Sandro Skansi

Introduction to Deep Learning

From Logical Calculus to Artificial Intelligence

Sandro Skansi
University of Zagreb
Zagreb
Croatia

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-73003-5 ISBN 978-3-319-73004-2 (eBook)
https://doi.org/10.1007/978-3-319-73004-2

Library of Congress Control Number: 2017963994

Preface

This textbook contains no new scientific results, and my only contribution was to compile existing knowledge and explain it with my examples and intuition. I have made a great effort to cover everything with citations while maintaining a fluent exposition, but in the modern world of the 'electron and the switch' it is very hard to properly attribute all ideas, since there is an abundance of quality material online (and the online world became very dynamic thanks to the social media). I will do my best to correct any mistakes and omissions for the second edition, and all corrections and suggestions will be greatly appreciated.

This book uses the feminine pronoun to refer to the reader regardless of the actual gender identity. Today, we have a highly imbalanced environment when it comes to artificial intelligence, and the use of the feminine pronoun will hopefully serve to alleviate the alienation and make the female reader feel more at home while reading this book.

Throughout this book, I give historical notes on when a given idea was first discovered. I do this to credit the idea, but also to give the reader an intuitive timeline. Bear in mind that this timeline can be deceiving, since the time an idea or technique was first invented is not necessarily the time it was adopted as a technique for machine learning. This is often the case, but not always.

This book is intended to be a first introduction to deep learning. Deep learning is a special kind of learning with deep artificial neural networks, although today deep learning and artificial neural networks are considered to be the same field. Artificial neural networks are a subfield of machine learning which is in turn a subfield of both statistics and artificial intelligence (AI). Artificial neural networks are vastly more popular in artificial intelligence than in statistics. Deep learning today is not happy with just addressing a subfield of a subfield, but tries to make a run for the whole AI. An increasing number of AI fields like reasoning and planning, which were once the bastions of logical AI (also called the *Good Old-Fashioned AI*, or GOFAI), are now being tackled successfully by deep learning. In this sense, one might say that deep learning is an approach in AI, and not just a subfield of a subfield of AI.

There is an old idea from Kendo[1] which seems to find its way to the new world of cutting-edge technology. The idea is that you learn a martial art in four stages: big, strong, fast, light. 'Big' is the phase where all movements have to be big and correct. One here focuses on correct techniques, and one's muscles adapt to the new movements. While doing big movements, they unconsciously start becoming strong. 'Strong' is the next phase, when one focuses on strong movements. We have learned how to do it correctly, and now we add strength, and subconsciously they become faster and faster. While learning 'Fast', we start 'cutting corners', and adopt a certain 'parsimony'. This parsimony builds 'Light', which means 'just enough'. In this phase, the practitioner is a master, who does everything correctly, and movements can shift from strong to fast and back to strong, and yet they seem effortless and light. This is the road to mastery of the given martial art, and to an art in general. Deep learning can be thought of an art in this metaphorical sense, since there is an element of continuous improvement. The present volume is intended not to be an all-encompassing reference, but it is intended to be the textbook for the "big" phase in deep learning. For the strong phase, we recommend [1], for the fast we recommend [2] and for the light phase, we recommend [3]. These are important works in deep learning, and a well-rounded researcher should read them all.

After this, the 'fellow' becomes a 'master' (and mastery is not the end of the road, but the true beginning), and she should be ready to tackle research papers, which are best found on `arxiv.com` under 'Learning'. Most deep learning researchers are very active on `arxiv.com`, and regularly publish their preprints. Be sure to check out also 'Computation and Language', 'Sound' and 'Computer Vision' categories depending on your desired specialization direction. A good practice is just to put the desired category on your web browser home screen and check it daily. Surprisingly, the `arxiv.com` 'Neural and Evolutionary Computation' is not the best place for finding deep learning papers, since it is a rather young category, and some researchers in deep learning do not tag their work with this category, but it will probably become more important as it matures.

The code in this book is Python 3, and most of the code using the library Keras is a modified version of the codes presented in [2]. Their book[2] offers a lot of code and some explanations with it, whereas we give a modest amount of code, rewritten to be intuitive and comment on it abundantly. The codes we offer have all been extensively tested, and we hope they are in working condition. But since this book is an introduction and we cannot assume the reader is very familiar with coding deep architectures, I will help the reader troubleshoot all the codes from this book. A complete list of bug fixes and updated codes, as well as contact details for submitting new bugs are available at the book's repository `github.com/skansi/dl_book`, so please check the list and the updated version of the code before submitting a new bug fix request.

---

[1] A Japanese martial art similar to fencing.

[2] This is the only book that I own two copies of, one eBook on my computer and one hard copy—it is simply that good and useful.

Artificial intelligence as a discipline can be considered to be a sort of 'philosophical engineering'. What I mean by this is that AI is a process of taking philosophical ideas and making algorithms that implement them. The term 'philosophical' is taken broadly as a term which also encompasses the sciences which recently[3] became independent sciences (psychology, cognitive science and structural linguistics), as well as sciences that are hoping to become independent (logic and ontology[4]).

Why is philosophy in this broad sense so interesting to replicate? If you consider what topics are interesting in AI, you will discover that AI, at the most basic level, wishes to replicate philosophical concepts, e.g. to build machines that can think, know stuff, understand meaning, act rationally, cope with uncertainty, collaborate to achieve a goal, handle and talk about objects. You will rarely see a definition of an AI agent using non-philosophical terms such as 'a machine that can route internet traffic', or 'a program that will predict the optimal load for a robotic arm' or 'a program that identifies computer malware' or 'an application that generates a formal proof for a theorem' or 'a machine that can win in chess' or 'a subroutine which can recognize letters from a scanned page'. The weird thing is, all of these are actual historical AI applications, and machines such as these always made the headlines.

But the problem is, once we got it to work, it was no longer considered 'intelligent', but merely an elaborate computation. AI history is full of such examples.[5] The systematic solution of a certain problem requires a full formal specification of the given problem, and after a full specification is made, and a known tool is applied to it,[6] it stops being considered a mystical human-like machine and starts being considered 'mere computation'. Philosophy deals with concepts that are inherently tricky to define such as knowledge, meaning, reference, reasoning, and all of them are considered to be essential for intelligent behaviour. This is why, in a broad sense, AI is the engineering of philosophical concepts.

