keep the first element -- which is a tensor.
grads = K.gradients(loss, model.input)[0]
```

A non-obvious trick to use for the gradient descent process to go smoothly is to normalize the gradient tensor, by dividing it by its L2 norm (the square root of the average of the square of the values in the tensor). This ensures that the magnitude of the updates done to the input image is always within a same range.

Listing 5.47 The gradient normalization trick

```
# We add 1e-5 before dividing so as to avoid accidentally dividing by 0.
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
```

Now we need a way to compute the value of the loss tensor and the gradient tensor, given an input image. We can define a Keras backend function to do this: `iterate` is a

function that takes a Numpy tensor (as a list of tensors of size 1) and returns a list of two Numpy tensors: the loss value and the gradient value.

Listing 5.48 Defining a Keras function for fetching Numpy output values given Numpy input values

```
iterate = K.function([model.input], [loss, grads])

# Let's test it:
import numpy as np
loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])
```

At this point we can define a Python loop to do stochastic gradient descent:

Listing 5.49 Loss maximization via stochastic gradient descent over the input parameters

```
# We start from a gray image with some noise
input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.

# Run gradient ascent for 40 steps
step = 1. # this is the magnitude of each gradient update
for i in range(40):
    # Compute the loss value and gradient value
    loss_value, grads_value = iterate([input_img_data])
    # Here we adjust the input image in the direction that maximizes the loss
    input_img_data += grads_value * step
```

The resulting image tensor will be a floating point tensor of shape (1, 150, 150, 3), with values that may not be integer within [0, 255]. Hence we would need to post-process this tensor to turn it into a displayable image. We do it with the following straightforward utility function:

Listing 5.50 Utility function to convert a tensor into a valid image

```
def deprocess_image(x):
    # normalize tensor: center on 0., ensure std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    # clip to [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)

    # convert to RGB array
    x *= 255
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

Now we have all the pieces, let's put them together into a Python function that takes as input a layer name and a filter index, and that returns a valid image tensor representing the pattern that maximizes the activation the specified filter:

Listing 5.51 Putting it all together: a function to generate filter visualizations

```

def generate_pattern(layer_name, filter_index, size=150):
    # Build a loss function that maximizes the activation
    # of the nth filter of the layer considered.
    layer_output = model.get_layer(layer_name).output
    loss = K.mean(layer_output[:, :, :, :, filter_index])

    # Compute the gradient of the input picture wrt this loss
    grads = K.gradients(loss, model.input)[0]

    # Normalization trick: we normalize the gradient
    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)

    # This function returns the loss and grads given the input picture
    iterate = K.function([model.input], [loss, grads])

    # We start from a gray image with some noise
    input_img_data = np.random.random((1, size, size, 3)) * 20 + 128.

    # Run gradient ascent for 40 steps
    step = 1.
    for i in range(40):
        loss_value, grads_value = iterate([input_img_data])
        input_img_data += grads_value * step

    img = input_img_data[0]
    return deprocess_image(img)

```

Let's try this:

Listing 5.52 Visualising the response patterns of filter 0 of block3_conv1

```
>>> plt.imshow(generate_pattern('block3_conv1', 0))
```

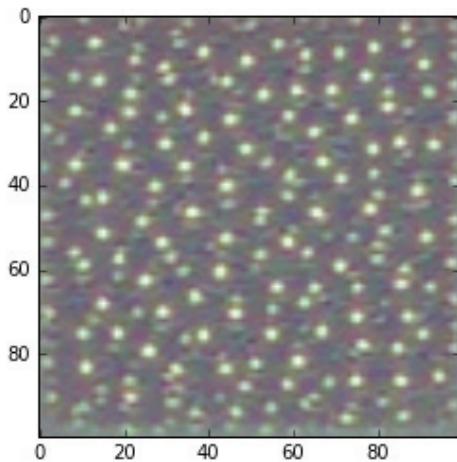


Figure 5.29 Pattern that the 0th channel in layer block3_conv1 maximally responds to

It seems that filter 0 in layer block3_conv1 is responsive to a polka dot pattern.

Now the fun part: we can start visualising every single filter in every layer. For simplicity, we will only look at the first 64 filters in each layer, and will only look at the first layer of each convolution block (block1_conv1, block2_conv1, block3_conv1, block4_conv1, block5_conv1). We will arrange the outputs on a 8x8 grid of 64x64 filter patterns, with some black margins between each filter pattern.

Listing 5.53 Generating of grid of all filter response patterns in a layer

```

layer_name = 'block1_conv1'
size = 64
margin = 5

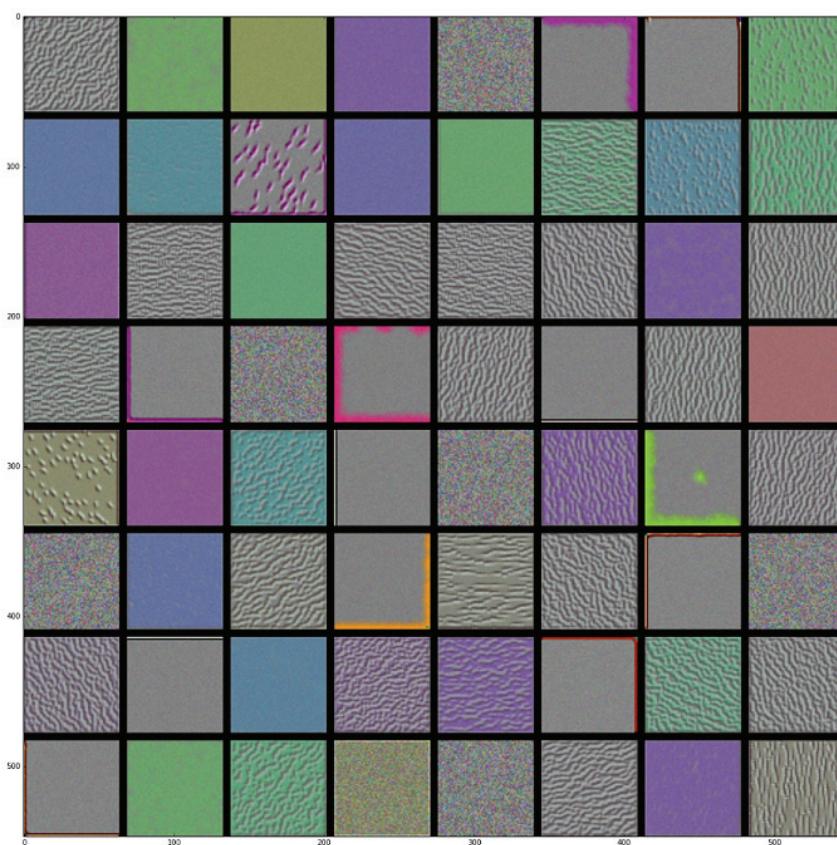
# This a empty (black) image where we will store our results.
results = np.zeros((8 * size + 7 * margin, 8 * size + 7 * margin, 3))

for i in range(8): # iterate over the rows of our results grid
    for j in range(8): # iterate over the columns of our results grid
        # Generate the pattern for filter `i + (j * 8)` in `layer_name`
        filter_img = generate_pattern(layer_name, i + (j * 8), size=size)

        # Put the result in the square `(i, j)` of the results grid
        horizontal_start = i * size + i * margin
        horizontal_end = horizontal_start + size
        vertical_start = j * size + j * margin
        vertical_end = vertical_start + size
        results[horizontal_start: horizontal_end, vertical_start: vertical_end, :] = filter_img

# Display the results grid
plt.figure(figsize=(20, 20))
plt.imshow(results)

```

**Figure 5.30 Filter patterns for layer block1_conv1**

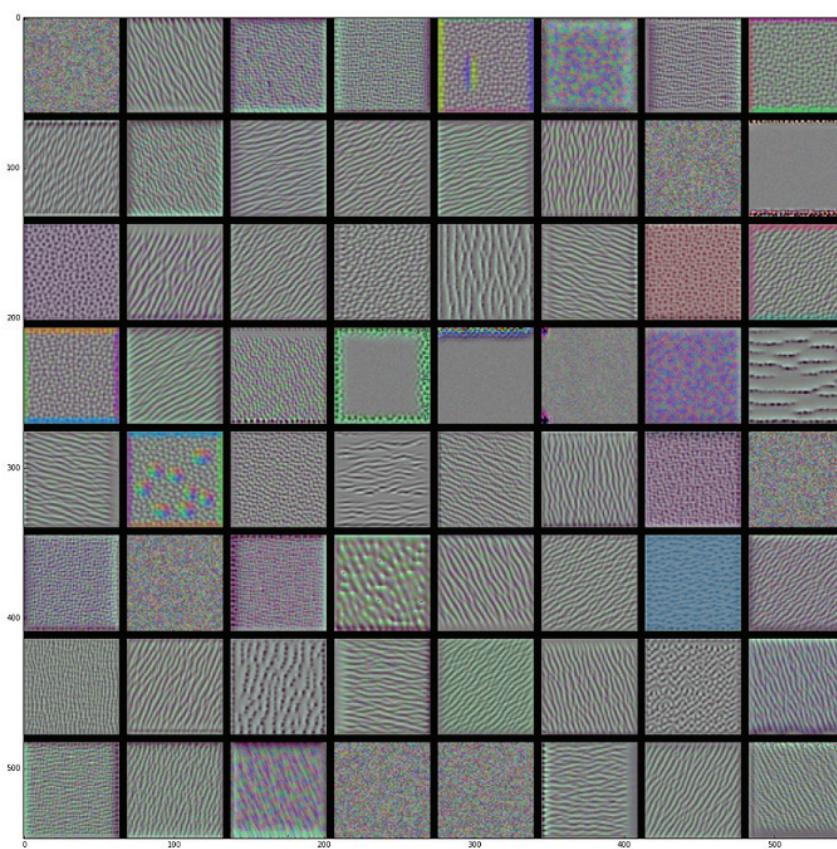


Figure 5.31 Filter patterns for layer block2_conv1

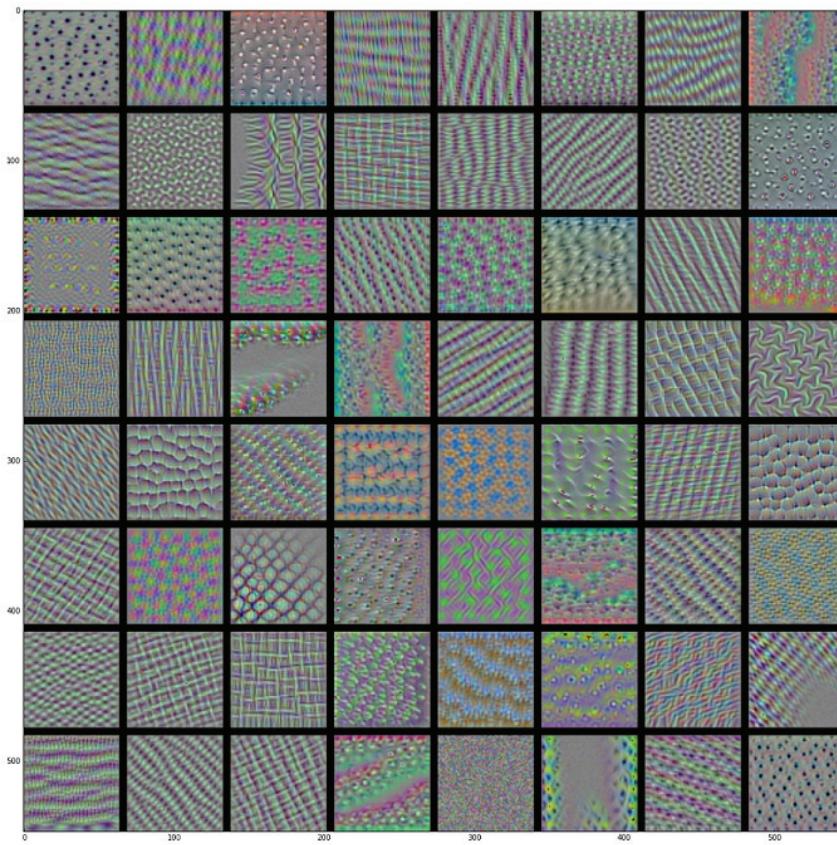


Figure 5.32 Filter patterns for layer block3_conv1

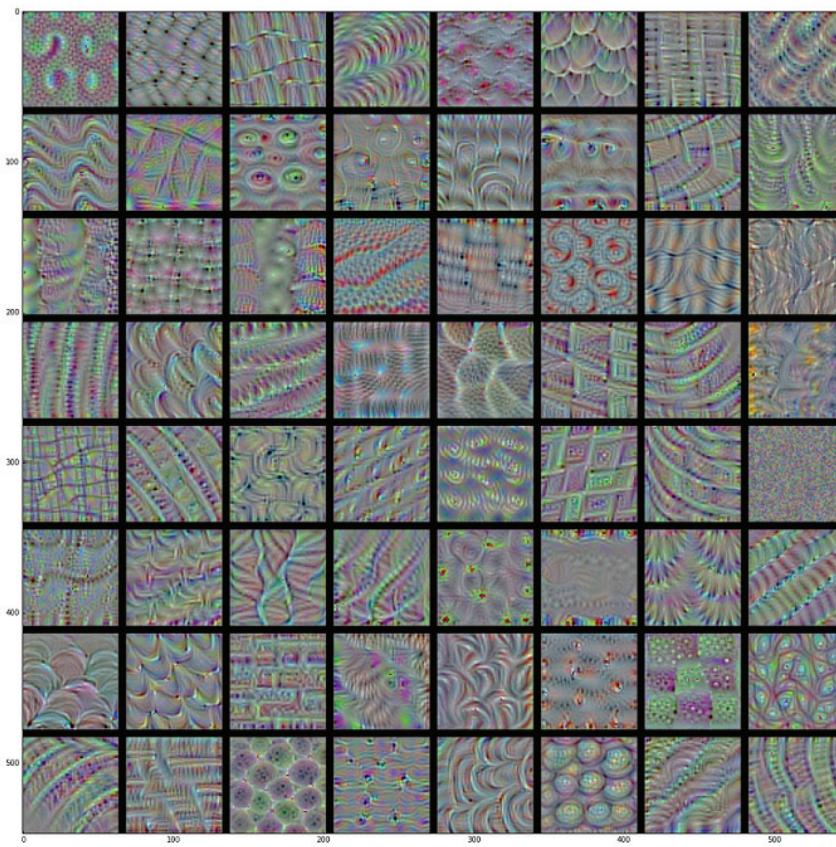


Figure 5.33 Filter patterns for layer block4_conv1

These filter visualizations tell us a lot about how convnet layers see the world: each layer in a convnet simply learns a collection of filters such that their inputs can be expressed as a combination of the filters. This is similar to how the Fourier transform decomposes signals onto a bank of cosine functions. The filters in these convnet filter banks get increasingly complex and refined as we go higher-up in the model:

- The filters from the first layer in the model (block1_conv1) encode simple directional edges and colors (or colored edges in some cases).
- The filters from block2_conv1 encode simple textures made from combinations of edges and colors.
- The filters in higher-up layers start resembling textures found in natural images: feathers, eyes, leaves, etc.

5.4.3 Visualizing heatmaps of class activation

We will introduce one more visualization technique, one that is useful for understanding which parts of a given image led a convnet to its final classification decision. This is helpful for "debugging" the decision process of a convnet, in particular in case of a classification mistake. It also allows you to locate specific objects in an image.

This general category of techniques is called "Class Activation Map" (CAM) visualization, and consists in producing heatmaps of "class activation" over input images. A "class activation" heatmap is a 2D grid of scores associated with a specific output class, computed for every location in any input image, indicating how important each

location is with respect to the class considered. For instance, given a image fed into one of our "cat vs. dog" convnet, Class Activation Map visualization allows us to generate a heatmap for the class "cat", indicating how cat-like different parts of the image are, and likewise for the class "dog", indicating how dog-like differents parts of the image are.

The specific implementation we will use is the one described in [Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization](arxiv.org/abs/1610.02391). It is very simple: it consists in taking the output feature map of a convolution layer given an input image, and weighing every channel in that feature map by the gradient of the class with respect to the channel. Intuitively, one way to understand this trick is that we are weighting a spatial map of "how intensely the input image activates different channels" by "how important each channel is with regard to the class", resulting in a spatial map of "how intensely the input image activates the class".

We will demonstrate this technique using the pre-trained VGG16 network again:

Listing 5.54 Loading the VGG16 network with pre-trained weights

```
from keras.applications.vgg16 import VGG16  
  
# Note that we are including the densely-connected classifier on top;  
# all previous times, we were discarding it.  
model = VGG16(weights='imagenet')
```

Let's consider the following image of two African elephants, possible a mother and its cub, strolling in the savanna (under a Creative Commons license):



Figure 5.34 Our test picture of African elephants

Let's convert this image into something the VGG16 model can read: the model was trained on images of size 224x244, preprocessed according to a few rules that are packaged in the utility function `keras.applications.vgg16.preprocess_input`. So we need to load the image, resize it to 224x224, convert it to a Numpy float32 tensor, and apply these pre-processing rules.

Listing 5.55 Pre-processing an input image for VGG16

```
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input, decode_predictions
import numpy as np

# The local path to our target image
img_path = '/Users/fchollet/Downloads/creative_commons_elephant.jpg'

# `img` is a PIL image of size 224x224
img = image.load_img(img_path, target_size=(224, 224))

# `x` is a float32 Numpy array of shape (224, 224, 3)
x = image.img_to_array(img)

# We add a dimension to transform our array into a "batch"
# of size (1, 224, 224, 3)
x = np.expand_dims(x, axis=0)

# Finally we preprocess the batch
# (this does channel-wise color normalization)
x = preprocess_input(x)
```

We can then run the pre-trained network on the image, and decode its prediction vector back to a human-readable format:

Listing 5.56 Predicting the class of our image

```
>>> preds = model.predict(x)
>>> print('Predicted:', decode_predictions(preds, top=3)[0])
Predicted: [u'n02504458', u'African_elephant', 0.92546833],
(u'n01871265', u'tusker', 0.070257246),
(u'n02504013', u'Indian_elephant', 0.0042589349)]
```

The top-3 classes predicted for this image are:

- African elephant (with 92.5% probability)
- Tusker (with 7% probability)
- Indian elephant (with 0.4% probability)

Thus our network has recognized our image as containing an undetermined quantity of African elephants. The entry in the prediction vector that was maximally activated is the one corresponding to the "African elephant" class, at index 386:

Listing 5.57 Retrieving the index of maximum prediction

```
>>> np.argmax(preds[0])
386
```

To visualize which parts of our image were the most "African elephant"-like, let's set up the Grad-CAM process:

Listing 5.58 Setting up the Grad-CAM algorithm

```
# This is the "african elephant" entry in the prediction vector
african_elephant_output = model.output[:, 386]

# This is the output feature map of the `block5_conv3` layer,
# the last convolutional layer in VGG16
last_conv_layer = model.get_layer('block5_conv3')

# This is the gradient of the "african elephant" class with regard to
# the output feature map of `block5_conv3`
grads = K.gradients(african_elephant_output, last_conv_layer.output)[0]

# This is a vector of shape (512,), where each entry
# is the mean intensity of the gradient over a specific feature map channel
pooled_grads = K.mean(grads, axis=(0, 1, 2))

# This function allows us to access the values of the quantities we just defined:
# `pooled_grads` and the output feature map of `block5_conv3`,
# given a sample image
iterate = K.function([model.input], [pooled_grads, last_conv_layer.output[0]])

# These are the values of these two quantities, as Numpy arrays,
# given our sample image of two elephants
pooled_grads_value, conv_layer_output_value = iterate([x])

# We multiply each channel in the feature map array
# by "how important this channel is" with regard to the elephant class
for i in range(512):
    conv_layer_output_value[:, :, i] *= pooled_grads_value[i]

# The channel-wise mean of the resulting feature map
# is our heatmap of class activation
heatmap = np.mean(conv_layer_output_value, axis=-1)
```

For visualization purpose, we will also normalize the heatmap between 0 and 1:

Listing 5.59 Heatmap post-processing

```
heatmap = np.maximum(heatmap, 0)
heatmap /= np.max(heatmap)
plt.matshow(heatmap)
```

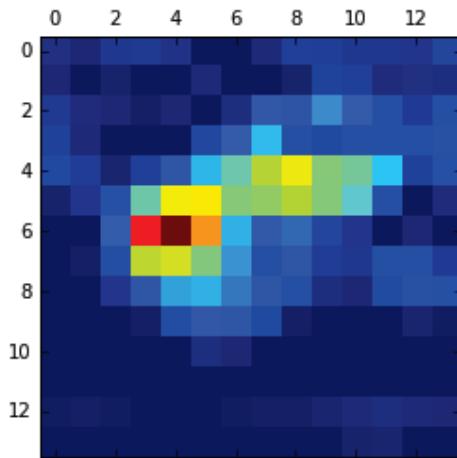


Figure 5.35 African elephant class activation heatmap over our test picture

Finally, we will use OpenCV to generate an image that superimposes the original image with the heatmap we just obtained:

Listing 5.60 Superimposing the heatmap with the original picture, and saving it to disk

```
import cv2

# We use cv2 to load the original image
img = cv2.imread(img_path)

# We resize the heatmap to have the same size as the original image
heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))

# We convert the heatmap to RGB
heatmap = np.uint8(255 * heatmap)

# We apply the heatmap to the original image
heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)

# 0.4 here is a heatmap intensity factor
superimposed_img = heatmap * 0.4 + img

# Save the image to disk
cv2.imwrite('/Users/fchollet/Downloads/elephant_cam.jpg', superimposed_img)
```



Figure 5.36 Superimposing the class activation heatmap with the original picture

This visualisation technique answers two important questions:

- Why did the network think this image contained an African elephant?

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

- Where is the African elephant located in the picture?

In particular, it is interesting the note that the ears of the elephant cub are strongly activated: this is probably how the network can tell the difference between African and Indian elephants.

5.5 Wrapping up: deep learning for computer vision

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

- Convnets are the best tool for attacking visual classification problems.
- They work by learning a hierarchy of modular patterns and concepts to represent the visual world.
- The representations they learn are easy to inspect—they are the opposite of a black box!

Additionally, you should have picked up practical skills:

- You are capable of training your own convnet from scratch to solve an image classification problem.
- You understand how to use visual data augmentation to fight overfitting.
- You know how to use a pre-trained convnet to do feature extraction and fine-tuning.
- You can generate visualizations of the filters learned by your convnets, as well as heatmaps of class activity.

Deep learning for text and sequences

Here's what you have learned in this chapter:

- How to tokenize text.
- What word embeddings are, and how to use them.
- What recurrent networks are, and how to use them.
- How to stack RNN layers and use bidirectional RNNs to build more powerful sequence processing models.
- How to use 1D convnets for sequence processing.
- How to combine 1D convnets and RNNs to process long sequences.

These techniques are widely applicable to any dataset of sequence data, from text to timeseries.

For instance, you could use RNNs for:

- Timeseries regression ("predicting the future").
- Timeseries classification.
- Anomaly detection in timeseries.
- Sequence labeling, e.g. identifying names or dates in sentences.
- ...

Similarly, you could use 1D convnets for:

- Machine translation (sequence-to-sequence convolutional models, like SliceNet).
- Document classification.
- Spelling correction.
- ...

Remember: if *global order matters* in your sequence data, then it is preferable to use a recurrent network to process it. This is typically the case for timeseries, where the recent past is likely to be more informative than the distant past. But if global ordering isn't fundamentally meaningful, then 1D convnets will turn out to work at least as well, while being cheaper. This is often the case for text data, where a keyword found at the beginning of a sentence is just as meaningful as a keyword found at the end.

6.1 Working with text data

Text is one of the most widespread form of sequence data. It can be understood either as a sequence of characters, or a sequence of words, albeit it is most common to work at the level of words. The deep learning sequence processing models that we will introduce in the next sections are able to leverage text to produce a basic form of natural language understanding, sufficient for applications ranging from document classification, sentiment analysis, author identification, or even question answering (in a constrained context). Of course, keep in mind throughout this chapter that none of the deep learning models you see truly "understands" text in a human sense, rather, these models are able to map the statistical structure of written language, which is sufficient to solve many simple textual tasks. Deep learning for natural language processing is simply pattern recognition applied to words, sentences, and paragraphs, in much the same way that computer vision is simply pattern recognition applied to pixels.

Like all other neural networks, deep learning models do not take as input raw text: they only work with numeric tensors. Vectorizing text is the process of transforming text into numeric tensors. This can be done in multiple ways:

- By segmenting text into words, and transforming each word into a vector.
- By segmenting text into characters, and transforming each character into a vector.
- By extracting "N-grams" of words or characters, and transforming each N-gram into a vector. "N-grams" are overlapping groups of multiple consecutive words or characters.

Collectively, the different units into which you can break down text (words, characters or N-grams) are called "tokens", and breaking down text into such tokens is called "tokenization". All text vectorization processes consist in applying some tokenization scheme, then associating numeric vectors with the generated tokens. These vectors, packed into sequence tensors, are what gets fed into deep neural networks. There are multiple ways to associate a vector to a token. In this section we will present two major ones: *one-hot encoding of tokens*, and *token embeddings* (typically used exclusively for words, and called "*word embeddings*"). In the remainder of this section, we will explain these techniques and show concretely how to use them to go from raw text to a Numpy tensor that you can send to a Keras network.

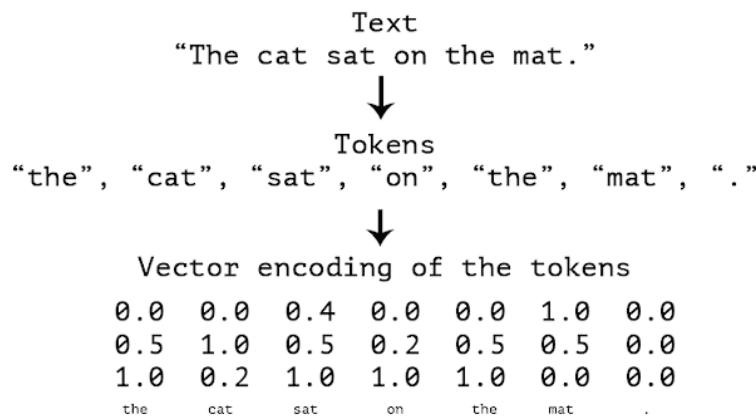


Figure 6.1 From text to tokens to vectors**NOTE****Understanding N-grams and "bag-of-words".**

Word N-grams are groups of N (or fewer) consecutive words that you can extract from a sentence. The same concept may also be applied to characters instead of words.

Here's a simple example. Consider the sentence: "*The cat sat on the mat*". It may be decomposed as the following set of 2-grams:

```
{"The", "The cat", "cat", "cat sat", "sat",  
 "sat on", "on", "on the", "the", "the mat", "mat"}
```

It may also be decomposed as the following set of 3-grams:

```
{"The", "The cat", "cat", "cat sat", "The cat sat",  
 "sat", "sat on", "on", "cat sat on", "on the", "the",  
 "sat on the", "the mat", "mat", "on the mat"}
```

Such a set is called a "bag-of-3-grams" (resp. 2-grams). The term "bag" here refers to the fact that we are dealing with a *set* of tokens rather than a list or sequence: the tokens have no specific order. This family of tokenization method is called "bag-of-words".

Because bag-of-words are not an order-preserving tokenization method (the tokens generated are understood as a set, not a sequence, and the general structure of the sentences is lost), bag-of-words tend to be used in shallow language processing models rather than in deep learning models. Extracting N-grams is a form of feature engineering, and deep learning does away with this kind of rigid and brittle feature engineering, replacing it with hierarchical feature learning. One-dimensional convnets and recurrent neural networks, introduced later in this chapter, are capable of learning representations for groups of words and characters without being explicitly told about the existence of such groups, simply by looking at continuous word or character sequences. For this reason, we will not be covering N-grams any further in this book. But do keep in mind that they are a powerful, unavoidable feature engineering tool when using lightweight shallow text processing models such as logistic regression and random forests.

6.1.1 One-hot encoding of words or characters

One-hot encoding is the most common, most basic way to turn a token into a vector. You already saw it in action in our initial IMDB and Reuters examples from chapter 3 (done with words, in our case). It consists in associating a unique integer index to every word, then turning this integer index i into a binary vector of size N , the size of the vocabulary, that would be all-zeros except for the i -th entry, which would be 1.

Of course, one-hot encoding can be done at the character level as well. To unambiguously drive home what one-hot encoding is and how to implement it, here are

two toy examples of one-hot encoding: one for words, the other for characters.

Listing 6.1 Word level one-hot encoding (toy example)

```
import numpy as np

# This is our initial data; one entry per "sample"
# (in this toy example, a "sample" is just a sentence, but
# it could be an entire document).
samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# First, build an index of all tokens in the data.
token_index = {}
for sample in samples:
    # We simply tokenize the samples via the `split` method.
    # in real life, we would also strip punctuation and special characters
    # from the samples.
    for word in sample.split():
        if word not in token_index:
            # Assign a unique index to each unique word
            token_index[word] = len(token_index) + 1
            # Note that we don't attribute index 0 to anything.

# Next, we vectorize our samples.
# We will only consider the first `max_length` words in each sample.
max_length = 10

# This is where we store our results:
results = np.zeros((len(samples), max_length, max(token_index.values()) + 1))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = token_index.get(word)
        results[i, j, index] = 1.
```

Listing 6.2 Character level one-hot encoding (toy example)

```
import string

samples = ['The cat sat on the mat.', 'The dog ate my homework.']
characters = string.printable # All printable ASCII characters.
token_index = dict(zip(range(1, len(characters) + 1), characters))

max_length = 50
results = np.zeros((len(samples), max_length, max(token_index.keys()) + 1))
for i, sample in enumerate(samples):
    for j, character in enumerate(sample):
        index = token_index.get(character)
        results[i, j, index] = 1.
```

Note that Keras has built-in utilities for doing one-hot encoding text at the word level or character level, starting from raw text data. This is what you should actually be using, as it will take care of a number of important features, such as stripping special characters from strings, or only taking into the top N most common words in your dataset (a common restriction to avoid dealing with very large input vector spaces).

Listing 6.3 Using Keras for word-level one-hot encoding

```
from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# We create a tokenizer, configured to only take
```

```
# into account the top-1000 most common words
tokenizer = Tokenizer(num_words=1000)
# This builds the word index
tokenizer.fit_on_texts(samples)

# This turns strings into lists of integer indices.
sequences = tokenizer.texts_to_sequences(samples)

# You could also directly get the one-hot binary representations.
# Note that other vectorization modes than one-hot encoding are supported!
one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')

# This is how you can recover the word index that was computed
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

A variant of one-hot encoding is the so-called "one-hot hashing trick", which can be used when the number of unique tokens in your vocabulary is too large to handle explicitly. Instead of explicitly assigning an index to each word and keeping a reference of these indices in a dictionary, one may hash words into vectors of fixed size. This is typically done with a very lightweight hashing function. The main advantage of this method is that it does away with maintaining an explicit word index, which saves memory and allows online encoding of the data (starting to generate token vectors right away, before having seen all of the available data). The one drawback of this method is that it is susceptible to "hash collisions": two different words may end up with the same hash, and subsequently any machine learning model looking at these hashes won't be able to tell the difference between these words. The likelihood of hash collisions decreases when the dimensionality of the hashing space is much larger than the total number of unique tokens being hashed.

Listing 6.4 Word-level one-hot encoding with hashing trick (toy example)

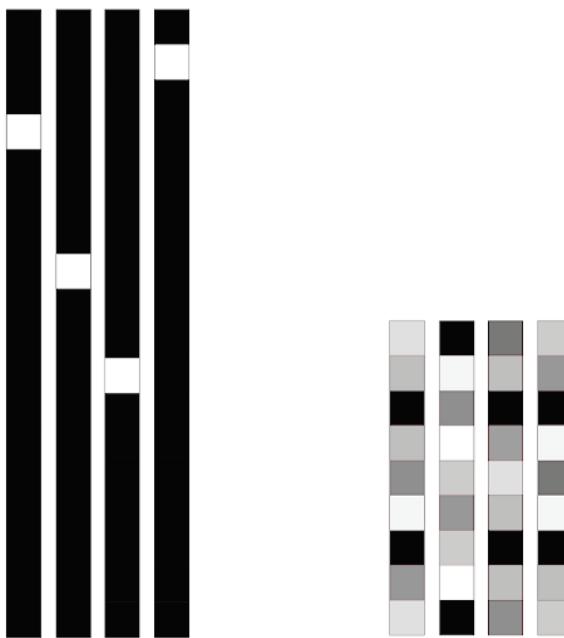
```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# We will store our words as vectors of size 1000.
# Note that if you have close to 1000 words (or more)
# you will start seeing many hash collisions, which
# will decrease the accuracy of this encoding method.
dimensionality = 1000
max_length = 10

results = np.zeros((len(samples), max_length, dimensionality))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        # Hash the word into a "random" integer index
        # that is between 0 and 1000
        index = abs(hash(word)) % dimensionality
        results[i, j, index] = 1.
```

6.1.2 Using word embeddings

Another popular and powerful way to associate a vector with a word is the use of dense "word vectors", also called "word embeddings". While the vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros) and very high-dimensional (same dimensionality as the number of words in the vocabulary), "word embeddings" are low-dimensional floating point vectors (i.e. "dense" vectors, as opposed to sparse vectors). Unlike word vectors obtained via one-hot encoding, word embeddings are learned from data. It is common to see word embeddings that are 256-dimensional, 512-dimensional, or 1024-dimensional when dealing with very large vocabularies. On the other hand, one-hot encoding words generally leads to vectors that are 20,000-dimensional or higher (capturing a vocabulary of 20,000 token in this case). So, word embeddings pack more information into far fewer dimensions.



One-hot word vectors: Word embeddings:

- Sparse
 - High-dimensional
 - Hard-coded
- Dense
 - Lower-dimensional
 - Learned from data

Figure 6.2 While word representations obtained from one-hot encoding or hashing are sparse, high-dimensional, and hard-coded, word embeddings are dense, relatively low-dimensional, and learned from data.

There are two ways to obtain word embeddings:

- Learn word embeddings jointly with the main task you care about (e.g. document classification or sentiment prediction). In this setup, you would start with random word vectors, then learn your word vectors in the same way that you learn the weights of a neural network.
- Load into your model word embeddings that were pre-computed using a different machine learning task than the one you are trying to solve. These are called "pre-trained word embeddings".

Let's take a look at both.

LEARNING WORD EMBEDDINGS WITH THE EMBEDDING LAYER

The simplest way to associate a dense vector to a word would be to pick the vector at random. The problem with this approach is that the resulting embedding space would have no structure: for instance, the words "accurate" and "exact" may end up with completely different embeddings, even though they are interchangeable in most sentences. It would be very difficult for a deep neural network to make sense of such a noisy, unstructured embedding space.

To get a bit more abstract: the geometric relationships between word vectors should reflect the semantic relationships between these words. Word embeddings are meant to map human language into a geometric space. For instance, in a reasonable embedding space, we would expect synonyms to be embedded into similar word vectors, and in general we would expect the geometric distance (e.g. L2 distance) between any two word vectors to relate to the semantic distance of the associated words (words meaning very different things would be embedded to points far away from each other, while related words would be closer). Even beyond mere distance, we may want specific *directions* in the embedding space to be meaningful. To make this clearer, let's look at a concrete example.

In figure 6.3, we embedded four words on a 2D plane, "cat", "dog", "wolf" and "tiger". With the vector representations we chose here, some semantic relationships between these words can be encoded as geometric transformations. For instance, a same vector allows to go from "cat" to "tiger" and from "dog" to "wolf": this vector could be interpreted as the "from pet to wild animal" vector. Similarly, another vector allows to go from "dog" to "cat" and from "wolf" to "tiger", which could be interpreted as a "from canine to feline" vector.

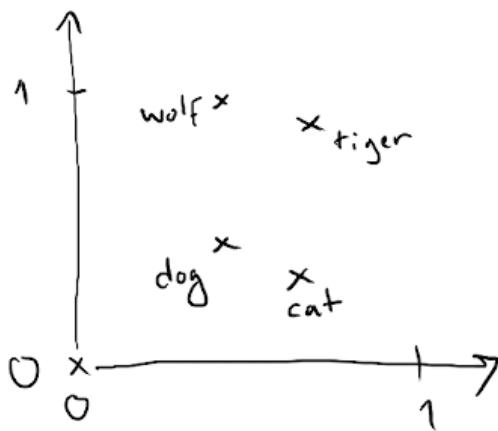


Figure 6.3 A toy example of a word embedding space

In real-world word embedding spaces, common examples of meaningful geometric transformations are "gender vectors" and "plural vector". For instance, by adding a "female vector" to the vector "king", one obtain the vector "queen". By adding a "plural

vector", one obtain "kings". Word embedding spaces typically feature thousands of such interpretable and potentially useful vectors.

Is there some "ideal" word embedding space that would perfectly map human language and could be used for any natural language processing task? Possibly, but in any case, we have yet to compute anything of the sort. Also, there isn't such a thing as "human language", there are many different languages and they are not isomorphic, as a language is the reflection of a specific culture and a specific context. But more pragmatically, what makes a good word embedding space depends heavily on your task: the perfect word embedding space for an English-language movie review sentiment analysis model may look very different from the perfect embedding space for an English-language legal document classification model, because the importance of certain semantic relationships varies from task to task.

It is thus reasonable to *learn* a new embedding space with every new task. Thankfully, backpropagation makes this really easy, and Keras makes it even easier. It's just about learning the weights a layer: the `Embedding` layer.

Listing 6.5 Instantiating an Embedding layer.

```
from keras.layers import Embedding

# The Embedding layer takes at least two arguments:
# the number of possible tokens, here 1000 (1 + maximum word index),
# and the dimensionality of the embeddings, here 64.
embedding_layer = Embedding(1000, 64)
```

The `Embedding` layer is best understood as a dictionary mapping integer indices (which stand for specific words) to dense vectors. It takes as input integers, it looks up these integers into an internal dictionary, and it returns the associated vectors. It's effectively a dictionary lookup.

```
word index -> Embedding layer -> corresponding word vector
```

The `Embedding` layer takes as input a 2D tensor of integers, of shape `(samples, sequence_length)`, where each entry is a sequence of integers. It can embed sequences of variable lengths, so for instance we could feed into our embedding layer above batches that could have shapes `(32, 10)` (batch of 32 sequences of length 10) or `(64, 15)` (batch of 64 sequences of length 15). All sequences in a batch must have the same length, though (since we need to pack them into a single tensor), so sequences that are shorter than others should be padded with zeros, and sequences that are longer should be truncated.

This layer returns a 3D floating point tensor, of shape `(samples, sequence_length, embedding_dimensionality)`. Such a 3D tensor can then be processed by a RNN layer or a 1D convolution layer (both will be introduced in the next sections).

When you instantiate an `Embedding` layer, its weights (its internal dictionary of token vectors) are initially random, just like with any other layer. During training, these word vectors will be gradually adjusted via backpropagation, structuring the space into something that the downstream model can exploit. Once fully trained, your embedding space will show a lot of structure—a kind of structure specialized for the specific problem you were training your model for.

Let's apply this idea to the IMDB movie review sentiment prediction task that you are already familiar with. With, let's quickly prepare the data. We will restrict the movie reviews to the top 10,000 most common words (like we did the first time we worked with this dataset), and cut the reviews after only 20 words. Our network will simply learn 8-dimensional embeddings for each of the 10,000 words, turn the input integer sequences (2D integer tensor) into embedded sequences (3D float tensor), flatten the tensor to 2D, and train a single `Dense` layer on top for classification.

Listing 6.6 Loading the IMDB data for use with an Embedding layer.

```
from keras.datasets import imdb
from keras import preprocessing

# Number of words to consider as features
max_features = 10000
# Cut texts after this number of words
# (among top max_features most common words)
 maxlen = 20

# Load the data as lists of integers.
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)

# This turns our lists of integers
# into a 2D integer tensor of shape `(samples, maxlen)`
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
```

Listing 6.7 Using an Embedding layer and classifier on the IMDB data.

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

model = Sequential()
# We specify the maximum input length to our Embedding layer
# so we can later flatten the embedded inputs
model.add(Embedding(10000, 8, input_length=maxlen))
# After the Embedding layer,
# our activations have shape `(samples, maxlen, 8)`.

# We flatten the 3D tensor of embeddings
# into a 2D tensor of shape `(samples, maxlen * 8)`
model.add(Flatten())

# We add the classifier on top
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
model.summary()

history = model.fit(x_train, y_train,
                      epochs=10,
                      batch_size=32,
                      validation_split=0.2)
```

We get to a validation accuracy of ~76%, which is pretty good considering that we are only look at the first 20 words in every review. But note that merely flattening the embedded sequences and training a single Dense layer on top leads to a model that treats each word in the input sequence separately, without considering inter-word relationships and structure sentence (e.g. it would likely treat both "*this movie is shit*" and "*this movie is the shit*" as being negative "reviews"). It would be much better to add recurrent layers or 1D convolutional layers on top of the embedded sequences to learn features that take into account each sequence as a whole. That's what we will focus on in the next few sections.

USING PRE-TRAINED WORD EMBEDDINGS

Sometimes, you have so little training data available that could never use your data alone to learn an appropriate task-specific embedding of your vocabulary. What to do then?

Instead of learning word embeddings jointly with the problem you want to solve, you could be loading embedding vectors from a pre-computed embedding space known to be highly structured and to exhibit useful properties—that captures generic aspects of language structure. The rationale behind using pre-trained word embeddings in natural language processing is very much the same as for using pre-trained convnets in image classification: we don't have enough data available to learn truly powerful features on our own, but we expect the features that we need to be fairly generic, i.e. common visual features or semantic features. In this case it makes sense to reuse features learned on a different problem.

Such word embeddings are generally computed using word occurrence statistics (observations about what words co-occur in sentences or documents), using a variety of techniques, some involving neural networks, others not. The idea of a dense, low-dimensional embedding space for words, computed in an unsupervised way, was initially explored by Bengio et al. in the early 2000s, but it only started really taking off in research and industry applications after the release of one of the most famous and successful word embedding scheme: the Word2Vec algorithm, developed by Mikolov at Google in 2013. Word2Vec dimensions capture specific semantic properties, e.g. gender.

There are various pre-computed databases of word embeddings that can download and start using in a Keras Embedding layer. Word2Vec is one of them. Another popular one is called "GloVe", developed by Stanford researchers in 2014. It stands for "Global Vectors for Word Representation", and it is an embedding technique based on factorizing a matrix of word co-occurrence statistics. Its developers have made available pre-computed embeddings for millions of English tokens, obtained from Wikipedia data or from Common Crawl data.

Let's take a look at how you can get started using GloVe embeddings in a Keras model. The same method will of course be valid for Word2Vec embeddings or any other word embedding database that you can download. We will also use this example to refresh the text tokenization techniques we introduced a few paragraphs ago: we will start from raw text, and work our way up.

6.1.3 Putting it all together: from raw text to word embeddings

We will be using a model similar to the one we just went over—embedding sentences in sequences of vectors, flattening them and training a Dense layer on top. But we will do it using pre-trained word embeddings, and instead of using the pre-tokenized IMDB data packaged in Keras, we will start from scratch, by downloading the original text data.

DOWNLOAD THE IMDB DATA AS RAW TEXT

First, head to ai.stanford.edu/~amaas/data/sentiment/ and download the raw IMDB dataset (if the URL isn't working anymore, just Google "IMDB dataset"). Uncompress it.

Now let's collect the individual training reviews into a list of strings, one string per review, and let's also collect the review labels (positive / negative) into a labels list:

Listing 6.8 Processing the labels of the raw IMDB data

```
import os

imdb_dir = '/Users/fchollet/Downloads/aclImdb'
train_dir = os.path.join(imdb_dir, 'train')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

TOKENIZE THE DATA

Let's vectorize the texts we collected, and prepare a training and validation split. We will merely be using the concepts we introduced earlier in this section.

Because pre-trained word embeddings are meant to be particularly useful on problems where little training data is available (otherwise, task-specific embeddings are likely to outperform them), we will add the following twist: we restrict the training data to its first 200 samples. So we will be learning to classify movie reviews after looking at just 200 examples...

Listing 6.9 Tokenizing the text of the raw IMDB data

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100 # We will cut reviews after 100 words
training_samples = 200 # We will be training on 200 samples
validation_samples = 10000 # We will be validating on 10000 samples
```

```

max_words = 10000 # We will only consider the top 10,000 words in the dataset

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

# Split the data into a training set and a validation set
# But first, shuffle the data, since we started from data
# where sample are ordered (all negative first, then all positive).
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]

```

DOWNLOAD THE GLOVE WORD EMBEDDINGS

Head to nlp.stanford.edu/projects/glove/ (where you can learn more about the GloVe algorithm), and download the pre-computed embeddings from 2014 English Wikipedia. It's a 822MB zip file named `glove.6B.zip`, containing 100-dimensional embedding vectors for 400,000 words (or non-word tokens). Un-zip it.

PRE-PROCESS THE EMBEDDINGS

Let's parse the un-zipped file (it's a `txt` file) to build an index mapping words (as strings) to their vector representation (as number vectors).

Listing 6.10 Parsing the GloVe word embeddings file

```

glove_dir = '/Users/fchollet/Downloads/glove.6B'

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))

```

Now let's build an embedding matrix that we will be able to load into an `Embedding` layer. It must be a matrix of shape `(max_words, embedding_dim)`, where each entry `i` contains the `embedding_dim`-dimensional vector for the word of index `i` in our reference word index (built during tokenization). Note that the index `0` is not supposed to stand for any word or token—it's a placeholder.

Listing 6.11 Preparing the GloVe word embeddings matrix

```

embedding_dim = 100

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if i < max_words:
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zeros.
            embedding_matrix[i] = embedding_vector

```

DEFINE A MODEL

We will be using the same model architecture as before:

Listing 6.12 Model definition

```

from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

```

LOAD THE GLOVE EMBEDDINGS IN THE MODEL

The `Embedding` layer has a single weight matrix: a 2D float matrix where each entry `i` is the word vector meant to be associated with index `i`. Simple enough. Let's just load the GloVe matrix we prepared into our `Embedding` layer, the first layer in our model:

Listing 6.13 Loading the matrix of pre-trained word embeddings into the Embedding layer

```

model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False

```

Additionally, we freeze the embedding layer (we set its `trainable` attribute to `False`), following the same rationale as what you are already familiar with in the context of pre-trained convnet features: when parts of a model are pre-trained (like our `Embedding` layer), and parts are randomly initialized (like our classifier), the pre-trained parts should not be updated during training to avoid forgetting what they already know. The large gradient update triggered by the randomly initialized layers would be very disruptive to the already learned features.

TRAIN AND EVALUATE

Let's compile our model and train it:

Listing 6.14 Training and evaluation

```

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                     epochs=10,
                     batch_size=32,
                     validation_data=(x_val, y_val))
model.save_weights('pre_trained_glove_model.h5')

```

Let's plot its performance over time:

Listing 6.15 Plotting results

```

import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()

```

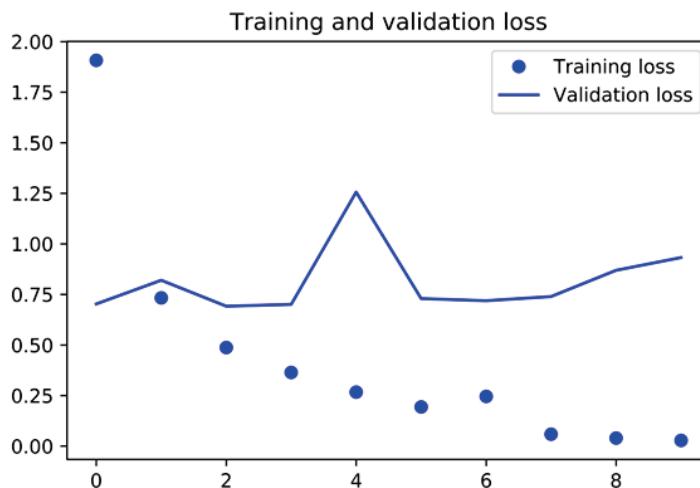


Figure 6.4 Training and validation loss when using pre-trained word embeddings

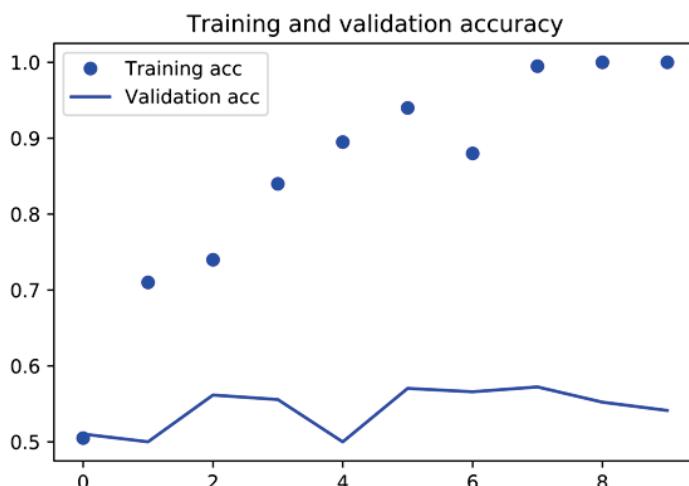


Figure 6.5 Training and validation accuracy when using pre-trained word embeddings

The model quickly starts overfitting, unsurprisingly given the small number of training samples. Validation accuracy has high variance for the same reason, but seems to reach high 50s.

Note that your mileage may vary: since we have so few training samples, performance is heavily dependent on which exact 200 samples we picked, and we picked them at random. If it worked really poorly for you, try picking a different random set of 200 samples, just for the sake of the exercise (in real life you don't get to pick your training data).

We can also try to train the same model without loading the pre-trained word embeddings and without freezing the embedding layer. In that case, we would be learning a task-specific embedding of our input tokens, which is generally more powerful than pre-trained word embeddings when lots of data is available. However, in our case, we have only 200 training samples. Let's try it:

Listing 6.16 Defining a training the same model without pre-trained word embeddings

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                      epochs=10,
                      batch_size=32,
                      validation_data=(x_val, y_val))
```

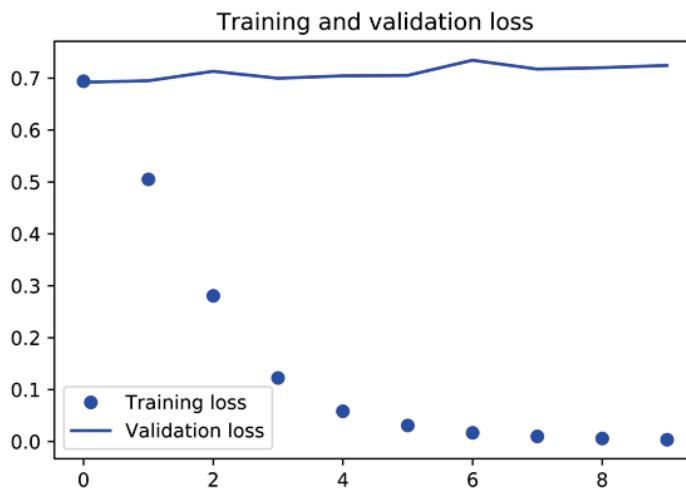


Figure 6.6 Training and validation loss without using pre-trained word embeddings

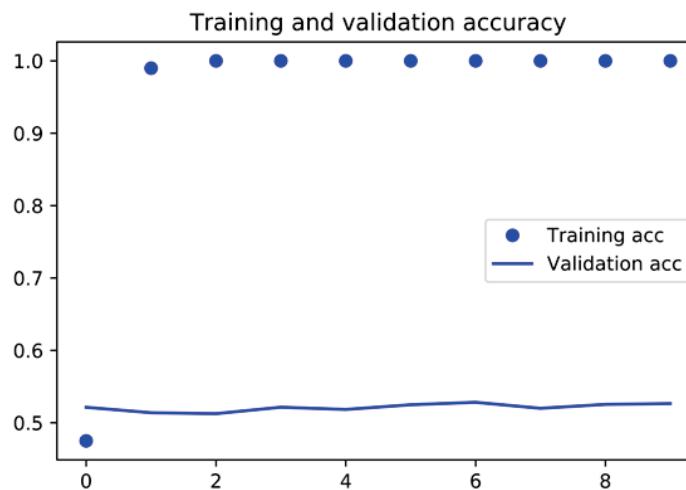


Figure 6.7 Training and validation accuracy without using pre-trained word embeddings

Validation accuracy stalls in the low 50s. So in our case, pre-trained word embeddings does outperform jointly learned embeddings. If you increase the number of training samples, this will quickly stop being the case—try it as an exercise.

Finally, let's evaluate the model on the test data. First, we will need to tokenize the test data:

Listing 6.17 Tokenizing the data of the test set

```
test_dir = os.path.join(imdb_dir, 'test')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(test_dir, label_type)
    for fname in sorted(os.listdir(dir_name)):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
```

```

    labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)

```

And let's load and evaluate the first model:

Listing 6.18 Evaluating the model on the test set

```

model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)

```

We get an appalling test accuracy of 56%. Working with just a handful of training samples is hard!

6.1.4 Wrapping up

Wrapping up: now you are be able to...

- Turn raw text into something that a neural network can process.
- Use the `Embedding` layer in a Keras model to learn task-specific token embeddings.
- Leverage pre-trained word embeddings to get an extra boost on small natural language processing problems.

6.2 Understanding recurrent neural networks

A major characteristic of all neural networks that you have seen so far, such as densely-connected networks and convnets, is that they had no memory. Each input shown to them gets processed independently, with no state kept in between inputs. With such networks, in order to process a sequence or a temporal series of data points, you have to show the entire sequence to the network at once, i.e. turn it into a single datapoint. For instance, this is what we have been doing in our IMDB example: an entire movie review would get transformed into a single large vector, and would be processed in one go. Such networks are called "feedforward networks".

By contrast, as you are reading the present sentence, you are processing it word by word, or rather, eye saccade by eye saccade, while keeping around memories of what came before—a fluid representation of the meaning that I am conveying with this sentence. Biological intelligence processes information incrementally while maintaining an internal model of what it is processing, built from past information and constantly getting updated as new information comes in.

Recurrent Neural Networks (RNNs) adopt the same principle, albeit in an extremely simplified version: they process sequences by iterating through the sequence elements and maintaining a "state" containing information relative to what they have seen so far. In effect, RNNs are a type of neural network that has an internal loop (Figure 6.8). The state of the RNN is reset in-between processing two different, independent sequences

(e.g. two different IMDB reviews), so we still consider one sequence as a single datapoint, a single input to the network—what changes is that this datapoint is no longer processed in a single step, rather, the network internally loops over sequence elements.

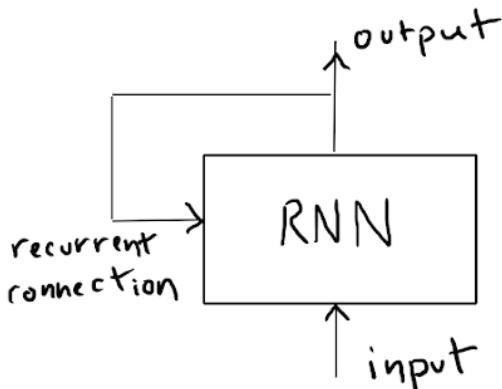


Figure 6.8 A recurrent network: a network with a loop.

To make these notions of loop and state completely clear, let's implement the forward pass of a toy RNN in Numpy. This RNN takes as input a sequence of vectors, which we will encode as a 2D tensor of size `(timesteps, input_features)`. It loops over timesteps, and at each timestep, it considers its current state at t , the input at t (of shape `(input_features,)`), and combines them to obtain the output at t . We then set the state for the next step to simply be this previous output. For the very first timestep, the "previous output" is not defined, hence there is no "current state", so we will initialize the state as an all-zero vector, called the "initial state" of the network.

In pseudo code, this is our RNN:

Listing 6.19 A pseudo-code simple RNN

```

state_t = 0 # This is the state at t.
for input_t in input_sequence: # We iterate over sequence elements.
    output_t = f(input_t, state_t) # `f` is our "step function"
    state_t = output_t # The previous output becomes the new state.
  
```

We can even flesh out a bit the function `f`: the transformation of the input and state into an output will be parametrized by two matrices, W and U , and a bias vector. It is very similar to the transformation operated by a densely connected layer in a feedforward network.

Listing 6.20 More detailed pseudo-code simple RNN

```

state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
  
```

To make these notions absolutely unambiguous, let's go ahead and write down a naive Numpy implementation of the forward pass of our simple RNN.

Listing 6.21 Numpy implementation of a simple RNN

```

timesteps = 100 # Number of timesteps in the input sequence
inputs_features = 32 # Dimensionality of the input feature space
output_features = 64 # Dimensionality of the output feature space

# This is our input data - just random noise for the sake of our example.
inputs = np.random.random((timesteps, features))

# This is our "initial state": an all-zero vector.
state_t = np.zeros((output_features,))

# Create random weight matrices
W = np.random.random((input_features, output_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))

successive_outputs = []
for input_t in inputs: # input_t is a vector of shape (input_features,)
    # We combine the input with the current state
    # (i.e. the previous output) to obtain the current output.
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)

    # We store this output in a list.
    successive_outputs.append(output_t)

    # We update the "state" of the network for the next timestep
    state_t = output_t

# The final output is a 2D tensor of shape (timesteps, output_features).
final_output_sequence = np.concatenate(successive_outputs, axis=0)

```

Easy enough: in summary, a RNN is just a *for loop* that reuses quantities computed during the previous iteration of the loop. Nothing more. Of course, there are many different RNNs fitting this definition that one could build—the example we just showed is one of the simplest RNN formulations out there. RNNs are characterized by their "step function", e.g. in our case, the function:

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```

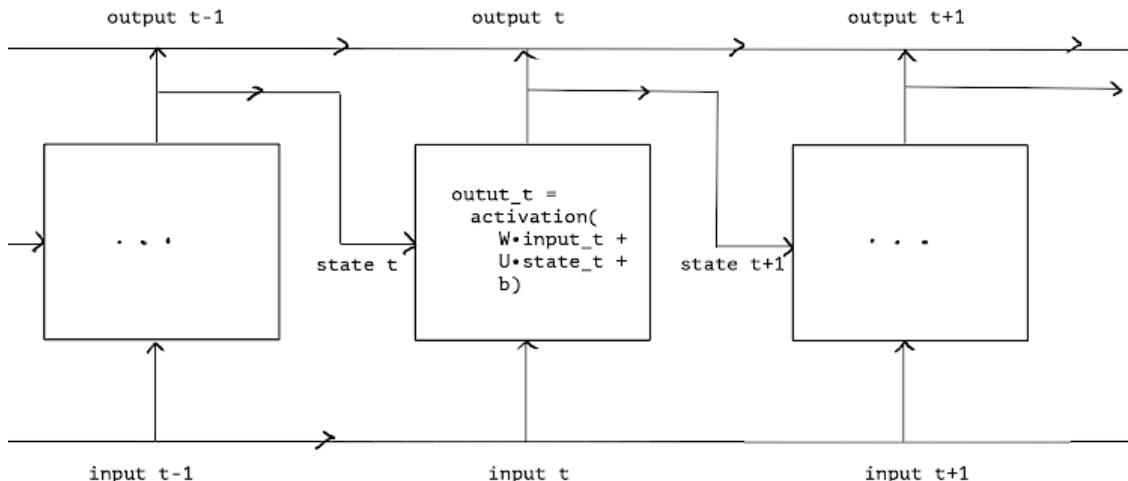


Figure 6.9 A simple RNN, unrolled over time.

Note: in our example, the final output is a 2D tensor of shape (timesteps,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

`output_features`), where each timestep is the output of the loop at time t . Each timestep t in the output tensor contains information about timesteps 0 to t in the input sequence—about the entire past. For this reason, in many cases you don’t need this full sequence of outputs, you just need the very last output (`output_t` at the end of the loop), since it already contains information about the entire sequence.

6.2.1 A first recurrent layer in Keras

The process we just naively implemented in Numpy corresponds to an actual Keras layer: the `SimpleRNN` layer:

Listing 6.22 The Keras SimpleRNN layer

```
from keras.layers import SimpleRNN
```

There is just one minor difference: `SimpleRNN` processes batches of sequences, like all other Keras layers, not just a single sequence like in our Numpy example. This means that it takes inputs of shape `(batch_size, timesteps, input_features)`, rather than `(timesteps, input_features)`.

Like all recurrent layers in Keras, `SimpleRNN` can be run in two different modes: it can return either the full sequences of successive outputs for each timestep (a 3D tensor of shape `(batch_size, timesteps, output_features)`), or it can return only the last output for each input sequence (a 2D tensor of shape `(batch_size, output_features)`). These two modes are controlled by the `return_sequences` constructor argument. Let’s take a look at an example:

Listing 6.23 Using SimpleRNN and returning the last state

```
>>> from keras.models import Sequential
>>> from keras.layers import Embedding, SimpleRNN
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32))
>>> model.summary()

Layer (type)                 Output Shape              Param #
=====
embedding_22 (Embedding)      (None, None, 32)        320000
=====
simplernn_10 (SimpleRNN)     (None, 32)                2080
=====
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

Listing 6.24 Using SimpleRNN and returning the full state sequence

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.summary()
```

Layer (type)	Output Shape	Param #
embedding_23 (Embedding)	(None, None, 32)	320000
simpleRNN_11 (SimpleRNN)	(None, None, 32)	2080
Total params:	322,080	
Trainable params:	322,080	
Non-trainable params:	0	

It is sometimes useful to stack several recurrent layers one after the other in order to increase the representational power of a network. In such a setup, you have to get all intermediate layers to return full sequences:

Listing 6.25 Stacking RNN layers on top of each other

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32)) # This last layer only returns the last outputs.
>>> model.summary()

Layer (type)          Output Shape         Param #
=====
embedding_24 (Embedding)    (None, None, 32)      320000
simpleRNN_12 (SimpleRNN)   (None, None, 32)      2080
simpleRNN_13 (SimpleRNN)   (None, None, 32)      2080
simpleRNN_14 (SimpleRNN)   (None, None, 32)      2080
simpleRNN_15 (SimpleRNN)   (None, 32)            2080
=====
Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0
```

Now let's try to use such a model on the IMDB movie review classification problem. First, let's preprocess the data:

Listing 6.26 Preparing the IMDB data

```
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 10000 # number of words to consider as features
 maxlen = 500 # cut texts after this number of words (among top max_features most common words)
 batch_size = 32

print('Loading data...')
(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_features)
print(len(input_train), 'train sequences')
print(len(input_test), 'test sequences')

print('Pad sequences (samples x time)')
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train shape:', input_train.shape)
print('input_test shape:', input_test.shape)
```

Let's train a simple recurrent network using an Embedding layer and a SimpleRNN layer:

Listing 6.27 Training our model including an Embedding layer and a SimpleRNN layer

```
from keras.layers import Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(input_train, y_train,
                      epochs=10,
                      batch_size=128,
                      validation_split=0.2)
```

Let's display the training and validation loss and accuracy:

Listing 6.28 Plotting results

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

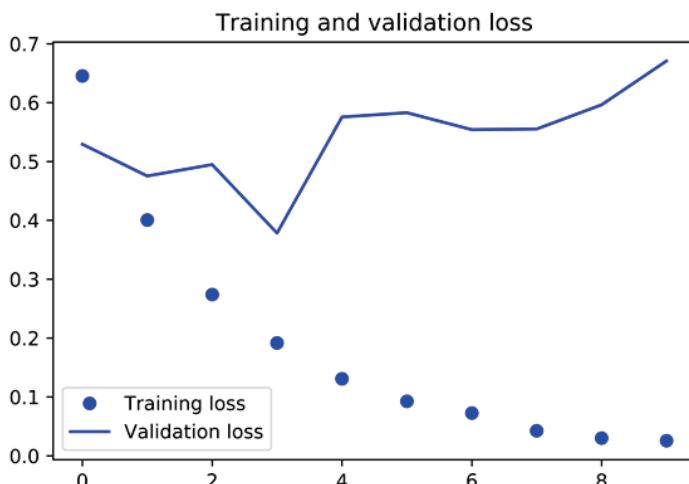
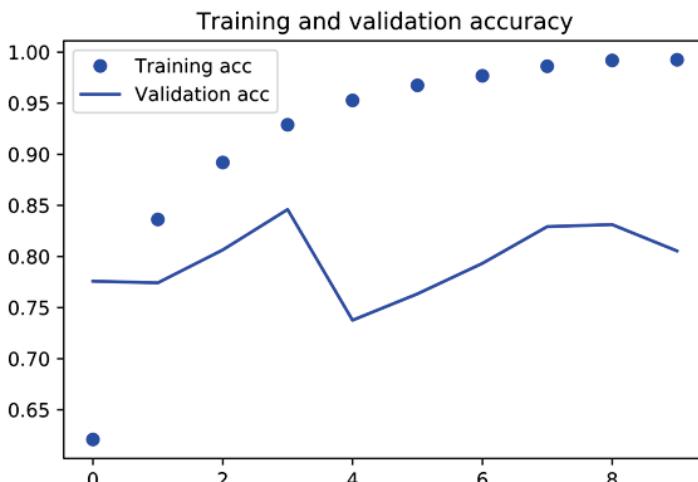


Figure 6.10 Training and validation loss on IMDB with SimpleRNN.**Figure 6.11 Training and validation accuracy on IMDB with SimpleRNN.**

As a reminder, in chapter 3, our very first naive approach to this very dataset got us to 88% test accuracy. Unfortunately, our small recurrent network doesn't perform very well at all compared to this baseline (only up to 85% validation accuracy). Part of the problem is that our inputs only consider the first 500 words rather than the full sequences—hence our RNN has access to less information than our earlier baseline model. The remainder of the problem is simply that `SimpleRNN` isn't very good at processing long sequences, like text. Other types of recurrent layers perform much better. Let's take a look at some more advanced layers.

6.2.2 Understanding the LSTM and GRU layers

`SimpleRNN` isn't the only recurrent layer available in Keras: there are two more, `LSTM` and `GRU`. In practice, you will always be using one of these two, as `SimpleRNN` is generally too simplistic to be of any real use. Indeed, `SimpleRNN` has a major issue: albeit it should theoretically be able to retain at time t information about inputs seen many timesteps before, in practice such long-term dependencies prove to be impossible to learn. This is due to the "vanishing gradient problem", an effect that is similar to what can be observed with non-recurrent networks (feedforward networks) that are many layers deep: as one keeps adding layers to a network, the network eventually becomes un-trainable. The theoretical reasons for this effect have been studied by Hochreiter, Schmidhuber, and Bengio in the early 1990s. The `LSTM` and `GRU` layers are designed to solve this very problem.

Let's consider the `LSTM` layer. The abbreviation stands for "Long-Short Term Memory". The underlying algorithm was developed by Hochreiter and Schmidhuber in 1997, the culmination of their research on the vanishing gradient problem.

It is a variant of the simple RNN you already know about, that adds a way to carry information across many timesteps. Imagine a conveyor belt running parallel to the sequence we are processing. Information from the sequence can jump on the conveyor

belt on at any point, get transported to a later timestep, and jump off, intact, when we need it. This is essentially what LSTM does: it saves information for later, thus preventing older signals to gradually vanish during processing.

To understand it detail, let's start from our simple RNN cell. Because we are going to have a lot of weight matrices, we will index our w and u matrices in the cell with the letter o (w_o and u_o). It's for "output".

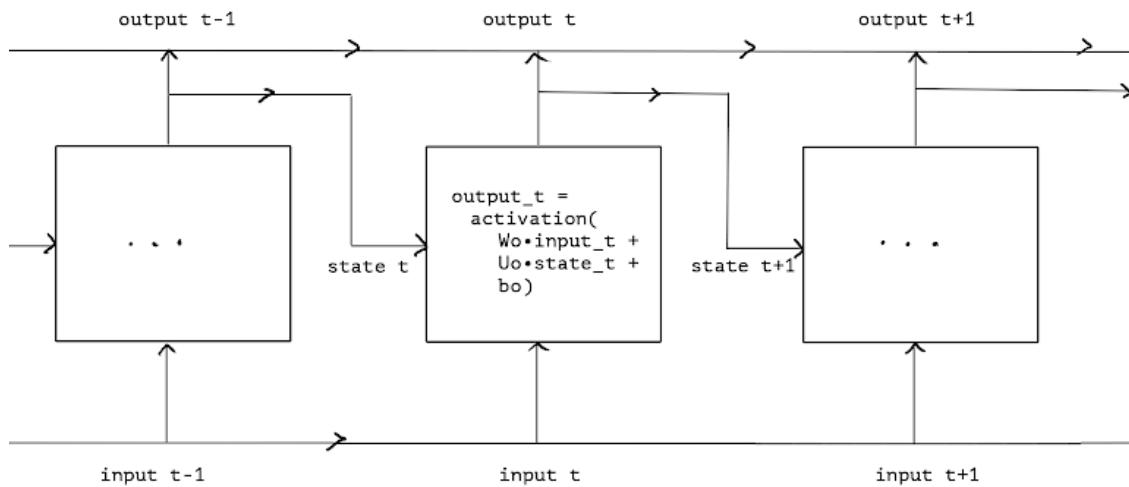


Figure 6.12 The starting point of a LSTM layer: a simple RNN.

Let's literally add to this picture an additional data flow that carries information across timesteps. We'll call its values at different timesteps c_t , c standing for "carry". This information will have the following impact on the cell: it will get combined with the input connection and the recurrent connection (via a dense transformation, i.e. a dot product with a weight matrix followed by a bias add and the application of an activation function), and it affects the state being sent to the next timestep (via an activation function and a multiplication operation). Conceptually, our "carry" dataflow is simply a way to modulate the next output and the next state. Super simple so far.

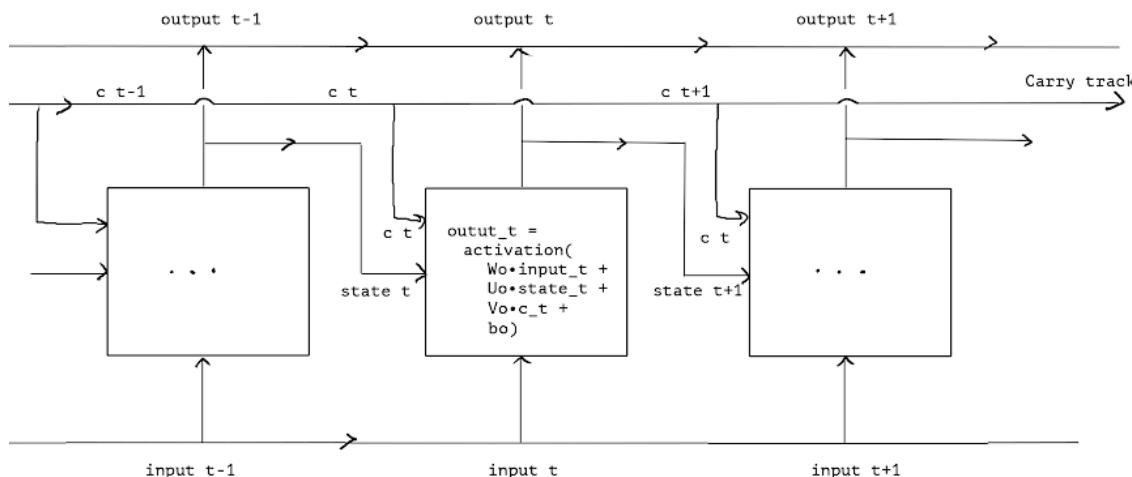


Figure 6.13 Going from a simple RNN to a LSTM: adding a "carry" track.

Now the subtlety: the way the next value of the carry dataflow is computed. It involves 3 distinct transformations. All three have the form of a simple RNN cell, i.e.:

```
y = activation(dot(state_t, U) + dot(input_t, W) + b)
```

But all three transformations have their own weight matrices, which we will index by the letters i , f , and k . Let's write down what we have so far (it may seem a bit arbitrary, but bear with me):

Listing 6.29 Pseudo-code details of the LSTM architecture (1/2)

```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(c_t, Vo) + bo)

i_t = activation(dot(state_t,Ui) + dot(input_t,Wi) + bi)
f_t = activation(dot(state_t,Uf) + dot(input_t,Wf) + bf)
k_t = activation(dot(state_t,Uk) + dot(input_t,Wk) + bk)
```

We obtain the new carry state (the next c_{t+1}) by simply combining i_t , f_t and k_t :

Listing 6.30 Pseudo-code details of the LSTM architecture (2/2)

```
c_{t+1} = i_t * k_t + c_t * f_t
```

Let's add this to our figure:

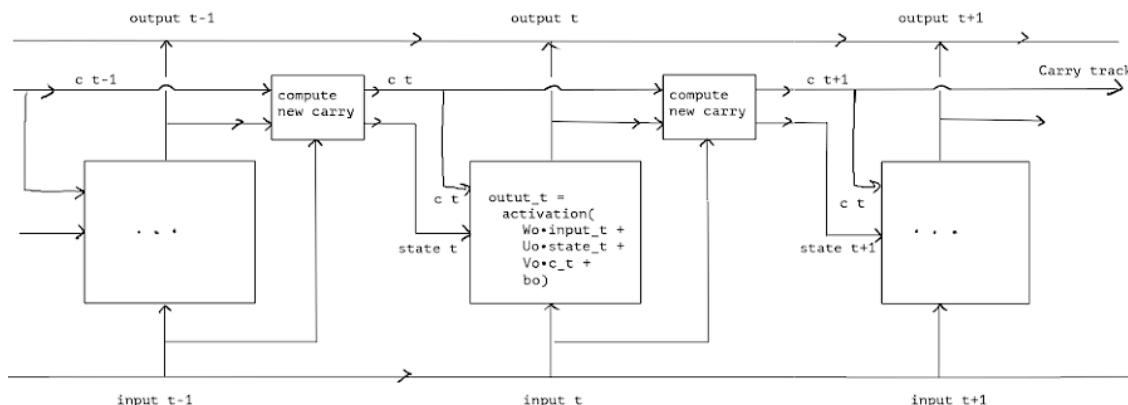


Figure 6.14 Anatomy of a LSTM.

That's it. Not so complicated after all, merely a tad complex.

Now if we want to get all philosophical, we can start interpreting what each of these operations are "meant" to do. For instance, one might say that multiplying c_t with f_t is a way to deliberately "forget" some irrelevant information in the carry dataflow. Meanwhile i_t and k_t provide information about the present, updating the carry track with new information. However, at the end of the day, these interpretations may not mean much, because what these operations *actually* do is determined by contents of the weights parametrizing them, and the weights are learned in an end-to-end fashion, anew with each training round, making it impossible to credit this or that operation with a

specific purpose. The specification of a RNN cell (what we just described above) determines your hypothesis space—the space in which you will search for a good model configuration during training—but it does not determine what the cell does; that is up to the cell weights. A same cell with different weights can be doing very different things. So the combination of operations making up a RNN cell is a better interpreted as set of *constraints* on your search, not as a *design* in an engineering sense.

As a researcher, it seems to me that the choice of such constraints—the question of how to implement RNN cells—is better left to optimization algorithms (like genetic algorithms or reinforcement learning processes) than to human engineers. And in the future, that's how we will build our networks anyway. In summary: you don't need to understand anything about the specific architecture of a LSTM cell; as a human, it shouldn't be your job to understand it. Just keep in mind what the LSTM cell is meant to do: allowing past information to be reinjected at a later time, thus fighting the vanishing gradient problem.

6.2.3 A concrete LSTM example in Keras

Now let's switch to more practical concerns: we will set up a model using a LSTM layer and train it on the IMDB data. Here's the network, similar to the one with SimpleRNN that we just presented. We only specify the output dimensionality of the LSTM layer, and leave every other argument (there are lots) to the Keras defaults. Keras has good defaults, and things will almost always "just work" without you having to spend time tuning parameters by hand.

Listing 6.31 Using the LSTM layer in Keras

```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(input_train, y_train,
                      epochs=10,
                      batch_size=128,
                      validation_split=0.2)
```

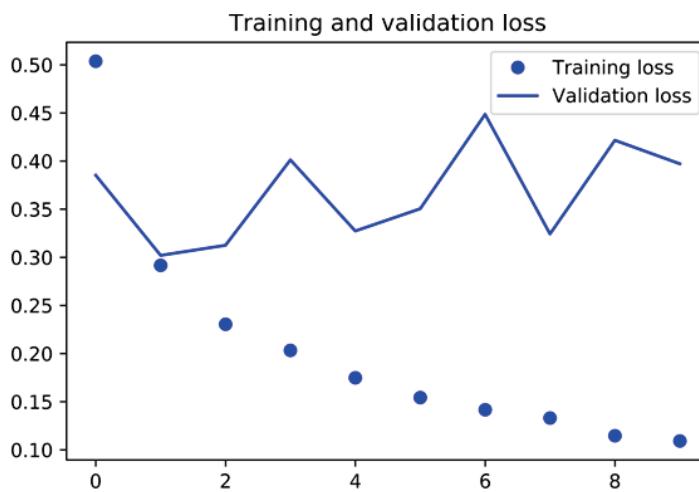


Figure 6.15 Training and validation loss on IMDB with LSTM.

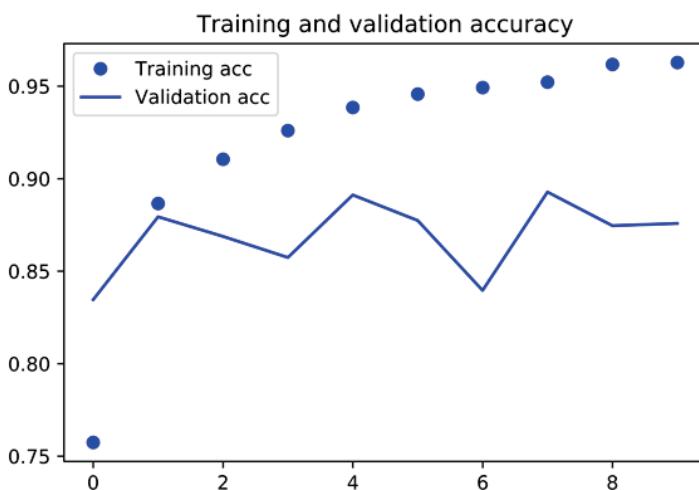


Figure 6.16 Training and validation accuracy on IMDB with LSTM.

Here's what we are getting this time: up to 89% validation accuracy. Not bad: certainly much better than the SimpleRNN network—that's largely because LSTM suffers much less from the vanishing gradient problem—and even slightly better than our fully-connected approach from chapter 3, even though we are looking at less data than we were in chapter 3—we are truncating sequences after 500 timesteps, whereas in chapter 3 we were considering full sequences.

However, it's not groundbreaking either for such a computationally intensive approach. Why isn't our LSTM performing better? One reason is that we did no effort to tune hyperparameters such as the embeddings dimensionality or the LSTM output dimensionality. Another may be lack of regularization. But honestly, the reason is mostly that analysing the global, long-term structure of the reviews (what LSTM is really good at) isn't very helpful for a sentiment analysis problem. Such a basic problem is very well solved by simply looking at what words occur in each review, and at what frequency. That's what our first fully-connected approach was looking at. But there are far more difficult natural language processing problems out there, where the strength of LSTM will become apparent: in particular, question answering and machine translation.

6.2.4 Wrapping up

To wrap up—now you understand:

- What RNNs are, and how they work.
- What LSTM is, and why it works better on long sequences than a naive RNN.
- How to use Keras RNN layers to process sequence data.

Next, we will review a number of more advanced features of RNNs, to get the most out of your deep learning sequence models.

6.3 Advanced usage of recurrent neural networks

In this section, we will review three advanced techniques for improving the performance and generalization power of recurrent neural networks. By the end of the section, you will know most of what there is to know about using recurrent networks with Keras. We will demonstrate all three concepts on a weather forecasting problem, where we have access to a timeseries of data points coming from sensors installed on the roof of a building, such as temperature, air pressure, and humidity, which we use to predict what the temperature will be 24 hours after the last data point collected. This is a fairly challenging problem that exemplifies many common difficulties encountered when working with timeseries.

We will cover the following techniques:

- *Recurrent dropout*, a specific, built-in way to use dropout to fight overfitting in recurrent layers.
- *Stacking recurrent layers*, to increase the representational power of the network (at the cost of higher computational loads).
- *Bidirectional recurrent layers*, which presents the same information to a recurrent network in different ways, increasing accuracy and mitigating forgetting issues.

6.3.1 A temperature forecasting problem

Until now, the only sequence data we have covered has been text data, for instance the IMDB dataset and the Reuters dataset. But sequence data is found in many more problems than just language processing. In all of our examples in this section, will be playing with a weather timeseries dataset recorded at the Weather Station at the Max-Planck-Institute for Biogeochemistry in Jena, Germany.¹.

Footnote 1 Olaf Kolle - www.bgc-jena.mpg.de/wetter/

In this dataset, fourteen different quantities (such air temperature, atmospheric pressure, humidity, wind direction, etc.) are recorded every ten minutes, over several years. The original data goes back to 2003, but we limit ourselves to data from 2009-2016. This dataset is perfect for learning to work with numerical timeseries. We will use it to build a model that takes as input some data from the recent past (a few days worth of data points) and predicts the air temperature 24 hours in the future.

The data can be downloaded and uncompressed via e.g.:

Listing 6.32 Downloading the Jena weather dataset

```
cd ~/Downloads
mkdir jena_climate
cd jena_climate
wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
unzip jena_climate_2009_2016.csv.zip
```

Let's take a look at the data:

Listing 6.33 Inspecting the data of the Jena weather dataset

```
import os

data_dir = '/users/fchollet/Downloads/jena_climate'
fname = os.path.join(data_dir, 'jena_climate_2009_2016.csv')

f = open(fname)
data = f.read()
f.close()

lines = data.split('\n')
header = lines[0].split(',')
lines = lines[1:]

print(header)
print(len(lines))
```

This outputs a count of 420,551 lines of data (each line is a timestep, i.e. a record of a date and 14 weather-related values), as well as the following header:

Listing 6.34 The header of the Jena weather data

```
["Date Time",
 "p (mbar)",
 "T (degC)",
 "Tpot (K)",
 "Tdew (degC)",
 "rh (%)",
 "VPmax (mbar)",
 "VPact (mbar)",
 "VPdef (mbar)",
 "sh (g/kg)",
 "H2OC (mmol/mol)",
 "rho (g/m**3)",
 "wv (m/s)",
 "max. wv (m/s)",
 "wd (deg)"]
```

Let's convert all of these 420,551 lines of data into a Numpy array:

Listing 6.35 Parsing the data

```
import numpy as np

float_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(',') [1:]]
    float_data[i, :] = values
```

For instance, here is the plot of temperature (in degrees Celsius) over time:

Listing 6.36 Plotting the temperature timeseries

```
from matplotlib import pyplot as plt

temp = float_data[:, 1] # temperature (in degrees Celsius)
plt.plot(range(len(temp)), temp)
```

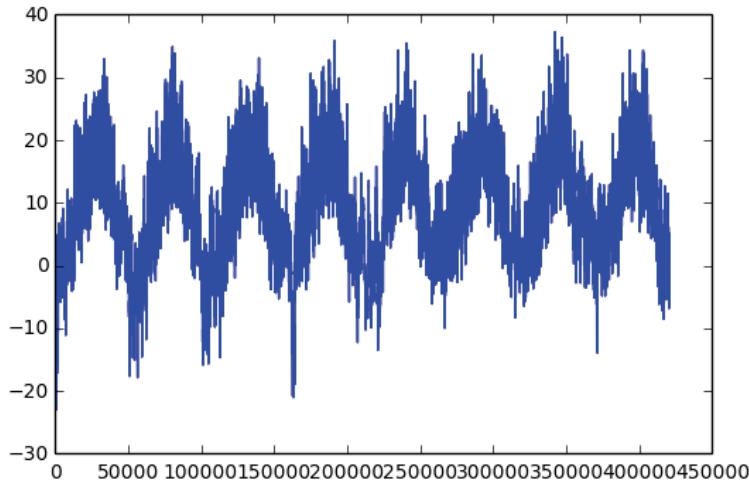


Figure 6.17 Temperature over the full temporal range of the dataset (oC).

On this plot, you can clearly see the yearly periodicity of temperature.

Here is a more narrow plot of the first ten days of temperature data (since the data is recorded every ten minutes, we get 144 data points per day):

Listing 6.37 Plotting the first ten days of the temperature timeseries

```
plt.plot(range(1440), temp[:1440])
```

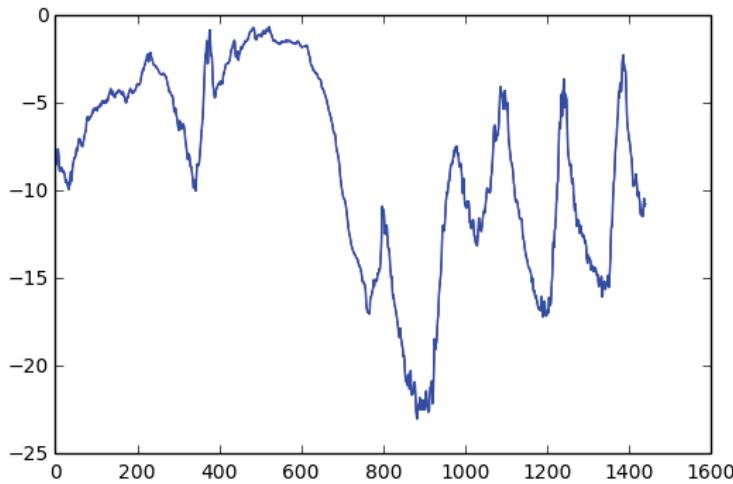


Figure 6.18 Temperature over the first ten days of the dataset (oC).

On this plot, you can see daily periodicity, especially evident for the last 4 days. We can also note that this ten-days period must be coming from a fairly cold winter month.

If we were trying to predict average temperature for the next month given a few month of past data, the problem would be easy, due to the reliable year-scale periodicity of the data. But looking at the data over a scale of days, the temperature looks a lot more chaotic. So is this timeseries predictable at a daily scale? Let's find out.

6.3.2 Preparing the data

The exact formulation of our problem will be the following: given data going as far back as `lookback` timesteps (a timestep is 10 minutes) and sampled every `steps` timesteps, can we predict the temperature in `delay` timesteps?

We will use the following parameter values:

- `lookback` = 720, i.e. our observations will go back 5 days.
- `steps` = 6, i.e. our observations will be sampled at one data point per hour.
- `delay` = 144, i.e. our targets will be 24 hour in the future.