But do not underestimate the engineering part. While philosophy is very prone to reexamining ideas, engineering is very progressive, and once a problem is solved, it is considered done. AI has the tendency to revisit old tasks and old problems (and this makes it very similar to philosophy), but it does require measurable progress, in the sense that new techniques have to bring something new (and this is its

[3]Philosophy is an old discipline, dating back at least 2300 years, and 'recently' here means 'in the last 100 years'.

[4]Logic, as a science, was considered independent (from philosophy and mathematics) by a large group of logicians for at least since Willard Van Orman Quine's lectures from the 1960s, but thinking of ontology as an independent discipline is a relatively new idea, and as far as I was able to pinpoint it, this intriguing and promising initiative came from professor Barry Smith form the Department of Philosophy of the University of Buffalo.

[5]John McCarthy was amused by this phenomenon and called it the 'look ma', no hands' period of AI history, but the same theme keeps recurring.

[6]Since new tools are presented as new tools for existing problems, it is not very common to tackle a new problem with newly invented tools.

engineering side). This novelty can be better results than the last result on that problem,[7] the formulation of a new problem[8] or results below the benchmark but which can be generalized to other problems as well.

Engineering is progressive, and once something is made, it is used and built upon. This means that we do not have to re-implement everything anew—there is no use in reinventing the wheel. But there is value to be gained in understanding the idea behind the invention of the wheel and in trying to make a wheel by yourself. In this sense, you should try to recreate the codes we will be exploring, and see how they work and even try to re-implement a completed Keras layer in plain Python. It is quite probable that if you manage your solution will be considerably slower, but you will have gained insight. When you feel you understand it as much as you would like, you should just use Keras or any other framework as building bricks to go on and build more elaborate things.

In today's world, everything worth doing is a team effort and every job is then divided in parts. My part of the job is to get the reader started in deep learning. I would be proud if a reader would digest this volume, put it on a shelf, become and active deep learning researcher and never consult this book again. To me, this would mean that she has learned everything there was in this book and this would entail that my part of the job of getting one started[9] in deep learning was done well. In philosophy, this idea is known as Wittgenstein's ladder, and it is an important practical idea that will supposedly help you in your personal exploration–exploitation balance.

I have also placed a few Easter eggs in this volume, mainly as unusual names in examples. I hope that they will make reading more lively and enjoyable. For all who wish to know, the name of the dog in Chap. 3 is Gabi, and at the time of publishing, she will be 4 years old. This book is written in plural, following the old academic custom of using *pluralis modestiae*, and hence after this preface I will no longer use the singular personal pronoun, until the very last section of the book.

I would wish to thank everyone who has participated in any way and made this book possible. In particular, I would like to thank Siniša Urošev, who provided valuable comments and corrections of the mathematical aspects of the book, and to Antonio Šajatović, who provided valuable comments and suggestions regarding memory-based models. Special thanks go to my wife Ivana for all the support she gave me. I hold myself (and myself alone) responsible for any omissions or mistakes, and I would greatly appreciate all feedback from readers.

Zagreb, Croatia — Sandro Skansi

---

[7]This is called the *benchmark* for a given problem, it is something you must surpass.

[8]Usually in the form of a new dataset constructed from a controlled version of a philosophical problem or set of problems. We will have an example of this in the later chapters when we will address the bAbI dataset.

[9]Or, perhaps, 'getting initiated' would be a better term—it depends on how fond will you become of deep learning.

References

1. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT press, Cambridge, 2016)
2. A. Gulli, S. Pal, *Deep Learning with Keras* (Packt publishing, Birmingham, 2017)
3. G. Montavon, G. Orr, K.R. Muller, *Neural Networks: Tricks of the Trade* (Springer, New York, 2012)

Contents

From Logic to Cognitive Science 1

1.1 The Beginnings of Artificial Neural Networks

Artificial intelligence has it roots in two philosophical ideas of Gottfried Leibniz, the great seventeenth-century philosopher and mathematician, viz. the *characteristica universalis* and the *calculus ratiocinator*. The *characteristica universalis* is an idealized language, in which all of science could in principle be translated. It would be language in which every natural language would translate, and as such it would be the language of pure meaning, uncluttered by linguistic technicalities. This language can then serve as a background for explicating rational thinking, in a manner so precise, a machine could be made to replicate it. The *calculus ratiocinator* would be a name for such a machine. There is a debate among historians of philosophy whether this would mean making a software or a hardware, but this is in fact a insubstantial question since to get the distinction we must understand the concept of an universal machine accepting different instructions for different tasks, an idea that would come from Alan Turing in 1936 [1] (we will return to Turing shortly), but would become clear to a wider scientific community only in the late 1970s with the advent of the personal computer. The ideas of the *characteristica universalis* and the *calculus ratiocinator* are Leibniz' central ideas, and are scattered throughout his work, so there is no single point to reference them, but we point the reader to the paper [2], which is a good place to start exploring.

The journey towards deep learning continues with two classical nineteenth century works in logic. This is usually omitted since it is not clearly related to neural networks, there was a strong influence, which deserves a couple of sentences. The first is John Stuart Mill's *System of Logic* from 1843 [3], where for the first time in history, logic is explored in terms of a manifestation of a mental process. This approach,

called *logical psychologism*, is still researched only in philosophical logic,[1] but even in philosophical logic it is considered a fringe theory. Mill's book never became an important work, and his work in ethics overshadowed his contribution to logical psychologism, but fortunately there was a second book, which was highly influential. It was the *Laws of Thought* by George Boole, published in 1854 [4]. In his book, Boole systematically presented logic as a system of formal rules which turned out to be a major milestone in the reshaping of logic as a formal science. Quickly after, formal logic developed, and today it is considered a native branch of both philosophy and mathematics, with abundant applications to computer science. The difference in these 'logics' is not in the techniques and methodology, but rather in applications. The core results of logic such as De Morgan's laws, or deduction rules for first-order logic, remain the same across all sciences. But exploring formal logic beyond this would take us away from our journey. What is important here is that during the first half of the twentieth century, logic was still considered to be something connected with the laws of thinking. Since thinking was the epitome of intelligence, it was only natural that artificial intelligence started out with logic.

Alan Turing, the father of computing, marked the first step of the birth of artificial intelligence with his seminal 1950 paper [5] by introducing the Turing test to determine whether a computer can be regarded intelligent. A Turing test is a test in natural language administered to a human (who takes the role of the referee). The human communicates with a person and a computer for five minutes, and if the referee cannot tell the two apart, the computer has passed the Turing test and it may be regarded as intelligent. There are many modifications and criticism, but to this day the Turing test is one of the most widely used benchmarks in artificial intelligence.