To get started, we need to do two things:

- Preprocess the data to a format a neural network can ingest. This is easy: the data is already numerical, so we don't need to do any vectorization. However each timeseries in the data is on a different scale (e.g. temperature is typically between -20 and +30, but pressure, measured in mbar, is around 1000). So we will normalize each timeseries independently so that they all take small values on a similar scale.
- Write a Python generator that takes our current array of float data and yields batches of data from the recent past, alongside with a target temperature in the future. Since the samples in our dataset are highly redundant (e.g. sample N and sample $N + 1$ will have most of their timesteps in common), it would be very wasteful to explicitly allocate every sample. Instead, we will generate the samples on the fly using the original data.

We preprocess the data by subtracting the mean of each timeseries and dividing by the standard deviation. We plan on using the first 200,000 timesteps as training data, so we compute the mean and standard deviation only on this fraction of the data:

Listing 6.38 Normalizing the data

```
mean = float_data[:200000].mean(axis=0)
float_data -= mean
std = float_data[:200000].std(axis=0)
float_data /= std
```

Now here is the data generator that we will use. It yields a tuple `(samples, targets)` where `samples` is one batch of input data and `targets` is the corresponding array of target temperatures. It takes the following arguments:

- `data`: The original array of floating point data, which we just normalized in the code snippet above.
- `lookback`: How many timesteps back should our input data go.
- `delay`: How many timesteps in the future should our target be.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

- `min_index` and `max_index`: Indices in the `data` array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another one for testing.
- `shuffle`: Whether to shuffle our samples or draw them in chronological order.
- `batch_size`: The number of samples per batch.
- `step`: The period, in timesteps, at which we sample data. We will set it 6 in order to draw one data point every hour.

Listing 6.39 Generator yielding timeseries samples and their targets

```
def generator(data, lookback, delay, min_index, max_index,
              shuffle=False, batch_size=128, step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(
                min_index + lookback, max_index, size=batch_size)
        else:
            if i + batch_size >= max_index:
                i = min_index + lookback
            rows = np.arange(i, min(i + batch_size, max_index))
            i += len(rows)

        samples = np.zeros((len(rows),
                           lookback // step,
                           data.shape[-1]))
        targets = np.zeros((len(rows),))
        for j, row in enumerate(rows):
            indices = range(rows[j] - lookback, rows[j], step)
            samples[j] = data[indices]
            targets[j] = data[rows[j] + delay][1]
        yield samples, targets
```

Now let's use our abstract generator function to instantiate three generators, one for training, one for validation and one for testing. Each will look at different temporal segments of the original data: the training generator looks at the first 200,000 timesteps, the validation generator looks at the following 100,000, and the test generator looks at the remainder.

Listing 6.40 Preparing the training, validation and test generators

```
lookback = 1440
step = 6
delay = 144
batch_size = 128

train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=200000,
                      shuffle=True,
                      step=step,
                      batch_size=batch_size)
val_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=200001,
                     max_index=300000,
                     step=step,
                     batch_size=batch_size)
```

```

test_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=300001,
                     max_index=None,
                     step=step,
                     batch_size=batch_size)

# This is how many steps to draw from `val_gen`
# in order to see the whole validation set:
val_steps = (300000 - 200001 - lookback) // batch_size

# This is how many steps to draw from `test_gen`
# in order to see the whole test set:
test_steps = (len(float_data) - 300001 - lookback) // batch_size

```

6.3.3 A common sense, non-machine learning baseline

Before we start leveraging black-box deep learning models to solve our temperature prediction problem, let's try out a simple common-sense approach. It will serve as a sanity check, and it will establish a baseline that we will have to beat in order to demonstrate the usefulness of more advanced machine learning models. Such common-sense baselines can be very useful when approaching a new problem for which there is no known solution (yet). A classic example is that of unbalanced classification tasks, where some classes can be much more common than others. If your dataset contains 90% of instances of class A and 10% of instances of class B, then a common sense approach to the classification task would be to always predict "A" when presented with a new sample. Such a classifier would be 90% accurate overall, and any learning-based approach should therefore beat this 90% score in order to demonstrate usefulness. Sometimes such elementary baseline can prove surprisingly hard to beat.

In our case, the temperature timeseries can safely be assumed to be continuous (the temperatures tomorrow are likely to be close to the temperatures today) as well as periodical with a daily period. Thus a common sense approach would be to always predict that the temperature 24 hours from now will be equal to the temperature right now. Let's evaluate this approach, using the Mean Absolute Error metric (MAE). Mean Absolute Error is simply equal to:

Listing 6.41 The Mean Absolute Error (MAE)

```
np.mean(np.abs(preds - targets))
```

Here's our evaluation loop:

Listing 6.42 Computing the common-sense baseline MAE

```

def evaluate_naive_method():
    batch_maes = []
    for step in range(val_steps):
        samples, targets = next(val_gen)
        preds = samples[:, -1, 1]
        mae = np.mean(np.abs(preds - targets))
        batch_maes.append(mae)
    print(np.mean(batch_maes))

```

```
evaluate_naive_method()
```

It yields a MAE of 0.29. Since our temperature data has been normalized to be centered on 0 and have a standard deviation of one, this number is not immediately interpretable. It translates to an average absolute error of $0.29 * \text{temperature_std}$ degrees Celsius, i.e. 2.57°C . That's a fairly large average absolute error—now the game is to leverage our knowledge of deep learning to do better.

Listing 6.43 Converting the MAE back to a Celsius error

```
celsius_mae = 0.29 * std[1]
```

6.3.4 A basic machine learning approach

In the same way that it is useful to establish a common sense baseline before trying machine learning approaches, it is useful to try simple and cheap machine learning models (such as small densely-connected networks) before looking into complicated and computationally expensive models such as RNNs. This is the best way to make sure that any further complexity we throw at the problem later on is legitimate and delivers real benefits.

Here is a simply fully-connected model in which we start by flattening the data, then run it through two `Dense` layers. Note the lack of activation function on the last `Dense` layer, which is typical for a regression problem. We use MAE as the loss. Since we are evaluating on the exact same data and with the exact same metric as with our common sense approach, the results will be directly comparable.

Listing 6.44 Training and evaluating a densely-connected model using the data generators

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Flatten(input_shape=(lookback // step, float_data.shape[-1])))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=20,
                               validation_data=val_gen,
                               validation_steps=val_steps)
```

Let's display the loss curves for validation and training:

Listing 6.45 Plotting results

```
import matplotlib.pyplot as plt
```

```

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()

```

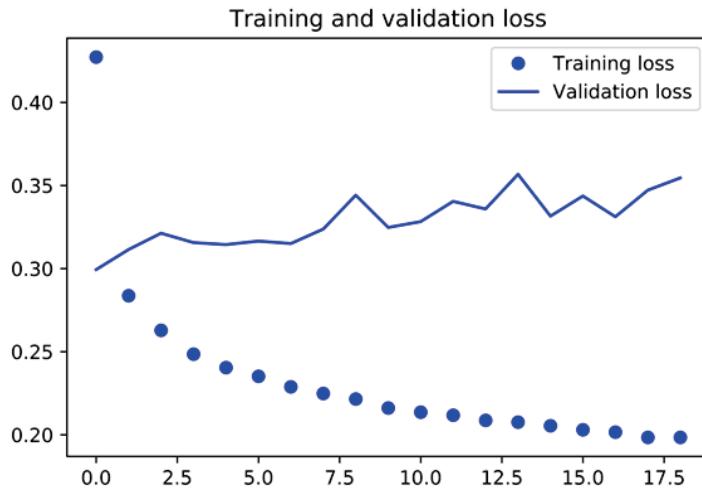


Figure 6.19 Training and validation loss on the Jena temperature forecasting task with a simple densely-connected network.

Some of our validation losses get close to the no-learning baseline, but not very reliably. This goes to show the merit of having had this baseline in the first place: it turns out not to be so easy to outperform. Our common sense contains already a lot of valuable information that a machine learning model does not have access to.

You may ask, if there exists a simple, well-performing model to go from the data to the targets (our common sense baseline), why doesn't the model we are training find it and improve on it? Simply put: because this simple solution is not what our training setup is looking for. The space of models in which we are searching for a solution, i.e. our hypothesis space, is the space of all possible 2-layer networks with the configuration that we defined. These networks are already fairly complicated. When looking for a solution with a space of complicated models, the simple well-performing baseline might be unlearnable, even if it's technically part of the hypothesis space. That is a pretty significant limitation of machine learning in general: unless the learning algorithm is hard-coded to look for a specific kind of simple model, parameter learning can sometimes fail to find a simple solution to a simple problem.

6.3.5 A first recurrent baseline

Our first fully-connected approach didn't do so well, but that doesn't mean machine learning is not applicable to our problem. The approach above consisted in first flattening the timeseries, which removed the notion of time from the input data. Let us instead look at our data as what it is: a sequence, where causality and order matter. We will try a recurrent sequence processing model—it should be the perfect fit for such sequence data, precisely because it does exploit the temporal ordering of data points, unlike our first approach.

Instead of the LSTM layer introduced in the previous section, we will use the GRU layer, developed by Cho et al. in 2014. GRU layers (which stands for "gated recurrent unit") work by leveraging the same principle as LSTM, but they are somewhat streamlined and thus cheaper to run, albeit they may not have quite as much representational power as LSTM. This trade-off between computational expensiveness and representational power is seen everywhere in machine learning.

Listing 6.46 Training and evaluating a GRU-based model

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=20,
                               validation_data=val_gen,
                               validation_steps=val_steps)
```

Let look at our results:

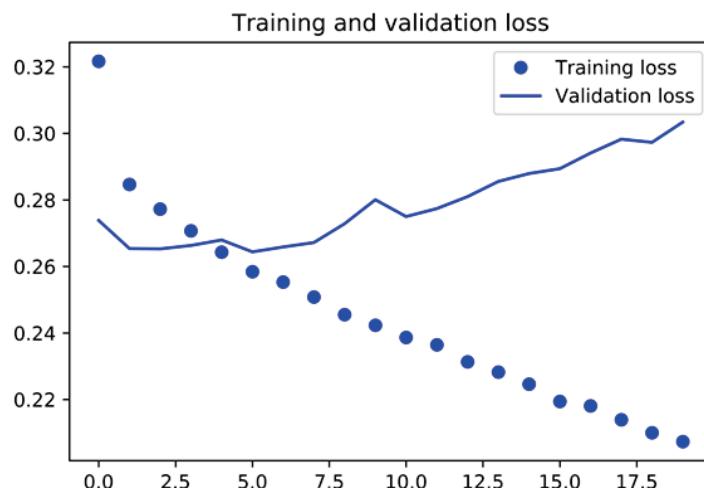


Figure 6.20 Training and validation loss on the Jena temperature forecasting task with a GRU.

Much better! We are able to significantly beat the common sense baseline, such demonstrating the value of machine learning here, as well as the superiority of recurrent networks compared to sequence-flattening dense networks on this type of task.

Our new validation MAE of ~ 0.265 (before we start significantly overfitting) translates to a mean absolute error of 2.35°C after de-normalization. That's a solid gain on our initial error of 2.57°C , but we probably still have a bit of margin for improvement.

6.3.6 Using recurrent dropout to fight overfitting

It is evident from our training and validation curves that our model is overfitting: the training and validation losses start diverging considerably after a few epochs. You are already familiar with a classic technique for fighting this phenomenon: dropout, consisting in randomly zeroing-out input units of a layer in order to break happenstance correlations in the training data that the layer is exposed to. How to correctly apply dropout in recurrent networks, however, is not a trivial question. It has long been known that applying dropout before a recurrent layer hinders learning rather than helping with regularization. In 2015, Yarin Gal, as part of his Ph.D. thesis on Bayesian deep learning, determined the proper way to use dropout with a recurrent network: the same dropout mask (the same pattern of dropped units) should be applied at every timestep, instead of a dropout mask that would vary randomly from timestep to timestep. What's more: in order to regularize the representations formed by the recurrent gates of layers such as GRU and LSTM, a temporally constant dropout mask should be applied to the inner recurrent activations of the layer (a "recurrent" dropout mask). Using the same dropout mask at every timestep allows the network to properly propagate its learning error through time; a temporally random dropout mask would instead disrupt this error signal and be harmful to the learning process.

Yarin Gal did his research using Keras and helped build this mechanism directly into Keras recurrent layers. Every recurrent layer in Keras has two dropout-related arguments: `dropout`, a float specifying the dropout rate for input units of the layer, and `recurrent_dropout`, specifying the dropout rate of the recurrent units. Let's add dropout and recurrent dropout to our GRU layer and see how it impacts overfitting. Because networks being regularized with dropout always take longer to fully converge, we train our network for twice as many epochs.

Listing 6.47 Training and evaluating a dropout-regularized GRU-based model

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
                     dropout=0.2,
                     recurrent_dropout=0.2,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
```

```
steps_per_epoch=500,
epochs=40,
validation_data=val_gen,
validation_steps=val_steps)
```

Let's take a look at our results:

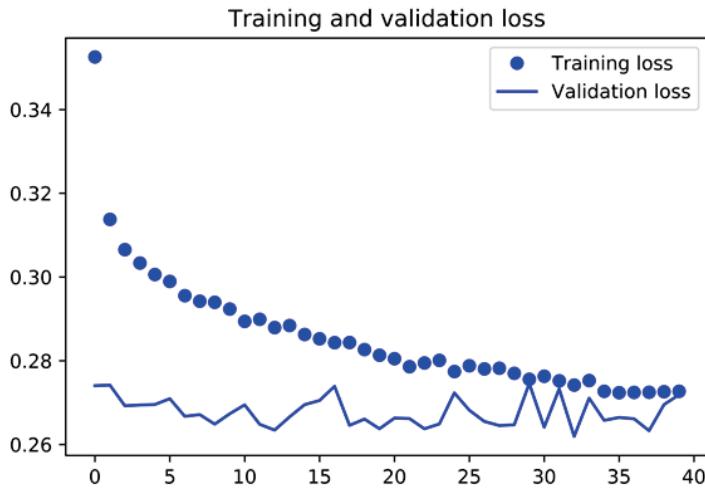


Figure 6.21 Training and validation loss on the Jena temperature forecasting task with a dropout-regularized GRU.

Great success; we are no longer overfitting during the first 30 epochs. However, while we have more stable evaluation scores, our best scores are not much lower than they were previously.

6.3.7 Stacking recurrent layers

Since we are no longer overfitting yet we seem to have hit a performance bottleneck, we should start considering increasing the capacity of our network. If you remember our description of the "universal machine learning workflow": it is a generally a good idea to increase the capacity of your network until overfitting becomes your primary obstacle (assuming that you are already taking basic steps to mitigate overfitting, such as using dropout). As long as you are not overfitting too badly, then you are likely under-capacity.

Increasing network capacity is typically done by increasing the number of units in the layers, or adding more layers. Recurrent layer stacking is a classic way to build more powerful recurrent networks: for instance, what currently powers the Google translate algorithm is a stack of seven large LSTM layers—that's huge.

To stack recurrent layers on top of each other in Keras, all intermediate layers should return their full sequence of outputs (a 3D tensor) rather than their output at the last timestep. This is done by specifying `return_sequences=True`:

Listing 6.48 Training and evaluating a dropout-regularized stacked GRU model

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
```

```

model.add(layers.GRU(32,
                     dropout=0.1,
                     recurrent_dropout=0.5,
                     return_sequences=True,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
                     dropout=0.1,
                     recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=40,
                               validation_data=val_gen,
                               validation_steps=val_steps)

```

Let's take a look at our results:

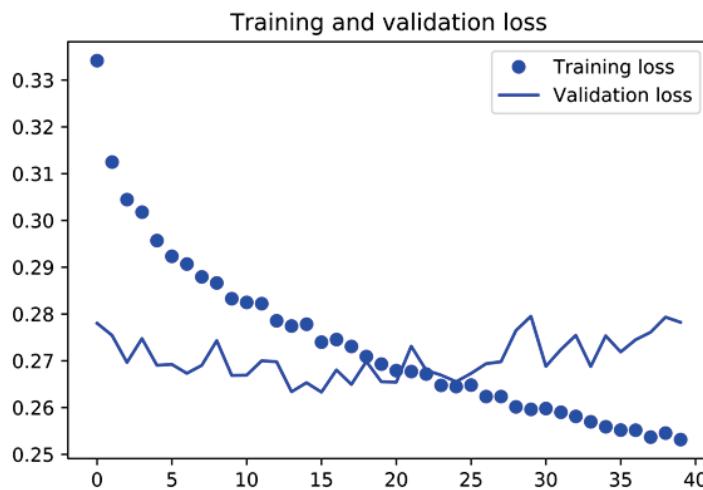


Figure 6.22 Training and validation loss on the Jena temperature forecasting task with a stacked GRU network.

We can see that the added layers does improve ours results by a bit, albeit not very significantly. We can draw two conclusions:

- Since we are still not overfitting too badly, we could safely increase the size of our layers, in quest for a bit of validation loss improvement. This does have a non-negligible computational cost, though.
- Since adding a layer did not help us by a significant factor, we may be seeing diminishing returns to increasing network capacity at this point.

6.3.8 Using bidirectional RNNs

The last technique that we will introduce in this section is called "bidirectional RNNs". A bidirectional RNN is common RNN variant which can offer higher performance than a regular RNN on certain tasks. It is frequently used in natural language processing—you could call it the Swiss army knife of deep learning for NLP.

RNNs are notably order-dependent, or time-dependent: they process the timesteps of their input sequences in order, and shuffling or reversing the timesteps can completely change the representations that the RNN will extract from the sequence. This is precisely the reason why they perform well on problems where order is meaningful, such as our

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

temperature forecasting problem. A bidirectional RNN exploits the order-sensitivity of RNNs: it simply consists of two regular RNNs, such as the GRU or LSTM layers that you are already familiar with, each processing input sequence in one direction (chronologically and antichronologically), then merging their representations. By processing a sequence both way, a bidirectional RNN is able to catch patterns that may have been overlooked by a one-direction RNN.

Remarkably, the fact that the RNN layers in this section have so far processed sequences in chronological order (older timesteps first) may have been an arbitrary decision. At least, it's a decision we made no attempt at questioning so far. Could it be that our RNNs could have performed well enough if it were processing input sequences in antichronological order, for instance (newer timesteps first)? Let's try this in practice and see what we get. All we need to do is write a variant of our data generator, where the input sequences get reverted along the time dimension (replace the last line with `yield samples[:, ::-1, :], targets`). Training the same one-GRU-layer network as we used in the first experiment in this section, we get the following results:

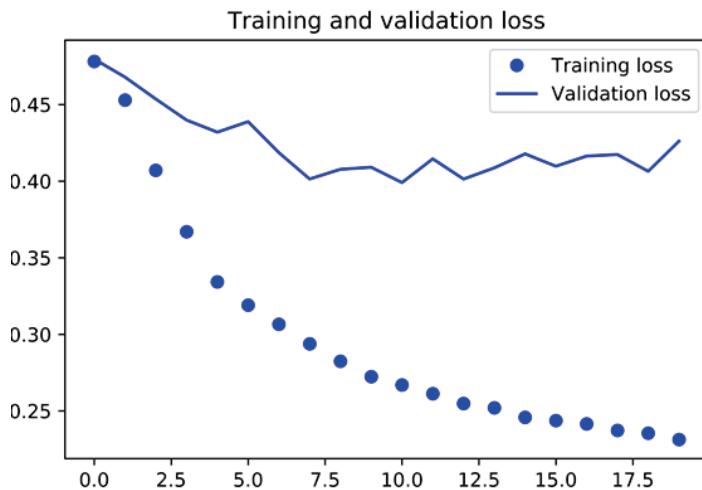


Figure 6.23 Training and validation loss on the Jena temperature forecasting task with a GRU trained on reversed sequences.

So the reversed-order GRU strongly underperforms even the common-sense baseline, indicating that the in our case chronological processing is very important to the success of our approach. This makes perfect sense: the underlying GRU layer will typically be better at remembering the recent past than the distant past, and naturally the more recent weather data points are more predictive than older data points in our problem (that is precisely what makes the common-sense baseline a fairly strong baseline). Thus the chronological version of the layer is bound to outperform the reversed-order version. Importantly, this is generally not true for many other problems, including natural language: intuitively, the importance of a word in understanding a sentence is not usually dependent on its position in the sentence. Let's try the same trick on the LSTM IMDB example from the previous section:

Listing 6.49 Training and evaluating a LSTM using reversed sequences on the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

IMDB data

```

from keras.datasets import imdb
from keras.preprocessing import sequence
from keras import layers
from keras.models import Sequential

# Number of words to consider as features
max_features = 10000
# Cut texts after this number of words (among top max_features most common words)
maxlen = 500

# Load data
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)

# Reverse sequences
x_train = [x[::-1] for x in x_train]
x_test = [x[::-1] for x in x_test]

# Pad sequences
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)

model = Sequential()
model.add(layers.Embedding(max_features, 128))
model.add(layers.LSTM(32))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                      epochs=10,
                      batch_size=128,
                      validation_split=0.2)

```

We get near-identical performance as the chronological-order LSTM we tried in the previous section.

Thus, remarkably, on such a text dataset, reversed-order processing works just as well as chronological processing, confirming our hypothesis that, albeit word order *does* matter in understanding language, *which* order you use isn't crucial. Importantly, a RNN trained on reversed sequences will learn different representations than one trained on the original sequences, in much the same way that you would have quite different mental models if time flowed backwards in the real world—if you lived a life where you died on your first day and you were born on your last day. In machine learning, representations that are *different* yet *useful* are always worth exploiting, and the more they differ the better: they offer a new angle from which to look at your data, capturing aspects of the data that were missed by other approaches, and thus they can allow to boost performance on a task. This is the intuition behind "ensembling", a concept that we will introduce in the next chapter.

A bidirectional RNN exploits this idea to improve upon the performance of chronological-order RNNs: it looks at its inputs sequence both ways, obtaining potentially richer representations and capturing patterns that may have been missed by the chronological-order version alone.

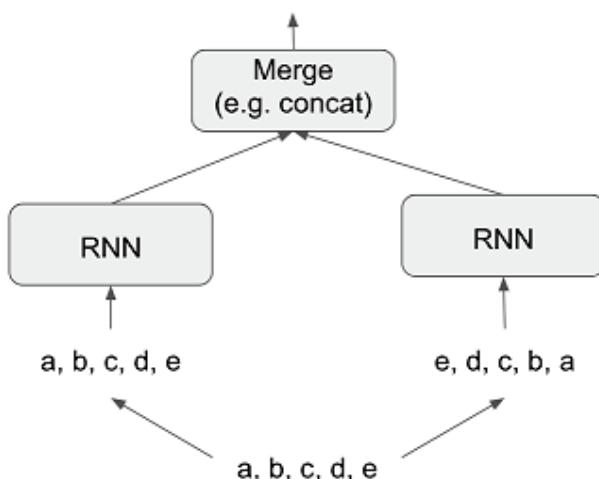


Figure 6.24 How a bidirectional RNN layer works.

To instantiate a bidirectional RNN in Keras, one would use the `Bidirectional` layer, which takes as first argument a recurrent layer instance. `Bidirectional` will create a second, separate instance of this recurrent layer, and will use one instance for processing the input sequences in chronological order and the other instance for processing the input sequences in reversed order. Let's try it on the IMDB sentiment analysis task:

Listing 6.50 Training and evaluating a bidirectional LSTM on the IMDB data

```

model = Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs=10, batch_size=128, validation_split=0.2)
  
```

It performs slightly better than the regular LSTM we tried in the previous section, going above 89% validation accuracy. It also seems to overfit faster, which is unsurprising since a bidirectional layer has twice more parameters than a chronological LSTM. With some regularization, the bidirectional approach would likely be a strong performer on this task.

Now let's try the same approach on the weather prediction task:

Listing 6.51 Training a bidirectional GRU on the Jena data

```

from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Bidirectional(
    layers.GRU(32), input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
  
```

```
steps_per_epoch=500,
epochs=40,
validation_data=val_gen,
validation_steps=val_steps)
```

It performs about as well as the regular GRU layer. It's easy to understand why: all of the predictive capacity must be coming from the chronological half of the network, since the anti-chronological half is known to be severely underperforming on this task (again, because the recent past matters much more than the distant past in this case).

6.3.9 Going even further

At this stage, there are still many other things you could try in order to improve performance on our weather forecasting problem:

- Adjust the number of units in each recurrent layer in the stacked setup. Our current choices are largely arbitrary and thus likely suboptimal.
- Adjust the learning rate used by our RMSprop optimizer.
- Try using LSTM layers instead of GRU layers.
- Try using a bigger densely-connected regressor on top of the recurrent layers, i.e. a bigger Dense layer or even a stack of Dense layers.
- Don't forget to eventually run the best performing models (in terms of validation MAE) on the test set! Least you start developing architectures that are overfitting to the validation set.

As usual: deep learning is more an art than a science, and while we can provide guidelines as to what is likely to work or not work on a given problem, ultimately every problem is unique and you will have to try and evaluate different strategies empirically. There is currently no theory that will tell you in advance precisely what you should do to optimally solve a problem. You must try and iterate.

6.3.10 Wrapping up

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

- As you first learned in Chapter 4, when approaching a new problem, it is good to first establish common sense baselines for your metric of choice. If you don't have a baseline to beat, you can't tell if you are making any real progress.
- Try simple models before expensive ones, to justify the additional expense. Sometimes a simple model will turn out to be your best option.
- On data where temporal ordering matters, recurrent networks are a great fit and easily outperform models that first flatten the temporal data.
- To use dropout with recurrent networks, one should use a time-constant dropout mask and recurrent dropout mask. This is built into Keras recurrent layers, so all you have to do is use the dropout and recurrent_dropout arguments of recurrent layers.
- Stacked RNNs provide more representational power than a single RNN layer. They are also much more expensive, and thus not always worth it. While they offer clear gains on complex problems (e.g. machine translation), they might not always be relevant to smaller, simpler problems.
- Bidirectional RNNs, which look at a sequence both ways, are very useful on natural language processing problems. However, they will not be strong performers on sequence data where the recent past is much more informative than the beginning of the sequence.

Note there are two important concepts that we will not cover in detail here: recurrent "attention", and sequence masking. Both tend to be especially relevant for natural language processing, and are not particularly applicable to our temperature forecasting problem. We will leave them for future study outside of this book.

One last remark to close this section—some readers are bound to want to take the techniques we introduced here and try them on the problem of forecasting the future price of securities on the stock market (or currency exchange rates, etc.). A warning: markets have very different statistical characteristics from natural phenomena such as weather patterns. Trying to use machine learning to beat markets while only having access to publicly available data is a very difficult endeavor, and you are likely to waste your time and resources with nothing to show for it. Always remember that when it comes to markets, past performance is not a good predictor of future returns—looking in the rearview mirror is a bad way to drive. Machine learning, on the other hand, is only applicable to datasets where past *is* a good predictor of the future.

6.4 Sequence processing with convnets

6.4.1 1D Convnets as an alternative to RNNs for sequence processing

In chapter 5, you learned about convolutional neural networks (convnets), and how they perform particularly well on computer vision problems, due to their ability to operate "convolutionally", extracting features from local input patches, allowing for representation modularity and data efficiency. The same properties that make convnets excel at computer vision also make them highly relevant to sequence processing. Indeed, time can be treated as a spatial dimension, like the height or the width of a 2D image.

Such 1D convnets can be competitive with RNNs on certain sequence processing problems, usually at a considerably cheaper computational cost. Recently, 1D convnets, typically used with dilated kernels, have been with great success for audio generation and machine translation. And besides these specific successes, it has long been known that small 1D convnets can offer a fast alternative to RNNs for simple tasks such as text classification or timeseries forecasting.

6.4.2 Understanding 1D convolution for sequence data

The convolution layers we introduced previously were 2D convolutions, extracting 2D patches out of image tensors and applying a same transformation to every patch. In the same way, we can use 1D convolutions, extracting local 1D patches (sub-sequences) out of sequences.

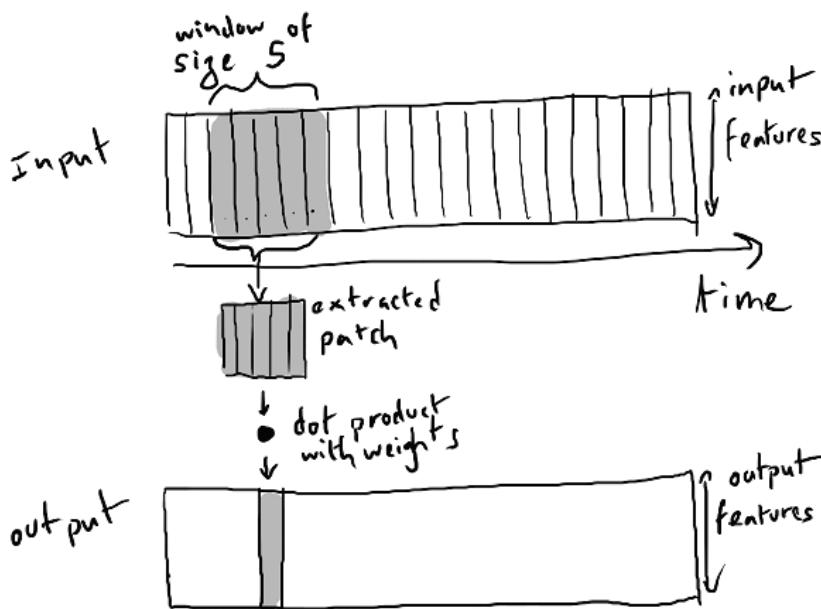


Figure 6.25 How 1D convolution works: each output timestep is obtained from a temporal patch in the input sequence.

Such 1D convolution layers will be able to recognize local patterns in a sequence. Because the same input transformation is performed on every patch, a pattern learned at a certain position in a sentence can later be recognized at a different position, making 1D convnets translation invariants (for temporal translations). For instance, a 1D convnet processing sequences of characters using convolution windows of size 5 should be able to learn words or word fragments of length 5 or lower, and should be able to recognize these words in any context in an input sequence. A character-level 1D convnet is thus capable to learn about word morphology.

6.4.3 1D Pooling for sequence data

You are already familiar with 2D pooling operations, such as 2D average pooling or max pooling, used in convnets to spatially downsample image tensors. The 2D pooling operation has a 1D equivalent, extracting 1D patches (subsequences) from an input and outputting the maximum value ("max pooling") or average value ("average pooling"). Just like in 2D convnets, this is used for reducing the length of 1D inputs ("subsampling").

6.4.4 Implementing a 1D convnet

In Keras, you would use a 1D convnet via the `Conv1D` layer, which has a very similar interface to `Conv2D`. It takes as input 3D tensors with shape `(samples, time, features)` and also returns similarly-shaped 3D tensors. The convolution window is a 1D window on the temporal axis, axis 1 in the input tensor.

Let's build a simple 2-layer 1D convnet and apply it to the IMDB sentiment classification task that you are already familiar with.

As a reminder, this is the code for obtaining and preprocessing the data:

Listing 6.52 Preparing the IMDB data

```

from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 10000 # number of words to consider as features
max_len = 500 # cut texts after this number of words (among top max_features most common words)

print('Loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)

```

1D convnets are structured in the same way as their 2D counter-parts that you have used in Chapter 5: they consist of a stack of Conv1D and MaxPooling1D layers, eventually ending in either a global pooling layer or a Flatten layer, turning the 3D outputs into 2D outputs, allowing to add one or more Dense layers to the model, for classification or regression.

One difference, though, is the fact that we can afford to use larger convolution windows with 1D convnets. Indeed, with a 2D convolution layer, a 3×3 convolution window contains $3 \times 3 = 9$ feature vectors, but with a 1D convolution layer, a convolution window of size 3 would only contain 3 feature vectors. We can thus easily afford 1D convolution windows of size 7 or 9.

This is our example 1D convnet for the IMDB dataset:

Listing 6.53 Training and evaluating a simple 1D convnet on the IMDB data

```

from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Embedding(max_features, 128, input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(lr=1e-4),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                      epochs=10,
                      batch_size=128,
                      validation_split=0.2)

```

Here are our training and validation results: validation accuracy is somewhat lower than that of the LSTM we used two sections ago, but runtime is faster, both on CPU and GPU (albeit the exact speedup will vary greatly depending on your exact configuration).

At that point, we could re-train this model for the right number of epochs (8), and run it on the test set. This is a convincing demonstration that a 1D convnet can offer a fast, cheap alternative to a recurrent network on a word-level sentiment classification task.

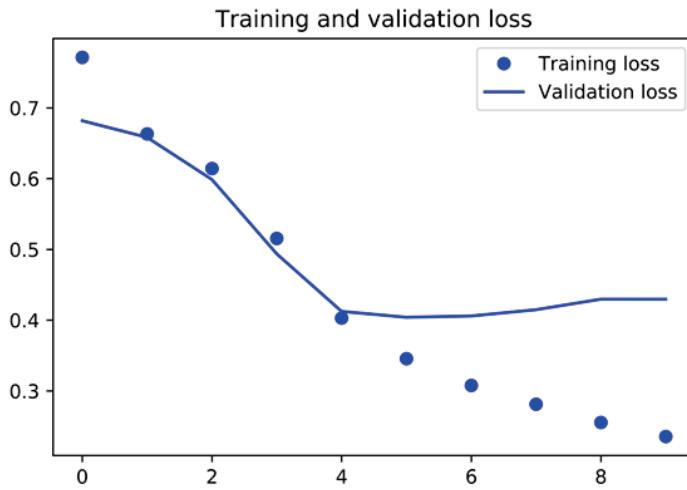


Figure 6.26 Training and validation loss on IMDB with a simple 1D convnet.

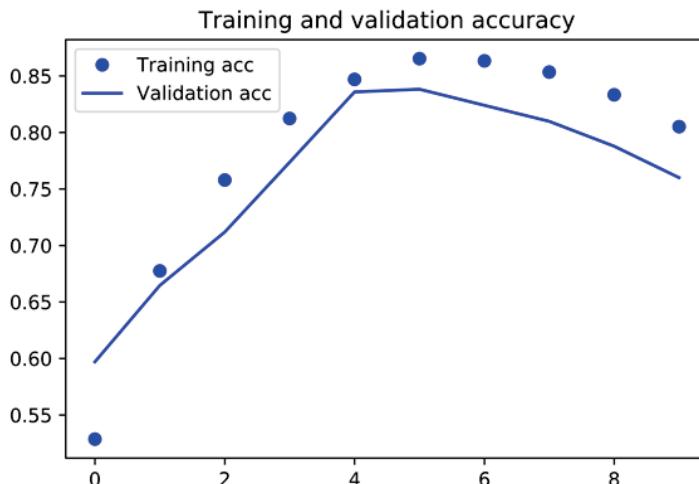


Figure 6.27 Training and validation accuracy on IMDB with a simple 1D convnet.

6.4.5 Combining CNNs and RNNs to process long sequences

Because 1D convnets process input patches independently, they are not sensitive to the order of the timesteps (beyond a local scale, the size of the convolution windows), unlike RNNs. Of course, in order to be able to recognize longer-term patterns, one could stack many convolution layers and pooling layers, resulting in upper layers that would "see" long chunks of the original inputs—but that's still a fairly weak way to induce order-sensitivity. One way to evidence this weakness is to try 1D convnets on the temperature forecasting problem from the previous section, where order-sensitivity was key to produce good predictions. Let's see:

Listing 6.54 Training and evaluating a simple 1D convnet on the Jena data

```
# We reuse the following variables defined in the last section:  
©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and  
other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.  
https://forums.manning.com/forums/deep-learning-with-python
```

Licensed to <null>

```
# float_data, train_gen, val_gen, val_steps

from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                      input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
cnn_history = model.fit_generator(train_gen,
                                    steps_per_epoch=500,
                                    epochs=20,
                                    validation_data=val_gen,
                                    validation_steps=val_steps)
```

Here are our training and validation Mean Absolute Errors:

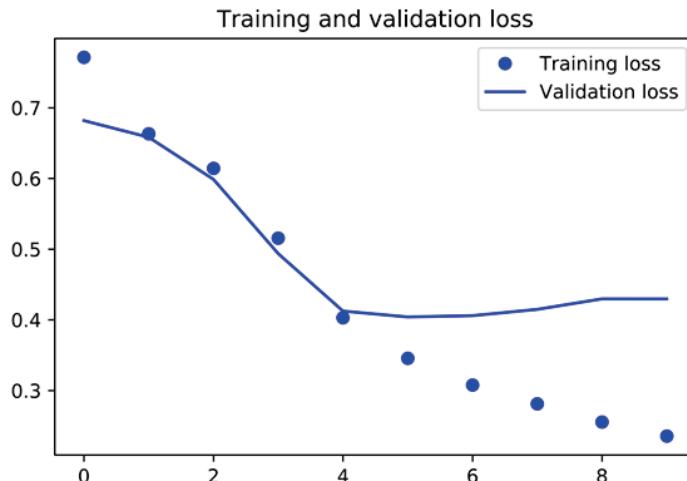


Figure 6.28 Training and validation loss on the Jena temperature forecasting task with a simple 1D convnet.

The validation MAE stays in the low 0.40s: we cannot even beat our common-sense baseline using the small convnet. Again, this is because our convnet looks for patterns anywhere in the input timeseries, and has no knowledge of the temporal position of a pattern it sees (e.g. towards the beginning, towards the end, etc.). Since more recent datapoints should be interpreted differently from older datapoints in the case of this specific forecasting problem, the convnet fails at producing meaningful results here. This limitation of convnets was not an issue on IMDB, because patterns of keywords that are associated with a positive or a negative sentiment will be informative independently of where they are found in the input sentences.

One strategy to combine the speed and lightness of convnets with the order-sensitivity of RNNs is to use a 1D convnet as a preprocessing step before a RNN. This is especially beneficial when dealing with sequences that are so long that they couldn't realistically be processed with RNNs, e.g. sequences with thousands of steps. The convnet will turn the

long input sequence into much shorter (downsampled) sequences of higher-level features. This sequence of extracted features then becomes the input to the RNN part of the network.

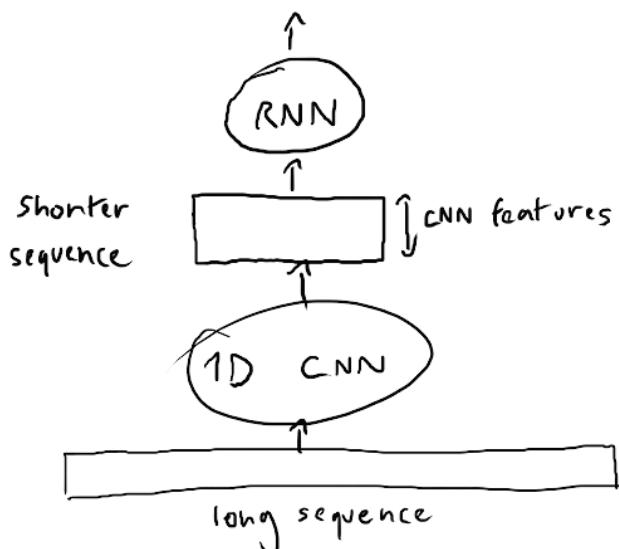


Figure 6.29 Combining a 1D convnet and a RNN for processing long sequences.

This technique is not seen very often in research papers and practical applications, possibly because it is not very well known. It is very effective and ought to be more common. Let's try this out on the temperature forecasting dataset. Because this strategy allows us to manipulate much longer sequences, we could either look at data from further back (by increasing the `lookback` parameter of the data generator), or look at high-resolution timeseries (by decreasing the `step` parameter of the generator). Here, we will chose (somewhat arbitrarily) to use a `step` twice smaller, resulting in twice longer timeseries, where the weather data is being sampled at a rate of one point per 30 minutes.

Listing 6.55 Preparing higher-resolution data generators for the Jena dataset

```

# We reuse the `generator` function defined at the previous section.

# This was previously set to 6 (one point per hour).
# Now 3 (one point per 30 min).
step = 3
lookback = 720 # Unchanged
delay = 144 # Unchanged

train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=200000,
                      shuffle=True,
                      step=step)
val_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=200001,
                     max_index=300000,
                     step=step)
test_gen = generator(float_data,

```

```

        lookback=lookback,
        delay=delay,
        min_index=300001,
        max_index=None,
        step=step)
val_steps = (300000 - 200001 - lookback) // 128
test_steps = (len(float_data) - 300001 - lookback) // 128

```

This is our model, starting with two Conv1D layers and following-up with a GRU layer:

Listing 6.56 Model combining a 1D convolutional base and a GRU layer

```

from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                      input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GRU(32, dropout=0.1, recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=20,
                               validation_data=val_gen,
                               validation_steps=val_steps)

```

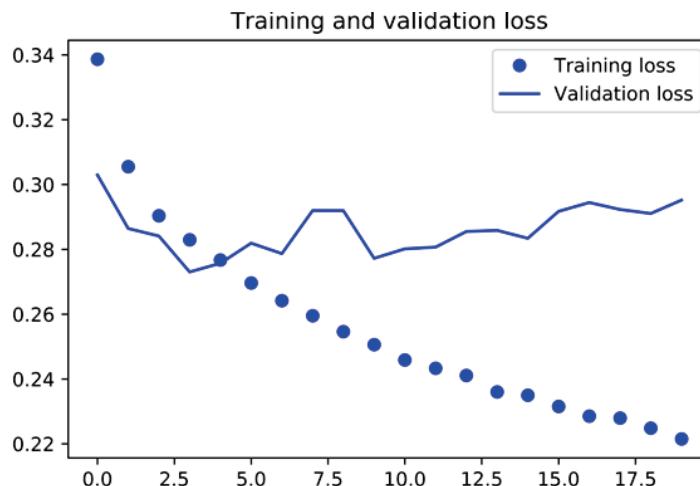


Figure 6.30 Training and validation loss on the Jena temperature forecasting task with a 1D convnet followed by a GRU.

Judging from the validation loss, this setup is not quite as good as the regularized GRU alone, but it's significantly faster. It is looking at twice more data, which in this case doesn't appear to be hugely helpful, but may be important for other datasets.

6.4.6 Wrapping up

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

- In the same way that 2D convnets perform well for processing visual patterns in 2D

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

space, 1D convnets perform well for processing temporal patterns. They offer a faster alternative to RNNs on some problems, in particular NLP tasks.

- Typically 1D convnets are structured much like their 2D equivalents from the world of computer vision: they consist of stacks of `Conv1D` layers and `MaxPooling1D` layers, eventually ending in a global pooling operation or flattening operation.
- Because RNNs are extremely expensive for processing very long sequences, but 1D convnets are cheap, it can be a good idea to use a 1D convnet as a preprocessing step before a RNN, shortening the sequence and extracting useful representations for the RNN to process.

One useful and important concept that we will not cover in these pages is that of 1D convolution with dilated kernels.

6.5 Wrapping up: deep learning for text and sequences

Here's what you have learned in this chapter:

- How to tokenize text.
- What word embeddings are, and how to use them.
- What recurrent networks are, and how to use them.
- How to stack RNN layers and use bidirectional RNNs to build more powerful sequence processing models.
- How to use 1D convnets for sequence processing.
- How to combine 1D convnets and RNNs to process long sequences.

These techniques are widely applicable to any dataset of sequence data, from text to timeseries.

For instance, you could use RNNs for:

- Timeseries regression ("predicting the future").
- Timeseries classification.
- Anomaly detection in timeseries.
- Sequence labeling, e.g. identifying names or dates in sentences.
- ...

Similarly, you could use 1D convnets for:

- Machine translation (sequence-to-sequence convolutional models, like SliceNet).
- Document classification.
- Spelling correction.
- ...

Remember: if *global order matters* in your sequence data, then it is preferable to use a recurrent network to process it. This is typically the case for timeseries, where the recent past is likely to be more informative than the distant past. But if global ordering isn't fundamentally meaningful, then 1D convnets will turn out to work at least as well, while being cheaper. This is often the case for text data, where a keyword found at the beginning of a sentence is just as meaningful as a keyword found at the end.