The second event that is considered the birth of artificial intelligence was the Dartmouth Summer Research Project on Artificial Intelligence. The participants were John McCarthy, Marvin Minsky, Julian Bigelow, Donald MacKay, Ray Solomonoff, John Holland, Claude Shannon, Nathanial Rochester, Oliver Selfridge, Allen Newell and Herbert Simon. Quoting the proposal, the conference was *to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it*.[2] This premise made a substantial mark in the years to come, and mainstream AI would become logical AI. This logical AI would go unchallenged for years, and would eventually be overthrown only in the 21 millennium by a new tradition, known today as deep learning. This tradition was actually older, founded more than a decade earlier in 1943, in a paper written by a logician of a different kind, and his co-author, a philosopher and psychiatrist. But, before we continue, let us take a small step back. The interconnection between logical rules and thinking was seen as directed. The common knowledge is that the logical rules are grounded in thinking. Artificial intelligence asked whether we can impersonate thinking in a machine with

[1] Today, this field of research can be found under a refreshing but very unusual name: 'logic in the wild'.

[2] The full text of the proposal is available at https://www.aaai.org/ojs/index.php/aimagazine/article/view/1904/1802.

logical rules. But there was another direction which is characteristic of philosophical logic: could we model thinking as a human mental process with logical rules? This is where the neural network history begins, with the seminal paper by Walter Pitts and Warren McCulloch titled *A Logical Calculus of Ideas Immanent in Nervous Activity* and published in the *Bulletin of Mathematical Biophysics*. A copy of the paper is available at `http://www.cs.cmu.edu/~epxing/Class/10715/reading/McCulloch.and.Pitts.pdf`, and we advise the student to try to read it to get a sense of how deep learning began.

Warren McCulloch was a philosopher, psychologist and psychiatrist by degree, but he would work in neurophysiology and cybernetics. He was a vivid character, embodying many academic stereotypes, and as such was a curious person whose interests could be described as interdisciplinary. He met the homeless Walter Pitts in 1942 when he got a job at the Department of Psychiatry at the University of Chicago, and invited Pitts to come to live with his family. They shared a lifelong interested in Leibniz, and they wanted to bring his ideas to fruition an create a machine which could implement logical reasoning.[3] The two men worked every night on their idea of capturing reasoning with a logical calculus inspired by the biological neurons. This meant constructing a formal neuron with capabilities similar to that of a Turing machine. The paper had only three references, and all of them are classical works in logic: Carnap's *Logical Syntax of Language* [6], Russell's and Whitehead's *Principa Mathematica* [7] and the Hilbert and Ackermann *Grundüge der Theoretischen Logik*. The paper itself approached the problem of neural networks as a logical one, proceeding from definitions, over lemmas to theorems.

Their paper introduced the idea of the artificial neural network, as well as some of the definitions we take for granted today. One of these is what would it mean for a logical predicate to be realizable on a neural network. They divided the neurons in two groups, the first called *peripheral afferents* (which are now called 'input neurons'), and the rest, which are actually output neurons, since at this time there was no hidden layer—the hidden layer came to play only in the 1970s and 1980s. Neurons can be in two states, firing and non-firing, and they define for every neuron i a predicate which is true when the neuron is firing at the moment t. This predicate is denoted as $N_i(t)$. The *solution* of a network is then an equivalence of the form $N_i(t) \equiv B$ where B is a conjunction of firings from the previous moment of the peripheral afferents, and i is not an input neuron. A sentence like this is realizable in a neural network if and only if the network can compute it, and all sentences for which there is a network which computes them are called a *temporal propositional expression* (TPE). Notice that TPEs have a logical characterization. The main result of the paper (asides from defining artificial neural networks) is that any TPE can be computed by an artificial neural network. This paper would be cited later by John von Neumann as a major influence in his own work. This is just a short and incomplete glimpse into this exciting historical paper, but let us return to the story of the second protagonist.

[3] This was 15 years before artificial intelligence was defined as a scientific field.

Walter Pitts was an interesting person, and, one could argue, *the* father of artificial neural networks. At the age of 12, he ran away from home and hid in a library, where he read *Principia Mathematica* [7] by the famous logician Bertrand Russell. Pitts contacted Russell, who invited him to come to study at Cambridge under his tutorship, but Pitts was still a child. Several years later, Pitts, now a teenager, found out that Russell was holding a lecture at the University of Chicago. He met with Russell in person, and Russell told him to go and meet his old friend from Vienna, the logician Rudolph Carnap, who was a professor there. Carnap gave Pitts his seminal book *Logical Syntax of Language*[4] [6], which would highly influence Pitts in the following years. After his initial contact with Carnap, Pitts disappeared for a year, and Carnap could not find him, but after he did, he used his academic influence to get Pitts a student job at the university, so that Pitts does not have to do menial jobs during days and ghostwrite student papers during nights just to survive.

Another person Pitts met during Russell was Jerome Lettvin, who at the time was a pre-med student there, and who would later become neurologist and psychiatrist by degree, but he will also write papers in philosophy and politics. Pitts and Lettvin became close friends, and would eventually write an influential paper together (along with McCulloch and Maturana) titled *What the Frog's Eye Tells the Frog's Brain* in 1959 [8]. Lettvin would also introduce Pitts to the mathematician Norbert Weiner from MIT who later became known as the father of cybernetics, a field colloquially known as 'the science of steering', dedicated to studying system control both in biological and artificial systems. Weiner invited Pitts to come to work at MIT (as a lecturer in formal logic) and the two men worked together for a decade. Neural networks were at this time considered to be a part of cybernetics, and Pitts and McCulloch were very active in the field, both attending the Macy conferences, with McCulloch becoming the president of the American Society for Cybernetics in 1967–1968. During his stay at Chicago, Pitts also met the theoretical physicist Nicolas Rashevsky, who was a pioneer in mathematical biophysics, a field which tried to explain biological processes with a combination of logic and physics. Physics might seem distant to neural networks, but in fact, we will soon discuss the role physicists played in the history of deep learning.