Advanced deep learning best practices

In this chapter, we cover more advanced techniques for designing and manipulating deep neural networks:

- The Keras functional API, with which you will be able to build graph-like models, share a same layer across different inputs, and use Keras models just like Python functions.
- Keras callbacks, and the TensorBoard browser-based visualization tool, to monitor models during training.
- Important best practices such as batch normalization, residual connections, hyperparameter optimization, and model ensembling.

These are powerful tools that will bring you closer to being able to develop state-of-art models on difficult problems.

7.1 Going beyond the *sequential* model: the Keras functional API

Until now, all neural networks introduced in this book have been implemented using the Sequential model. The Sequential model makes the assumption that the network has exactly one input and exactly one output, and that it consists of a linear stack of layers.

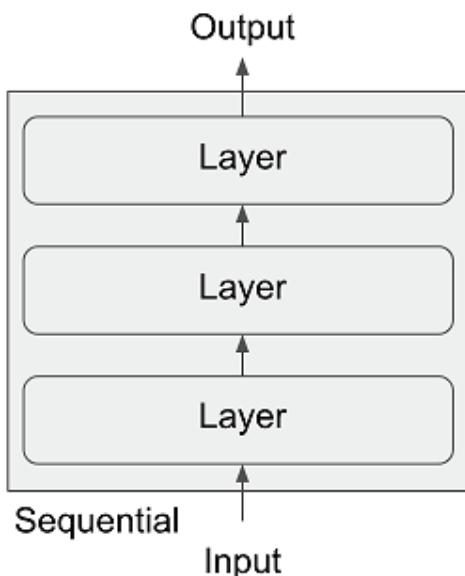


Figure 7.1 A Sequential model: a linear stack of layers.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

This is a very commonly verified assumption; this configuration is in fact so common that we have been able to cover many topics and practical applications in these pages so far using only the `Sequential` model class. However, this set of assumptions is too inflexible in a number of cases. Some networks require several independent inputs, some others require multiple outputs, and some networks have internal branching between layers making them look like *graphs* of layers rather than linear stacks of layers.

Some tasks, for instance, require *multi-modal* inputs: they merge data coming from different input sources, processing each type of data using different kinds of neural layers. Imagine a deep learning model trying to predict the most likely market price of a second-hand piece of clothing, using as input: 1) some user-provided metadata (such as the brand, the age, etc.), 2) a user-provided text description, and 3) a picture of the item. If we only had the metadata available, we could one-hot encode it and use a densely-connected network to predict the price. If we only had the text description available, we could use a RNN or a 1D convnet. If we only had the picture, we could use a 2D convnet. But how can all leverage all three at the same time? A naive approach would be to train three separate models, and then do a weighted average of their predictions. However, this may well be suboptimal, because the information extracted by the models may be high redundant. A better way is to *jointly* learn a more accurate model of the data by using a model that can see all available input modalities simultaneously: a model with three input branches (see Figure 7.2).

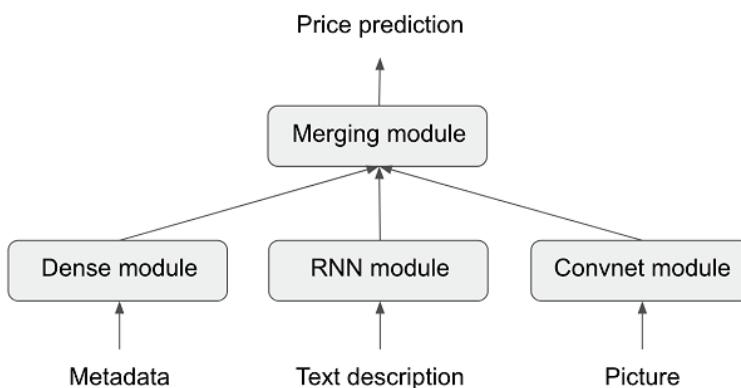


Figure 7.2 A multi-input model.

Similarly, some tasks require predicting multiple target attributes of some input data. Given the text of a novel or short-story, one might want to automatically classify it by genre (e.g. romance, thriller) but also predict the approximate date it was written. Of course, one could simply train two separate models, one for the genre and one for the date. However, because these attributes are not statistically independent, we can build a better model by learning to *jointly* predict both genre and date at the same time. Such a joint model would then have two outputs, or two "heads" (Figure 7.3). Due to correlations between genre and date, knowing the date of a novel will help the model learn rich and accurate representations of the space of novel genres, and reciprocally.

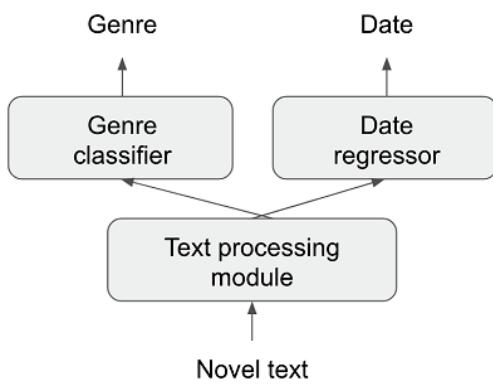


Figure 7.3 A multi-output (or multi-head) model.

Additionally, many recent neural architectures require non-linear network topology: networks structured as directed acyclic graphs. The Inception family of networks (developed by Szegedy et al. at Google), for instance, relies on "Inception modules", where the input is processed by several parallel convolutional branches whose outputs then get merged back into a single tensor (Figure 7.4). There is also the recent trend of adding "residual connections" to a model, which started with the ResNet family of networks (developed by He et al at Microsoft). A residual connection consists simply in reinjecting previous representations into the downstream flow of data, by adding a past output tensor to later output tensor (Figure 7.5), which helps prevent information loss along the data processing flow. And there are many more examples of such graph-like networks.

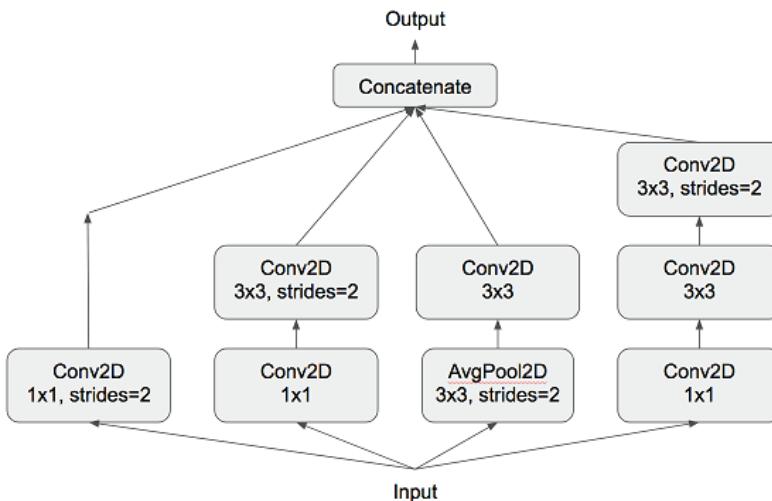


Figure 7.4 An Inception module: a subgraph of layers with several parallel convolutional branches.

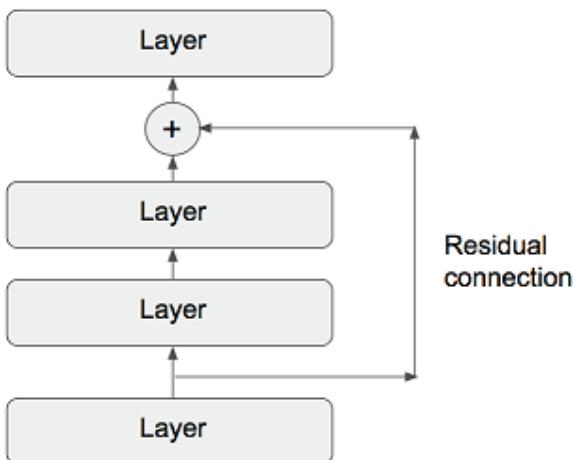


Figure 7.5 A residual connection: reinjection of prior information downstream via feature map addition.

These three important use cases—multi-input models, multi-output models, and graph-like models—are not possible when using only the `Sequential` model class in Keras. But there is also a different, far more general and flexible way to use Keras: the *functional API*. This section explains in detail what it is, what it can do, and how to use it.

7.1.1 Introduction to the functional API

In the functional API, you are directly manipulating tensors, and you use layers as *functions* that take tensors and return tensors (hence the name "functional API").

Listing 7.1 Calling layers as function in the functional API

```

from keras import Input, layers

# This is a tensor.
input_tensor = Input(shape=(32,))

# A layer is a function.
dense = layers.Dense(32, activation='relu')

# A layer may be called on a tensor, and it returns a tensor.
output_tensor = dense(input_tensor)
  
```

Let's start with a minimal example: we will show side by side a simple `Sequential` model and its equivalent in the functional API.

Listing 7.2 The functional API equivalent to a Sequential model

```

from keras.models import Sequential, Model
from keras import layers
from keras import Input

# A Sequential model, which you already know all about.
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))

# Its functional equivalent.
  
```

```

input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)

# The Model class turns an input tensor and output tensor into a model
model = Model(input_tensor, output_tensor)

# Let's look at it!
model.summary()

```

This is what our call to `model.summary()` displays:

Listing 7.3 Summary of the functional model

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	(None, 64)	0
dense_1 (Dense)	(None, 32)	2080
dense_2 (Dense)	(None, 32)	1056
dense_3 (Dense)	(None, 10)	330
<hr/>		
Total params: 3,466		
Trainable params: 3,466		
Non-trainable params: 0		

The only part that may seem a bit magical at this point is instantiating a `Model` object using only an input tensor and output tensor. Behind the scenes, Keras will retrieve every layer that was involved in going from `input_tensor` to `output_tensor`, bringing them together into a graph-like data structure—a `Model`. Of course, the reason it works is because `output_tensor` was indeed obtained by repeatedly transforming `input_tensor`. If you tried to build a model from inputs and outputs that were not related, you would get a `RuntimeError`:

Listing 7.4 A graph disconnection error

```

>>> unrelated_input = Input(shape=(32,))
>>> bad_model = Model(unrelated_input, output_tensor)
RuntimeError: Graph disconnected: cannot obtain value for tensor Tensor("input_1:0", shape=(?, 64), c

```

This error tells you, in essence, that Keras was not able to reach `input_1` from the provided output tensor.

When it comes to compiling, training or evaluating such an instance of `Model`, the API is the same as that of `Sequential`:

Listing 7.5 Training models built with the functional API: business as usual

```

# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# Generate dummy Numpy data to train on

```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

```

import numpy as np
x_train = np.random.random((1000, 64))
y_train = np.random.random((1000, 10))

# Train the model for 10 epochs
model.fit(x_train, y_train, epochs=10, batch_size=128)

# Evaluate the model
score = model.evaluate(x_train, y_train)

```

7.1.2 Multi-input models

The functional API can be used to build models that have multiple inputs. Typically, such models will at some point "merge" their different input branches using a layer that can combine several tensors, i.e. by adding them, concatenating them, etc. This is usually done via a Keras "merge operation" such as `keras.layers.add`, `keras.layers.concatenate`, etc. Let's take a look at a very simple example of a multi-input model: a question-answering model.

A typical question-answering model has two inputs: a natural language question, and a text snippet (such as a news article) providing information to be used for answering the question. The model must then produce an answer: in the simplest possible setup, this is simply a one-word answer obtained via a softmax over some predefined vocabulary. This is presented in Figure 7.6.

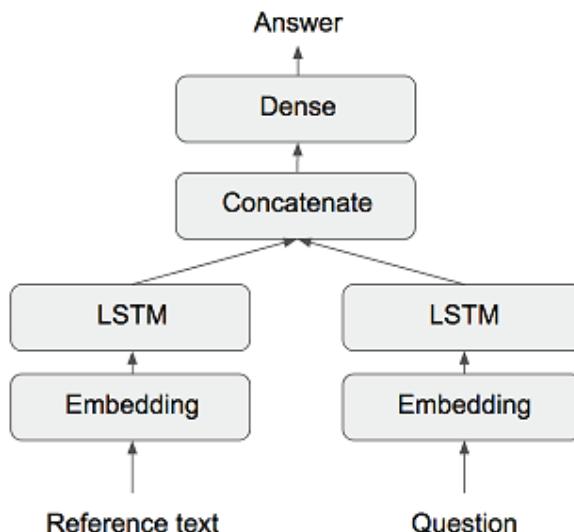


Figure 7.6 A question-answering model

Here is an example of how we can build such a model with the functional API: we set up two independent branches, encoding the text input and the question input as representation vectors, then we concatenate these vectors, and finally, we add a softmax classifier on top of the concatenated representations.

Listing 7.6 Functional API implementation of a two-input question-answering model

```

from keras.models import Model
from keras import layers

```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

```

from keras import Input

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

# Our text input is a variable-length sequence of integers.
# Note that we can optionally name our inputs!
text_input = Input(shape=(None,), dtype='int32', name='text')

# Which we embed into a sequence of vectors of size 64
embedded_text = layers.Embedding(64, text_vocabulary_size)(text_input)

# Which we encoded in a single vector via a LSTM
encoded_text = layers.LSTM(32)(embedded_text)

# Same process (with different layer instances) for the question
question_input = Input(shape=(None,), dtype='int32', name='question')
embedded_question = layers.Embedding(32, question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

# We then concatenate the encoded question and encoded text
concatenated = layers.concatenate([encoded_text, encoded_question], axis=-1)

# And we add a softmax classifier on top
answer = layers.Dense(answer_vocabulary_size, activation='softmax')(concatenated)

# At model instantiation, we specify the two inputs and the output:
model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])

```

Now, how do we train this two-input model? There are two possible APIs: you could feed as inputs to the model a list of Numpy arrays, or you could feed it a dictionary mapping input names to Numpy arrays. Naturally, the latter option is only available if you gave names to your inputs.

Listing 7.7 Feeding data to a multi-input model

```

import numpy as np

# Let's generate some dummy Numpy data
text = np.random.randint(1, text_vocabulary_size, size=(num_samples, max_length))
question = np.random.randint(1, question_vocabulary_size, size=(num_samples, max_length))

# Answers are one-hot encoded, not integers
answers = np.random.randint(0, 1, size=(num_samples, answer_vocabulary_size))

# Fitting using a list of inputs
model.fit([text, question], answers, epochs=10, batch_size=128)

# Fitting using a dictionary of inputs (only if inputs were named!)
model.fit({'text': text, 'question': question}, answers,
          epochs=10, batch_size=128)

```

7.1.3 Multi-output models

In the same way, the functional API can be used to build models with multiple outputs (or multiple "heads", as sometimes described in the literature). A simple example would be a network that attempts to simultaneously predict different properties of the data: let's say, a network that takes as input a series of social media posts from one single anonymous person, and tries to predict attributes of that person, such as age, gender, or income level (Figure 7.7).

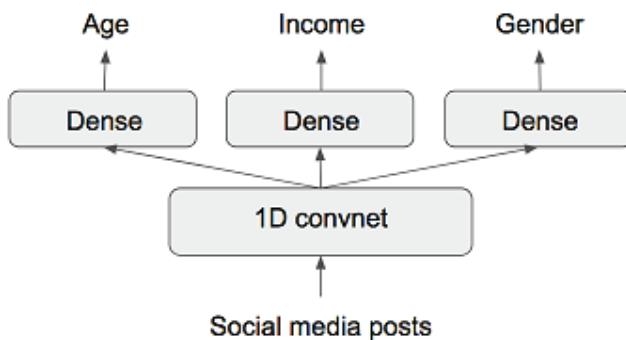


Figure 7.7 A social media model with three heads

Listing 7.8 Functional API implementation of a three-output model

```

from keras import layers
from keras import Input
from keras.models import Model

vocabulary_size = 50000
num_income_groups = 10

posts_input = Input(shape=(None,), dtype='int32', name='posts')
embedded_posts = layers.Embedding(256, vocabulary_size)(posts_input)
x = layers.Conv1D(128, 5, activation='relu')(embedded_posts)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation='relu')(x)

# Note that we are giving names to the output layers.
age_prediction = layers.Dense(1, name='age')(x)
income_prediction = layers.Dense(num_income_groups, activation='softmax', name='income')(x)
gender_prediction = layers.Dense(1, activation='sigmoid', name='gender')(x)

model = Model(posts_input, [age_prediction, income_prediction, gender_prediction])
  
```

Importantly, training such a model requires the ability to specify different loss functions for different "heads" of the network: for instance, age prediction is a scalar regression task, but gender prediction is a binary classification task, requiring a different training procedure. However, since gradient descent requires us to minimize a *scalar*, we must combine these losses into a single value in order to be able to train the model. The simplest way to combine different losses is simply to sum them all. In Keras, you can use either a list or a dictionary of losses in `compile` to specify different objects for different

outputs, and the resulting loss values get summed into a global loss, which is what gets minimized during training.

Listing 7.9 Compilation options of a multi-output model: multiple losses

```
model.compile(optimizer='rmsprop',
              loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'])

# Equivalent (only possible if you gave names to the output layers!):
model.compile(optimizer='rmsprop',
              loss={'age': 'mse',
                    'income': 'categorical_crossentropy',
                    'gender': 'binary_crossentropy'})
```

Note that it is also possible to assign different importances to the loss values in their contribution to the final loss. This is useful in particular if the different losses take values on different scales. For instance, the MSE loss used for the age regression task could take a typical value around 3-5, while the crossentropy loss used for the gender classification task could be as low as 0.1. In such a situation, in order to make the contribution of the different losses more balanced, you would assign a weight of 10 to the crossentropy loss, and a weight of 0.25 to the MSE loss. Very imbalanced loss contributions would cause the model representations to be optimized preferentially for the task with the largest individual loss, at the expense of the other tasks.

Listing 7.10 Compilation options of a multi-output model: loss weighting

```
model.compile(optimizer='rmsprop',
              loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'],
              loss_weights=[0.25, 1., 10.])

# Equivalent (only possible if you gave names to the output layers!):
model.compile(optimizer='rmsprop',
              loss={'age': 'mse',
                    'income': 'categorical_crossentropy',
                    'gender': 'binary_crossentropy'},
              loss_weights={'age': 0.25,
                           'income': 1.,
                           'gender': 10.})
```

Much like in the case of multi-input models, passing Numpy data to the model for training can be done either via a list of arrays or via a dictionary of arrays:

Listing 7.11 Feeding data to a multi-output model

```
# age_targets, income_targets and gender_targets are assumed to be Numpy arrays
model.fit(posts, [age_targets, income_targets, gender_targets],
          epochs=10, batch_size=64)

# Equivalent (only possible if you gave names to the output layers!):
model.fit(posts, {'age': age_targets,
                  'income': income_targets,
                  'gender': gender_targets},
          epochs=10, batch_size=64)
```

7.1.4 Directed acyclic graphs of layers

With the functional API, not only can you build models with multiple inputs and multiple outputs, you can also implement networks with a complex internal topology. Neural networks in Keras are allowed to be arbitrary *directed acyclic graphs* of layers. The qualifier "acyclic" is important: these graphs cannot have cycles, i.e. it is impossible for a tensor x to become the input of one of the layers that generated x). The only processing "loops" that are allowed (i.e. recurrent connections) are those internal to recurrent layers.

Several common neural network components are implemented as graphs. Two notable ones are Inception modules, and residual connections. To better understand how the functional API can be used to build graphs of layers, let's take a look at how we can implement both of them in Keras.

INCEPTION MODULES

Inception is a popular type of network architecture for convolutional neural networks, developed by Christian Szegedy and colleagues at Google in 2013-2014, inspired by the earlier "network-in-network" architecture. It consists of a stack of modules which themselves look like small independent networks, split into several parallel branches. The most basic form of an inception module has three to four branches starting with a 1x1 convolution, following up with a 3x3 convolution, and ending with the concatenation of the resulting features. This setup helps the network separately learn spatial features and channel-wise features, which is more efficient than learning them jointly. More complex versions of an Inception module are also possible, typically involving pooling operations, different spatial convolution sizes (e.g. 5x5 instead of 3x3 on some branches), and branches without a spatial convolution (only a 1x1 convolution). An example of such a module, taken from Inception V3, is provided in Figure 7.8.

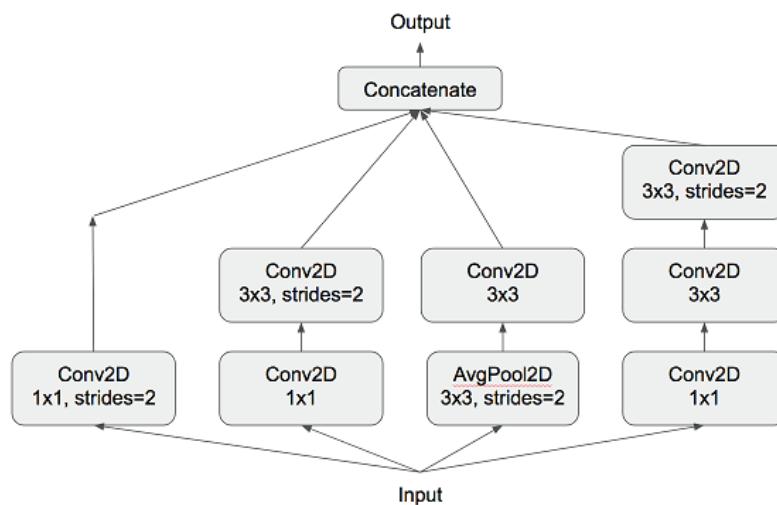


Figure 7.8 An Inception module

NOTE**The purpose of 1x1 convolutions (also called pointwise convolution).**

You already know that convolutions extract spatial patches around every tile in an input tensor, and apply a same transformation to each patch. An edge case is when the patches extracted consist of a single tile. The convolution operation then becomes equivalent to running each tile vector through a `Dense` layer: it will compute features that mix together information from the channels of the input tensor, but it will not mix information across space at all (since it is looking at one tile at a time). Such 1x1 convolutions are featured in Inception modules, where they contribute to factoring out channel-wise feature learning and space-wise feature learning—a reasonable thing to do if you assume that each channel is highly auto-correlated across space, but different channels might not be highly correlated with each other.

Here is how one would implement the module featured in Figure 7.4 using the functional API:

Listing 7.12 Implementing an Inception module with the functional API

```
from keras import layers

# We assume the existence of a 4D input tensor `x`

# Every branch has the same stride value (2), which is necessary to keep all
# branch outputs the same size, so as to be able to concatenate them.
branch_a = layers.Conv2D(128, 1, activation='relu', strides=2)(x)

# In this branch, the striding occurs in the spatial convolution layer
branch_b = layers.Conv2D(128, 1, activation='relu')(x)
branch_b = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_b)

# In this branch, the striding occurs in the average pooling layer
branch_c = layers.AveragePooling2D(3, strides=2, activation='relu')(x)
branch_c = layers.Conv2D(128, 3, activation='relu')(branch_c)

branch_d = layers.Conv2D(128, 1, activation='relu')(x)
branch_d = layers.Conv2D(128, 3, activation='relu')(branch_d)
branch_d = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_d)

# Finally, we concatenate the branch outputs to obtain the module output
output = layers.concatenate([branch_a, branch_b, branch_c, branch_d], axis=-1)
```

Note that the full Inception V3 architecture is available in Keras as `keras.applications.inception_v3.InceptionV3`, including weights pre-trained on the ImageNet dataset. Another closely related model available as part of the Keras applications module is `Xception`. "Xception", which stands for "extreme inception", is a convnet architecture loosely inspired by Inception. It takes the idea of separating the learning of channel-wise and space-wise features to its logical extreme, and replaces Inception modules with depthwise separable convolutions, consisting in a depthwise convolution (a spatial convolution where every input channel is handled separately) followed by a pointwise convolution (i.e. a 1x1 convolution)—effectively, an extreme

form of an Inception module, where spatial features and channel-wise features are fully separated. Xception has roughly the same number of parameters of Inception V3, but it shows better runtime performance and higher accuracy on ImageNet as well as other large-scale datasets, due to a more efficient use of model parameters.

RESIDUAL CONNECTIONS

Residual connections are a very common graph-like network component found in many post-2015 network architectures, including Xception. They were introduced by He et al from Microsoft in their winning entry in the ILSVRC ImageNet challenge in late 2015. They tackle two common problems that plague any large-scale deep learning model: vanishing gradients, and representational bottlenecks. In general, adding residual connections to any model that has over ten layers is likely to be beneficial.

A residual connection simply consist of making the output of an earlier layer available as input to a later layer, effectively creating a shortcut in a sequential network (Figure 7.5). Rather than being concatenated to the later activation, the earlier output is summed with the later activation, which assumes that both activations have the same size. In case of differing sizes, one may use a linear transformation to reshape the earlier activation into the target shape (e.g. a `Dense` layer without an activation, or for convolutional feature maps, a 1×1 convolution without an activation).

Here is how you would implement a residual connection in Keras:

Listing 7.13 Implementing a residual connection when feature map sizes are the same: using identity residual connections

```
from keras import layers

# We assume the existence of a 4D input tensor `x`
x = ...
# We apply some transformation to `x`
y = layers.Conv2D(128, 3, activation='relu')(x)
y = layers.Conv2D(128, 3, activation='relu')(y)
y = layers.Conv2D(128, 3, activation='relu')(y)

# We add the original `x` back to the output features
y = layers.add([y, x])
```

Listing 7.14 Implementing a residual connection when feature map sizes differ: using a linear residual connection

```
from keras import layers

# We assume the existence of a 4D input tensor `x`
x = ...
y = layers.Conv2D(128, 3, activation='relu')(x)
y = layers.Conv2D(128, 3, activation='relu')(y)
y = layers.MaxPooling2D(2, strides=2)(y)

# We use a 1x1 convolution to linearly downsample
# the original `x` tensor to the same shape as `y`
residual = layers.Conv2D(1, strides=2)(x)

# We add the residual tensor back to the output features
y = layers.add([y, residual])
```

NOTE**Representational bottlenecks in deep learning**

In a sequential model, each successive representation layer is built on top of the previous one, which means that it only has access to information contained in the activation of the previous layer. If one layer happens to be too small (e.g. have features that are too low-dimensional), then the model will be constrained by how much information can be crammed into the activations of this layer. You can think of it with a signal processing analogy: if you have an audio processing pipeline that consists of a series of operations that each take as input the output of the previous operation, then if one operation happens to crop your signal to a low frequency range (e.g. 0-15 kHz), the operations downstream will never be able to recover the dropped frequencies. Any loss of information is permanent. Residual connections, by reinjecting earlier information downstream, partially solve this issue for deep learning models.

NOTE**Vanishing gradients in deep learning**

Backpropagation, the master algorithm used to train deep neural networks, works by propagating a feedback signal from the output loss down to earlier layers. If this feedback signal has to be propagated through a very deep stack of layers, the signal may become very tenuous or even be lost entirely, rendering the network untrainable—this is the problem of "vanishing gradients". This problem occurs both with very deep networks and with recurrent networks over very long sequences—in both cases, a feedback signal must be propagated through a long series of operations. You are already familiar with the solution that the LSTM layer uses to address this problem in recurrent networks: it introduces a "carry track" that propagates information in parallel to the main processing track. Residual connections work in a very similar way in feedforward deep networks, but they are even simpler: they introduce a purely linear information carry track parallel to the main layer stack, thus helping to propagate gradients through arbitrarily deep stacks of layers.

7.1.5 Layer weight sharing

One more important feature of the functional API is the ability to reuse a layer instance several time. When you call a layer instance twice, instead of instantiating a new layer for each call, you are reusing the same weights with every call. This allows you to build models that have shared branches—several branches that all share the same knowledge and perform the same operations, i.e. that share the same representations, and learn these representations simultaneously for different sets of inputs.

One example would be a model that attempts to assess the semantic similarity between two sentences. The model would have two inputs (the two sentences to compare) and would output a score between 0 and 1, where 0 are unrelated sentences and 1 are sentences that either identical or mere reformulations of each other. Such a model

could be useful in many applications, including de-duplicating natural language queries in a dialog system.

In this setup, the two input sentences are interchangeable, because semantic similarity is a symmetrical relationship: the similarity of A to B is identical to the similarity of B to A. For this reason, it would not make sense to learn two independent models for processing each input sentence. Rather, we would like to process both with one single LSTM layer. The representations of this LSTM layer (its weights) would be learned based on both inputs simultaneously. This is what we would call a "siamese LSTM" model, or simply a "shared LSTM".

Here is how we would implement such a model using layer sharing in the Keras functional API:

Listing 7.15 Layer weight sharing (i.e. layer reuse) with the functional API: implementing a siamese LSTM model

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

# We instantiate a single LSTM layer, once
lstm = layers.LSTM(32)

# Building the left branch of the model
# ----

# Inputs are variable-length sequences of vectors of size 128
left_input = Input(shape=(None, 128))
left_output = lstm(left_input)

# Building the right branch of the model
# ----

right_input = Input(shape=(None, 128))
# When we call an existing layer instance,
# we are reusing its weights
right_output = lstm(right_input)

# Building the classifier on top
# ----

merged = layers.concatenate([left_output, right_output], axis=-1)
predictions = layers.Dense(1, activation='sigmoid')(merged)

# Instantiating and training the model
# ----

model = Model([left_input, right_input], predictions)
# When you train such a model, the weights of the `lstm` layer
# are updated based on both inputs.
model.fit([left_data, right_data], targets)
```

Naturally, a layer instance may be used more than once—it can be called arbitrarily many times, reusing the same set of weights every time.

7.1.6 Models as layers

Importantly, in the functional API, models can be used as you would use layers—effectively, you may think of a model as a "bigger layer". This is true of both the Sequential and Model classes. This means that you can call a model on an input tensor, and retrieve an output tensor:

Listing 7.16 Models as layers, i.e. models as functions

```
y = model(x)
```

If the model has multiple input tensors and multiple output tensors, it should be called with a list of tensors:

Listing 7.17 Calling a multi-output, multi-input model

```
y1, y2 = model([x1, x2])
```

When you call a model instance, you are reusing the weights of the model—exactly like what happens when you call a layer instance. Simply calling an instance, whether it is a layer instance or a model instance, will always reuse the existing learned representations of the instance—which is quite intuitive.

One simple practical example of what you can build by reusing a model instance would be a vision model that uses a dual camera as its input: two parallel cameras, a few centimeters (one inch) apart from each other. Such a model could be capable of perceiving depth, which can be useful in many applications. You shouldn't need two independent models for extracting visual features from the left camera and from the right camera, before merging the two feeds. Such low-level processing can be shared across the two inputs, i.e. done via layers that use the same weights and thus share the same representations. Here is how you would implement this in Keras:

Listing 7.18 Implementing a siamese vision model (shared convolutional base)

```
from keras import layers
from keras import applications
from keras import Input

# Our base image processing model will be the Xception network
# (convolutional base only).
xception_base = applications.Xception(weights=None, include_top=False)

# Our inputs are 250x250 RGB images.
left_input = Input(shape=(250, 250, 3))
right_input = Input(shape=(250, 250, 3))

# We call the same vision model twice!
left_features = xception_base(left_input)
right_input = xception_base(right_input)

# The merged features contain information from both
# the right visual feed and the left visual feed
merged_features = layers.concatenate([left_features, right_input], axis=-1)
```

7.1.7 Wrapping up

This concludes our introduction to the Keras functional API, an essential tool for building advanced deep neural network architectures. Now you know:

- When to step out of the `Sequential` API: whenever you need anything more than a linear stack of layers.
- How to build Keras models with several inputs, several outputs, and complex internal network topology, using the Keras functional API.
- How to reuse the weights of a layer or model across different processing branches, by calling a same layer or model instance several times.

7.2 Inspecting and monitoring deep learning models: using Keras callbacks and TensorBoard

In this section, we review ways to gain greater access and control over what goes on inside your model during training.

Launching a training run on a large dataset, for tens of epochs, using `model.fit()` or `model.fit_generator()`, can a bit like launching a paper plane: past the initial impulse, you don't have any control over its trajectory or its landing spot. If you want to avoid bad outcomes (and thus wasted paper planes), it is smarter to use not a paper plane, but a drone that will be able to sense its environment, send data back to its operator, and automatically make stirring decisions based on its current state. The techniques we present here will transform your call to `model.fit()` from a paper plane into a smart autonomous drone able to self-introspect and dynamically take action.

7.2.1 Using callbacks to act on a model during training

When training a model, there are many things you can't predict from the start. In particular, you cannot tell how many epochs will be needed to get to an optimal validation loss. In our examples so far, we have always adopted the strategy of training for enough epochs that we would start overfitting, using our first run to figure out the proper number of epochs to train for, then finally launch a new training run from scratch using this optimal number. Of course, this is quite wasteful.

A much better way to handle this would be stop training when we measure that the validation loss has stopped improving. This can be achieved using a Keras "callback". A callback is an object (a class instance implementing specific methods) that is passed to the model in the call to `fit` and that is called by the model at various points during training. It has access to all the data available about the state of the model and its performance, and it is capable to take action, for instance interrupting training, saving a model, loading a different weight set, or otherwise altering the state of the model.

Callbacks can be used for:

- Model checkpointing: saving the current weights of the model at different points during training.
- Early stopping: interrupting training when the validation loss has stopped improving (and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

of course, saving the best model obtained during training).

- Dynamically adjusting the value of certain parameters during training, such as the learning rate of the optimizer.
- Logging the training and validation metrics during training, or visualizing the representations learned by the model as they get updated. In fact, the Keras progress bar that you are familiar with is itself a callback!
- And more...

There are a number of built-in callbacks found in the `keras.callbacks` module (non-exhaustive list):

Listing 7.19 Some of the built-in Keras callbacks

```
keras.callbacks.ModelCheckpoint
keras.callbacks.EarlyStopping
keras.callbacks.LearningRateScheduler
keras.callbacks.ReduceLROnPlateau
keras.callbacks.CSVLogger
```

Let's review a few of them to give you an idea of how to use them: `ModelCheckpoint`, `EarlyStopping`, and `ReduceLROnPlateau`.

THE MODELCHECKPOINT AND EARLYSTOPPING CALLBACKS

You can use the `EarlyStopping` callback to interrupt training once a target metric being monitored has stopped improving for a fixed number of epochs. For instance, this callback allows you to interrupt training as soon as you start overfitting, thus allowing you to avoid having to retrain your model for a smaller number of epochs. This callback is typically used in combination with `ModelCheckpoint`, which allows to continually save the model during training (and optionally, to save only the current best model so far, i.e. the version of the model that achieved the best performance at the end of an epoch).

Listing 7.20 Using Callbacks: example with EarlyStopping and ModelCheckpoint

```
import keras

# Callbacks are passed to the model fit the `callbacks` argument in `fit`,
# which takes a list of callbacks. You can pass any number of callbacks.
callbacks_list = [
    # This callback will interrupt training when we have stopped improving
    keras.callbacks.EarlyStopping(
        # This callback will monitor the validation accuracy of the model
        monitor='acc',
        # Training will be interrupted when the accuracy
        # has stopped improving for *more* than 1 epochs (i.e. 2 epochs)
        patience=1,
    ),
    # This callback will save the current weights after every epoch
    keras.callbacks.ModelCheckpoint(
        filepath='my_model.h5', # Path to the destination model file
        # The two arguments below mean that we will not overwrite the
        # model file unless `val_loss` has improved, which
        # allows us to keep the best model every seen during training.
        monitor='val_loss',
        save_best_only=True,
    )
]
```

```
# Since we monitor `acc`, it should be part of the metrics of the model.
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

# Note that since the callback will be monitor validation accuracy,
# we need to pass some `validation_data` to our call to `fit`.
model.fit(x, y,
           epochs=10,
           batch_size=32,
           callbacks=callbacks_list,
           validation_data=(x_val, y_val))
```

THE REDUCELRONPLATEAU CALLBACK

You can use this callback to reduce the learning rate when the validation loss has stopped improving. Reducing or increasing the learning rate in case of a "loss plateau" is an effective strategy to get out of local minima during training.

Listing 7.21 Using the ReduceLROnPlateau Callback

```
callbacks_list = [
    keras.callbacks.ReduceLROnPlateau(
        # This callback will monitor the validation loss of the model
        monitor='val_loss',
        # It will divide the learning by 10 when it gets triggered
        factor=0.1,
        # It will get triggered after the validation loss has stopped improving
        # for at least 10 epochs
        patience=10,
    )
]

# Note that since the callback will be monitor validation loss,
# we need to pass some `validation_data` to our call to `fit`.
model.fit(x, y,
           epochs=10,
           batch_size=32,
           callbacks=callbacks_list,
           validation_data=(x_val, y_val))
```

WRITING YOUR OWN CALLBACK

If you need to take any specific action during training that isn't covered by one of the built-in callbacks, then you should write your own callback.

Callbacks are implemented by subclassing the class `keras.callbacks.Callback`. You can then implement any number of the following transparently-named methods, which get called at various points during training:

Listing 7.22 Overview of Callback methods

```
# Called at the start of every epoch
on_epoch_begin
# Called at the end of every epoch
on_epoch_end

# Called right before processing each batch
on_batch_begin
# Called right after processing each batch
on_batch_end

# Called at the start of training
```

```
on_train_begin
# Called at the end of training
on_train_end
```

These methods all get called with a `logs` argument, a dictionary containing information about what the previous batch, epoch, or training run: training and validation metrics, etc. Additionally, the callback has access to the following attributes:

- `self.model`, the model instance from which the callback is being called.
- `self.validation_data`, the value of what was passed to `fit` as validation data.

Here is a simple example of a custom callback, where we save to disk (as Numpy arrays) the activations of every layer of the model at the end of every epoch, computed on the first sample of the validation set:

Listing 7.23 Writing a custom Callback

```
import keras
import numpy as np

class ActivationLogger(keras.callbacks.Callback):

    def set_model(self, model):
        # This method is called by the parent model
        # before training, to inform the callback
        # of what model will be calling it
        self.model = model
        layer_outputs = [layer.output for layer in model.layers]
        # This is a model instance that returns the activations of every layer
        self.activations_model = keras.models.Model(model.input, layer_outputs)

    def on_epoch_end(self, epoch, logs=None):
        if self.validation_data is None:
            raise RuntimeError('Requires validation_data.')

        # Obtain first input sample of the validation data
        validation_sample = self.validation_data[0][0:1]
        activations = self.activations_model.predict(validation_sample)
        # Save arrays to disk
        f = open('activations_at_epoch_' + str(epoch) + '.npz', 'w')
        np.savez(f, activations)
        f.close()
```

This is all you need to know about callbacks—the rest is technical details, which can be easily looked up. Now you are equipped to perform any sort of logging or pre-programmed intervention on a Keras model during training.

7.2.2 Introduction to TensorBoard: the TensorFlow visualization framework

To do good research, or develop good models, you need to get rich and frequent feedback about what is going on inside your models during your experiments. That's the point of running experiments: to get information about how well a model performs—as much information as possible. Making progress is an iterative process, a loop: you start with an idea, then you express it as an experiment, attempting to validate or invalidate your idea. You run this experiment, then you process the information generated by the experiment. This inspires your next idea. The more iterations of this loop you are able to run, the more refined and powerful your ideas become. Keras helps you go from idea to experiment in the least possible time, and fast GPUs help you get from experiment to result as fast as possible. But what about processing the experiment results? That's where TensorBoard comes in.

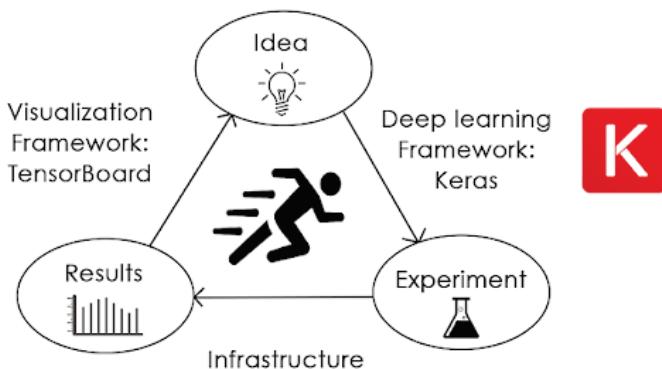


Figure 7.9 The loop of progress

In the next few paragraphs, we introduce TensorBoard, a browser-based visualization tool that comes packaged with TensorFlow. Note that it is only available for your Keras models when you are using Keras with the TensorFlow backend.

The key purpose of TensorBoard is to help you visually monitor everything that goes on inside your model during training. If you are monitoring more information than just the final loss of your model, then you can develop a clearer vision of the model does or doesn't do, and you can make progress faster. TensorBoard gives you access to several neat features, all inside your browser:

- Visually monitoring your metrics during training.
- Visualizing your model architecture.
- Visualizing histograms of activations and gradients.
- Exploring embeddings in 3D.

Let's demonstrate these features on a simple example. We will be training a 1D convnet on the IMDB sentiment analysis task.

Below is our model, similar to the one you have seen in the last section of Chapter 6. We only consider the top 2000 words in the IMDB vocabulary, to make word embedding visualization more tractable.

Listing 7.24 A simple text classification model that we will use with a TensorBoard Callback

```

import keras
from keras import layers
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 2000 # number of words to consider as features
max_len = 500 # cut texts after this number of words (among top max_features most common words)

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)

model = keras.models.Sequential()
model.add(layers.Embedding(max_features, 128, input_length=max_len, name='embed'))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
model.summary()
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

```

Before we start using TensorBoard, we need to create a directory where we will store log files generated by TensorBoard:

Listing 7.25 Create a directory for the log files that the TensorBoard Callback will generate

```
mkdir my_log_dir
```

Let's launch the training, with a TensorBoard callback instance. This callback will write log events to disk at the specified location.

Listing 7.26 Training our model together with a TensorBoard Callback

```

callbacks = [
    keras.callbacks.TensorBoard(
        # Log files will be written at this location
        log_dir='my_log_dir',
        # We will record activation histograms every 1 epoch
        histogram_freq=1,
        # We will record embedding data every 1 epoch
        embeddings_freq=1,
    )
]
history = model.fit(x_train, y_train,
                     epochs=20,
                     batch_size=128,
                     validation_split=0.2,
                     callbacks=callbacks)

```

At this point, you can launch the TensorBoard server from the command line, instructing it to read the logs the callback is currently writing. The `tensorboard` utility should have been automatically installed on your machine the moment you installed

TensorFlow (e.g. via pip).

Listing 7.27 Launching the TensorBoard server from the command line

```
tensorboard --logdir=my_log_dir
```

You can then browse to localhost:6006 and look at your model training:

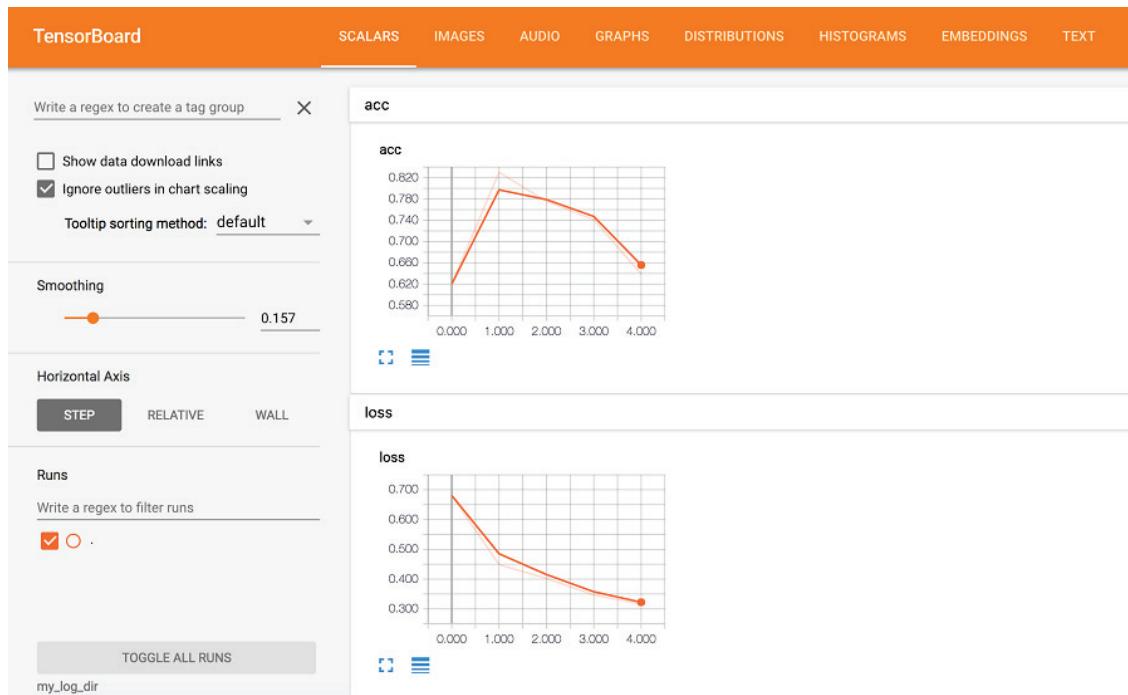


Figure 7.10 TensorBoard: metrics monitoring

Besides live graphs of the training and validation metrics, you get access to the Histograms tab, where you can find pretty visualizations of histograms of activation values taken by your layers:

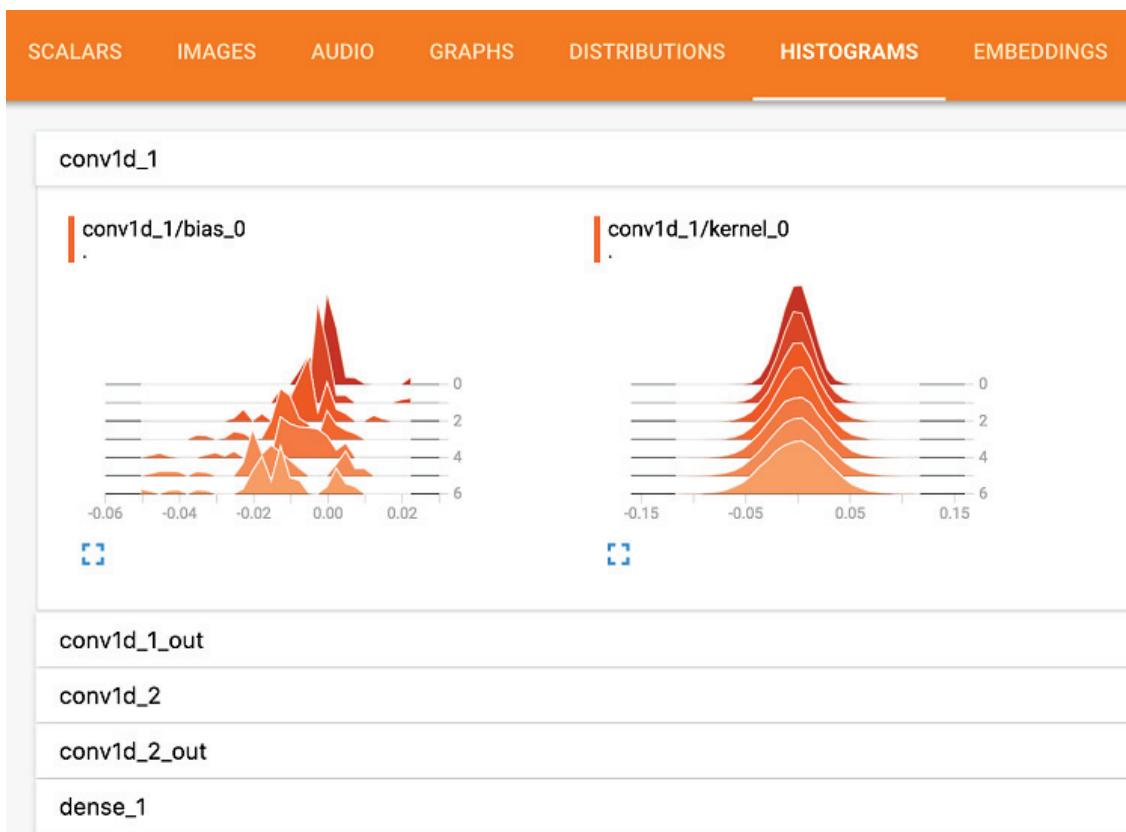


Figure 7.11 TensorBoard: activation histograms

The Embeddings tab gives you a way to inspect the embedding locations and spatial relationships of the 10,000 words in our input vocabulary, as learned by our initial Embedding layer. Since the embedding space is actually 128-dimensional, TensorBoard automatically reduces it 2D or 3D using a dimensionality reduction algorithm of your choice: either PCA or T-SNE. Here, in our point cloud, we can clearly see two clusters: words with a positive connotation and words with a negative connotation. The visualization makes it immediately obvious that embeddings trained jointly with a specific objective result in models that are completely specific to the underlying task—that's the reason why using pre-trained generic word embeddings is rarely a good idea.

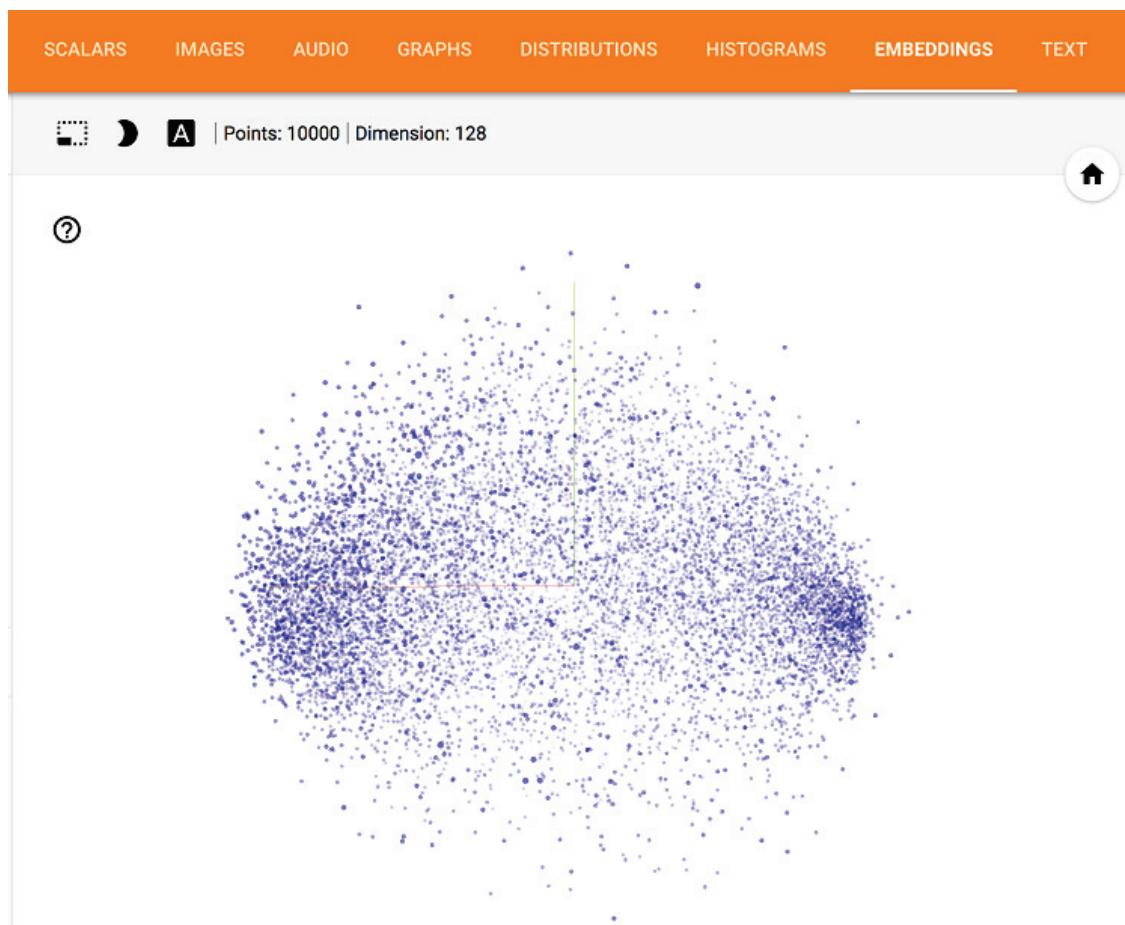


Figure 7.12 TensorBoard: interactive 3D word embedding visualization

The `Graphs` tab shows an interactive visualization of the graph of low-level TensorFlow operations underlying your Keras model:

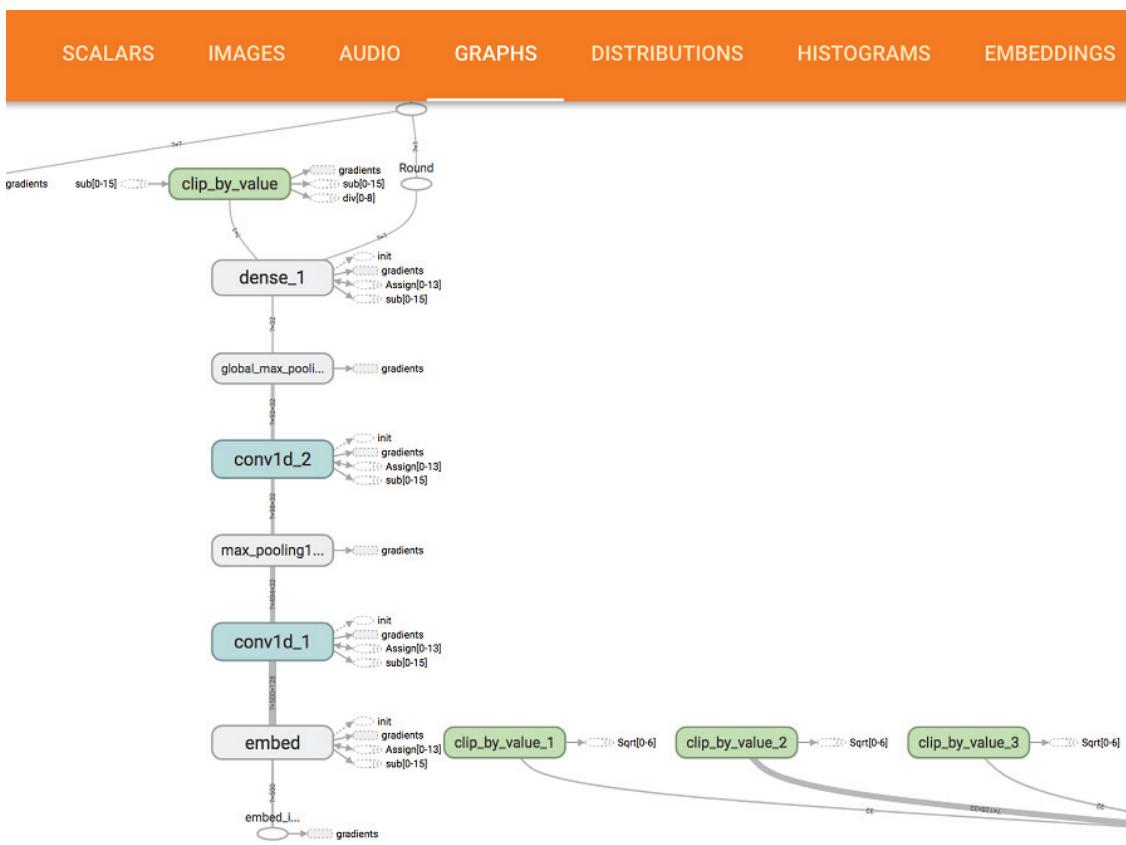


Figure 7.13 TensorBoard: TensorFlow graph visualization

As you can see, there is a lot more going on there than you would expect. The model we just built may look simple when defined in Keras—a small stack of basic layers—but under the hood, we need to construct a fairly complex graph structure to make it work. A lot of it is related to the gradient descent process. This complexity differential between what you see and what you are actually manipulating is precisely the key motivation for using Keras as your way of building models, instead of working with raw TensorFlow to define everything from scratch. Keras makes your workflow dramatically simpler.

Note that Keras also provides another, cleaner way to plot your models, as graphs of layers rather than graphs of TensorFlow ops: the utility `keras.utils.plot_model`. Using it requires to have installed the Python libraries `pydot` or `pydot-ng`, as well as the library `graphviz`. Let's take a quick look:

Listing 7.28 Visualizing model topology using `plot_model`

```
from keras.utils import plot_model
plot_model(model, to_file='model.png')
```

This creates the following PNG image:

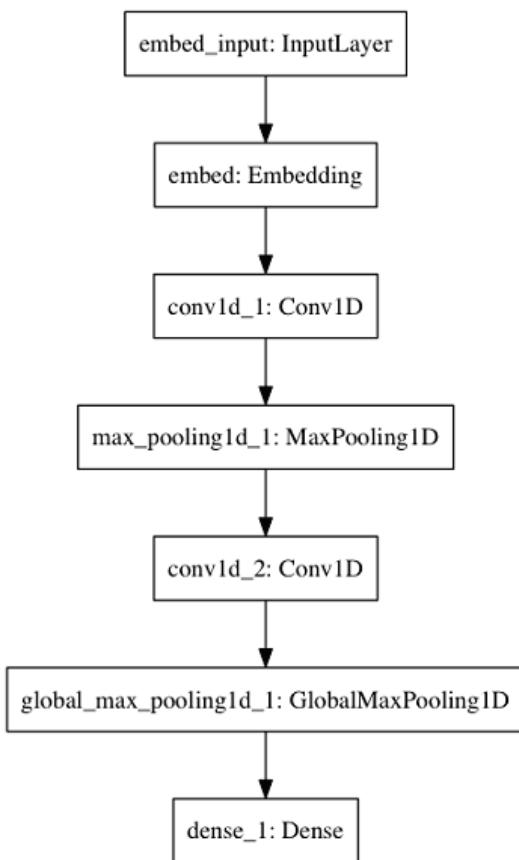


Figure 7.14 A model plot as a graph of layers, generated with `plot_model`

You also have the option of displaying shape information in the graph of layers:

Listing 7.29 Visualizing model topology using `plot_model` and the `show_shapes` option:

```
from keras.utils import plot_model  
plot_model(model, show_shapes=True, to_file='model.png')
```

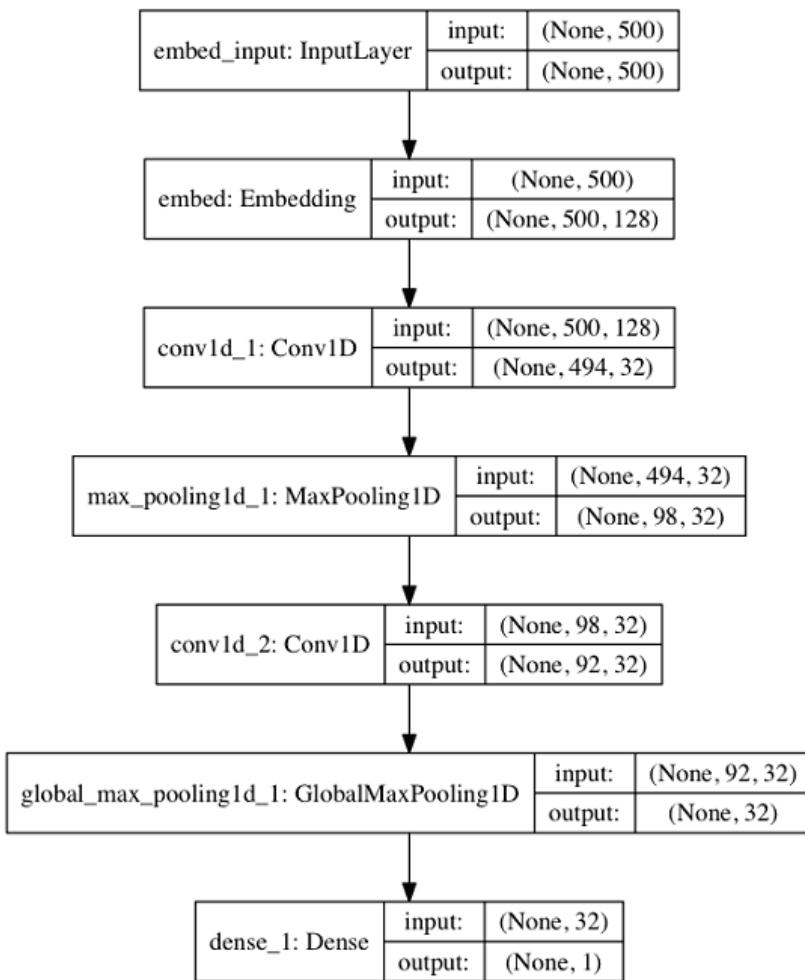


Figure 7.15 A model plot with shape information

7.2.3 Wrapping up

- Keras callbacks provide a simple way to monitor models during training, and automatically take action based on the state of the model.
- When using TensorFlow, TensorBoard is a great way to visualize model activity in your browser. You can use it in Keras models via the `TensorBoard` callback.

7.3 Getting the most out of your models

Trying out architectures blindly works well enough if you just need something that works okay. Here we go beyond "works okay" into "works great and wins machine learning competitions". Here's a quick explainer on a set of must-know techniques for building state-of-the-art deep learning models.

7.3.1 Advanced architecture patterns

We've already covered one important design pattern in detail in the previous section: residual connections. Here are two more that you should know about. These patterns are especially relevant when building high-performing deep convnets. However, they are commonly found in many other types of architectures as well.

BATCH NORMALIZATION

"Normalization" is a broad category of methods that seek to make different samples seen by a machine learning model more similar to each other, which helps the model learn and generalize well to new data. The most common form of data normalization is one that you have already encountered several times in this book already: centering the data on 0 by subtracting the mean from the data, and giving it a unit standard deviation by dividing the data by its standard deviation. In effect, this makes the assumption that the data follows a normal (or Gaussian) distribution, and makes sure that this distribution is centered and scaled to unit variance.

Listing 7.30 Typical data normalization process

```
normalized_data = (data - np.mean(data, axis=...)) / np.std(data, axis=...)
```

In previous examples, we were only normalizing data before feeding it into our models. However, data normalization should still be a concern after every transformation operated by the network: even if the data coming in a `Dense` or `Conv2D` network has 0-mean and unit variance, there are no reasons to expect a priori that this will still be the case for the data coming out.

Batch normalization is a type of layer (`BatchNormalization` in Keras) introduced in 2015 by Ioffe and Szegedy, capable of adaptively normalizing data even as its mean and variance change over time during training. It works by internally maintaining an exponential moving average of the batch-wise mean and variance of the data seen during training. The main effect of batch normalization is that it helps with gradient propagation—much like residual connections—and thus it allows for deeper networks. Some very deep networks can only be trained if they include multiple `BatchNormalization` layers. For instance, `BatchNormalization` is used liberally in many of the advanced convnets architectures that come packaged with Keras, such as ResNet50, InceptionV3 and Xception.

The `BatchNormalization` layer is typically used after a convolutional or densely-connected layer:

Listing 7.31 Using the BatchNormalization layer

```
# After a Conv layer:  
conv_model.add(layers.Conv2D(32, 3, activation='relu'))  
conv_model.add(layers.BatchNormalization())  
  
# After a Dense layer:  
dense_model.add(layers.Dense(32, activation='relu'))  
dense_model.add(layers.BatchNormalization())
```

The `BatchNormalization` layer takes an `axis` argument, which specifies the features axis which should be normalized. This argument defaults to `-1`, the last axis in the input tensor. This is the correct value when using `Dense` layers, `Conv1D` layers, RNN

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

layers, as well as Conv2D layers with `data_format` set to "channels_last". However, in the niche use case of Conv2D layers with `data_format` set to "channels_first", the features axis is axis number 1, and the `axis` argument in BatchNormalization should then be set to 1 accordingly.

A recent improvement over regular batch normalization has been "batch renormalization", introduced by Ioffe in 2017. It offers clear benefits over batch normalization, at no apparent cost. It is still too early to tell, as I am writing these lines, whether it will come to completely supplant batch normalization—but I would say it is rather likely. Even more recently, Klambauer et al introduced "self-normalizing neural networks", which manage to keep data normalized after going through any Dense layer, by using a specific activation function (`selu`) and a specific initializer (`lecun_normal`). This scheme, while highly interesting, is limited to densely-connected networks for now, and its usefulness has not yet been broadly replicated.

DEPTHWISE SEPARABLE CONVOLUTION

What if I told you that there is a layer you can use as a drop-in replacement for Conv2D, that will make your model lighter (fewer trainable weight parameters), faster (fewer floating point operations), and perform a few percent better on its task? That is precisely what the *depthwise separable convolution* layer does (`separableConv2D`). A depthwise separable convolution performs a spatial convolution on each channel of its input, independently, before mixing output channels via a "pointwise" convolution (a 1x1 convolution). This is equivalent to separating the learning of spatial features and the learning of channel-wise features, which makes a lot of sense if you assume that spatial locations in the input are highly correlated, while its different channels are fairly independent. It requires significantly fewer parameters, and involves fewer computations, thus resulting in smaller and speedier models. And because it's a more representationally efficient way to perform convolution, it tends to learn better representations using less data, resulting in better-performing models.

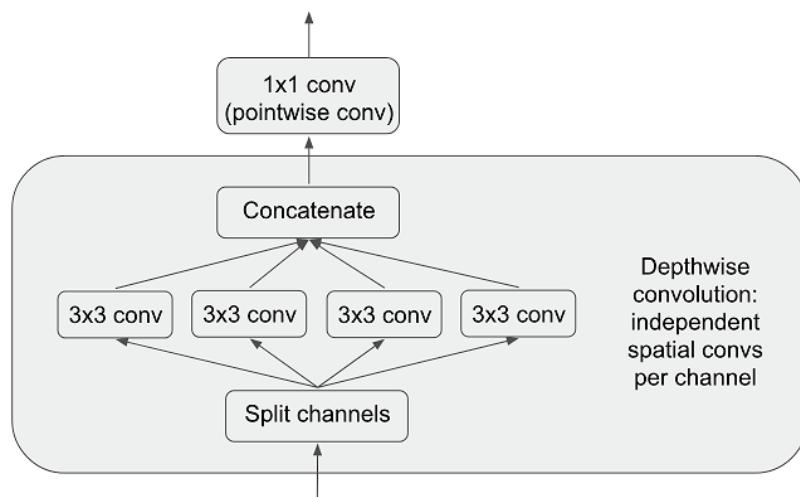


Figure 7.16 A depthwise separable convolution: a depthwise convolution followed by a pointwise convolution

These advantages become especially important when training small models from scratch on limited data. For instance, here is how you would build a lightweight depthwise separable convnet for an image classification task (softmax categorical classification) on a small dataset:

Listing 7.32 A small depthwise separable convnet

```
model = Sequential()
model.add(SeparableConv2D(32, activation='relu', input_shape=(height, width, channels)))
model.add(SeparableConv2D(64, activation='relu'))
model.add(MaxPooling2D(2))

model.add(SeparableConv2D(64, activation='relu'))
model.add(SeparableConv2D(128, activation='relu'))
model.add(MaxPooling2D(2))

model.add(SeparableConv2D(64, activation='relu'))
model.add(SeparableConv2D(128, activation='relu'))
model.add(GlobalAveragePooling2D())

model.add(Dense(32, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
```

When it comes to larger-scale models, depthwise separable convolutions are the basis of the Xception architecture, a high-performing convnet that comes packaged with Keras. You can read more about the theoretical grounding for depthwise separable convolutions and Xception in my paper "*Xception: deep learning with depthwise separable convolutions*" (CVPR 2017).

7.3.2 Hyperparameter optimization

When building a deep learning model, there are many seemingly arbitrary decisions that you have to make: how many layers should you stack? How many units or filters should go into each layer? Should you use `relu` as activation, or a different function? Should you use `BatchNormalization` after a given layer? How much dropout should you use? And so on... These architecture-level parameters are called "hyperparameters", to distinguish them from the parameters of a model, which are trained via backpropagation.

In practice, experienced machine learning engineers and researchers are able to build some intuition over time as to what works and what doesn't when it comes to these choices—they develop "hyperparameter tuning" skills. But there are no formal rules. If you want to get to the very limit of what can be achieved on a given task, you cannot just be content with arbitrary choices made a fallible human. Your initial decisions are almost always suboptimal, even if you have a good intuition. You could refine them by tweaking your choices by hand and retraining your model repeatedly—in fact, that's what machine learning engineers and researchers spend most of their time on. But it shouldn't be your job as a human to fiddle with hyperparameters all day—that is better left to a machine.

Thus, you need to explore the space of possible decisions automatically, systematically, in a principled way. You need to search through architecture space, and

find the best performing ones empirically. That's what the field of automatic hyperparameter optimization is about—it's an entire field of research, and an important one.

The process of optimizing hyperparameters typically looks like this:

- Pick a set of hyperparameters (automatically).
- Build the corresponding model.
- Fit it to your training data, and measure the final performance on the validation data.
- Pick the next set of hyperparameters to try (automatically).
- Repeat. ...
- Eventually measure performance on your test data.

The key to this process is the algorithm that uses this history of validation performance given various sets of hyperparameters to pick the next set of hyperparameters to evaluate. Many different techniques are possible: bayesian optimization, genetic algorithms, simple random search...

Training the weights of a model is relatively easy: you compute a loss function on a mini-batch of data, then use the backpropagation algorithm to move the weights in the right direction. Updating hyperparameters, on the other hand, is extremely challenging. Indeed, consider that:

- Computing the feedback signal (does this set of hyperparameter lead to a high-performing model on this task?) can be extremely expensive: it requires creating and training a new model from scratch on your dataset.
- The hyperparameter space is typically made of discrete decisions, and is thus not continuous, not differentiable. Hence, one typically cannot do gradient descent in hyperparameter space. Instead, one has to rely on gradient-free optimization techniques, which naturally are far less efficient than gradient descent.

Because these challenges are hard and the field is still young, we currently only have access to very limited tools to optimize our models. Often, it turns out that random search (picking the hyperparameters to evaluate at random, repeatedly) is the best solution, despite being the most naive one. However, one tool that I have found reliably better than random search is *Hyperopt*, a Python library for hyperparameter optimization which internally uses trees of Parzen estimators to predict sets of hyperparameters that are likely to work well. Another library called Hyperas integrates Hyperopt for use with Keras models. Do check it out.

One very important issue to keep in mind when doing automatic hyperparameter optimization at scale, is that of validation set overfitting. Since you are updating your hyperparameters based on a signal that is computed using your validation data, you are effectively training them on the validation data, and thus they will quickly overfit to the validation data. Always keep this in mind.

Overall—hyperparameter optimization is a powerful technique that is an absolute requirement to get to state-of-the-art models on any task, or to win machine learning competitions. Think about it: once upon a time, people would handcraft the features that

went into shallow machine learning models. That was very much suboptimal. Now deep learning automates the task of hierarchical feature engineering—features are now learned using a feedback signal, not hand-tuned, and that's the way it should be. In the very same way, we should not handcraft our model architectures, we should optimize them in a principled way. As I write these lines, the field of automatic hyperparameter optimization is still very young and immature, as deep learning was some years ago, but I would expect it to boom in the next few years.

7.3.3 Model ensembling

One last powerful technique for obtain the best possible results on a task is *model ensembling*. Ensembling consists in pooling together the predictions of a set of different models, in order produce better predictions. If you look at machine learning competitions out there, in particular on Kaggle, in all of them the winners are using very large ensembles of models, which inevitable beat any single model, no matter how good.

Ensembling relies on the assumption that different good models trained independently are likely to be good for *different reasons*: each model is looking at slightly different aspects of the data to make its predictions, getting hold of part of the "truth", but not all of it. You may be familiar with the parable of the blind men and the elephant: an ancient story of a group of blind men who come across an elephant for the first time, and try to understand what the elephant is by touching it. Each man touches a different part of the elephant's body—just one part, such as the trunk, or a leg. Then the men describe to each other what an elephant is: "it's like a snake", "like a pillar or a tree"... These blind men are essentially machine learning models trying to understand the manifold of the training data, each from its own perspective, using its own assumptions (provided by the unique architecture of the model and the unique random weight initialization). Each of them gets part of the truth of the data, but not the whole truth. By pooling their perspectives together, one can get a far more accurate description of the data. The elephant is a combination of parts: not any single blind man gets it quite right, but interviewed together, they can tell a fairly accurate story.

Let's use classification as an example. The easiest way to pool together the predictions of a set of classifiers (to "ensemble the classifiers") is to average their predictions at inference time:

Listing 7.33 Naive model ensembling: averaging model predictions

```
# Use 4 different models to compute initial predictions.
preds_a = model_a.predict(x_val)
preds_b = model_b.predict(x_val)
preds_c = model_c.predict(x_val)
preds_d = model_d.predict(x_val)

# This new prediction array should be more accurate
# than any of the initial ones.
final_preds = 0.25 * (preds_a + preds_b + preds_c + preds_d)
```

This will only work if the classifiers are more or less equally good. If one of their is

significantly worse than the other, then the final predictions may not be as good as the best classifier of the group.

A smarter way to ensemble classifiers is to do a weighted average, where the weights are learned on the validation data—typically the better classifiers will be given a higher weight, and the worse classifiers will be given a lower weight. To search for a good set of ensembling weights, one could use random search, or a simple optimization algorithm such as Nelder-Mead.

Listing 7.34 Model ensembling via a weighted average with optimized weights

```
preds_a = model_a.predict(x_val)
preds_b = model_b.predict(x_val)
preds_c = model_c.predict(x_val)
preds_d = model_d.predict(x_val)

# These weights (0.5, 0.25, 0.1, 0.15) are assumed
# to be learned empirically.
final_preds = 0.5 * preds_a + 0.25 * preds_b + 0.1 * preds_c + 0.15 * preds_d
```

There are many possible variants one can imagine: you could do an average of an exponential of the predictions, for instance. In general, a simple weighted average with weights optimized on the validation data provides a very strong baseline.

The key to making ensembling work is the *diversity* of the set of classifiers. Diversity is strength. If all of your blind men had only touched the elephant's trunk, they would agree that elephants are like snakes, and would forever stay ignorant of the truth of the elephant. Diversity is what makes ensembling work. In machine learning terms, if all of your models are biased in the same way, then your ensemble will retain this same bias. If your models are *biased in different ways*, the biases will cancel each other out and the ensemble will be more robust and more accurate.

For this reason, you should be ensembling models that are *as good as possible* while being *as different as possible*. This typically means using very different architectures or even different brands of machine learning approaches altogether. One thing that is largely *not* worth doing, is ensembling a same network trained several times independently, from different random initializations. If the only difference between your models is their random initialization and the order in which they have been exposed to the training data, then your ensemble will be low-diversity and will only provide a tiny improvement over any single model.

One thing that I have found to work well in practice—but which does not generalize to every problem domain—is the use of an ensemble of tree-based methods (such as random forests or gradient boosted trees) and deep neural networks. In 2014, partnering with Andrei Kolev, I took the fourth place in the Higgs Boson decay detection challenge on Kaggle using an ensemble of various tree models and deep neural networks. Remarkably, one of the models in the ensemble originated from a quite different method than the others (it was a Regularized Greedy Forest), and had a significantly worse score than the others. Unsurprisingly, it was assigned a small weight in the ensemble. But to

my surprise, it turned out to improve the overall ensemble by a large factor, simply because it was so different from every other model: it provided information that the other models did not have access to. That's precisely the point of ensembling. It's not so much about how good your best model is, it is about the diversity of your set of candidate models.

In recent times, one style of basic ensemble that has been very successful in practice is the "wide and deep" category of models, blending deep learning with shallow learning, consisting in jointly training a deep neural net with a large linear model. The joint training of a family of diverse models is yet another option to achieve model ensembling.

7.3.4 Wrapping up

- When building high-performing deep convnets, you will need to leverage residual connections, batch normalization, and depthwise separable convolutions. In the future, it is likely that depthwise separable convolutions will end up completely replacing regular convolutions, whether for 1D, 2D or 3D applications, due to their higher representational efficiency.
- Building deep nets requires making many small hyperparameter and architecture choices, which together define how good your model will end up being. Rather than basing these choices on intuition or random chance, it is better to systematically search through hyperparameter space to find optimal choices. At this time, the process is expensive, and the tools to do it are not very good. But maybe the Hyperopt or Hyperas libraries can help you. When doing hyperparameter optimization, be mindful of validation set overfitting!
- Winning machine learning competitions or otherwise obtaining the best possible results on a task can only be done with large ensembles of models. Ensembling via a well-optimized weighted average is usually good enough. Remember: diversity is strength; it is largely pointless to ensemble very similar models, the best ensembles are set of models that are as dissimilar as possible (while having as much predictive power as possible, naturally).

7.4 Wrapping up: advanced deep learning best practices

In this chapter, you have learned:

- How to build models as arbitrary graphs of layers.
- How to reuse layers ("layer weight sharing").
- How to use models as Python functions ("model templating").
- How to use Keras callbacks to monitor your models during training and take action based on model state.
- How to use TensorBoard to visualize metrics, activation histograms, and even embedding spaces.
- What Batch Normalization, Depthwise Separable Convolution, and Residual Connections are.
- Why you should use hyperparameter optimization and model ensembling.

With these new tools, you are better equipped to use deep learning in the real world and start building highly competitive deep learning models.

Generative deep learning



The potential of artificial intelligence to emulate human thought processes goes beyond passive tasks such as object recognition, or mostly reactive tasks such as driving a car. It extends well into creative activities. When I first made the claim that in a not-so-distant future, most of the cultural content that we consume will be created with heavy help from AIs, I was met with utter disbelief, even from long-time machine learning practitioners. That was in 2014. Fast forward three years, and the disbelief has receded—at an incredible speed. In the summer of 2015, you were entertained by Google’s Deep Dream algorithm turning an image into a psychedelic mess of dog eyes and pareidolic artifacts; in 2016 you used the Prisma application to turn your photos into paintings of various styles. In the summer of 2016, a first experimental short movie, *Sunspring*, was directed using a script written by a LSTM—complete with dialogue lines. Maybe you even recently listened to music tentatively generated by a neural network.

Granted, the artistic productions we have seen from AI so far are all fairly low-quality. AI is not anywhere close to rivaling human screenwriters, painters and composers. But replacing humans was always besides the point: artificial intelligence is not about replacing our own intelligence with something else, it is about bringing into our lives and work *more* intelligence, intelligence of a different kind. In many fields, but especially in creative ones, AI will be used by humans as a tool to augment their own capabilities: more *augmented* intelligence than *artificial* intelligence.

A large part of artistic creation consists of simple pattern recognition and technical skill. And that is precisely the part of the process that many find less attractive, even skippable. That’s where AI comes in. Our perceptual modalities, our language, our artworks all have statistical structure. Learning this structure is precisely what deep learning algorithms excel at. Machine learning models can learn the statistical "latent space" of images or music or even stories, and they can then "sample" from this space, creating new artworks with similar characteristics as what the model has seen in its training data. Naturally, such sampling is hardly an act of artistic creation in itself. It is a mere mathematical operation: the algorithm has no grounding in human life, human emotions, our experience of the world; instead it learns from an "experience" that has little in common with ours. It is only our interpretation, as human spectators, that will

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

give meaning to what the model generates. But in the hands of a skilled artist, algorithmic generation can be steered to become meaningful—and beautiful. Latent space sampling can become a brush that empowers the artist, augments our creative affordances, expands the space of what we can imagine. What's more, it can make artistic creation more accessible by eliminating the need for technical skill and practice—setting up a new medium of pure expression, factoring art apart from craft.

Iannis Xenakis, a visionary pioneer of electronic and algorithmic music, beautifully expressed this same idea in the 1960s, in the context of the application of automation technology to music composition:

"Freed from tedious calculations, the composer is able to devote himself to the general problems that the new musical form poses and to explore the nooks and crannies of this form while modifying the values of the input data. For example, he may test all instrumental combinations from soloists to chamber orchestras, to large orchestras. With the aid of electronic computers the composer becomes a sort of pilot: he presses the buttons, introduces coordinates, and supervises the controls of a cosmic vessel sailing in the space of sound, across sonic constellations and galaxies that he could formerly glimpse only as a distant dream."

In this chapter, we will explore under various angles the potential of deep learning to augment artistic creation. We will review sequence data generation (which can be used to generate text or music), Deep Dreams, and image generation using both Variational Auto-Encoders and Generative Adversarial Networks. We will get your computer to dream up content never seen before, and maybe, we will get you to dream too, about the fantastic possibilities that lie at the intersection of technology and art.

You will find five sections in this chapter:

- Text generation with LSTM, where you will use the recurrent networks you discovered in Chapter 7 to dream up a pastiche of Nietzschean philosophy, character by character.
- Deep Dreams, where you will find out what dreams look like when all you know of the world is the ImageNet dataset.
- Neural style transfer, where you will learn to apply the style of a famous painting to your vacation pictures.
- Variational Autoencoders, where you find out about "latent spaces" of images, and how to use them for creating new images.
- Adversarial Networks—deep networks that fight each other in a quest to produce the most realistic pictures possible.

Let's get started.

8.1 Text generation with LSTM

In this section, we present how recurrent neural networks can be used to generate sequence data. We will use text generation as an example, but the exact same techniques can be generalized to any kind of sequence data: you could apply it to sequences of musical notes in order to generate new music, you could apply it to timeseries of brush stroke data (e.g. recorded while an artist paints on an iPad) to generate paintings stroke-by-stroke, and so on.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

Sequence data generation is no way limited to artistic content generation, either. It has been successfully applied to speech synthesis, and dialog generation for chatbots. The "smart reply" feature that Google released in 2016, capable of automatically generating a selection of quick replies to your emails or text messages, is powered by similar techniques.

8.1.1 A brief history of generative recurrent networks

In late 2014, few people had ever heard the abbreviation "LSTM", even in the machine learning community. Successful applications of sequence data generation with recurrent networks only started appearing in the mainstream in 2016. But these techniques actually have a fairly long history, starting with the development of the LSTM algorithm by Hochreiter in 1997. This new algorithm was used early on to generate text character by character.

In 2002, Douglas Eck, then at Schmidhuber's lab in Switzerland, applied LSTM to music generation for the first time, with promising results. Douglas Eck is now a researcher at Google Brain, and in 2016 he started a new research group there, called Magenta, focused on applying modern deep learning techniques to produce engaging music. Sometimes, good ideas take fifteen years to get started.

In the late 2000s and early 2010, Alex Graves did important pioneering work on using recurrent networks for sequence data generation. In particular, his 2013 work on applying Recurrent Mixture Density Networks to generate human-like handwriting using timeseries of pen positions, is seen by some as a turning point. This specific application of neural networks at that specific moment in time captured for me the notion of "machines that dream" and was a significant inspiration around the time I started developing Keras. Alex Graves left a similar commented-out remark hidden in a 2013 LaTeX file uploaded to the preprint server Arxiv.org: "*generating sequential data is the closest computers get to dreaming*". Several years later, we have come to take a lot of these developments for granted, but at the time, it was hard to watch Graves' demonstrations and not walk away awe-inspired by the possibilities.

Since then, recurrent neural networks have been successfully used for music generation, dialogue generation, image generation, speech synthesis, molecule design, and were even used to produce a movie script that was then cast with real live actors.

8.1.2 How can we generate sequence data?

The universal way to generate sequence data in deep learning is to train a network (usually either a RNN or a convnet) to predict the next token or next few tokens in a sequence, using the previous tokens as input. For instance, given the input "*the cat is on the ma*", the network would be trained to predict the target "*t*", the next character. As usual when working with text data, "tokens" are typically words or characters, and any such network that can model the probability of the next token given the previous ones is called a *language model*. A language model captures the *latent space* of language, i.e. its statistical structure.

Once we have such a trained language model, we can *sample* from it, i.e. generate new sequences: we would feed it some initial string of text (called "conditioning data"), ask it to generate the next character or the next word (we could even generate several tokens at once), then add the generated output back to the input data, and repeat the process many times (see Figure 8.1). This loop allows to generate sequences of arbitrary length that reflect the structure of the data that the model was trained on, i.e. sequences that look *almost* like human-written sentences. In our case, we will take a LSTM layer, feed it with strings of N characters extracted from a text corpus, and train it to predict character $N+1$. The output of our model will be a softmax over all possible characters: a probability distribution for the next character. This LSTM would be called a "character-level neural language model".

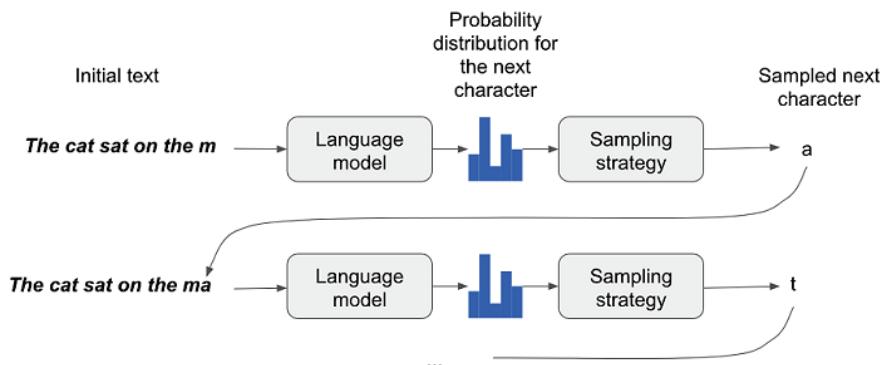


Figure 8.1 The process of character-by-character text generation using a language model

8.1.3 The importance of the sampling strategy

When generating text, the way we pick the next character is crucially important. A naive approach would be "greedy sampling", consisting in always choosing the most likely next character. However, such an approach would result in very repetitive and predictable strings that don't look like coherent language. A more interesting approach would consist in making slightly more surprising choices, i.e. introducing randomness in the sampling process, for instance by sampling from the probability distribution for the next character. This would be called "stochastic sampling" (you recall that "stochasticity" is what we call "randomness" in this field). In such a setup, if "e" has a probability 0.3 of being the next character according to the model, we would pick it 30% of the time. Note that greedy sampling can itself be cast as sampling from a probability distribution: one where a certain character has probability 1 and all others have probability 0.

Sampling probabilistically from the softmax output of the model is neat, as it allows even unlikely characters to be sampled some of the time, generating more interesting-looking sentences and even sometimes showing creativity by coming up with new, realistic-sounding words that didn't occur in the training data. But there is one issue with this strategy: it doesn't offer a way to *control the amount of randomness* in the sampling process.

Why would we want more or less randomness? Consider an extreme case: pure

random sampling, i.e. drawing the next character from a uniform probability distribution, where every character is equally likely. This scheme would have maximum randomness; in other words, this probability distribution would have maximum "entropy". Naturally, it would not produce anything interesting. At the other extreme, greedy sampling, which doesn't produce anything interesting either, has no randomness whatsoever: the corresponding probability distribution has minimum entropy. Sampling from the "real" probability distribution, i.e. the distribution that is output by the model's softmax function, constitutes an intermediate point in between these two extremes. However, there are many other intermediate points of higher or lower entropy that one might want to explore. Less entropy will give the generated sequences a more predictable structure (and thus they will potentially be more realistic-looking) while more entropy will result in more surprising and creative sequences. When sampling from generative models, it is always good to explore different amounts of randomness in the generation process. Since the ultimate judge of the interestingness of the generated data is us, humans, interestingness is highly subjective and there is no telling in advance where the point of optimal entropy lies.

In order to control the amount of stochasticity in the sampling process, let's introduce a parameter called "softmax temperature" that characterizes the entropy of the probability distribution used for sampling, or in other words, that characterizes how surprising or predictable our choice of next character will be. Given a temperature value, a new probability distribution is computed from the original one (the softmax output of the model) by reweighting it in the following way:

Listing 8.1 Reweighting a probability distribution to a different "temperature"

```
import numpy as np

def reweight_distribution(original_distribution, temperature=0.5):
    """Reweight a probability distribution to increase or decrease entropy.