Pitts would remain connected with the University, but he had minor jobs there due to his lack of formal academic credentials, and in 1944 was hired by the Kellex Corporation (with the help of Weiner), which participated in the Manhattan project. He detested the authoritarian General Groves (head of the Manhattan project), and would play pranks to mock the strict and sometimes meaningless rules that he enacted. He was granted an Associate of Arts degree (2-year degree) by the University of Chicago as a token of recognition of his 1943 paper, and this would remain the only academic degree he ever earned. He has never been fond of the usual academic procedures and this posed a major problem in his formal education. As an illustration, Pitts attended a

[4]The author has a fond memory of this book, but beware: here be dragons. The book is highly complex due to archaic notation and a system quite different from today's logic, but it is a worthwhile read if you manage to survive the first 20 pages.

course taught by professor Wilfrid Rall (the pioneer of computational neuroscience), and Rall remembered Pitts as 'an oddball who felt compelled to criticize exam questions rather than answer them'.

In 1952, Norbert Weiner broke all relations with McCulloch, which devastated Pitts. Weiner wife accused McCulloch that his boys (Pitts and Lettvin) seduced their daughter, Barbara Weiner. Pitts turned to alcohol to the point that he could not take care of his dog anymore,[5] and succumbed to cirrhosis complications in 1969, at the age of 46. McCulloch died the same year at the age of 70. Both of the Pitts' papers we mentioned remain to this day two of the most cited papers in all of science. It is interesting to note that even though Pitts had direct or mediated contact with most of the pioneers of AI, Pitts himself never thought about his work as geared towards building a machine replica of the mind, but rather as a quest to formalize and better understand human thinking [9], and that puts him squarely in the realm of what is known today as philosophical logic.[6]

The story of Walter Pitts is a story of influences of ideas and of collaboration between scientists of different backgrounds, and in a way a neural network nicely symbolizes this interaction. One of the main aims of this book is to (re-)introduce neural networks and deep learning to all the disciplines[7] which contributed to the birth and formation of the field, but currently shy away from it. The majority of the story about Walter Pitts we presented is taken from a great article named *The man who tried to redeem the world with logic* by Amanda Gefter published in Nautilus [10] and the paper *Walter Pitts* by Neil R. Smalheiser [9], both of which we highly recommend.[8]

1.2 The XOR Problem

In the 1950s, the Dartmouth conference took place and the interest of the newly born field of artificial intelligence in neural networks was evident from the very conference manifest. Marvin Minsky, one of the founding fathers of AI and participant to the Dartmouth conference was completing his dissertation at Princeton in 1954, and the title was *Neural Nets and the Brain Model Problem*. Minsky's thesis addressed several technical issues, but it became the first publication which collected all up to date results and theorems on neural networks. In 1951, Minsky built a machine (funded

[5] A Newfoundland, name unknown.

[6] An additional point here is the great influence of Russell and Carnap on Pitts. It is a great shame that many logicians today do not know of Pitts, and we hope the present volume will help bring the story about this amazing man back to the community from which he arose, and that he will receive the place he deserves.

[7] And any other scientific discipline which might be interested in studying or using deep neural networks.

[8] Also, there is a webpage on Pitts `http://www.abstractnew.com/2015/01/walter-pitts-tribute-to-unknown-genius.html` worth visiting.

by the Air Force Office of Scientific Research) which implemented neural networks called SNARC (Stochastic Neural Analog Reinforcement Calculator), which was the first major computer implementation of a neural network. As a bit of trivia, Marvin Minsky was an advisor to Arthur C. Clarke's and Stanley Kubrick's *2001: A Space Odyssey* movie. Also, Isaac Asimov claimed that Marvin Minsky was one of two people he has ever met whose intelligence surpassed his own (the other one being Carl Sagan). Minsky will return to our story soon, but first let us present another hero of deep learning.

Frank Rosenblatt received his PhD in Psychology at Cornell University in 1956. Rosenblatt made a crucial contribution to neural networks, by discovering the *perceptron learning rule*, a rule which governs *how* to update the weights of neural networks, which we shall explore in detail in the forthcoming chapters. His perceptrons were initially developed as a program on an IBM 704 computer at Cornell Aeronautical Laboratory in 1957, but Rosenblatt would eventually develop the Mark I Perceptron, a computer built with the sole purpose of implementing neural networks with the perceptron rule. But Rosenblatt did more than just implement the perceptron. His 1962 book *Principles of Neurodynamics* [11] explored a number of architectures, and his paper [12] explored the idea of multilayered networks similar to modern convolutional networks, which he called *C-system*, which might be seen as the theoretical birth of deep learning. Rosenblatt died in 1971 on his 43rd birthday in a boating accident.

There were two major trends underlying the research in the 1960s. The first one was the results that were delivered by programs working on symbolic reasoning, using deductive logical systems. The two most notable were the Logic Theorist by Herbert Simon, Cliff Shaw and Allen Newell, and their later program, the General Problem Solver [13]. Both programs produced working results, something neural networks did not. Symbolic systems were also appealing since they seemed to provide control and easy extensibility. The problem was not that neural networks were not giving any result, just that the results they have been giving (like image classification) were not really considered that intelligent at the time, compared to symbolic systems that were proving theorems and playing chess—which were the hallmark of human intelligence. The idea of this intelligence hierarchy was explored by Hans Moravec in the 1980s [14], who concluded that symbolic thinking is considered a rare and desirable aspect of intelligence in humans, but it comes rather natural to computers, which have much more trouble with reproducing 'low-level' intelligent behaviour that many humans seem to exhibit with no trouble, such as recognizing that an animal in a photo is a dog, and picking up objects.[9]

The second trend was the Cold War. Starting with 1954, the US military wanted to have a program to automatically translate Russian documents and academic papers.

[9]Even today people consider playing chess or proving theorems as a higher form of intelligence than for example gossiping, since they point to the rarity of such forms of intelligence. The rarity of an aspect of intelligence does not directly correlate with its computational properties, since problems that are computationally easy to describe are easier to solve regardless of the cognitive rarity in humans (or machines for that matter).

Funding was abundant, but many technically inclined researchers underestimated the linguistic complexities involved in extracting meaning from words. A famous example was the back and forth translation from English to Russian and back to English of the phrase 'the spirit was willing but the flesh was weak' which produced the sentence 'the vodka was good, but the meat was rotten'. In 1964, there were some concerns about wasting government money in a dead end, so the National Research Council formed the Automatic Language Processing Advisory Committee or ALPAC [13]. The ALPAC report from 1966 cut funding to all machine translation projects, and without the funding, the field lingered. This in turn created turmoil in the whole AI community.

But the final stroke which nearly killed off neural networks came in 1969, from Marvin Minsky and Seymour Papert [15], in their monumental book *Perceptrons: An Introduction to Computational Geometry*. Remember that McCulloch and Pitts proved that a number of logical functions can be computed with a neural network. It turns out, as Minsky and Papert showed in their book, they missed a simple one, the equivalence. The computer science and AI community tend to favour looking at this problem as the XOR function, which is the negation of an equivalence, but it really does not matter, since the only thing different is how you place the labels.

It turns out that perceptrons, despite the peculiar representations of the data they process, are only linear classifiers. The perceptron learning procedure is remarkable, since it is guaranteed to converge (terminate), but it did not add a capability of capturing nonlinear regularities to the neural network. The XOR is a nonlinear problem, but this is not clear at first.[10] To see the problem, imagine[11] a simple 2D coordinate system, with only 0 and 1 on both axes. The XOR of 0 and 0 is 0, and write an O at coordinates (0, 0). The XOR of 0 and 1 is 1, and now write an X at the coordinates (0,1). Continue with $\mathrm{XOR}(1, 0) = 1$ and $\mathrm{XOR}(1, 1) = 0$. You should have two Xs and two Os. Now imagine you are the neural network, and you have to find out how to draw a curve to separate the Xs from Os. If you can draw anything, it is easy. But you are not a modern neural network, but a perceptron, and you must use a straight line—no curves. It soon becomes obvious that this is impossible.[12] The problem with the perceptron was the linearity. The idea of a multilayered perceptron was here, but it was impossible to build such a device with the perceptron learning rule. And so, seemingly, no neural network could handle (learn to compute) even the basic logical operations, something symbolic systems could do in an instant. A quiet darkness fell across the neural networks, lasting many years. One might wonder what was happening in the USSR at this time, and the short answer is that cybernetics, as neural networks were still called in the USSR in this period, was considered a bourgeois pseudoscience. For a more detailed account, we refer the reader to [16].

---

[10] The view is further dimmed by the fact that the perceptron could process an image (at least rudimentary), which intuitively seems to be quite harder than simple logical operations.

[11] Pick up a pen and paper and draw along.

[12] If you wish to try the equivalence instead of XOR, you should do the same but with $\mathrm{EQUIV}(0, 0) = 1$, $\mathrm{EQUIV}(0, 1) = 0$, $\mathrm{EQUIV}(1, 0) = 0$, $\mathrm{EQUIV}(1, 1) = 1$, keeping the Os for 0 and Xs for 1. You will see it is literally the same thing as XOR in the context of our problem.

1.3 From Cognitive Science to Deep Learning

But the idea of neural networks lingered on in the minds of only a handful of believers. But there were processes set in motion which would enable their return in style. In the context of neural networks, the 1970s were largely uneventful. But there we two trends present which would help the revival of the 1980s. The first one was the advent of cognitivism in psychology and philosophy. Perhaps the most basic idea that cognitivism brought in the mainstream is the idea that the mind, as a complex system made from many interacting parts, should explored on its own (independent of the brain), but with formal methods.[13] While the neurological reality that determines cognition should not be ignored, it can be helpful to build and analyse systems that try to recreate portions of the neurological reality, and at the same time they should be able to recreate some of the behaviour. This is a response to both Skinner's behaviourism [18] in psychology of the 1950s, which aimed to focus a scientific study of the mind as a black box processor (everything else is purely speculation[14]) and to the dualism of the mind and brain which was strongly implied by a strict formal study of knowledge in philosophy (particularly as a response to Gettier [19]).

Perhaps one of the key ideas in the whole scientific community at that time was the idea of a paradigm shift in science, proposed by Thomas Kuhn in 1962 [20], and this was undoubtedly helpful to the birth of cognitive science. By understanding the idea of the paradigm shift, for the first time in history, it felt legitimate to abandon a state-of-the-art method for an older, underdeveloped idea and then dig deep into that idea and bring it to a whole new level. In many ways, the shift proposed by cognitivism as opposed to the older behavioural and causal explanations was a shift from studying an immutable structure towards the study of a mutable change. The first truly cognitive turn in the so-called cognitive sciences is probably the turn made in linguistics by Chomsky's universal grammar [21] and his earlier and ingenious attack on Skinner [22]. Among other early contributions to the cognitive revolution, we find the most interesting one the paper from our old friends [23]. This paradigm shift happened across six disciplines (the cognitive sciences), which would become the founding disciplines of cognitive science: anthropology, computer science, linguistic, neuroscience, philosophy and psychology.

The second was another setback in funding caused by a government report. It was the paper *Artificial Intelligence: A General Survey* by James Lighthill [24], which was presented to the British Science Research Council in 1973, and became widely known as the Lighthill report. Following the Lighthill report, the British government would close all but three AI departments in the UK, which forced many scientists to abandon their research projects. One of the three AI departments that survived was Edinburgh. The Lighthill report enticed one Edinburgh professor to issue a statement, and in this statement, cognitive science was referenced for the first time

[13] A great exposition of the cognitive revolution can be found in [17].

[14] It must be acknowledged that Skinner, by insisting on focusing only on the objective and measurable parts of the behaviour, brought scientific rigor into the study of behaviour, which was previously mainly a speculative area of research.

in history, and its scope was roughly defined. It was Christopher Longuet-Higgins, Fellow of the Royal Society, a chemist by formal education, who began work in AI in 1967 when he took a job at the University of Edinburgh, where he joined the Theoretical Psychology Unit. In his reply,[15] Longuet-Higgins asked a number of important questions. He understood that Lighthill wanted the AI community to give a proper justification of AI research. The logic was simple, if AI does not work, why do we want to keep it? Longuet-Higgins provided an answer, which was completely in the spirit of McCulloch and Pitts: we need AI not to build machines (although that would be nice), but to understand humans. But Lighthill was aware of this line of thought, and he has acknowledged in his report that some aspects, in particular neural networks, are scientifically promising. He thought that the study of neural networks can be understood and reclassified as *Computer-based studies of the central nervous system*, but it had to abide by the latest findings of neuroscience, and model neurons as they are, and not weird variations of their simplifications. This is where Longuet-Higgins diverged from Lighthill. He used an interesting metaphor: just like hardware in computers is only a part of the whole system, so is actual neural brain activity, and to study what a computer does, one needs to look at the software, and so to see what a human does, one need to look at mental processes, and *how they interact*. Their interaction is the basis of cognition, all processes taking parts are cognitive processes, and AI needs to address the question of their interaction in a precise and formal way. This is the true knowledge gained from AI research: understanding, modelling and formalizing the interactions of cognitive processes. An this is why we need AI as a field and all of its simplified and sometimes inaccurate and weird models. This is the true scientific gain from AI, and not the technological, martial and economic gain that was initially promised to obtain funding.