    # Arguments
        original_distribution: A 1D Numpy array of probability values.
            Must sum to one.
        temperature: Factor quantifying the entropy of the output distribution.

    # Returns
        A re-weighted version of the original distribution.
    """
    distribution = np.log(original_distribution) / temperature
    distribution = np.exp(distribution)
    # The sum of the distribution may no longer be 1!
    # Thus we divide it by its sum to obtain the new distribution.
    return distribution / np.sum(distribution)
```

Higher "temperatures" result in sampling distributions of higher entropy, that will generate more surprising and unstructured generated data, while a lower temperature will result in less randomness and much more predictable generated data.

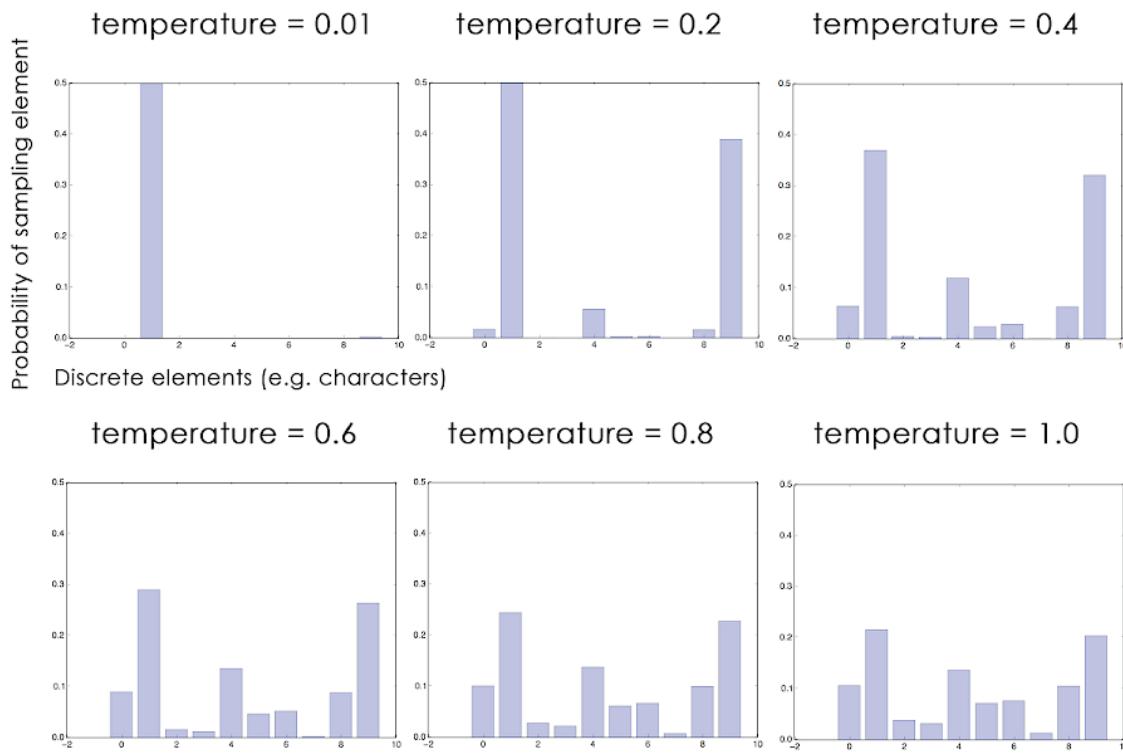


Figure 8.2 Different reweightings of a same probability distribution. Low temperature = more deterministic, high temperature = more random.

8.1.4 Implementing character-level LSTM text generation

Let's put these ideas in practice in a Keras implementation. The first thing we need is a lot of text data that we can use to learn a language model. You could use any sufficiently large text file or set of text files—Wikipedia, the Lord of the Rings, etc. In this example we will use some of the writings of Nietzsche, the late-19th century German philosopher (translated to English). The language model we will learn will thus be specifically a model of Nietzsche's writing style and topics of choice, rather than a more generic model of the English language.

PREPARING THE DATA

Let's start by downloading the corpus and converting it to lowercase:

Listing 8.2 Downloading and parsing our initial text file

```
import keras
import numpy as np

path = keras.utils.get_file(
    'nietzsche.txt',
    origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
text = open(path).read().lower()
print('Corpus length:', len(text))
```

Next, we will extract partially-overlapping sequences of length `maxlen`, one-hot encode them and pack them in a 3D Numpy array `x` of shape (`sequences`, `maxlen`, `unique_characters`). Simultaneously, we prepare a array `y` containing the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

corresponding targets: the one-hot encoded characters that come right after each extracted sequence.

Listing 8.3 Vectorizing sequences of characters

```
# Length of extracted character sequences
maxlen = 60

# We sample a new sequence every `step` characters
step = 3

# This holds our extracted sequences
sentences = []

# This holds the targets (the follow-up characters)
next_chars = []

for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i:i + maxlen])
    next_chars.append(text[i + maxlen])
print('Number of sequences:', len(sentences))

# List of unique characters in the corpus
chars = sorted(list(set(text)))
print('Unique characters:', len(chars))
# Dictionary mapping unique characters to their index in `chars`
char_indices = dict((char, chars.index(char)) for char in chars)

# Next, one-hot encode the characters into binary arrays.
print('Vectorization...')
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
        y[i, char_indices[next_chars[i]]] = 1
```

BUILDING THE NETWORK

Our network is a single LSTM layer followed by a Dense classifier and softmax over all possible characters. But let us note that recurrent neural networks are not the only way to do sequence data generation; 1D convnets also have proven extremely successful at it in recent times.

Listing 8.4 A single-layer LSTM model for next-character prediction

```
from keras import layers

model = keras.models.Sequential()
model.add(layers.LSTM(128, input_shape=(maxlen, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))
```

Since our targets are one-hot encoded, we will use `categorical_crossentropy` as the loss to train the model:

Listing 8.5 The model compilation configuration

```
optimizer = keras.optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

TRAINING THE LANGUAGE MODEL AND SAMPLING FROM IT

Given a trained model and a seed text snippet, we generate new text by repeatedly:

- 1) Drawing from the model a probability distribution over the next character given the text available so far
- 2) Reweighting the distribution to a certain "temperature"
- 3) Sampling the next character at random according to the reweighted distribution
- 4) Adding the new character at the end of the available text

This is the code we use to reweight the original probability distribution coming out of the model, and draw a character index from it (the "sampling function"):

Listing 8.6 Function for sampling the next character given the model's predictions

```
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Finally, this is the loop where we repeatedly train and generated text. We start generating text using a range of different temperatures after every epoch. This allows us to see how the generated text evolves as the model starts converging, as well as the impact of temperature in the sampling strategy.

Listing 8.7 The text generation loop

```
import random
import sys

for epoch in range(1, 60):
    print('epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    model.fit(x, y,
               batch_size=128,
               epochs=1)

    # Select a text seed at random
    start_index = random.randint(0, len(text) - maxlen - 1)
    generated_text = text[start_index: start_index + maxlen]
    print('--- Generating with seed: "' + generated_text + '"')

    for temperature in [0.2, 0.5, 1.0, 1.2]:
        print('----- temperature:', temperature)
        sys.stdout.write(generated_text)

        # We generate 400 characters
        for i in range(400):
            sampled = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(generated_text):
                sampled[0, t, char_indices[char]] = 1.

            preds = model.predict(sampled, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_char = chars[next_index]
```

```

generated_text += next_char
generated_text = generated_text[1:]

sys.stdout.write(next_char)
sys.stdout.flush()
print()

```

Here is what we get at epoch 20, long before the model has fully converged. We used the random seed text "new faculty, and the jubilation reached its climax when kant".

With temperature=0.2:

```

new faculty, and the jubilation reached its climax when kant and such a man in the same time the spir
as a man is the sunligh and subject the present to the superiority of the special pain the most man a
special conscience the special and nature and such men the subjection of the special men, the most s
intellect of the subjection of the same things and

```

With temperature=0.5:

```

new faculty, and the jubilation reached its climax when kant in the eterned and such man as it is als
experience of off the basis the superiory and the special morty of the strength, in the langus, as w
discless the mankind, with a subject and fact all you have to be the stand and lave no comes a trover
conscience the superiority, and when one must be w

```

With temperature=1.0:

```

new faculty, and the jubilation reached its climax when kant, as a periliting of manner to all defin
hicable and ont him artiar resull
too such as if ever the proping to makes as cneience. to been juden, all every could coldiciousnike
which might thiod was account, indifferent germin, that everythery certain destruction, intellect intc
and a lessority o

```

At epoch 60, the model has mostly converged and the text starts looking significantly more coherent:

With temperature=0.2:

```

cheerfulness, friendliness and kindness of a heart are the sense of the spirit is a man with the sens
self-end and self-concerning the subjection of the strengthorixes--the subjection of the subjection c
self-concerning the feelings in the superiority in the subjection of the subjection of the spirit is
subjection and said to the strength of the sense of the

```

With temperature=0.5:

```

cheerfulness, friendliness and kindness of a heart are the part of the soul who have been the art of
will not say, which is it the higher the and with religion of the frences. the life of the spirit amc
strengthre of the sense the conscience of men of precisely before enough presumption, and can mankindc
subjection of the sense and suffering and the

```

With temperature=1.0:

```

cheerfulness, friendliness and kindness of a heart are spiritual by the ciuture for the
entalled is, he astraged, or errors to our you idstood--and it needs, to think by spars to whole the
raals! it was
name, for example but voludd atu-especity"--or rank onee, or even all "solett increessic of the worlc
implussionnal tragedy experience, transf, or insiderar, --must hast
if desires of the strubction is be stronges

```

As you can see, a low temperature results in extremely repetitive and predictable text, but where local structure is highly realistic: in particular, all words (a word being a local pattern of characters) are real English words. With higher temperatures, the generated text becomes more interesting, surprising, even creative; it may sometimes invent completely new words that sound somewhat plausible (such as "eterned" or "troveration"). With a high temperature, the local structure starts breaking down and most words look like semi-random strings of characters. Without a doubt, here 0.5 is the most interesting temperature for text generation in this specific setup. Always experiment with multiple sampling strategies! A clever balance between learned structure and randomness is what makes generation interesting.

Note that by training a bigger model, longer, on more data, you can achieve generated samples that will look much more coherent and realistic than ours. But of course, don't expect to ever generate any meaningful text, other than by random chance: all we are doing is sampling data from a statistical model of which characters come after which characters. Language is a communication channel, and there is a distinction between what communications are about, and the statistical structure of the messages in which communications are encoded. To evidence this distinction, here is a thought experiment: what if human language did a better job at compressing communications, much like our computers do with most of our digital communications? Then language would be no less meaningful, yet it would lack any intrinsic statistical structure, thus making it impossible to learn a language model like we just did.

TAKE AWAYS

- We can generate discrete sequence data by training a model to predict the next tokens(s) given previous tokens.
- In the case of text, such a model is called a "language model" and could be based on either words or characters.
- Sampling the next token requires balance between adhering to what the model judges likely, and introducing randomness.
- One way to handle this is the notion of *softmax temperature*. Always experiment with different temperatures to find the "right" one.

8.2 Deep Dream

"Deep Dream" is an artistic image modification technique that leverages the representations learned by convolutional neural networks. It was first released by Google in the summer of 2015, as an implementation written using the Caffe deep learning library (this was several months before the first public release of TensorFlow). It quickly became an Internet sensation thanks to the trippy pictures it could generate, full of algorithmic pareidolia artifacts, bird feathers and dog eyes—a by-product of the fact that the Deep Dream convnet was trained on ImageNet, where dog breeds and bird species are vastly over-represented.

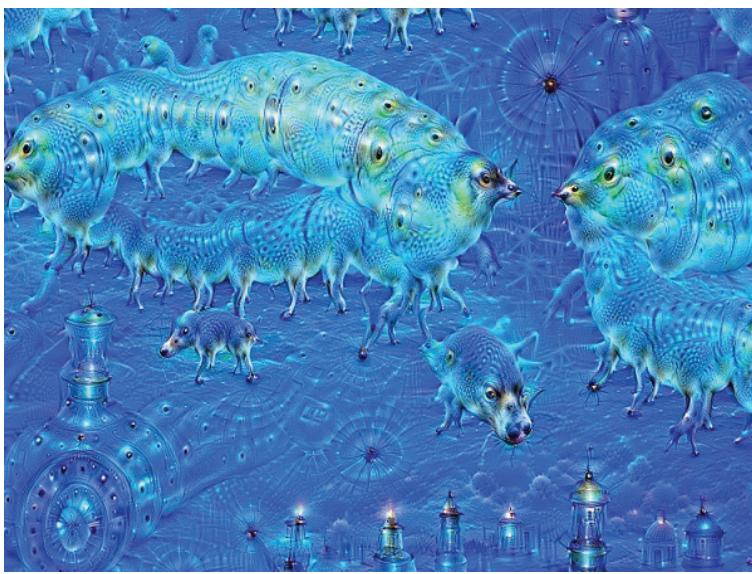


Figure 8.3 Example of a Deep Dream output image

The Deep Dream algorithm is almost identical to the convnet filter visualization technique that we introduced in Chapter 5, consisting in running a convnet "in reverse", i.e. doing gradient ascent on the input to the convnet in order to maximize the activation of a specific filter in an upper layer of the convnet. Deep Dream leverages this same idea, with a few simple differences:

- With Deep Dream, we try to maximize the activation of entire layers rather than that of a specific filter, thus mixing together visualizations of large numbers of features at once.
- We start not from a blank, slightly noisy input, but rather from an existing image—thus the resulting feature visualizations will latch unto pre-existing visual patterns, distorting elements of the image in a somewhat artistic fashion.
- The input images get processed at different scales (called "octaves"), which improves the quality of the visualizations.

Let's make our own Deep Dreams.

8.2.1 Implementing Deep Dream in Keras

We will start from a convnet pre-trained on ImageNet. In Keras, we have many such convnets available: VGG16, VGG19, Xception, ResNet50... albeit the same process is doable with any of these, your convnet of choice will naturally affect your visualizations, since different convnet architectures result in different learned features. The convnet used in the original Deep Dream release was an Inception model, and in practice Inception is known to produce very nice-looking Deep Dreams, so we will use the InceptionV3 model that comes with Keras.

Listing 8.8 Loading the pre-trained InceptionV3 model

```
from keras.applications import inception_v3
from keras import backend as K

# We will not be training our model,
# so we use this command to disable all training-specific operations
K.set_learning_phase(0)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

```
# Build the InceptionV3 network.
# The model will be loaded with pre-trained ImageNet weights.
model = inception_v3.InceptionV3(weights='imagenet',
                                   include_top=False)
```

Next, we compute the "loss", the quantity that we will seek to maximize during the gradient ascent process. In Chapter 5, for filter visualization, we were trying to maximize the value of a specific filter in a specific layer. Here we will simultaneously maximize the activation of all filters in a number of layers. Specifically, we will maximize a weighted sum of the L2 norm of the activations of a set of high-level layers. The exact set of layers we pick (as well as their contribution to the final loss) has a large influence on the visuals that we will be able to produce, so we want to make these parameters easily configurable. Lower layers result in geometric patterns, while higher layers result in visuals in which you can recognize some classes from ImageNet (e.g. birds or dogs). We'll start from a somewhat arbitrary configuration involving four layers—but you will definitely want to explore many different configurations later on:

Listing 8.9 Setting up the Dream configuration

```
# Dict mapping layer names to a coefficient
# quantifying how much the layer's activation
# will contribute to the loss we will seek to maximize.
# Note that these are layer names as they appear
# in the built-in InceptionV3 application.
# You can list all layer names using `model.summary()` .
layer_contributions = {
    'mixed2': 0.2,
    'mixed3': 3.,
    'mixed4': 2.,
    'mixed5': 1.5,
}
```

Now let's define a tensor that contains our loss, i.e. the weighted sum of the L2 norm of the activations of the layers listed above.

Listing 8.10 Defining the loss to be maximized

```
# Get the symbolic outputs of each "key" layer (we gave them unique names).
layer_dict = dict([(layer.name, layer) for layer in model.layers])

# Define the loss.
loss = K.variable(0.)
for layer_name in layer_contributions:
    # Add the L2 norm of the features of a layer to the loss.
    coeff = layer_contributions[layer_name]
    activation = layer_dict[layer_name].output

    # We avoid border artifacts by only involving non-border pixels in the loss.
    scaling = K.prod(K.cast(K.shape(activation), 'float32'))
    loss += coeff * K.sum(K.square(activation[:, 2:-2, 2:-2, :])) / scaling
```

Now we can set up the gradient ascent process:

Listing 8.11 The gradient ascent process

```
# This holds our generated image
dream = model.input

# Compute the gradients of the dream with regard to the loss.
grads = K.gradients(loss, dream)[0]

# Normalize gradients.
grads /= K.maximum(K.mean(K.abs(grads)), 1e-7)

# Set up function to retrieve the value
# of the loss and gradients given an input image.
outputs = [loss, grads]
fetch_loss_and_grads = K.function([dream], outputs)

def eval_loss_and_grads(x):
    outs = fetch_loss_and_grads([x])
    loss_value = outs[0]
    grad_values = outs[1]
    return loss_value, grad_values

def gradient_ascent(x, iterations, step, max_loss=None):
    for i in range(iterations):
        loss_value, grad_values = eval_loss_and_grads(x)
        if max_loss is not None and loss_value > max_loss:
            break
        print('...Loss value at', i, ':', loss_value)
        x += step * grad_values
    return x
```

Finally, here is the actual Deep Dream algorithm.

First, we define a list of "scales" (also called "octaves") at which we will process the images. Each successive scale is larger than previous one by a factor 1.4 (i.e. 40% larger): we start by processing a small image and we increasingly upscale it (Figure 8.4).

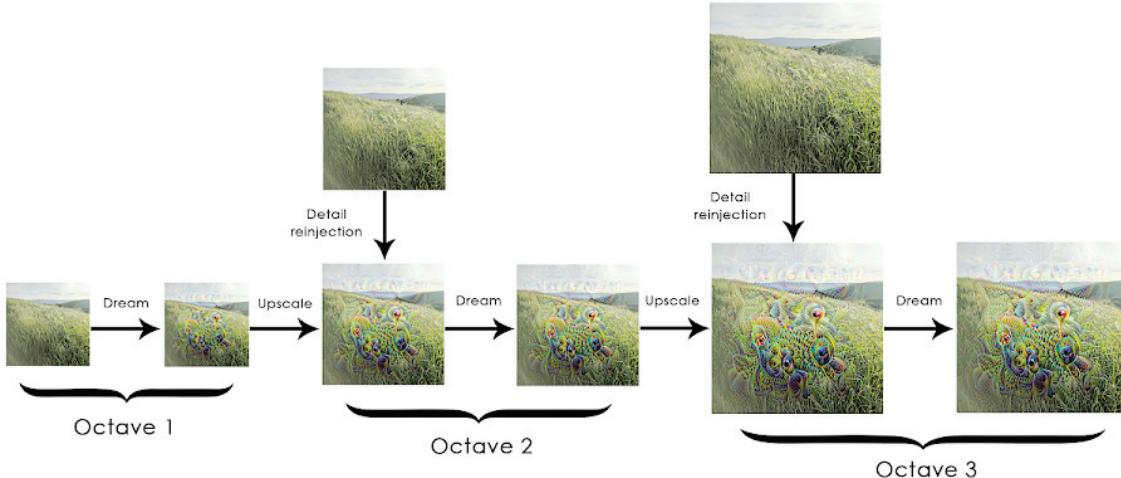


Figure 8.4 The Deep Dream process: successive scales of spatial processing (octaves) and detail reinjection upon upscaling

Then, for each successive scale, from the smallest to the largest, we run gradient ascent to maximize the loss we have previously defined, at that scale. After each gradient ascent run, we upscale the resulting image by 40%.

To avoid losing a lot of image detail after each successive upscaling (resulting in increasingly blurry or pixelated images), we leverage a simple trick: after each upscaling, we reinject the lost details back into the image, which is possible since we know what the

original image should look like at the larger scale. Given a small image S and a larger image size L, we can compute the difference between the original image (assumed larger than L) resized to size L and the original resized to size S—this difference quantifies the details lost when going from S to L.

Listing 8.12 Running gradient ascent over different successive scales

```
import numpy as np

# Playing with these hyperparameters will also allow you to achieve new effects

step = 0.01 # Gradient ascent step size
num_octave = 3 # Number of scales at which to run gradient ascent
octave_scale = 1.4 # Size ratio between scales
iterations = 20 # Number of ascent steps per scale

# If our loss gets larger than 10,
# we will interrupt the gradient ascent process, to avoid ugly artifacts
max_loss = 10.

# Fill this to the path to the image you want to use
base_image_path = '...'

# Load the image into a Numpy array
img = preprocess_image(base_image_path)

# We prepare a list of shape tuples
# defining the different scales at which we will run gradient ascent
original_shape = img.shape[1:3]
successive_shapes = [original_shape]
for i in range(1, num_octave):
    shape = tuple([int(dim / (octave_scale ** i)) for dim in original_shape])
    successive_shapes.append(shape)

# Reverse list of shapes, so that they are in increasing order
successive_shapes = successive_shapes[::-1]

# Resize the Numpy array of the image to our smallest scale
original_img = np.copy(img)
shrunk_original_img = resize_img(img, successive_shapes[0])

for shape in successive_shapes:
    print('Processing image shape', shape)
    img = resize_img(img, shape)
    img = gradient_ascent(img,
                          iterations=iterations,
                          step=step,
                          max_loss=max_loss)
    upscaled_shrunk_original_img = resize_img(shrunk_original_img, shape)
    same_size_original = resize_img(original_img, shape)
    lost_detail = same_size_original - upscaled_shrunk_original_img

    img += lost_detail
    shrunk_original_img = resize_img(original_img, shape)
    save_img(img, fname='dream_at_scale_' + str(shape) + '.png')

save_img(img, fname='final_dream.png')
```

Note that the code above leverages the following straightforward auxiliary Numpy functions, which all do just as their name suggests. They require to have SciPy installed.

Listing 8.13 Auxiliary functions

```
import scipy
from keras.preprocessing import image
```

```
def resize_img(img, size):
    img = np.copy(img)
    factors = (1,
               float(size[0]) / img.shape[1],
               float(size[1]) / img.shape[2],
               1)
    return scipy.ndimage.zoom(img, factors, order=1)

def save_img(img, fname):
    pil_img = deprocess_image(np.copy(img))
    scipy.misc.imsave(fname, pil_img)

def preprocess_image(image_path):
    # Util function to open, resize and format pictures
    # into appropriate tensors.
    img = image.load_img(image_path)
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = inception_v3.preprocess_input(img)
    return img

def deprocess_image(x):
    # Util function to convert a tensor into a valid image.
    if K.image_data_format() == 'channels_first':
        x = x.reshape((3, x.shape[2], x.shape[3]))
        x = x.transpose((1, 2, 0))
    else:
        x = x.reshape((x.shape[1], x.shape[2], 3))
    x /= 2.
    x += 0.5
    x *= 255.
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

Note that because the original InceptionV3 network was trained to recognize concepts in images of size 299x299, and given that the process involves downscaling the images by a reasonable factor, our Deep Dream implementation will produce much better results on images that are somewhere between 300x300 and 400x400. Regardless, it is still possible to run the same code on images of any size and any ratio.

Starting from this photograph (taken in the small hills between the San Francisco bay and the Google campus), we obtain the following Deep Dream:



©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

Figure 8.5 Running our Deep Dream code on an example image

I strongly suggest that you explore what you can do by adjusting which layers you are using in your loss. Layers that are lower in the network contain more local, less abstract representations and will lead to more geometric-looking dream patterns. Layers higher-up will lead to more recognizable visual patterns based on the most common objects found in ImageNet, such as dog eyes, bird feathers, and so on. You can use random generation of the parameters in our `layer_contributions` dictionary in order to quickly explore many different layer combinations.

Here is a range of results obtained using different layer configurations, from an image of a delicious homemade pastry:

**Figure 8.6 Trying a range of Deep Dream configurations on an example image**

8.2.2 Take aways

- Deep Dream consists in running a network "in reverse" to generate inputs based on the representations learned by the convnet.
- The results produced are fun, and share some similarity with the visual artifacts induced in humans by the disruption of the visual cortex via psychedelics.
- Note that the process is not specific to image models, nor even to convnets. It could be done for speech, music, and more.

8.3 Neural style transfer

Besides Deep Dream, another major development in deep learning-driven image modification that happened in the summer of 2015 is neural style transfer, introduced by Leon Gatys et al. The neural style transfer algorithm has undergone many refinements and spawned many variations since its original introduction, including a viral smartphone app, called Prisma. For simplicity, this section focuses on the formulation described in the original paper.

Neural style transfer consists in applying the "style" of a reference image to a target image, while conserving the "content" of the target image:

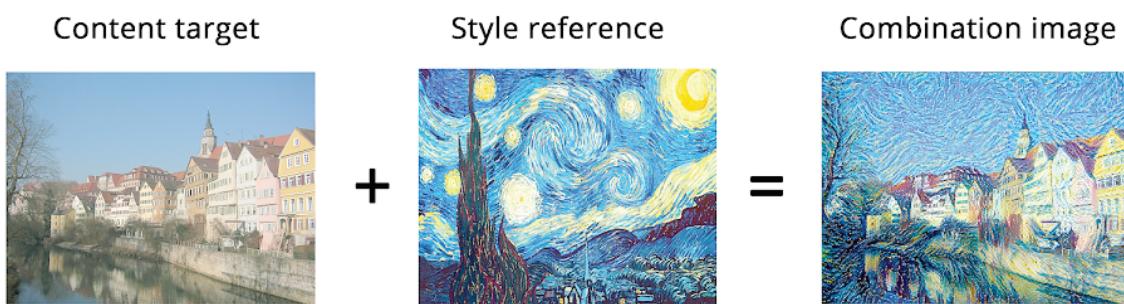


Figure 8.7 A style transfer example

What is meant by "style" is essentially textures, colors, and visual patterns in the image, at various spatial scales, while the "content" is the higher-level macrostructure of the image. For instance, blue-and-yellow circular brush strokes are considered to be the "style" in the above example using Starry Night by Van Gogh, while the buildings in the Tuebingen photograph are considered to be the "content".

The idea of style transfer, tightly related to that of texture generation, has had a long history in the image processing community prior to the development of neural style transfer in 2015. However, as it turned out, the deep learning-based implementations of style transfer offered results unparalleled by what could be previously achieved with classical computer vision techniques, and triggered an amazing renaissance in creative applications of computer vision.

The key notion behind implementing style transfer is same idea that is central to all deep learning algorithms: we define a loss function to specify what we want to achieve, and we minimize this loss. We know what we want to achieve: conserve the "content" of the original image, while adopting the "style" of the reference image. If we were able to mathematically define content and style, then an appropriate loss function to minimize would be the following:

Listing 8.14 Schematic formulation of a "style transfer" loss

```
loss = distance(style(reference_image) - style(generated_image)) +
      distance(content(original_image) - content(generated_image))
```

Where `distance` is a norm function such as the L2 norm, `content` is a function that

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

takes an image and computes a representation of its "content", and `style` is a function that takes an image and computes a representation of its "style".

Minimizing this loss would cause `style(generated_image)` to be close to `style(reference_image)`, while `content(generated_image)` would be close to `content(generated_image)`, thus achieving style transfer as we defined it.

A fundamental observation made by Gatys et al is that deep convolutional neural networks offer precisely a way to mathematically define the `style` and `content` functions. Let's see how.

8.3.1 The content loss

As you already know, activations from earlier layers in a network contain *local* information about the image, while activations from higher layers contain increasingly *global* and *abstract* information. Formulated in a different way, the activations of the different layers of a convnet provide a decomposition of the contents of an image over different spatial scales. Therefore we expect the "content" of an image, which is more global and more abstract, to be captured by the representations of a top layer of a convnet.

A good candidate for a content loss would thus be to consider a pre-trained convnet, and define as our loss the L2 norm between the activations of a top layer computed over the target image and the activations of the same layer computed over the generated image. This would guarantee that, as seen from the top layer of the convnet, the generated image will "look similar" to the original target image. Assuming that what the top layers of a convnet see is really the "content" of their input images, then this does work as a way to preserve image content.

8.3.2 The style loss

While the content loss only leverages a single higher-up layer, the style loss as defined in the Gatys et al. paper leverages multiple layers of a convnet: we aim at capturing the appearance of the style reference image at all spatial scales extracted by the convnet, not just any single scale.

For the style loss, the Gatys et al. paper leverages the "Gram matrix" of a layer's activations, i.e. the inner product between the feature maps of a given layer. This inner product can be understood as representing a map of the correlations between the features of a layer. These feature correlations capture the statistics of the patterns of a particular spatial scale, which empirically corresponds to the appearance of the textures found at this scale.

Hence the style loss aims at preserving similar internal correlations within the activations of different layers, across the style reference image and the generated image. In turn, this guarantees that the textures found at different spatial scales will look similar across the style reference image and the generated image.

8.3.3 In short

In short, we can use a pre-trained convnet to define a loss that will:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

- Preserve content by maintaining similar high-level layer activations between the target content image and the generated image. The convnet should "see" both the target image and the generated image as "containing the same things".
- Preserve style by maintaining similar *correlations* within activations for both low-level layers and high-level layers. Indeed, feature correlations capture *textures*: the generated and the style reference image should share the same textures at different spatial scales.

Now let's take a look at a Keras implementation of the original 2015 neural style transfer algorithm. As you will see, it shares a lot of similarities with the Deep Dream implementation we developed in the previous section.

8.3.4 Neural style transfer in Keras

Neural style transfer can be implemented using any pre-trained convnet. Here we will use the VGG19 network, used by Gatys et al in their paper. VGG19 is a simple variant of the VGG16 network we introduced in Chapter 5, with three more convolutional layers.

This is our general process:

- Set up a network that will compute VGG19 layer activations for the style reference image, the target image, and the generated image at the same time.
- Use the layer activations computed over these three images to define the loss function described above, which we will minimize in order to achieve style transfer.
- Set up a gradient descent process to minimize this loss function.

Let's start by defining the paths to the two images we consider: the style reference image and the target image. To make sure that all images processed share similar sizes (widely different sizes would make style transfer more difficult), we will later resize them all to a shared height of 400px.

Listing 8.15 Defining some initial variables

```
from keras.preprocessing.image import load_img, img_to_array

# This is the path to the image you want to transform.
target_image_path = 'img/portrait.jpg'
# This is the path to the style image.
style_reference_image_path = 'img/transfer_style_reference.jpg'

# Dimensions of the generated picture.
width, height = load_img(target_image_path).size
img_height = 400
img_width = int(width * img_height / height)
```

We will need some auxiliary functions for loading, pre-processing and post-processing the images that will go in and out of the VGG19 convnet:

Listing 8.16 Auxiliary functions

```
import numpy as np
from keras.applications import vgg19

def preprocess_image(image_path):
    img = load_img(image_path, target_size=(img_height, img_width))
    img = img_to_array(img)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

```

img = np.expand_dims(img, axis=0)
img = vgg19.preprocess_input(img)
return img

def deprocess_image(x):
    # Remove zero-center by mean pixel
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    # 'BGR'->'RGB'
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype('uint8')
    return x

```

Let's set up the VGG19 network. It takes as input a batch of three images: the style reference image, the target image, and a placeholder that will contain the generated image. A placeholder is simply a symbolic tensor, the values of which are provided externally via Numpy arrays. The style reference and target image are static, and thus defined using `K.constant`, while the values contained in the placeholder of the generated image will change over time.

Listing 8.17 Loading the pre-trained VGG19 network and applying to our three images

```

from keras import backend as K

target_image = K.constant(preprocess_image(target_image_path))
style_reference_image = K.constant(preprocess_image(style_reference_image_path))

# This placeholder will contain our generated image
combination_image = K.placeholder((1, img_height, img_width, 3))

# We combine the 3 images into a single batch
input_tensor = K.concatenate([target_image,
                             style_reference_image,
                             combination_image], axis=0)

# We build the VGG19 network with our batch of 3 images as input.
# The model will be loaded with pre-trained ImageNet weights.
model = vgg19.VGG19(input_tensor=input_tensor,
                     weights='imagenet',
                     include_top=False)
print('Model loaded.')

```

Let's define the content loss, meant to make sure that the top layer of the VGG19 convnet will have a similar view of the target image and the generated image:

Listing 8.18 The content loss, operating on the features of the target image and the generated "combination" image

```

def content_loss(base, combination):
    return K.sum(K.square(combination - base))

```

Now, here's the style loss. It leverages an auxiliary function to compute the Gram matrix of an input matrix, i.e. a map of the correlations found in the original feature matrix.

Listing 8.19 The style loss, operating on the features of the style reference image and the generated "combination" image

```

def gram_matrix(x):
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram

def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_height * img_width
    return K.sum(K.square(S - C)) / (4. * (channels ** 2) * (size ** 2))

```

To these two loss components, we add a third one, the "total variation loss". It is meant to encourage spatial continuity in the generated image, thus avoiding overly pixelated results. You could interpret it as a regularization loss.

Listing 8.20 The total variation loss, operating on the pixels of the generated "combination" image

```

def total_variation_loss(x):
    a = K.square(
        x[:, :img_height - 1, :img_width - 1, :] - x[:, 1:, :img_width - 1, :])
    b = K.square(
        x[:, :img_height - 1, :img_width - 1, :] - x[:, :img_height - 1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))

```

The loss that we minimize is a weighted average of these three losses. To compute the content loss, we only leverage one top layer, the `block5_conv2` layer, while for the style loss we use a list of layers than spans both low-level and high-level layers. We add the total variation loss at the end.

Depending on the style reference image and content image you are using, you will likely want to tune the `content_weight` coefficient, the contribution of the content loss to the total loss. A higher `content_weight` means that the target content will be more recognizable in the generated image.

Listing 8.21 Defining the final loss that we will minimize

```

# Dict mapping layer names to activation tensors
outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
# Name of layer used for content loss
content_layer = 'block5_conv2'
# Name of layers used for style loss
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']
# Weights in the weighted average of the loss components
total_variation_weight = 1e-4
style_weight = 1.
content_weight = 0.025

# Define the loss by adding all components to a `loss` variable
loss = K.variable(0.)

```

```

layer_features = outputs_dict[content_layer]
target_image_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :]
loss += content_weight * content_loss(target_image_features,
                                         combination_features)
for layer_name in style_layers:
    layer_features = outputs_dict[layer_name]
    style_reference_features = layer_features[1, :, :, :]
    combination_features = layer_features[2, :, :, :]
    sl = style_loss(style_reference_features, combination_features)
    loss += (style_weight / len(style_layers)) * sl
loss += total_variation_weight * total_variation_loss(combination_image)

```

Finally, we set up the gradient descent process. In the original Gatys et al. paper, optimization is performed using the L-BFGS algorithm, so that is also what we will use here. This is a key difference from the Deep Dream example in the previous section. The L-BFGS algorithms comes packaged with SciPy. However, there are two slight limitations with the SciPy implementation:

- It requires to be passed the value of the loss function and the value of the gradients as two separate functions.
- It can only be applied to flat vectors, whereas we have a 3D image array.

It would be very inefficient for us to compute the value of the loss function and the value of gradients independently, since it would lead to a lot of redundant computation between the two. We would be almost twice slower than we could be by computing them jointly. To by-pass this, we set up a Python class named `Evaluator` that will compute both loss value and gradients value at once, will return the loss value when called the first time, and will cache the gradients for the next call.

Listing 8.22 Setting up the gradient descent process

```

# Get the gradients of the generated image wrt the loss
grads = K.gradients(loss, combination_image)[0]

# Function to fetch the values of the current loss and the current gradients
fetch_loss_and_grads = K.function([combination_image], [loss, grads])

class Evaluator(object):

    def __init__(self):
        self.loss_value = None
        self.grads_values = None

    def loss(self, x):
        assert self.loss_value is None
        x = x.reshape((1, img_height, img_width, 3))
        outs = fetch_loss_and_grads([x])
        loss_value = outs[0]
        grad_values = outs[1].flatten().astype('float64')
        self.loss_value = loss_value
        self.grads_values = grad_values
        return self.loss_value

    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.grads_values)
        self.loss_value = None
        self.grads_values = None
        return grad_values

```

```
evaluator = Evaluator()
```

Finally, we can run the gradient ascent process using SciPy's L-BFGS algorithm, saving the current generated image at each iteration of the algorithm (here, a single iteration represents 20 steps of gradient ascent):

Listing 8.23 The style transfer loop

```
from scipy.optimize import fmin_l_bfgs_b
from scipy.misc import imsave
import time

result_prefix = 'my_result'
iterations = 20

# Run scipy-based optimization (L-BFGS) over the pixels of the generated image
# so as to minimize the neural style loss.
# This is our initial state: the target image.
# Note that `scipy.optimize.fmin_l_bfgs_b` can only process flat vectors.
x = preprocess_image(target_image_path)
x = x.flatten()
for i in range(iterations):
    print('Start of iteration', i)
    start_time = time.time()
    x, min_val, info = fmin_l_bfgs_b(evaluator.loss, x,
                                       fpprime=evaluator.grads, maxfun=20)
    print('Current loss value:', min_val)
    # Save current generated image
    img = x.copy().reshape((img_height, img_width, 3))
    img = deprocess_image(img)
    fname = result_prefix + '_at_iteration_%d.png' % i
    imsave(fname, img)
    end_time = time.time()
    print('Image saved as', fname)
    print('Iteration %d completed in %ds' % (i, end_time - start_time))
```

Here's what we get:

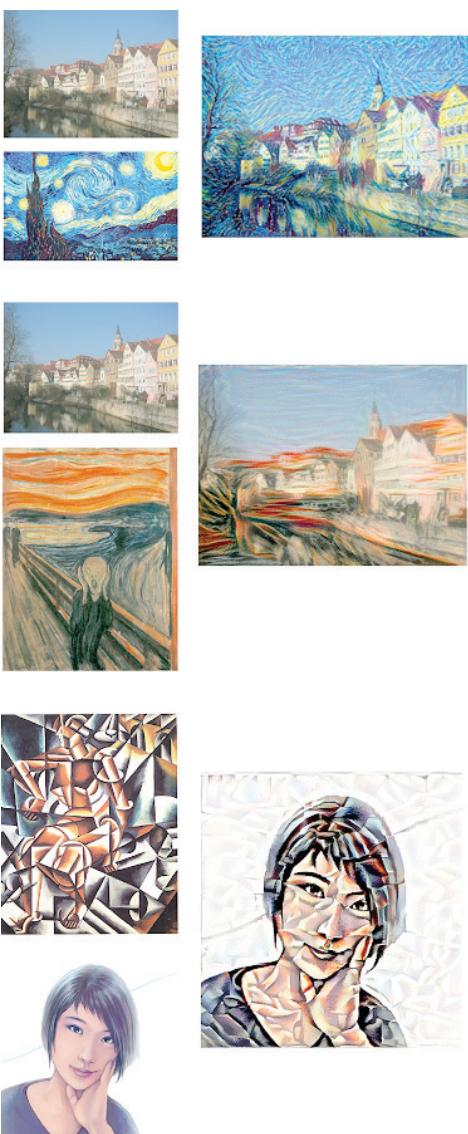


Figure 8.8 Some example results

Keep in mind that what this technique achieves is merely a form of image re-texturing, or texture transfer. It will work best with style reference images that are strongly textured and highly self-similar, and with content targets that don't require high levels of details in order to be recognizable. It would typically not be able to achieve fairly abstract feats such as "transferring the style of one portrait to another". The algorithm is closer to classical signal processing than to AI, so don't expect it to work like magic!

Additionally, do note that running this style transfer algorithm is quite slow. However, the transformation operated by our setup is simple enough that it can be learned by a small, fast feedforward convnet as well—as long as you have appropriate training data available. Fast style transfer can thus be achieved by first spending a lot of compute cycles to generate input-output training examples for a fixed style reference

image, using the above method, and then training a simple convnet to learn this style-specific transformation. Once that is done, stylizing a given image is instantaneous: it's just a forward pass of this small convnet.

8.3.5 Take aways

- Style transfer consists in creating a new image that preserves the "contents" of a target image while also capturing the "style" of a reference image.
- "Content" can be captured by the high-level activations of a convnet.
- "Style" can be captured by the internal correlations of the activations of different layers of a convnet.
- Hence deep learning allows style transfer to be formulated as an optimization process using a loss defined with a pre-trained convnet.
- Starting from this basic idea, many variants and refinements are possible!

8.4 Generating images with Variational Autoencoders

Sampling from a latent space of images to create entirely new images, or edit existing ones, is currently the most popular and successful application of creative AI. In this section and the next one, we review some of the high-level concepts pertaining to image generation, alongside implementations details relative to the two main techniques in this domain: Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs). The techniques we present here are not specific to images—one could develop latent spaces of sound, music, or even text, using GANs or VAEs—but in practice the most interesting results have been obtained with pictures, and that is what we focus on here.

8.4.1 Sampling from latent spaces of images

The key idea of image generation is to develop a low-dimensional *latent space* of representations (which naturally is a vector space, i.e. a geometric space), where any point can be mapped to a realistic-looking image. The module capable of realizing this mapping, taking as input a latent point and outputting an image, i.e. a grid of pixels, is called a generator (in the case of GANs) or a decoder (in the case of VAEs). Once such a latent space has been developed, one may sample points from it, either deliberately or at random, and by mapping them to image space, generate images never seen before.

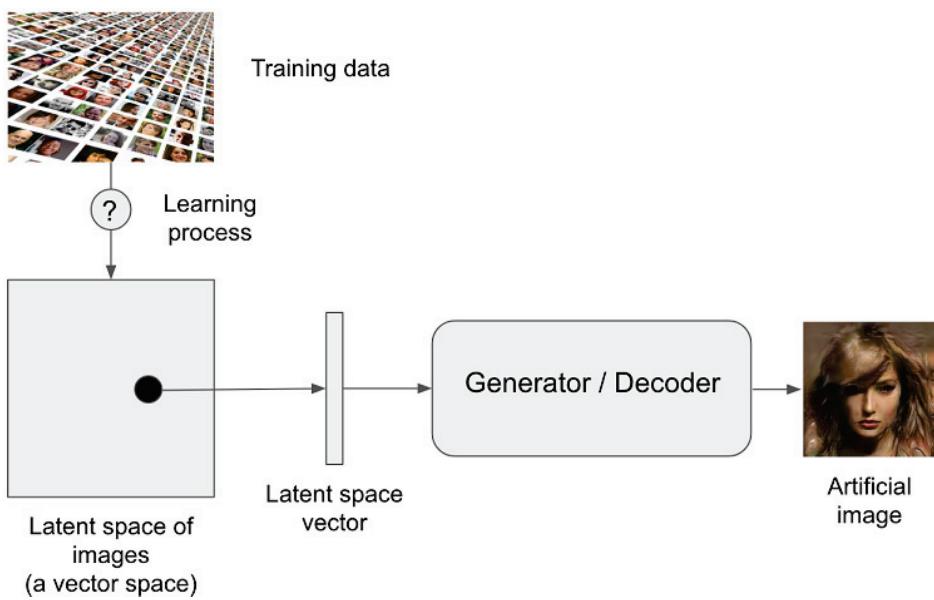


Figure 8.9 Learning a latent vector space of images and using it to sample new images

GANs and VAEs are simply two different strategies for learning such latent spaces of image representations, with each its own characteristics. VAEs are great for learning latent spaces that are well-structured, where specific directions encode a meaningful axis of variation in the data. GANs generate images that can potentially be highly realistic, but the latent space they come from may not have as much structure and continuity.

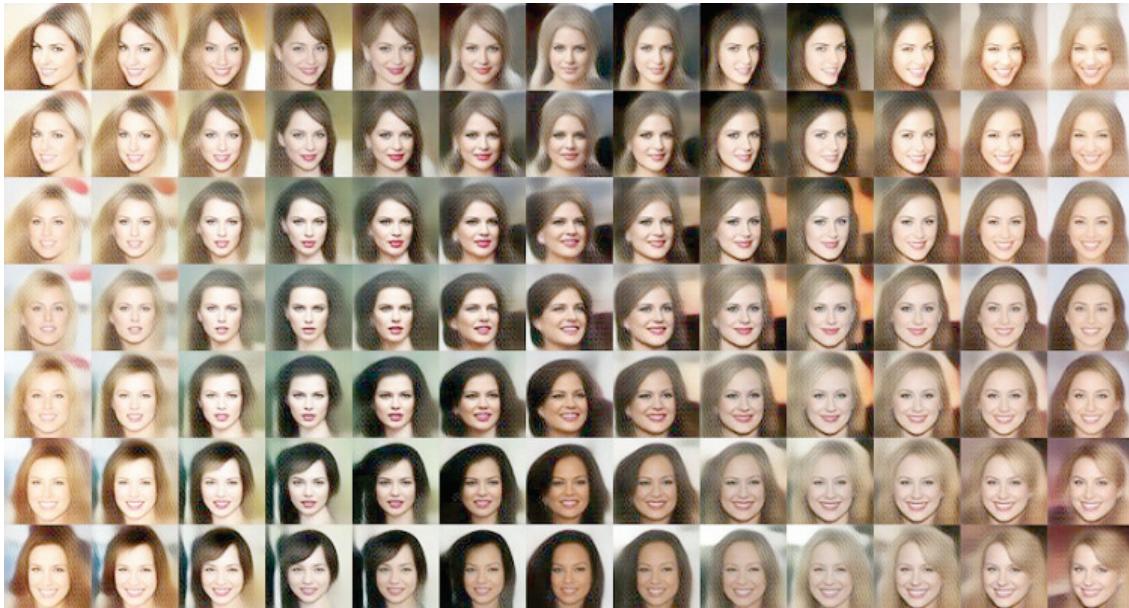


Figure 8.10 A continuous space of faces generated by Tom White using VAEs

8.4.2 Concept vectors for image editing

We already hinted at the idea of a "concept vector" when we covered word embeddings in Chapter 6. The idea is still the same: given a latent space of representations, or an embedding space, certain directions in the space may encode interesting axes of variation in the original data. In a latent space of images of faces, for instance, there may be a "smile vector" s , such that if latent point z is the embedded representation of a certain face, then latent point $z + s$ is the embedded representation of the same face, smiling. Once one has identified such a vector, it then becomes possible to edit images by projecting them into the latent space, moving their representation in a meaningful way, then decoding them back to image space. There are concept vectors for essentially any independent dimension of variation in image space—in the case of faces, one may discover vectors for adding sunglasses to a face, removing glasses, turning a male face into female face, etc.

Here is an example of a "smile vector", a concept vector discovered by Tom White from the Victoria University School of Design in New Zealand, using VAEs trained on a dataset of faces of celebrities (the CelebA dataset):



Figure 8.11 The smile vector

8.4.3 Variational autoencoders

Variational autoencoders, simultaneously discovered by Kingma & Welling in December 2013, and Rezende, Mohamed & Wierstra in January 2014, are a kind of generative model that is especially appropriate for the task of image editing via concept vectors. They are a modern take on autoencoders—a type of network that aims to "encode" an input to a low-dimensional latent space then "decode" it back—that mixes ideas from deep learning with Bayesian inference.

A classical image autoencoder takes an image, maps it to a latent vector space via an "encoder" module, then decode it back to an output with the same dimensions as the original image, via a "decoder" module. It is then trained by using as target data the *same images* as the input images, meaning that the autoencoder learns to reconstruct the

original inputs. By imposing various constraints on the "code", i.e. the output of the encoder, one can get the autoencoder to learn more or less interesting latent representations of the data. Most commonly, one would constraint the code to be very low-dimensional and sparse (i.e. mostly zeros), in which case the encoder acts as a way to compress the input data into fewer bits of information.

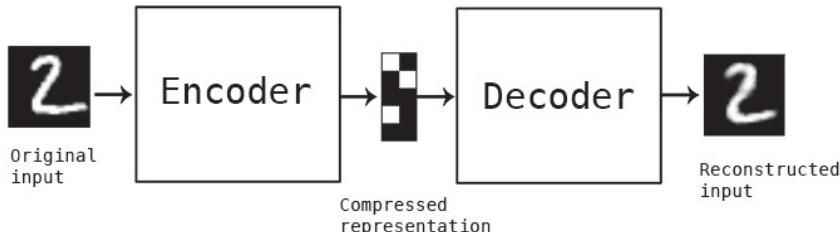


Figure 8.12 An autoencoder: mapping an input x to a compressed representation, then decoding it back as x' .

In practice, such classical autoencoders don't lead to particularly useful or well-structured latent spaces. They're not particularly good at compression, either. For these reasons, they have largely fallen out of fashion over the past years. Variational autoencoders, however, augment autoencoders with a little bit of statistical magic that forces them to learn continuous, highly structured latent spaces. They have turned out to be a very powerful tool for image generation.

A VAE, instead of compressing its input image into a fixed "code" in the latent space, turns the image into the parameters of a statistical distribution: a mean and a variance. Essentially, this means that we are assuming that the input image has been generated by a statistical process, and that the randomness of this process should be taken into account during encoding and decoding. The VAE then uses the mean and variance parameters to randomly sample one element of the distribution, and decodes that element back to the original input. The stochasticity of this process improves robustness and forces the latent space to encode meaningful representations everywhere, i.e. every point sampled in the latent will be decoded to a valid output.

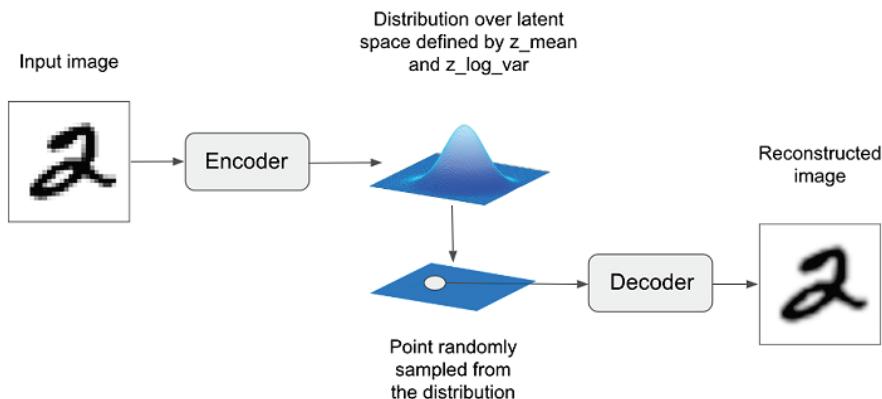


Figure 8.13 A VAE maps an image to two vectors, z_mean and z_log_sigma , which define a probability distribution over the latent space, used to sample a latent point to decode.

In technical terms, here is how a variational autoencoder works. First, an encoder ©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

module turns the input samples `input_img` into two parameters in a latent space of representations, which we will note `z_mean` and `z_log_variance`. Then, we randomly sample a point `z` from the latent normal distribution that is assumed to generate the input image, via `z = z_mean + exp(z_log_variance) * epsilon`, where `epsilon` is a random tensor of small values. Finally, a decoder module will map this point in the latent space back to the original input image. Because `epsilon` is random, the process ensures that every point that is close to the latent location where we encoded `input_img` (`z-mean`) can be decoded to something similar to `input_img`, thus forcing the latent space to be continuously meaningful. Any two close points in the latent space will decode to highly similar images. Continuity, combined with the low dimensionality of the latent space, forces every direction in the latent space to encode a meaningful axis of variation of the data, making the latent space very structured and thus highly suitable to manipulation via concept vectors.

The parameters of a VAE are trained via two loss functions: first, a reconstruction loss that forces the decoded samples to match the initial inputs, and a regularization loss, which helps in learning well-formed latent spaces and reducing overfitting to the training data.

Let's quickly go over a Keras implementation of a VAE. Schematically, it looks like this:

Listing 8.24 Schematic formulation of a Variational Autoencoder (VAE)

```
# Encode the input into a mean and variance parameter
z_mean, z_log_variance = encoder(input_img)

# Draw a latent point using a small random epsilon
z = z_mean + exp(z_log_variance) * epsilon

# Then decode z back to an image
reconstructed_img = decoder(z)

# Instantiate a model
model = Model(input_img, reconstructed_img)

# Then train the model using 2 losses:
# a reconstruction loss and a regularization loss
```

Here is the encoder network we will use: a very simple convnet which maps the input image `x` to two vectors, `z_mean` and `z_log_variance`.

Listing 8.25 The VAE encoder network, mapping images to the parameters of a probability distribution over the latent space

```
import keras
from keras import layers
from keras import backend as K
from keras.models import Model
import numpy as np

img_shape = (28, 28, 1)
batch_size = 16
latent_dim = 2 # Dimensionality of the latent space: a plane
```

```

input_img = keras.Input(shape=img_shape)

x = layers.Conv2D(32, 3,
                  padding='same', activation='relu')(input_img)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu',
                  strides=(2, 2))(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
shape_before_flattening = K.int_shape(x)

x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)

z_mean = layers.Dense(latent_dim)(x)
z_log_var = layers.Dense(latent_dim)(x)

```

Here is the code for using `z_mean` and `z_log_var`, the parameters of the statistical distribution assumed to have produced `input_img`, to generate a latent space point `z`. Here, we wrap some arbitrary code (built on top of Keras backend primitives) into a Lambda layer. In Keras, everything needs to be a layer, so code that isn't part of a built-in layer should be wrapped in a Lambda (or else, in a custom layer).

Listing 8.26 The latent space sampling function

```

def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                               mean=0., stddev=1.)
    return z_mean + K.exp(z_log_var) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_var])

```

This is the decoder implementation: we reshape the vector `z` to the dimensions of an image, then we use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

Listing 8.27 The VAE decoder network, mapping latent space points to images

```

# This is the input where we will feed `z`.
decoder_input = layers.Input(K.int_shape(z)[1:])

# Upsample to the correct number of units
x = layers.Dense(np.prod(shape_before_flattening[1:]),
                 activation='relu')(decoder_input)

# Reshape into an image of the same shape as before our last `Flatten` layer
x = layers.Reshape(shape_before_flattening[1:])(x)

# We then apply then reverse operation to the initial
# stack of convolution layers: a `Conv2DTranspose` layers
# with corresponding parameters.
x = layers.Conv2DTranspose(32, 3,
                         padding='same', activation='relu',
                         strides=(2, 2))(x)
x = layers.Conv2D(1, 3,
                 padding='same', activation='sigmoid')(x)
# We end up with a feature map of the same size as the original input.

```

```
# This is our decoder model.
decoder = Model(decoder_input, x)

# We then apply it to `z` to recover the decoded `z`.
z_decoded = decoder(z)
```

The dual loss of a VAE doesn't fit the traditional expectation of a sample-wise function of the form `loss(input, target)`. Thus, we set up the loss by writing a custom layer with internally leverages the built-in `add_loss` layer method to create an arbitrary loss.

Listing 8.28 A custom layer used to compute the VAE loss

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

    def vae_loss(self, x, z_decoded):
        x = K.flatten(x)
        z_decoded = K.flatten(z_decoded)
        xent_loss = keras.metrics.binary_crossentropy(x, z_decoded)
        kl_loss = -5e-4 * K.mean(
            1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
        return K.mean(xent_loss + kl_loss)

    def call(self, inputs):
        x = inputs[0]
        z_decoded = inputs[1]
        loss = self.vae_loss(x, z_decoded)
        self.add_loss(loss, inputs=inputs)
        # We don't use this output.
        return x

    # We call our custom layer on the input and the decoded output,
    # to obtain the final model output.
y = CustomVariationalLayer()([input_img, z_decoded])
```

Finally, we instantiate and train the model. Since the loss has been taken care of in our custom layer, we don't specify an external loss at compile time (`loss=None`), which in turns means that we won't pass target data during training (as you can see we only pass `x_train` to the model in `fit`).

Listing 8.29 Training the VAE

```
from keras.datasets import mnist

vae = Model(input_img, y)
vae.compile(optimizer='rmsprop', loss=None)
vae.summary()

# Train the VAE on MNIST digits
(x_train, _), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.astype('float32') / 255.
x_test = x_test.reshape(x_test.shape + (1,))

vae.fit(x=x_train, y=None,
        shuffle=True,
        epochs=10,
        batch_size=batch_size,
        validation_data=(x_test, None))
```

Once such a model is trained—e.g. on MNIST, in our case—we can use the decoder network to turn arbitrary latent space vectors into images:

Listing 8.30 Sampling a grid of points from the 2D latent space and decoding them to images

```
import matplotlib.pyplot as plt
from scipy.stats import norm

# Display a 2D manifold of the digits
n = 15 # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
# Linearly spaced coordinates on the unit square were transformed
# through the inverse CDF (ppf) of the Gaussian
# to produce values of the latent variables z,
# since the prior of the latent space is Gaussian
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
        z_sample = np.tile(z_sample, batch_size).reshape(batch_size, 2)
        x_decoded = decoder.predict(z_sample, batch_size=batch_size)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure, cmap='Greys_r')
plt.show()
```

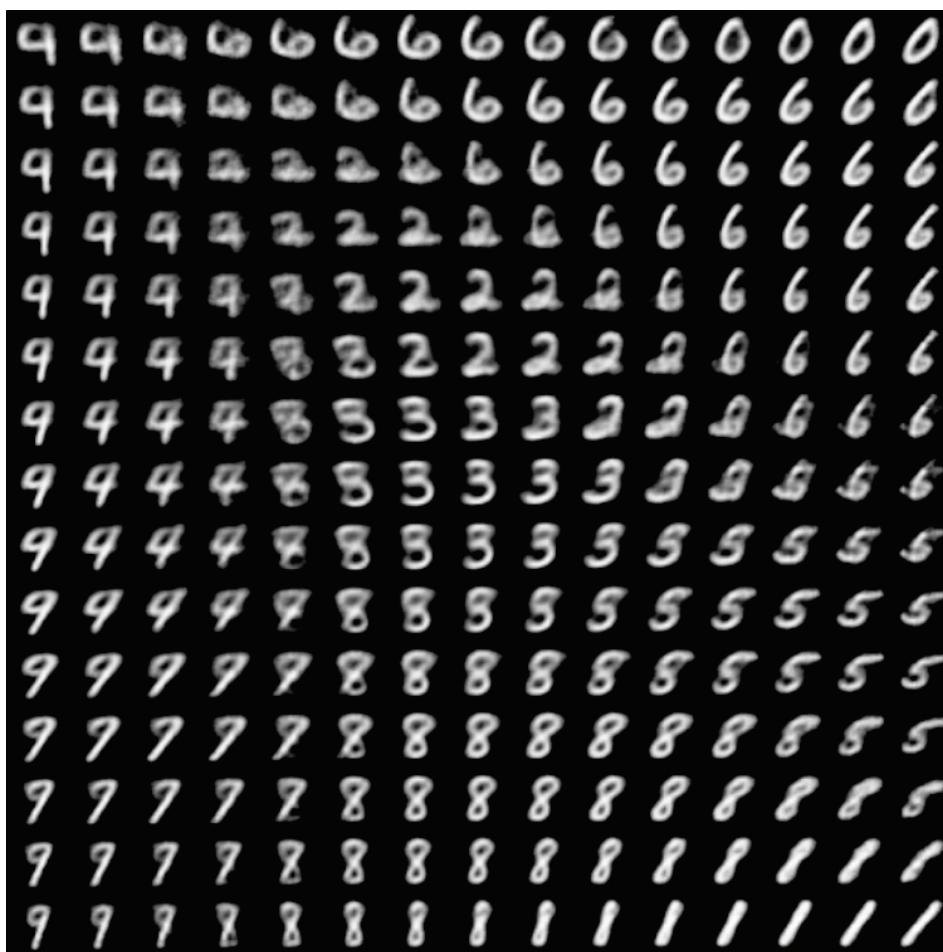


Figure 8.14 Grid of digits decoded from the latent space

The grid of sampled digits shows a completely continuous distribution of the different digit classes, with one digit morphing into another as you follow a path through latent space. Specific directions in this space have a meaning, e.g. there is a direction for "four-ness", "one-ness", etc.

In the next section, we cover in detail the other major tool for generating artificial images: generative adversarial networks (GANs).

8.4.4 Take aways

Image generation with deep learning is done by learning latent spaces that capture statistical information about a dataset of images. By sampling points from the latent space, and "decoding" them, one can generate never-seen-before images. There are two major tools to do this: VAEs and GANs.

- VAEs result in highly structured, continuous latent representations. For this reason, they work well for doing all sort of image edition in latent space, like face swapping, turning a frowning face into a smiling face, and so on. They also work nicely for doing latent space based animations, i.e. animating a walk along a cross section of the latent space, showing a starting image slowly morphing into different images in a continuous way.
- GANs enable the generation of realistic single-frame images, but may not induce latent spaces with solid structure and high continuity.

Most successful practical applications I have seen with images actually rely on VAEs, but GANs are extremely popular in the world of academic research—at least circa 2016-2017. You will find out how they work and how to implement one in the next section.

To play further with image generation, I suggest working with the `CelebA` dataset, "Large-scale Celeb Faces Attributes". It's a free-to-download image dataset with more than 200,000 celebrity portraits. It's great for experimenting with concept vectors in particular. It beats MNIST for sure.

8.5 Introduction to generative adversarial networks

Generative Adversarial Networks (GANs), introduced in 2014 by Ian Goodfellow, are an alternative to VAEs for learning latent spaces of images. They enable the generation of fairly realistic synthetic images by forcing the generated images to be statistically almost indistinguishable from real ones.

An intuitive way to understand GANs is to imagine a forger trying to create a fake Picasso painting. At first, the forger is pretty bad at the task. He mixes some of his fakes with authentic Picassos, and shows them all to an art dealer. The art dealer makes an authenticity assessment for each painting, and gives the forger feedback about what makes a Picasso look like a Picasso. The forger goes back to his atelier to prepare some new fakes. As times goes on, the forger becomes increasingly competent at imitating the style of Picasso, and the art dealer becomes increasingly expert at spotting fakes. In the end, we have on our hands some excellent fake Picassos.

That's what GANs are: a forger network network and an expert network, each being trained to best the other. As such, a GAN is made of two parts:

- A *generator network*, which takes as input a random vector (a random point in the latent space) and decodes it into a synthetic image.
- A *discriminator network* (also called *adversary*), which takes as input an image (real or synthetic), and must predict whether the image came from the training set or was created by the generator network.

The generator network is trained to be able to fool the discriminator network, and thus it evolves towards generating increasingly realistic images as training goes on: artificial images that look indistinguishable from real ones—to the extent that it is impossible for the discriminator network to tell the two apart. Meanwhile, the discriminator is constantly adapting to the gradually improving capabilities of the generator, which sets a very high bar of realism for the generated images. Once training is over, the generator is capable of turning any point in its input space into a believable image. Unlike VAEs, this latent space has less explicit guarantees of meaningful structure, and in particular, it isn't continuous.

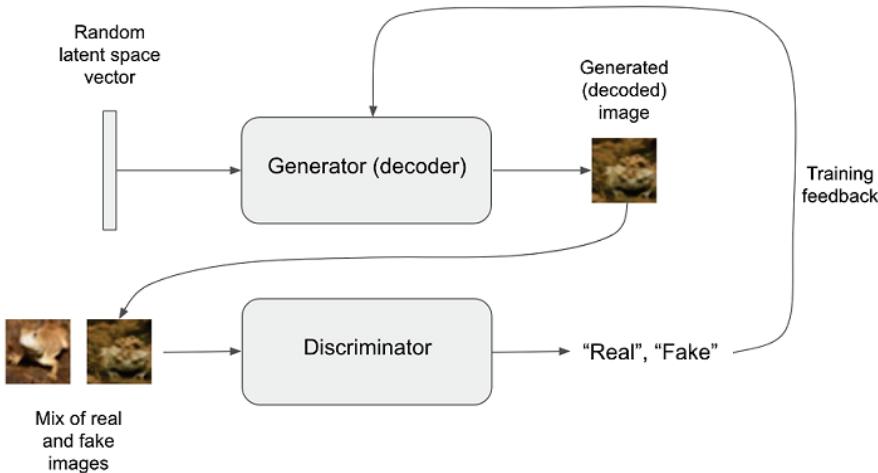


Figure 8.15 A generator transforms random latent vectors into images, while a discriminator seeks to tell apart real images from generated ones. The generator is trained to fool the discriminator.

Remarkably, a GAN is a system where the optimization minimum isn't fixed—unlike in any other training setup you have encountered in this book before. Normally, gradient descent consists in rolling down some hills in a static loss landscape. However, with a GAN, every step taken down the hill changes the entire landscape by a bit. It's a dynamic system where the optimization process is seeking not a minimum, but rather an equilibrium between two forces. For this reason, GANs are notoriously very difficult to train—getting a GAN to work require lots of careful tuning of the model architecture and training parameters.



Figure 8.16 Latent space dwellers. Images generated by Mike Tyka using a multi-staged GAN trained on a dataset of faces. Website: www.miketyka.com

8.5.1 A schematic GAN implementation

In what follows, we explain how to implement a GAN in Keras, in its barest form—since GANs are quite advanced, diving deeply into the technical details would be out of scope for us. Our specific implementation will be a deep convolutional GAN, or DCGAN: a GAN where the generator and discriminator are deep convnets. In particular, it leverages a Conv2DTranspose layer for image upsampling in the generator.

We will train our GAN on images from CIFAR10, a dataset of 50,000 32x32 RGB images belonging to 10 classes (5,000 images per class). To make things even easier, we will only use images belonging to the class "frog".

Schematically, our GAN looks like this:

- A generator network maps vectors of shape (`latent_dim,`) to images of shape (32, 32, 3).
- A discriminator network maps images of shape (32, 32, 3) to a binary score estimating the probability that the image is real.
- A gan network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus this gan network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, as we would train any regular image classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the gan model. This means that, at every step, we move the weights of the generator in a direction that will make the discriminator more likely to classify as "real" the images decoded by the generator. I.e. we train the generator to fool the discriminator.

8.5.2 A bag of tricks

Training GANs and tuning GAN implementations is notoriously difficult. There are a number of known "tricks" that one should keep in mind. Like most things in deep learning, it is more alchemy than science: these tricks are really just heuristics, not theory-backed guidelines. They are backed by some level of intuitive understanding of the phenomenon at hand, and they are known to work well empirically, albeit not necessarily in every context.

Here are a few of the tricks that we leverage in our own implementation of a GAN generator and discriminator below. It is not an exhaustive list of GAN-related tricks; you will find many more across the GAN literature.

- We use `tanh` as the last activation in the generator, instead of `sigmoid`, which would be more commonly found in other types of models.
- We sample points from the latent space using a *normal distribution* (Gaussian distribution), not a uniform distribution.
- Stochasticity is good to induce robustness. Since GAN training results in a dynamic equilibrium, GANs are likely to get "stuck" in all sorts of ways. Introducing randomness during training helps prevent this. We introduce randomness in two ways: 1) we use dropout in the discriminator, 2) we add some random noise to the labels for the discriminator.
- Sparse gradients can hinder GAN training. In deep learning, sparsity is often a desirable property, but not in GANs. There are two things that can induce gradient sparsity: 1) max pooling operations, 2) ReLU activations. Instead of max pooling, we recommend using strided convolutions for downsampling, and we recommend using a `LeakyReLU` layer instead of a ReLU activation. It is similar to ReLU but it relaxes sparsity constraints by allowing small negative activation values.
- In generated images, it is common to see "checkerboard artifacts" caused by unequal coverage of the pixel space in the generator. To fix this, we use a kernel size that is

divisible by the stride size, whenever we use a strided Conv2DTranspose or Conv2D in both the generator and discriminator.

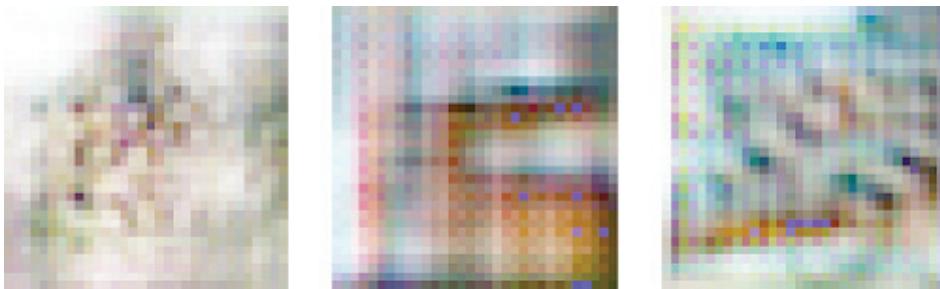


Figure 8.17 "Checkerboard artifacts" caused by mismatching strides and kernel sizes, resulting in unequal pixel space coverage: one of the many "gotchas" of GANs.

8.5.3 The generator

First, we develop a generator model, which turns a vector (from the latent space—during training it will be sampled at random) into a candidate image. One of the many issues that commonly arise with GANs is that the generator gets stuck with generated images that look like noise. A possible solution is to use dropout on both the discriminator and generator.

Listing 8.31 The GAN generator network

```
import keras
from keras import layers
import numpy as np

latent_dim = 32
height = 32
width = 32
channels = 3

generator_input = keras.Input(shape=(latent_dim,))

# First, transform the input into a 16x16 128-channels feature map
x = layers.Dense(128 * 16 * 16)(generator_input)
x = layers.LeakyReLU()(x)
x = layers.Reshape((16, 16, 128))(x)

# Then, add a convolution layer
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

# Upsample to 32x32
x = layers.Conv2DTranspose(256, 4, strides=2, padding='same')(x)
x = layers.LeakyReLU()(x)

# Few more conv layers
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

# Produce a 32x32 1-channel feature map
x = layers.Conv2D(channels, 7, activation='tanh', padding='same')(x)
generator = keras.models.Model(generator_input, x)
generator.summary()
```

8.5.4 The discriminator

Then, we develop a discriminator model, that takes as input a candidate image (real or synthetic) and classifies it into one of two classes, either "generated image" or "real image that comes from the training set".

Listing 8.32 The GAN discriminator network

```

discriminator_input = layers.Input(shape=(height, width, channels))
x = layers.Conv2D(128, 3)(discriminator_input)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Flatten()(x)

# One dropout layer - important trick!
x = layers.Dropout(0.4)(x)

# Classification layer
x = layers.Dense(1, activation='sigmoid')(x)

discriminator = keras.models.Model(discriminator_input, x)
discriminator.summary()

# To stabilize training, we use learning rate decay
# and gradient clipping (by value) in the optimizer.
discriminator_optimizer = keras.optimizers.RMSprop(lr=0.0008, clipvalue=1.0, decay=1e-8)
discriminator.compile(optimizer=discriminator_optimizer, loss='binary_crossentropy')

```

8.5.5 The adversarial network

Finally, we setup the GAN, which chains the generator and the discriminator. This is the model that, when trained, will move the generator in a direction that improves its ability to fool the discriminator. This model turns latent space points into a classification decision, "fake" or "real", and it is meant to be trained with labels that are always "these are real images". So training gan will updates the weights of generator in a way that makes discriminator more likely to predict "real" when looking at fake images. Very importantly, we set the discriminator to be frozen during training (non-trainable): its weights will not be updated when training gan. If the discriminator weights could be updated during this process, then we would be training the discriminator to always predict "real", which is not what we want!

Listing 8.33 The adversarial network

```

# Set discriminator weights to non-trainable
# (will only apply to the `gan` model)
discriminator.trainable = False

gan_input = keras.Input(shape=(latent_dim,))
gan_output = discriminator(generator(gan_input))
gan = keras.models.Model(gan_input, gan_output)

gan_optimizer = keras.optimizers.RMSprop(lr=0.0004, clipvalue=1.0, decay=1e-8)
gan.compile(optimizer=gan_optimizer, loss='binary_crossentropy')

```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

8.5.6 How to train your DCGAN

Now we can start training. To recapitulate, this is schematically what the training loop looks like:

Listing 8.34 Schematic formulation of GAN training

```
for each epoch:  
    * Draw random points in the latent space (random noise).  
    * Generate images with `generator` using this random noise.  
    * Mix the generated images with real ones.  
    * Train `discriminator` using these mixed images, with corresponding targets, either "real" (for  
    * Draw new random points in the latent space.  
    * Train `gan` using these random vectors, with targets that all say "these are real images". This
```

Let's implement it:

Listing 8.35 Implementing GAN training

```
import os  
from keras.preprocessing import image  
  
# Load CIFAR10 data  
(x_train, y_train), (_, _) = keras.datasets.cifar10.load_data()  
  
# Select frog images (class 6)  
x_train = x_train[y_train.flatten() == 6]  
  
# Normalize data  
x_train = x_train.reshape(  
    (x_train.shape[0],) + (height, width, channels)).astype('float32') / 255.  
  
iterations = 10000  
batch_size = 20  
save_dir = 'your_dir'  
  
# Start training loop  
start = 0  
for step in range(iterations):  
    # Sample random points in the latent space  
    random_latent_vectors = np.random.normal(size=(batch_size, latent_dim))  
  
    # Decode them to fake images  
    generated_images = generator.predict(random_latent_vectors)  
  
    # Combine them with real images  
    stop = start + batch_size  
    real_images = x_train[start: stop]  
    combined_images = np.concatenate([generated_images, real_images])  
  
    # Assemble labels discriminating real from fake images  
    labels = np.concatenate([np.ones((batch_size, 1)),  
                           np.zeros((batch_size, 1))])  
    # Add random noise to the labels - important trick!  
    labels += 0.05 * np.random.random(labels.shape)  
  
    # Train the discriminator  
    d_loss = discriminator.train_on_batch(combined_images, labels)  
  
    # sample random points in the latent space  
    random_latent_vectors = np.random.normal(size=(batch_size, latent_dim))  
  
    # Assemble labels that say "all real images"  
    misleading_targets = np.zeros((batch_size, 1))
```

```

# Train the generator (via the gan model,
# where the discriminator weights are frozen)
a_loss = gan.train_on_batch(random_latent_vectors, misleading_targets)

start += batch_size
if start > len(x_train) - batch_size:
    start = 0

# Occasionally save / plot
if step % 100 == 0:
    # Save model weights
    gan.save_weights('gan.h5')

    # Print metrics
    print('discriminator loss:', d_loss)
    print('adversarial loss:', a_loss)

    # Save one generated image
    img = image.array_to_img(generated_images[0] * 255., scale=False)
    img.save(os.path.join(save_dir, 'generated_frog' + str(step) + '.png'))

    # Save one real image, for comparison
    img = image.array_to_img(real_images[0] * 255., scale=False)
    img.save(os.path.join(save_dir, 'real_frog' + str(step) + '.png'))

```

When training, you may see your adversarial loss start increasing considerably while your discriminative loss will tend to zero, i.e. your discriminator may end up dominating your generator. If that's the case, try reducing the discriminator learning rate and increase the dropout rate of the discriminator.

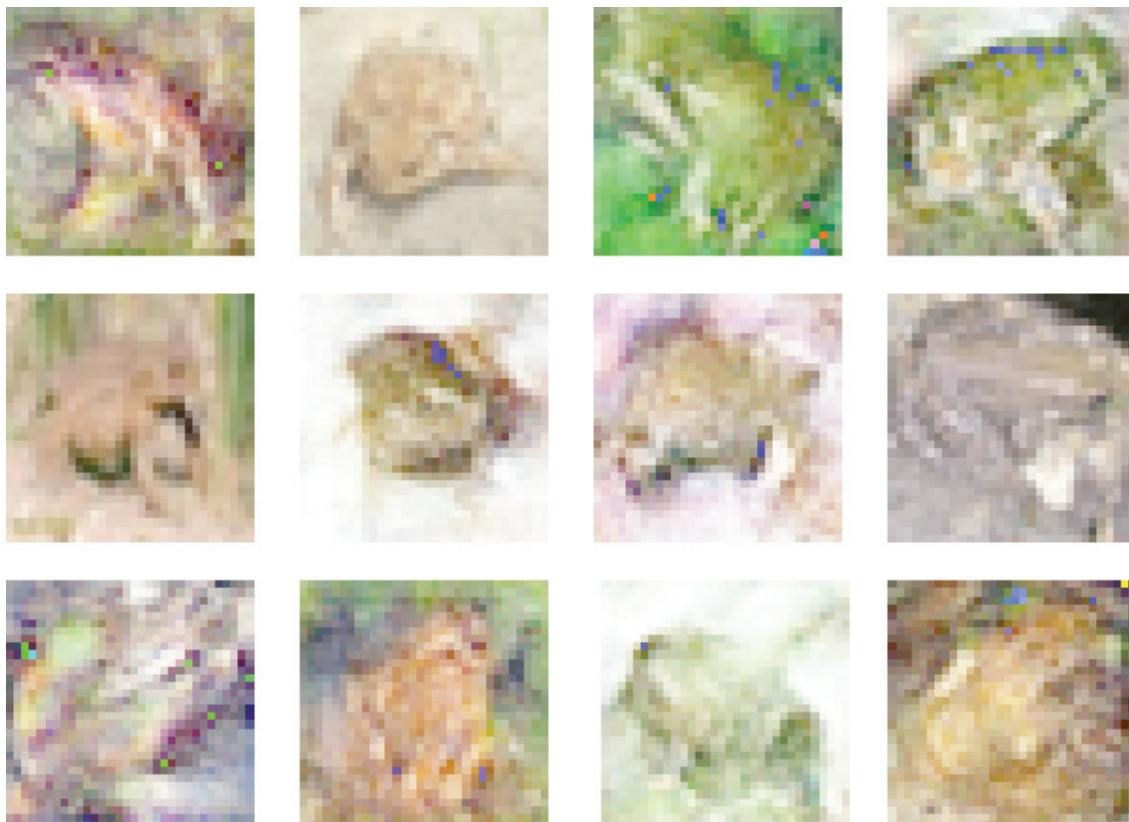


Figure 8.18 Play the discriminator: in each row, two images were dreamed up by our GAN, and one image comes from the training set. Can you tell them apart? (answers - real images: middle, top, bottom, middle)

8.5.7 Take-aways

- GANs consist in a generator network coupled with a discriminator network. The discriminator is trained to tell apart the output of the generator and real images from a training dataset, while the generator is trained to fool the discriminator. Remarkably, the generator never sees images from the training set directly; the information it has about the data comes from the discriminator.
- GANs are difficult to train, because training a GAN is a dynamic process rather than a simple descent process with a fixed loss landscape. Getting a GAN to train correctly requires leveraging a number of heuristic tricks, as well as extensive tuning.
- GANs can potentially produce highly realistic images. However, unlike VAEs, the latent space that they learn does not have a neat continuous structure, and thus may not be suited for certain practical applications, such as image editing via latent space concept vectors.

8.6 Wrapping up: generative deep learning

This is the end of the chapter on creative applications of deep learning, where deep nets go beyond simply annotating existing content, and start generating their own. You have just learned:

- How to generate sequence data, one timestep at a time. This is applicable to text generation, but also to note-by-note music generation, or any other type of timeseries data.
- How Deep Dreams work: by maximizing convnet layer activations through gradient ascent in input space.
- How to perform style transfer, where a content image and a style image get combined to produce interesting-looking results.
- What GANs and VAEs are, how they can be used for dreaming up new images, and how latent space "concept vectors" could be used for image edition.

These few techniques only cover the very basics of this fast-expanding field. There's a lot more to discover out there—generative deep learning would be deserving of an entire book of its own.

Conclusions

You have almost reached the end of this book. In this last chapter, we focus on summarizing and reviewing core concepts, while expanding your horizons beyond the relatively basic notions you have learned so far. Understanding deep learning and AI is a journey, and finishing this book is merely the first step of it. I want to make sure that you realize this, and that you are properly equipped for taking the next steps of this journey on your own.

This chapter is structured in four sections:

- A bird's eye view of what you should take away from this book. This should refresh your memory on some of the concepts you have previously learned.
- An overview of some key limitations of deep learning. To use a tool appropriately, you should not only understand what it can do, but also be aware of what it won't do.
- Speculative thoughts about the future evolution of the fields of deep learning, machine learning and AI. This should be especially interesting to you if you would like to get into fundamental research.
- A short list of resources and strategies for learning further about AI and staying up to date with new advances.

9.1 Key concepts in review

This section aims at briefly synthesizing the key take-aways from this book. If you ever need a quick refresher to help you recall what you've learned in this book, you can just read these few pages.

9.1.1 Different brands of approaches to AI

First of all, deep learning is not synonymous with AI, nor even with machine learning. AI is an ancient and very broad field that can generally be defined as "all attempts to automate cognitive processes". The automation of thought. This can range from the very basic, like an Excel spreadsheet, to the very advanced, like a humanoid robot able to walk and talk.

Machine learning is a specific subfield of AI that aims at automatically developing programs (called "models") purely from exposure to training data. This process of turning data into a program is called "learning". Albeit machine learning has been around for a

long time, it only started taking off in the 1990s.

Deep learning is one of many branches of machine learning, where the models are long chains of geometric functions, chained one after the other. These operations are structured into modules called "layers": deep learning models are typically just stacks of layers, or more generally graphs of layers. These layers are parametrized by "weights", which are the parameters that are learned during training. The "knowledge" of a model is stored in its weights, and process of "learning" consists in finding good values for these weights.

Even though deep learning is just one approach to machine learning among many, it isn't on an equal footing with the others. Deep learning is a breakout success. Here's why.

9.1.2 What makes deep learning special within machine learning

In the span of just a few years, deep learning has achieved tremendous breakthroughs across a wide range of tasks that have been historically perceived as extremely difficult for computers, especially in machine perception: extracting useful information from images, videos, sound, and more. Given sufficient training data (in particular, training data appropriately labeled by humans), it is possible to extract from perceptual data almost anything that a human could extract. Hence it is sometimes said that deep learning has "solved perception", albeit that is only true for a fairly narrow definition of "perception".

Due to its unprecedented technical successes, deep learning has single-handedly brought about the third and by far the largest "AI summer", a period of intense interest, investment, and hype, in the field of AI. As this book is being written, we are in the middle of it. Whether this period will end in the near future, and what happens after it ends, is a topic of debate. One thing is certain: in stark contrast with previous AI summers, deep learning has been providing enormous business value to a number of large technology companies, enabling human-level speech recognition, smart assistants, human-level image classification, vastly improved machine translation, and more. The hype may (and likely will) recess, but the sustained economic and technological impact of deep learning will remain. In that sense, deep learning could be analogous to the Internet: it may be overly hyped up for a few years, but in the longer term it will still be a major revolution that will transform our economy and our lives.

I am particularly optimistic about deep learning, because, even if we were to make no further technological progress in the next decade, simply deploying existing algorithms to every problem where they are applicable would already be a game-changer for most industries. Deep learning is nothing short of a revolution. And as it happens, progress is currently happening at an incredibly fast rate—due to an exponential investment in resources and headcount. From where I stand, the future looks bright. It does seem, however, that short-term expectations are somewhat over-optimistic. Deploying deep learning to the full extent of its potential will take well over a decade.

9.1.3 How to think about deep learning

The most surprising thing about deep learning is how simple it is. Ten years ago, no one expected that we would achieve such amazing results on machine perception problems by using simple parametric models trained with gradient descent. Now, it turns out that all you need is *sufficiently large* parametric models trained with gradient descent on *sufficiently many* examples. As Feynman once said about the universe, "*It's not complicated, it's just a lot of it*".

In deep learning, everything is a vector, i.e. everything is a *point* in a *geometric space*. Model inputs (it could be text, images, etc) and targets are first "vectorized", i.e. turned into some initial input vector space and target vector space. Each layer in a deep learning model operates one simple geometric transformation on the data that goes through it. Together, the chain of layers of the model forms one very complex geometric transformation, broken down into a series of simple ones. This complex transformation attempts to map the input space to the target space, one point at a time. This transformation is parametrized by the weights of the layers, which are iteratively updated based on how well the model is currently performing. A key characteristic of this geometric transformation is that it must be *differentiable*, which is required in order for us to be able to learn its parameters via gradient descent. Intuitively, this means that the geometric morphing from inputs to outputs must be smooth and continuous—a significant constraint.

The whole process of applying this complex geometric transformation to the input data can be visualized in 3D by imagining a person trying to uncrumple a paper ball: the crumpled paper ball is the manifold of the input data that the model starts with. Each movement operated by the person on the paper ball is similar to a simple geometric transformation operated by one layer. The full uncrumpling gesture sequence is the complex transformation of the entire model. Deep learning models are mathematical machines for uncrumpling complicated manifolds of high-dimensional data.

That's the magic of deep learning: turning meaning into vectors, into geometric spaces, then incrementally learning complex geometric transformations that map one space to another. All you need are spaces of sufficiently high dimensionality in order to capture the full scope of the relationships found in the original data.

The whole thing hinges on one single core idea: *that meaning is derived from the pairwise relationship between things* (between words in a language, between pixels in an image, etc) and that *these relationships can be captured by a distance function*. But do note that whether or not the brain implements meaning via geometric spaces is an entirely separate question. Vector spaces are very efficient to work with from a computational standpoint, but different data structures for intelligence can easily be envisioned, in particular graphs. In fact, "neural networks" initially emerged from the very idea of using graphs as a way to encode meaning, which is why they are named "neural networks", and the surrounding field of research used to be called "connectionism". Nowadays the name "neural network" is still around purely for historical reasons—it is an extremely

misleading name since they are in fact neither neural nor networks. In particular, they have hardly anything to do with the brain. A more appropriate name could have been "layered representations learning" or "hierarchical representations learning". Or maybe even "deep differentiable models" or "chained geometric transforms", in order to emphasize the fact that continuous geometric space manipulation is at their core.

9.1.4 Key enabling technologies

The technological revolution that is currently unfolding did not start with any single breakthrough invention. Rather, like any other revolution, it is the product of a vast accumulation of enabling factors—slow at first, then all of a sudden. In the case of deep learning, we can point out the following key factors:

- Incremental algorithmic innovations, first spread over two decades (starting with backpropagation), then happening increasingly faster as more research effort was poured into deep learning after 2012.
- The availability of large amounts of perceptual data, a requirement in order to realize that "sufficiently large models trained on sufficiently large data are all you need". This is in turn a by-product of the rise of the consumer Internet and Moore's law applied to storage media.
- The availability of fast, highly parallel computation hardware at a low price, especially the GPUs produced by NVIDIA—first gaming GPUs, then chips designed from the ground up for deep learning, as CEO Jensen Huang took note early on of the deep learning boom and decided to bet the company's future on it.
- A complex stack of software layers that makes this computational power available to humans: the CUDA language, frameworks like TensorFlow that do automatic differentiation, and finally Keras that makes deep learning accessible to most people.

In the future, deep learning will not only be used by specialists—researchers, grad students, and expensive engineers with an academic profile. Rather, it will be a tool in the toolbox of every developer, much like web technology today. Everyone needs to build intelligent apps: much like every business today needs a website, every product will need to intelligently make sense of user-generated data. Making this future happen requires us to build tools that make deep learning radically easy to use, that make deep learning accessible to anyone with basic coding abilities. Keras is the first major step in that direction.

9.1.5 The universal machine learning workflow

Having access to an extremely powerful tool for creating models that map any input space to any target space is great, but the difficult part of the machine learning workflow is often everything that comes before designing and training such models (and, for production models, what comes after as well). Understanding the problem domain so as to be able to determine what to attempt to predict, given what data, and how to measure success, is a prerequisite for any successful application of machine learning, and it isn't something that advanced tools like Keras or TensorFlow will help you with. As a reminder, here's a quick summary of the typically machine learning workflow as described in Chapter 4:

- First, define the problem: what data is available, and what are you trying to predict? Will you need to collect more data, to hire people to manually label a dataset?
- Identify a way to reliably measure success on your goal. For simple tasks, this may be prediction accuracy, but in many cases it will require sophisticated domain-specific metrics.
- Prepare the validation process that you will use to evaluate your models. In particular, you should define a training set, validation set, and test set. Your validation and test set labels should not "leak" into your training data: for instance, with temporal prediction, the validation and test data should be posterior to the training data.
- Vectorize your data, by turning it into vectors and preprocessing it in a way that makes it more easily approachable by a neural network (normalization, etc).
- Develop a first model that beats a trivial common-sense baseline—thus demonstrating that machine learning can work on your problem. This might not always be the case!
- Gradually refine your model architecture by tuning hyperparameters and adding regularization. Make changes based on your performance on the validation data only, not the test data nor the training data. Remember that you should manage to get your model to overfit (thus identifying a model capacity level that is above that you need), and only then start adding regularization or start downsizing your model.
- Mind "validation set overfitting" when turning hyperparameters, i.e. the fact that your hyperparameters might end up being over-specialized to your validation set. Avoiding this is precisely the purpose of having a separate test set!

9.1.6 Key network architectures

The three families of network architectures that you should be familiar with are densely-connected networks, convolutional networks, and recurrent networks. Each type of network is meant for a specific input modality: a network architecture (dense, convolutional, recurrent) encodes *assumptions* about the structure of the data, a *hypothesis space* within which the search for a good model will proceed. Whether or not a given architecture will work on a given problem depends entirely on the match between the structure of the data and the assumptions of the network architecture.

These different network types can be easily combined to achieve larger multi-modal networks, much like one would combine together Lego bricks. In a way, deep learning layers are Lego for information processing. Here is a quick overview of the mapping between input modalities and appropriate network architecture:

- Vector data: Densely-connected network (`Dense` layers).
- Image data: 2D convnets.
- Sound data (e.g. waveform): Either 1D convnets (preferred) or RNNs.
- Text data: Either 1D convnets (preferred) or RNN.
- Timeseries data: Either RNNs (preferred) or 1D convnets.
- Other types of sequence data: Either RNNs or 1D convnets. Prefer RNNs if data ordering is strongly meaningful (e.g. for timeseries, but not for text).
- Video data: Either 3D convnets (if you need to capture motion effects) or a combination of a frame-level convnet for feature extraction followed by either a RNN or a 1D convnet to process the resulting sequences.
- Volumetric data: 3D convnets.

Now, let's quickly review the specificities of each network architecture.

DENSELY-CONNECTED NETWORKS

Densely-connected networks are just a stack of Dense layers, meant to process vector data (i.e. batches of vectors). They assume no specific structure in the input features: they are called "densely-connected" because the units of a Dense layer are "connected" to every other unit; the layer will attempt to map relationships between any two input features, which is unlike a 2D convolution layer, for instance, which only looks at *local* relationships.

Densely-connected networks are most commonly used for categorical data (e.g. where the input features are lists of attributes), for instance the Boston Housing dataset we covered in Chapter 3. They are also used as the final classification or regression stage of most networks. For instance, the convnets we covered in Chapter 5 typically ended with one or two Dense layers, and so did our recurrent networks from Chapter 6.

Remember: to perform *binary classification*, end your stack of layers with a Dense layer with a single unit and a sigmoid activation, and use binary_crossentropy as loss. Your targets should be either 0 or 1.