Before the turn of the decade, another thing happened, but it went unnoticed. Up until now, the community knew how to train a single-layer neural network, and that having a hidden layer would greatly increase the power of neural networks. The problem was, nobody knew how to train a neural network with more than one layer. In 1975, Paul Werbos [25], an economist by degree, discovered backpropagation, a way to propagate the error back through the hidden (middle) layer. His discovery went unnoticed, and was rediscovered by David Parker [26], who published the result in 1985. Yann LeCun also discovered backpropagation in 1985 and published in [27]. Backpropagation was discovered for the last time in San Diego, by Rumelhart, Hinton and Williams [28], which takes us to the next part of our story, the 1980s, in sunny San Diego, to the cognitive era of deep learning.

The San Diego circle was composed of several researchers. Geoffrey Hinton, a psychologist, was a PhD student of Christopher Longuet-Higgins back in the Edinburgh AI department, and there he was looked down upon by the other faculty, because he wanted to research neural networks, so he called them *optimal networks*

[15] The full text of the reply is available from http://www.chilton-computing.org.uk/inf/literature/reports/lighthill_report/p004.htm.

to avoid problems.[16] After graduating (1978), he came to San Diego as a visiting scholar to the Cognitive Science program at UCSD. There the academic climate was different, and the research in neural networks was welcome. David Rumelhart was one of the leading figures in UCSD. A mathematical psychologist, he is one of the founding fathers of cognitive science, and the person who introduced artificial neural networks as a major topic in cognitive science, under the name of *connectionism*, which had wide philosophical appeal, and is still one of the major theories in the philosophy of mind. Terry Sejnowski, a physicist by degree and later professor of computational biology, was another prominent figure in UCSD at the time, and he co-authored a number of seminal papers with Rumelhart and Hinton. His doctoral advisor, John Hopfield was another physicist who became interested in neural networks, and improved an popularized a recurrent neural network model called the *Hopfield Network* [29]. Jeffrey Elman, a linguist and cognitive science professor at UCSD, who would introduce Elman networks a couple of years later, and Michael I. Jordan, a psychologist, mathematician and cognitive scientist who would introduce Jordan networks (both of these networks are commonly called *simple recurrent networks* in today's literature), also belonged to the San Diego circle.

This leads us to the 1990s and beyond. The early 1990s were largely uneventful, as the general support of the AI community shifted towards *support vector machines* (SVM). These machine learning algorithms are mathematically well founded, as opposed to neural networks which were interesting from a philosophical standpoint, and mainly developed by psychologists and cognitive scientists. To the larger AI community, which still had a lot of the GOFAI drive for mathematical precision, they were uninteresting, and SVMs seemed to produce better results as well. A good reference book for SVMs is [30]. In the late 1990s, two major events occurred, which produced neural networks which are even today the hallmark of deep learning. The *long short-term memory* was invented by Hochreiter and Schmidhuber [31] in 1997, which continue to be one of the most widely used recurrent neural network architectures and in 1998 LeCun, Bottou, Bengio and Haffner produced the first convolutional neural network called LeNet-5 which achieved significant results on the MNIST dataset [32]. Both convolutional neural networks and LSTMs went unnoticed by the larger AI community, but the events were set in motion for neural networks to come back one more time. The final event in the return of neural networks was the 2006 paper by Hinton, Osindero and Teh [33] which introduced *deep belief networks* (DMB) which produces significantly better results on the MNIST dataset. After this paper, the rebranding of deep neural networks to deep learning was complete, and a new period in AI history would begin. Many new architectures followed, and some of them we will be exploring in this book, while some we leave to the reader to explore by herself. We prefer not to write to much about recent history, since it is still actual and there is a lot of factors at stake which hinder objectivity.

[16] The full story about Hinton and his struggles can be found at http://www.chronicle.com/article/The-Believers/190147.

For an exhaustive treatment of the history of neural networks, we point the reader to the paper by Jürgen Schmidhuber [34].

1.4 Neural Networks in the General AI Landscape

We have explored the birth of neural networks from philosophical logic, the role psychology and cognitive science played in their development and their grand return to mainstream computer science and AI. One question that is particularly interesting is where do artificial neural networks live in the general AI landscape. There are two major societies that provide a formal classification of AI, which is used in their publications to classify a research paper, the American Mathematical Society (AMS) and the Association for Computing Machinery (ACM). The AMS maintains the so-called *Mathematics Subject Classification 2010* which divides AI into the following subfields[17]: General, Learning and adaptive systems, Pattern recognition and speech recognition, Theorem proving, Problem solving, Logic in artificial intelligence, Knowledge representation, Languages and software systems, Reasoning under uncertainty, Robotics, Agent technology, Machine vision and scene understanding and Natural language processing. The ACM classification[18] for AI provides, in addition to subclasses of AI, their subclasses as well. The subclasses of AI are: Natural language processing, knowledge representation and reasoning, planning and scheduling, search methodologies, control methods, philosophical/theoretical foundations of AI, distributed artificial intelligence and computer vision. Machine learning is a parallel category to AI, not subordinated to it.

What can be concluded from these two classifications is that there are a few broad fields of AI, inside which all other fields can be subsumed:

- Knowledge representation and reasoning,
- Natural language processing,
- Machine Learning,
- Planning,
- Multi-agent systems,
- Computer vision,
- Robotics,
- Philosophical aspects.

In the simplest possible view, deep learning is a name for a specific class of artificial neural networks, which in turn are a special class of machine learning algorithms, applicable to natural language processing, computer vision and robotics. This is a very simplistic view, and we think it is wrong, not because it is not true (it is true), but

[17] See http://www.ams.org/msc/.
[18] See http://www.acm.org/about/class/class/2012.

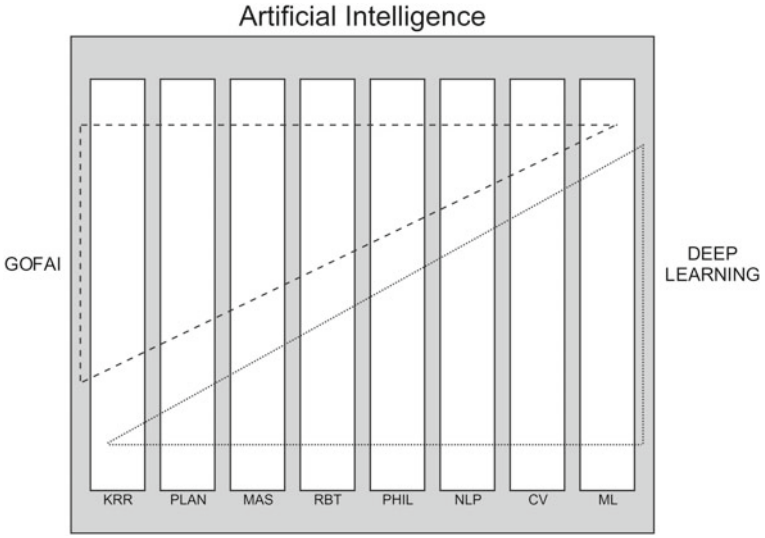


Fig. 1.1 Vertical and horizontal components of AI

because it misses an important aspect. Recall the Good Old-Fashioned AI (GOFAI), and consider what it is. Is it a subdiscipline of AI? The best answer it to think of subdivisions of AI as vertical components, and of GOFAI as a horizontal component that spans considerably more work in knowledge representation and reasoning than in computer vision (see Fig. 1.1). Deep learning, in our thinking, constitutes a second horizontal component, trying to unify across disciplines just as GOFAI did. Deep learning and GOFAI are in a way contenders to the whole AI, wanting to address all questions of AI with their respective methods: they both have their 'strongholds',[19] but they both try to encompass as much of AI as they can. The idea of deep learning being a separate influence is explored in detail in [35], where the deep learning movement is called 'connectionist tribe'.

1.5 Philosophical and Cognitive Aspects

So far, we have explored neural networks from a historical perspective, but there are two important things we have not explained. First, what the word 'cognitive' means. The term itself comes from neuroscience [36], where it has been used to characterize outward manifestations of mental behaviour which originates in the cortex. The what exactly comprises these abilities is non-debatable, since neuroscience grounds this division upon neural activity. A cognitive process in the context of AI is then an

[19] Knowledge representation and reasoning for GOFAI, machine learning for deep learning.

imitation of any mental process taking place in the human cortex. Philosophy also wants to abstract away from the brain, and define its terms in a more general setting. A working definition of 'cognitive process' might be: any process taking place in a similar way in the brain and the machine. This definition commits us to define 'similar way', and if we take artificial neural networks to be a simplified version of the real neuron, this might work for our needs here.

This leads us to the bigger issue. Some cognitive processes are simpler, and we could model them easily. Advances in deep learning sweep away one cognitive process at the time, but there is one major cognitive process eludes deep learning—reasoning. Capturing and describing reasoning is the very core of philosophical logic, and formal logic as the main method for a rigorous treatment of reasoning has been the cornerstone of GOFAI. Will deep learning ever conquer reasoning? Or is learning simply a process fundamentally different from reasoning? This would mean that reasoning is not learnable *in principle*. This discussion resonates the old philosophical dispute between rationalists and empiricists, where rationalists argued (in different ways) that there is a logical framework in our minds prior to any learning. A formal proof that no machine learning system could learn reasoning which is considered a distinctly human cognitive process would have a profound technological, philosophical and even theological significance.

The question about learning to reason can be rephrased. It is widely believed that dogs cannot learn relations.[20] A dog would then be an example of a trainable cognitive system incapable of learning relations. Suppose we want to teach a dog the relation 'smaller'. We could devise a training setting where we hand the dog two different objects, and the dog should pick the smaller one when hearing the command 'smaller' (and he is rewarded for the right pick). But the task for the dog is very complex: he has to realize that 'smaller' is not a name of a single object which changes reference from one training sample to the next, but something immaterial that comes into existence when you have both objects, and then resolves to refer to a single object (the smaller one). If you think about it like that, the difficulties of learning relations become clearer.

Logic is inherently relational, and everything there is a relation. Relational reasoning is accomplished by formal rules and poses no problem. But logic has the very same problem (but seen from the other side): how to learn content for relations? The usual procedure was to hand define entities and relations and then perhaps add a dynamical factor which would modify them over time. But the divide between patterns and relations exists on both sides.

[20]Whether this is true or not, is irrelevant for our discussion. The literature on animal cognitive abilities is notoriously hard to find as there are simply not enough academic studies connecting animal cognition and ethology. We have isolated a single paper dealing with limitations of dog learning [37], and therefore we would not dare to claim anything categorical—just hypothetical.

The paper that exposed this major philosophical issue in artificial neural networks and connectionism, is the seminal paper by Fodor and Pylyshyn [38]. They claimed that thinking and reasoning as a phenomena is inherently rule-based (symbolic, relational), and this was not so much a natural mental faculty but a complex ability that evolved as a tool for preserving truth and (to a lesser extent) predicting future events. They pose it as a challenge to connectionism: if connectionism will be able to reason, the only way it will be able to do so (since reasoning is inherently rule-based) is by making an artificial neural network which produces a system of rules. This would not be 'connectionist reasoning' but symbolic reasoning whose symbols are assigned meaningful things thanks to artificial neural networks. Artificial neural networks fill in the content, but the reasoning itself is still symbolic.

You might notice that the validity of this argument rests on the idea that thinking is inherently rule-based, so the most easy way to overcome their challenge it is to dispute this initial assumption. If thinking and reasoning would not be completely rule-based, it would mean that they have aspects that are processed 'intuitively', and not derived by rules. Connectionists have made an incremental but important step in bridging the divide. Consider the following reasoning: 'it is to long for a walk, I better take my van', 'I forgot that my van is at the mechanic, I better take my wife's car'. Notice that we have deliberately not framed this as a classic syllogism, but in a form similar to the way someone would actually think and reason.[21] Notice that what makes this thinking valid,[22] is the possibility of equating 'car' with 'van' as similar.[23] Word2vec [39] is a neural language model which learns numerical vectors for a given word and a context (several words around it), and this is learned from texts. The choice of texts is the 'big picture'. A great feature of word2vec is that it clusters words by semantic similarity in the big picture. This is possible since semantically similar words share a similar immediate context: both Bob and Alice can be hungry, but neither can Plato nor the number 4. But substituting similar for similar is just proto-inference, the major incremental advance towards connectionist reasoning made possible by word2vec is the native calculations it enables. Suppose that $v(x)$ is the function which maps x (which is a string) to its learned vector. Once trained, the word vectors word2vec generates are special in the sense that one can calculate with them like $v(king) - v(man) + v(woman) \approx v(queen)$. This is called *analogical reasoning* or *word analogies*, and it is the first major landmark in developing a purely connectionist approach to reasoning.