```
model = Sequential()
model.add(Dense(32, activation='relu', input_shape=(num_input_features,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy')
```

To perform *single-label categorical classification* (where each sample has exactly one class, no more), end your stack of layers with a Dense layer with a number of units equal to the number of classes, and a softmax activation. If your targets are one-hot encoded, use categorical_crossentropy as loss; if they are integers, use sparse_categorical_crossentropy.

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

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
```

To perform *multi-label categorical classification* (where each sample can have several classes), end your stack of layers with a Dense layer with a number of units equal to the number of classes and a sigmoid activation, and use binary_crossentropy as loss. Your targets should be one-hot encoded.

```
model = Sequential()
model.add(Dense(32, activation='relu', input_shape=(num_input_features,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(num_classes, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy')
```

To perform *regression* towards a vector of continuous values, end your stack of layers with a `Dense` layer with a number of units equal to the number of values you are trying to predict (often a single one, like the price of a house), and no activation. There are several losses that can be used for regression, most commonly `mean_squared_error` (MSE) and `mean_absolute_error` (MAE).

```
model = Sequential()
model.add(Dense(32, activation='relu', input_shape=(num_input_features,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(num_values))

model.compile(optimizer='rmsprop', loss='mse')
```

CONVNETS

Convolution layers look at spatially local patterns, by applying a same geometric transformation to different spatial locations ("patches") in an input tensor. This results in representations that are *translation-invariant*, making convolution layers highly data-efficient and modular. This idea is applicable to spaces of any dimensionality: 1D (sequences), 2D (images), 3D (volumes), etc. You can use the `Conv1D` layer to process sequences (especially text; it doesn't work as well on timeseries that often do not follow the translation invariance assumption), the `Conv2D` layer to process images, and the `Conv3D` layers to process volumes.

Convnets, or convolutional networks, consist of stacks of convolution and max-pooling layers. The pooling layers allow to spatially downsample the data, which is required to keep feature maps to a reasonable size as the number of features grows, and to allow subsequent convolution layers to "see" a greater spatial extent of the inputs. Convnets are often ended with either a `Flatten` operation or a global pooling layer, turning spatial feature maps into vectors, followed by `Dense` layers to achieve classification or regression.

Note that it is highly likely that regular convolutions will soon be mostly (or completely) replaced by an equivalent but faster and even representationally efficient alternative, the depthwise separable convolution (`SeparableConv2D` layer). This is true for 3D, 2D or 1D inputs. When building a new network from scratch, using depthwise separable convolutions is definitely the way to go. The `SeparableConv2D` layer can be used as a drop-in replacement for `Conv2D`, resulting in a smaller and faster network that will also perform better on its task.

Here is what a typical image classification network would look like (categorical classification, in this case):

```
model = Sequential()
model.add(SeparableConv2D(32, activation='relu',
                        input_shape=(height, width, channels)))
model.add(SeparableConv2D(64, activation='relu'))
model.add(MaxPooling2D(2))

model.add(SeparableConv2D(64, activation='relu'))
```

```

model.add(SeparableConv2D(128, activation='relu'))
model.add(MaxPooling2D(2))

model.add(SeparableConv2D(64, activation='relu'))
model.add(SeparableConv2D(128, activation='relu'))
model.add(GlobalAveragePooling2D())

model.add(Dense(32, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

```

RNNs

Recurrent neural networks (RNNs) work by processing sequences of inputs one timestep at a time, and maintaining a "state" throughout (a state is typically just a vector, or set of vectors, i.e. a point in a geometric space of states). They should be used preferentially over 1D convnets in the case of sequences where patterns of interest are not invariant by temporal translation (for instance, timeseries data where the recent past is more important than the distant past).

There are three RNN layers available in Keras: `SimpleRNN`, `GRU` and `LSTM`. For most practical purposes, you should be using either `GRU` or `LSTM`. `LSTM` is the more powerful of the two, but also the most expensive; you can see `GRU` as a simpler and cheaper alternative to it.

In order to stack multiple RNN layers on top of each other, each layer prior to the last one should return the full sequence of its outputs (to each input timestep will correspond an output timestep); if you are not stacking any further RNN layers, then it is common to only return the last output, which contains information about the entire sequence.

```

# Single RNN layer for binary classification
# of vector sequences
model = Sequential()
model.add(LSTM(32, input_shape=(num_timesteps, num_features)))
model.add(Dense(num_classes, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy')

```

```

# Stacked RNN for binary classification
# of vector sequences
model = Sequential()
model.add(LSTM(32, return_sequences=True,
              input_shape=(num_timesteps, num_features)))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(32))
model.add(Dense(num_classes, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy')

```

9.1.7 The space of possibilities

What will you build with deep learning? Remember, building deep learning models is like playing with Lego bricks: layers can be plugged together to map essentially anything to anything, given that you have appropriate training data available and that the mapping is achievable via a continuous geometric transformation of reasonable complexity. The space of possibilities is infinite. Here are a few examples to inspire you to think beyond the basic classification or regression tasks that have been traditionally the bread and butter of machine learning.

We sort our suggested applications by input and output modalities. Do note that quite a few of them are stretching the limits of what is possible—while a model could be trained on these tasks in every case, in some cases such a model would probably not generalize very far from its training data. In the next couple sections, we address how these limitations could be lifted in the future.

MAPPING VECTOR DATA TO VECTOR DATA

- Predictive healthcare: mapping patient medical records to predictions of patient outcome.
- Behavioral targeting: mapping a set of website attributes with data on how long a user will spend on the website.
- Product quality control: apping a set of attributes relative to an instance of a manufactured product with the probability that the product will fail by next year.

9.1.8 Mapping image data to vector data

- Doctor assistant: mapping slides of medical images with a prediction about the presence of a tumor.
- Self-driving: mapping car dashcam video frames to steering wheel angle commands.
- Board game AI: mapping Go or Chess boards to to next player move.
- Diet helper: mapping pictures of a dish with its calorie count.
- Age prediction: mapping selfies to the age of the person.

9.1.9 Mapping timeseries data to vector data

- Weather prediction: mapping timeseries of weather data in a grid of locations of weather data the following week at a specific location.
- Brain-computer interfaces: mapping timeseries of magnetoencephalogram (MEG) data to computer commands.
- Behavioral targeting: mapping timeseries of user interactions on a website with the probability that a user will buy something.

MAPPING TEXT TO TEXT

- Smart reply: mapping emails to possible one-line replies.
- Question answering: mapping general knowledge questions to answers.
- Summarization: mapping long articles to short summary of the article.

MAPPING IMAGES TO TEXT

- Captioning: mapping images to short captions describing the contents of the image.

MAPPING TEXT TO IMAGES

- Conditioned image generation: mapping short text descriptions to images matching the description.
- Logo generation/selection: mapping the name and description of a company to the company's logo.

MAPPING IMAGES TO IMAGES

- Super-resolution: mapping downsized images to higher-resolution versions of the same images.
- Visual depth sensing: mapping images of indoor environments to maps of depth predictions.

MAPPING IMAGES AND TEXT TO TEXT

- Visual QA: mapping images and natural language questions about the contents of the images, to natural language answers.

MAPPING VIDEO AND TEXT TO TEXT

- Video QA: mapping short videos and natural language questions about the contents of the video, to natural language answers.

Almost anything is possible—but not quite *anything*. Let's see in the next section what you *can't* do with deep learning.

9.2 *The limitations of deep learning*

The space of applications that can be implemented with deep learning is nearly infinite. And yet, many more applications are completely out of reach for current deep learning techniques—even given vast amounts of human-annotated data. Say, for instance, that you could assemble a dataset of hundreds of thousands—even millions—of English language descriptions of the features of a software product, as written by a product manager, as well as the corresponding source code developed by a team of engineers to meet these requirements. Even with this data, you could *not* train a deep learning model to simply read a product description and generate the appropriate codebase. That's just one example among many. In general, anything that requires reasoning—like programming, or applying the scientific method—long-term planning, and algorithmic-like data manipulation, is out of reach for deep learning models, no matter how much data you throw at them. Even learning a sorting algorithm with a deep neural network is tremendously difficult.

This is because a deep learning model is "just" *a chain of simple, continuous*

geometric transformations mapping one vector space into another. All it can do is map one data manifold X into another manifold Y, assuming the existence of a learnable continuous transform from X to Y. So even though a deep learning model can be interpreted as a kind of program, inversely *most programs cannot be expressed as deep learning models*—for most tasks, either there exists no corresponding deep neural network that solves the task, or even if there exists one, it may not be *learnable*, i.e. the corresponding geometric transform may be far too complex, or there may not be appropriate data available to learn it.

Scaling up current deep learning techniques by stacking more layers and using more training data can only superficially palliate some of these issues. It will not solve the more fundamental problem that deep learning models are very limited in what they can represent, and that most of the programs that one may wish to learn cannot be expressed as a continuous geometric morphing of a data manifold.

9.2.1 The risk of anthropomorphizing machine learning models

One very real risk with contemporary AI is that of misinterpreting what deep learning models do, and overestimating their abilities. A fundamental feature of the human mind is our "theory of mind", our tendency to project intentions, beliefs and knowledge on the things around us. Drawing a smiley face on a rock suddenly makes it "happy"—in our minds. Applied to deep learning, this means that when we are able to somewhat successfully train a model to generate captions to describe pictures, for instance, we are led to believe that the model "understands" the contents of the pictures, as well as the captions it generates. We then proceed to be very surprised when any slight departure from the sort of images present in the training data causes the model to start generating completely absurd captions.



The boy is holding a baseball bat.

Figure 9.1 Failure of a deep learning-based image captioning system.

In particular, this is highlighted by "adversarial examples", which are input samples to a deep learning network that are designed to trick the model into misclassifying them.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-learning-with-python>

Licensed to <null>

You are already aware that it is possible to do gradient ascent in input space to generate inputs that maximize the activation of some convnet filter, for instance—this was the basis of the filter visualization technique we introduced in Chapter 5, as well as the Deep Dream algorithm from Chapter 8. Similarly, through gradient ascent, one can slightly modify an image in order to maximize the class prediction for a given class. By taking a picture of a panda and adding to it a "gibbon" gradient, we can get a neural network to classify this panda as a gibbon. This evidences both the brittleness of these models, and the deep difference between the input-to-output mapping that they operate and our own human perception.

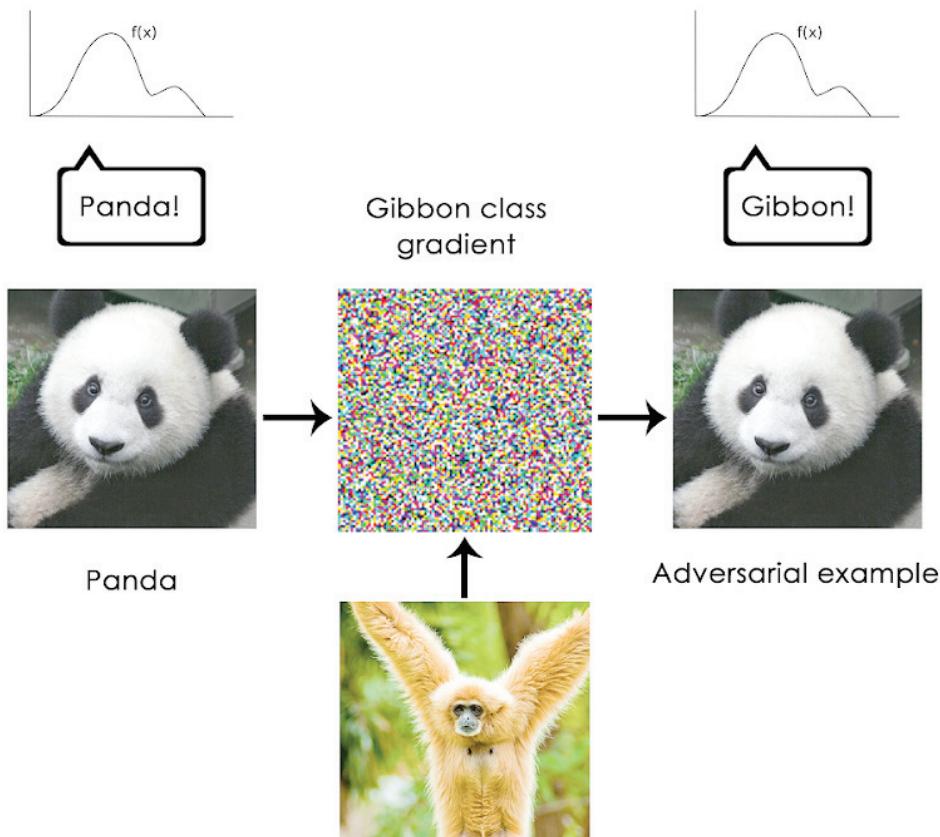


Figure 9.2 An adversarial example: imperceptible changes in an image can upend a model's classification of the image

In short, deep learning models do not have any understanding of their input, at least not in any human sense. Our own understanding of images, sounds, and language, is grounded in our sensorimotor experience as humans. Machine learning models have no access to such experiences and thus cannot "understand" their inputs in any human-relatable way. By annotating large numbers of training examples to feed into our models, we get them to learn a geometric transform that maps data to human concepts on this specific set of examples, but this mapping is just a simplistic sketch of the original model in our minds, the one developed from our experience as embodied agents—it is like a dim image in a mirror.

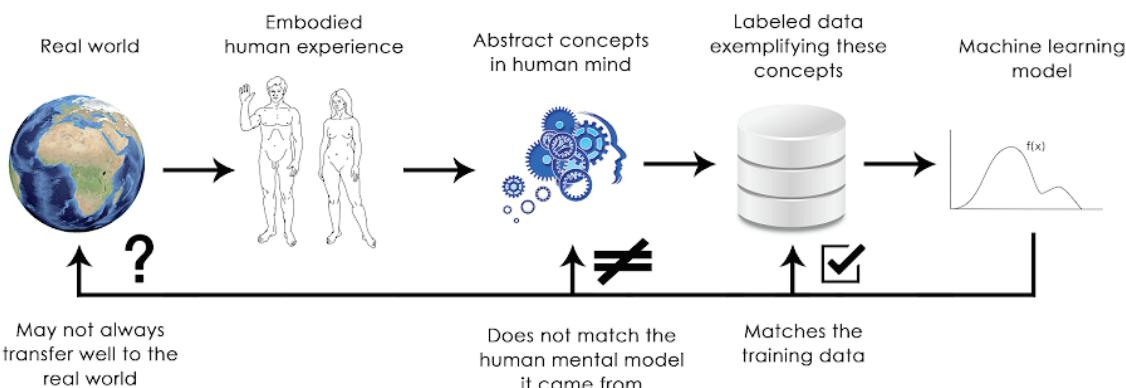


Figure 9.3 Current machine learning models: like a dim image in a mirror

As a machine learning practitioner, always be mindful of this, and never fall into the trap of believing that neural networks understand the task they perform—they don't, at least not in a way that would make sense to us. They were trained on a different, far narrower task than the one we wanted to teach them: that of merely mapping training inputs to training targets, point by point. Show them anything that deviates from their training data, and they will break in the most absurd ways.

9.2.2 Local generalization versus extreme generalization

There just seems to be fundamental differences between the straightforward geometric morphing from input to output that deep learning models do, and the way that humans think and learn. It isn't just the fact that humans learn by themselves from embodied experience instead of being presented with explicit training examples. Aside from the different learning processes, there is a fundamental difference in the nature of the underlying representations.

Humans are capable of far more than mapping immediate stimuli to immediate responses, like a deep net, or maybe an insect, would do. They maintain complex, *abstract models* of their current situation, of themselves, of other people, and can use these models to anticipate different possible futures and perform long-term planning. They are capable of merging together known concepts to represent something they have never experienced before—like picturing a horse wearing jeans, for instance, or imagining what they would do if they won the lottery. This ability to handle hypotheticals, to expand our mental model space far beyond what we can experience directly, in a word, to perform *abstraction* and *reasoning*, is arguably the defining characteristic of human cognition. I call it "extreme generalization": an ability to adapt to novel, never experienced before situations, using very little data or even no new data at all.

This stands in sharp contrast with what deep nets do, which I would call "local generalization": the mapping from inputs to outputs performed by deep nets quickly stops making sense if new inputs differ even slightly from what they saw at training time. Consider, for instance, the problem of learning the appropriate launch parameters to get a rocket to land on the moon. If you were to use a deep net for this task, whether training using supervised learning or reinforcement learning, you would need to feed it with

thousands or even millions of launch trials, i.e. you would need to expose it to a *dense sampling* of the input space, in order to learn a reliable mapping from input space to output space. By contrast, humans can use their power of abstraction to come up with physical models—rocket science—and derive an *exact* solution that will get the rocket on the moon in just one or few trials. Similarly, if you developed a deep net controlling a human body, and wanted it to learn to safely navigate a city without getting hit by cars, the net would have to die many thousands of times in various situations until it could infer that cars are dangerous, and develop appropriate avoidance behaviors. Dropped into a new city, the net would have to relearn most of what it knows. On the other hand, humans are able to learn safe behaviors without having to die even once—again, thanks to their power of abstract modeling of hypothetical situations.

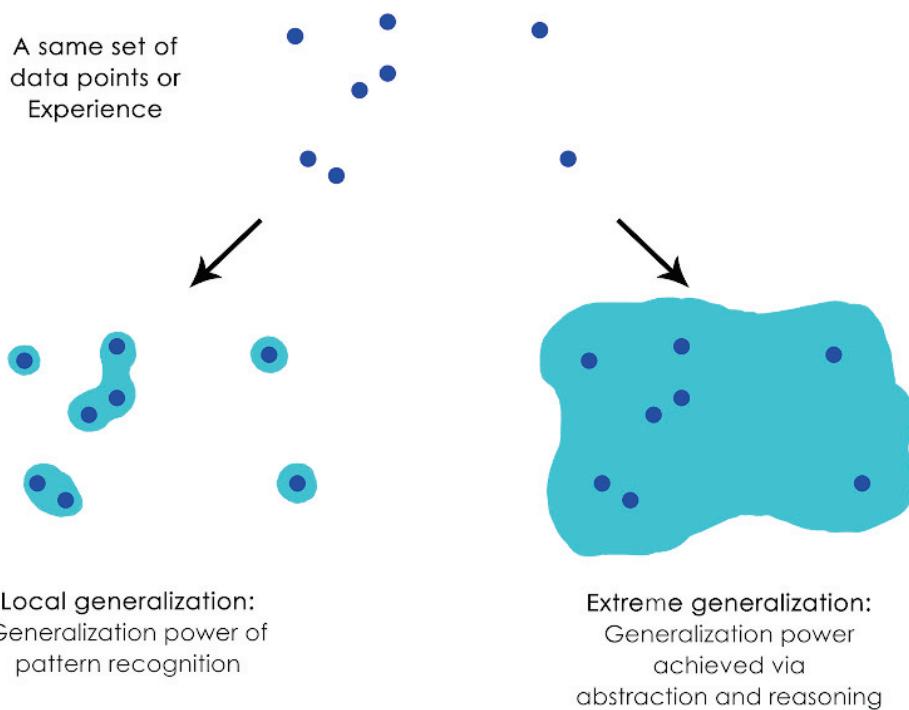


Figure 9.4 Local generalization vs. extreme generalization

In short, despite our progress on machine perception, we are still very far from human-level AI: our models can only perform *local generalization*, adapting to new situations that must stay very close from past data, while human cognition is capable of *extreme generalization*, quickly adapting to radically novel situations, or planning very far for long-term future situations.

9.2.3 Take-aways

Here's what you should remember: the only real success of deep learning so far has been the ability to map space X to space Y using a continuous geometric transform, given large amounts of human-annotated data. Doing this well is a game-changer for essentially every industry, but it is still a very long way from human-level AI.

To lift some of these limitations and start competing with human brains, we need to

move away from straightforward input-to-output mappings, and on to *reasoning* and *abstraction*. A likely appropriate substrate for abstract modeling of various situations and concepts is that of computer programs. We have said before that machine learning models could be defined as "learnable programs"; currently we can only learn programs that belong to a very narrow and specific subset of all possible programs. But what if we could learn *any* program, in a modular and reusable way? Let's see in the next section what the road ahead may look like.

9.3 The future of deep learning

This is a more speculative section aimed at opening up horizons to people who want to get into a research program or start doing independent research. Given what we know of how deep nets work, of their limitations, and of the current state of the research landscape, can we predict where things are headed in the medium term? Here are some purely personal thoughts. Note that I don't have a crystal ball, so a lot of what I anticipate might fail to become reality. I am sharing these predictions not because I expect them to be proven completely right in the future, but because they are interesting and actionable in the present.

At a high-level, the main directions in which I see promise are:

- Models closer to general-purpose computer programs, built on top of far richer primitives than our current differentiable layers—this is how we will get to *reasoning* and *abstraction*, the fundamental weakness of current models.
- New forms of learning that make the above possible—allowing models to move away from just differentiable transforms.
- Models that require less involvement from human engineers—it shouldn't be your job to tune knobs endlessly.
- Greater, systematic reuse of previously learned features and architectures; meta-learning systems based on reusable and modular program subroutines.

Additionally, do note that these considerations are not specific to the sort of supervised learning that has been the bread and butter of deep learning so far—rather, they are applicable to any form of machine learning, including unsupervised, self-supervised, and reinforcement learning. It is not fundamentally important where your labels come from or what your training loop looks like; these different branches of machine learning are just different facets of a same construct.

Let's dive in.

9.3.1 Models as programs

As we noted in the previous section, a necessary transformational development that we can expect in the field of machine learning is a move away from models that perform purely *pattern recognition* and can only achieve *local generalization*, towards models capable of *abstraction* and *reasoning*, that can achieve *extreme generalization*. Current AI programs that are capable of basic forms of reasoning are all hard-coded by human programmers: for instance, software that relies on search algorithms, graph manipulation, formal logic. In DeepMind's AlphaGo, for example, most of the "intelligence" on display is designed and hard-coded by expert programmers (e.g. Monte-Carlo tree search); learning from data only happens in specialized submodules (value networks and policy networks). But in the future, such AI systems may well be fully learned, with no human involvement.

What could be the path to make this happen? Consider a well-known type of network: RNNs. Importantly, RNNs have slightly less limitations than feedforward networks. That is because RNNs are a bit more than a mere geometric transformation: they are geometric transformations *repeatedly applied inside a for loop*. The temporal `for` loop is itself hard-coded by human developers: it is a built-in assumption of the network. Naturally, RNNs are still extremely limited in what they can represent, primarily because each step they perform is still just a differentiable geometric transformation, and the way they carry information from step to step is via points in a continuous geometric space (state vectors). Now, imagine neural networks that would be "augmented" in a similar way with programming primitives such as `for` loops—but not just a single hard-coded `for` loop with a hard-coded geometric memory, rather, a large set of programming primitives that the model would be free to manipulate to expand its processing function, such as `if` branches, `while` statements, variable creation, disk storage for long-term memory, sorting operators, advanced datastructures like lists, graphs, and hashtables, and many more. The space of programs that such a network could represent would be far broader than what can be represented with current deep learning models, and some of these programs could achieve superior generalization power.

In a word, we will move away from having on one hand "hard-coded algorithmic intelligence" (handcrafted software) and on the other hand "learned geometric intelligence" (deep learning). We will have instead a blend of formal algorithmic modules that provide *reasoning and abstraction* capabilities, and geometric modules that provide *informal intuition and pattern recognition* capabilities. The whole system would be learned with little or no human involvement.

A related subfield of AI that I think may be about to take off in a big way is that of *program synthesis*, in particular neural program synthesis. Program synthesis consists in automatically generating simple programs, by using a search algorithm (possibly genetic search, as in genetic programming) to explore a large space of possible programs. The search stops when a program is found that matches the required specifications, often provided as a set of input-output pairs. As you can see, is it highly reminiscent of

machine learning: given "training data" provided as input-output pairs, we find a "program" that matches inputs to outputs and can generalize to new inputs. The difference is that instead of learning parameter values in a hard-coded program (a neural network), we generate *source code* via a discrete search process.

I would definitely expect this subfield to see a wave of renewed interest in the next few years. In particular, I would expect the emergence of a crossover subfield in-between deep learning and program synthesis, where we would not quite be generating programs in a general-purpose language, but rather, where we would be generating neural networks (geometric data processing flows) *augmented* with a rich set of algorithmic primitives, such as `for` loops—and many others. This should be far more tractable and useful than directly generating source code, and it would dramatically expand the scope of problems that can be solved with machine learning—the space of programs that we can generate automatically given appropriate training data. A blend of symbolic AI and geometric AI. Contemporary RNNs can be seen as a prehistoric ancestor to such hybrid algorithmic-geometric models.

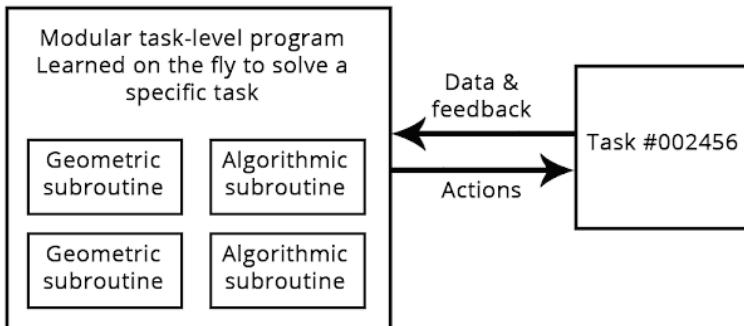


Figure 9.5 A learned program relying on both geometric (pattern recognition, intuition) and algorithmic (reasoning, search, memory) primitives.

9.3.2 Beyond backpropagation and differentiable layers

If machine learning models become more like programs, then they will mostly no longer be differentiable—certainly, these programs will still leverage continuous geometric layers as subroutines, which will be differentiable, but the model as a whole would not be. As a result, using backpropagation to adjust weight values in a fixed, hard-coded network, cannot be the method of choice for training models in the future—at least, it cannot be the whole story. We need to figure out to train non-differentiable systems efficiently. Current approaches include genetic algorithms, "evolution strategies", certain reinforcement learning methods, and ADMM (alternating direction method of multipliers). Naturally, gradient descent is not going anywhere—gradient information will always be useful for optimizing differentiable parametric functions. But our models will certainly become increasingly more ambitious than mere differentiable parametric functions, and thus their automatic development (the "learning" in "machine learning") will require more than backpropagation.

Besides, backpropagation is end-to-end, which is a great thing for learning good

chained transformations, but is rather computationally inefficient since it doesn't fully leverage the modularity of deep networks. To make something more efficient, there is one universal recipe: introduce modularity and hierarchy. So we can make backprop itself more efficient by introducing decoupled training modules with some synchronization mechanism between them, organized in a hierarchical fashion. This strategy is somewhat reflected in DeepMind's recent work on "synthetic gradients". I would expect more work along these lines in the near future.

One can imagine a future where models that would be globally non-differentiable (but would feature differentiable parts) would be trained—grown—using an efficient search process that would not leverage gradients, while the differentiable parts would be trained even faster by taking advantage of gradients using some more efficient version of backpropagation.

9.3.3 Automated machine learning

In the future, model architectures will be learned, rather than handcrafted by engineer-artisans. Learning architectures automatically goes hand in hand with the use of richer sets of primitives and program-like machine learning models.

Currently, most of the job of a deep learning engineer consists in munging data with Python scripts, then lengthily tuning the architecture and hyperparameters of a deep network to get a working model—or even, to get to a state-of-the-art model, if the engineer is so ambitious. Needless to say, that is not an optimal setup. But AI can help there too. Unfortunately, the data munging part is tough to automate, since it often requires domain knowledge as well as a clear high-level understanding of what the engineer wants to achieve. Hyperparameter tuning, however, is a simple search procedure, and we already know what the engineer wants to achieve in this case: it is defined by the loss function of the network being tuned. It is already common practice to set up basic "AutoML" systems that will take care of most of the model knob tuning. I even set up my own years ago to win Kaggle competitions.

At the most basic level, such a system would simply tune the number of layers in a stack, their order, and the number of units or filters in each layer. This is commonly done with libraries such as Hyperopt, which we discussed in Chapter 7. But we can also be far more ambitious, and attempt to learn an appropriate architecture from scratch, with as few constraints as possible. This is possible via reinforcement learning, for instance, or genetic algorithms.

Another important AutoML direction is to learn model architecture jointly with model weights. Because training a new model from scratch every time we try a slightly different architecture is tremendously inefficient, a truly powerful AutoML system would manage to evolve architectures at the same time as the features of the model are being tuned via backprop on the training data. Such approaches are already starting to emerge as I am writing these lines.

When this starts happening, the jobs of machine learning engineers will not disappear—rather, engineers will move higher up the value creation chain. They will start

putting a lot more effort into crafting complex loss functions that truly reflect business goals, and understanding deeply how their models impact the digital ecosystems in which they are deployed (e.g. the users that consume the model’s predictions and generate the model’s training data) —problems that currently only the largest company can afford to consider.

9.3.4 Lifelong learning and modular subroutine reuse

If models get more complex and are built on top of richer algorithmic primitives, then this increased complexity will require higher reuse between tasks, rather than training a new model from scratch every time we have a new task or a new dataset. Indeed, a lot of datasets would not contain enough information to develop a new complex model from scratch, and it will become necessary to leverage information coming from previously encountered datasets. Much like you don’t learn English from scratch every time you open a new book—that would be impossible. Besides, training models from scratch on every new task is very inefficient due to the large overlap between the current tasks and previously encountered tasks.

Additionally, a remarkable observation that has been made repeatedly in recent years is that training a *same* model to do several loosely connected tasks at the same time results in a model that is *better at each task*. For instance, training a same neural machine translation model to cover both English-to-German translation and French-to-Italian translation will result in a model that is better at each language pair. Training an image classification model jointly with an image segmentation model, sharing the same convolutional base, results in a model that is better at both tasks. And so on. This is fairly intuitive: there is always *some* information overlap between these seemingly disconnected tasks, and the joint model has thus access to a greater amount of information about each individual task than a model trained on that specific task only.

What we currently do along the lines of model reuse across tasks is to leverage pre-trained weights for models that perform common functions, like visual feature extraction. You saw this in action in Chapter 5. In the future, I would expect a generalized version of this to be commonplace: we would not only leverage previously learned features (submodel weights), but also model architectures and training procedures. As models become more like programs, we would start reusing *program subroutines*, like the functions and classes found in human programming languages.

Think of the process of software development today: once an engineer solves a specific problem (HTTP queries in Python, for instance), they will package it as an abstract and reusable library. Engineers that face a similar problem in the future can simply search for existing libraries, download one and use it in their own project. In a similar way, in the future, meta-learning systems will be able to assemble new programs by sifting through a global library of high-level reusable blocks. When the system would find itself developing similar program subroutines for several different tasks, it would come up with an “abstract”, reusable version of the subroutine and would store it in the global library. Such a process would implement the capability for *abstraction*, a

necessary component for achieving "extreme generalization": a subroutine that is found to be useful across different tasks and domains can be said to "abstract" some aspect of problem-solving. This definition of "abstraction" is similar to the notion of abstraction in software engineering. These subroutines could be either geometric (deep learning modules with pre-trained representations) or algorithmic (closer to the libraries that contemporary software engineers manipulate).

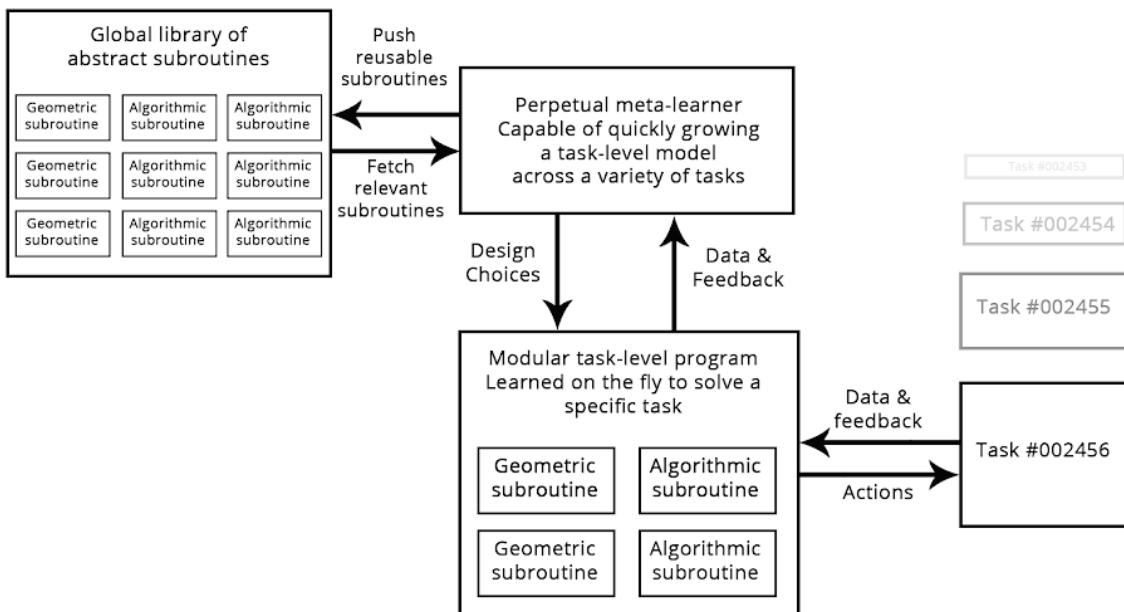


Figure 9.6 A meta-learner capable of quickly developing task-specific models using reusable primitives (both algorithmic and geometric), thus achieving "extreme generalization".

9.3.5 In summary: the long-term vision

In short, here is my long-term vision for machine learning:

- Models will be more like programs, and will have capabilities that go far beyond the continuous geometric transformations of the input data that we currently work with. These programs will arguably be much closer to the abstract mental models that humans maintain about their surroundings and themselves, and they will be capable of stronger generalization due to their rich algorithmic nature.
- In particular, models will blend *algorithmic modules* providing formal reasoning, search, and abstraction capabilities, with *geometric modules* providing informal intuition and pattern recognition capabilities. AlphaGo (a system that required a lot of manual software engineering and human-made design decisions) provides an early example of what such a blend between symbolic and geometric AI could look like.
- They will be *grown* automatically rather than hard-coded by human engineers, using modular parts stored in a global library of reusable subroutines—a library evolved by learning high-performing models on thousands of previous tasks and datasets. As frequent problem-solving patterns are identified by the meta-learning system, they would be turned into a reusable subroutine—much like functions and classes in software engineering—and added to the global library. This achieves the capability for *abstraction*.
- This global library and associated model-growing system will be able to achieve some form of human-like "extreme generalization": given a new task, a new situation, the

system would be able to assemble a new working model appropriate for the task using very little data, thanks to 1) rich program-like primitives that generalize well and 2) extensive experience with similar tasks. In the same way that humans can learn to play a complex new video game using very little play time because they have experience with many previous games, and because the models derived from this previous experience are abstract and program-like, rather than a basic mapping between stimuli and action.

- As such, this perpetually learning model-growing system could be interpreted as an AGI—an Artificial General Intelligence. But don't expect any singularitarian robot apocalypse to ensue: that's a pure fantasy, coming from a long series of profound misunderstandings of both intelligence and technology. This critique, however, does not belong in this book.

9.4 Staying up to date in a fast-moving field

As final parting words, I would like to give you some pointers on how to keep learning and updating your knowledge and skills after you've turned the last page. The field of modern deep learning, as we know it today, is only a few years old—despite a long, slow prehistory stretching back decades. With an exponential increase in financial resources and research headcount since 2013, the field as a whole has been moving at a frenetic pace. You cannot hope that what you've learned in this book will forever stay relevant, or that it will be all that will need for the rest of your career.

Thankfully, there are plenty of free online resources that you can leverage to stay up to date and expand your horizons. Here are a few.

9.4.1 Practice on real-world problems using Kaggle

One very effective way to acquire real-world experience is to try your hand at machine learning competitions on kaggle.com. The only real way to learn is through practice and actual coding—that's the philosophy of this book, and Kaggle competitions are the natural continuation of this. On Kaggle, you will find an array of constantly renewed data science competitions, a lot of them involving deep learning, prepared by companies interested in obtaining novel solutions to some of their most challenging machine learning problems. There are fairly large money prizes offered to top entrants.

Most competitions are won using either the XGBoost library (for shallow machine learning) or Keras (for deep learning). So you will fit right in! By participating in a few competitions, maybe as part of a team, you will become more familiar with the practical side of some of the advanced best practices we have described in this book, especially hyperparameter tuning, avoiding validation set overfitting, and model ensembling.

9.4.2 Read about the latest developments on Arxiv

Deep learning research, in contrast with some other scientific fields, takes places completely in the open. Papers are made publicly and freely accessible as soon as they are finalized, and a lot of related software gets open-sourced. arxiv.org (pronounced "archive"—that x stands for a greek chi) is an open-access preprint server for Physics, Mathematics, and Computer Science research papers. It has become the de-facto way to stay up-to-date on the bleeding edge of machine learning and deep learning. The large majority of deep learning researchers will upload any paper they write to Arxiv shortly after completion. This allows them to "plant a flag" and claim a specific finding without waiting for a conference acceptance (which takes half a year), which is necessary given the very fast pace of research and the intense competition in the field. It also allows the field to move extremely fast: all new findings are immediately available for all to see and to build upon.

An important downside is that the sheer quantity of new papers getting posted every day on Arxiv makes it impossible to even skim through them all, and the fact that they are not peer-reviewed makes it difficult to identify those that are both important and high-quality. It is difficult, and getting increasingly more difficult, to find the signal in the noise. Currently, there's isn't a good solution to this problem. But some tools can help: an auxiliary website called arxiv-sanity.com can serve a recommendation engine for new papers and can help you keep track of new developments within a specific narrow vertical of deep learning. Additionally, you can use Google Scholar to keep track of the publications of your favorite authors.

9.4.3 Explore the Keras ecosystem

With about 150,000 users as of June 2017, and fast growing, Keras has a large ecosystem of tutorials, guides and related open-source projects.

- Your main reference for working with Keras itself is the online documentation at keras.io.
- The Keras source code can be found at github.com/fchollet/keras.
- You can ask for help and join deep learning discussions on the Keras Slack channel, kerasteam.slack.com.
- The Keras blog, blog.keras.io, offers Keras tutorials and other deep learning-related articles.

Also, you can follow me on Twitter: @fchollet.

9.5 Final words

That's the end of Deep Learning with Python! I hope you have learned a thing or two about machine learning, deep learning, Keras, and maybe even cognition in general. Learning is a lifelong journey, especially in the field of AI, where we have far more unknowns on our hands than certitudes. So please go on learning, questioning, and researching. Never stop. Because for all the progress made so far, it seems like most of the fundamental questions in AI remain unanswered. Many have not even been properly asked yet.