We will be exploring reasoning in the final chapter of the book in the context of question answering. We will be exploring also energy-based models and memory models, and the best current take on the issue of reasoning is with memory-based

[21] Plato defined *thinking* (in his *Sophist*) as the soul's conversation with itself, and this is what we want to model, whereas the rule-based approach was championed by Aristotle in his *Organon*. More succinctly, we are trying to reframe reasoning in platonic terms instead of using the dominant Aristotelian paradigm.

[22] At this point, we deliberately avoid talking of 'valid inference' and use the term 'valid thinking'.

[23] Note that this interchangeability dependent on the big picture. If I need to move a piano, I could not do it with a car, but if I need to fetch groceries, I can do it with either the car or the van.

models. This is perhaps surprising since in the normal cognitive setting (undoubtedly under the influence of GOFAI), we consider memory (knowledge) and reasoning as two rather distinct aspects, but it seems that neural networks and connectionism do not share this dichotomy.

References

1. A.M. Turing, On computable numbers, with an application to the entscheidungsproblem. Proc. Lond. Math. Soc. **42**(2), 230–265 (1936)
2. V. Peckhaus, Leibniz's influence on 19th century logic. in *The Stanford Encyclopedia of Philosophy*, ed. by E.N. Zalta (2014)
3. J.S. Mill, *A System of Logic Ratiocinative and Inductive: Being a connected view of the Principles of Evidence and the Methods of Scientific Investigation* (1843)
4. G. Boole, *An Investigation of the Laws of Thought* (1854)
5. A.M. Turing, Computing machinery and intelligence. Mind **59**(236), 433–460 (1950)
6. R. Carnap, *Logical Syntax of Language* (Open Court Publishing, 1937)
7. A.N. Whitehead, B. Russell, *Principia Mathematica* (Cambridge University Press, Cambridge, 1913)
8. J.Y. Lettvin, H.R. Maturana, W.S. McCulloch, W.H. Pitts, What the frog's eye tells the frog's brain. Proc. IRE **47**(11), 1940–1959 (1959)
9. N.R. Smalheiser, Walter pitts. Perspect. Biol. Med. **43**(1), 217–226 (2000)
10. A. Gefter, The man who tried to redeem the world with logic. Nautilus **21** (2015)
11. F. Rosenblatt, *Principles of Neurodynamics: perceptrons and the theory of brain mechanisms* (Spartan Books, Washington, 1962)
12. F. Rosenblatt, Recent work on theoretical models of biological memory, in *Computer and Information Sciences II*, ed. by J.T. Tou (Academic Press, 1967)
13. S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd edn. (Pearsons, London, 2010)
14. H. Moravec, *Mind Children: The Future of Robot and Human Intelligence* (Harvard University Press, Cambridge, 1988)
15. M. Minsky, S. Papert, *Perceptrons: An Introduction to Computational Geometry* (MIT Press, Cambridge, 1969)
16. L.R. Graham, *Science in Russia and the Soviet Union. A Short History* (Cambridge University Press, Cambridge, 2004)
17. S. Pinker, *The Blank Slate* (Penguin, London, 2003)
18. B.F. Skinner, *The Possibility of a Science of Human Behavior* (The Free House, New York, 1953)
19. E.L. Gettier, Is justified true belief knowledge? Analysis **23**, 121–123 (1963)
20. T.S. Kuhn, *The Structure of Scientific Revolutions* (University of Chicago Press, Chicago, 1962)
21. N. Chomsky, *Aspects of the Theory of Syntax* (MIT Press, Cambridge, 1965)
22. N. Chomsky, A review of B. F. Skinner's verbal behavior. Language **35**(1), 26–58 (1959)
23. A. Newell, J.C. Shaw, H.A. Simon, Elements of a theory of human problem solving. Psychol. Rev. **65**(3), 151–166 (1958)
24. J. Lighthill, Artificial intelligence: a general survey, in *Artificial Intelligence: A Paper Symposium*, Science Research Council (1973)
25. Paul J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences* (Harvard University, Cambridge, 1975)

26. D.B. Parker, Learning-logic. Technical Report No. 47 (MIT Center for Computational Research in Economics and Management Science, Cambridge, 1985)
27. Y. LeCun, Une procédure d'apprentissage pour réseau a seuil asymmetrique. Proc. Cogn. **85**, 599–604 (1985)
28. D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning internal representations by error propagation. Parallel Distrib. Process. **1**, 318–362 (1986)
29. J.J. Hopfield, Neural networks and physical systems with emergent collective computational abilities. Proc. Natl. Acad. Sci. USA **79**(8), 2554–2558 (1982)
30. N. Cristianini, J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods* (Cambridge University Press, Cambridge, 2000)
31. S. Hochreiter, J. Schmidhuber, Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
32. Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
33. G.E. Hinton, S. Osindero, Y.-W. Teh, A fast learning algorithm for deep belief nets. Neural Comput. **18**(7), 1527–1554 (2006)
34. J. Schmidhuber, Deep learning in neural networks: an overview. Neural Netw. **61**, 85–117 (2015)
35. P. Domingos, *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World* (2015)
36. M.S. Gazzanga, R.B. Ivry, G.R. Mangun, *Cognitive Neuroscience: The Biology of Mind*, 4th edn. (W. W. Norton and Company, New York, 2013)
37. A. Santos, Limitations of prompt-based training. J. Appl. Companion Anim. Behav. **3**(1), 51–55 (2009)
38. J. Fodor, Z. Pylyshyn, Connectionism and cognitive architecture: a critical analysis. Cognition **28**, 3–71 (1988)
39. T. Mikolov, T. Chen, G. Corrado, J, Dean, Efficient estimation of word representations in vector space, in *ICLR Workshop* (2013), arXiv:1301.3781

Mathematical and Computational Prerequisites

2

2.1 Derivations and Function Minimization

In this chapter, we give most of the mathematical preliminaries necessary to understand the later chapters. The main engine of deep learning is called *backpropagation* and it consists mainly of gradient descent, which is a move along the gradient, and the gradient is a vector of derivations. And the first section of this chapter is about derivations, and by the end of it, the reader should know what a gradient is and what is gradient descent. We will not return to this topic, but we will make heavy use of it in all the remaining chapters of this book.

One basic notational convention we will be using is ':='; '$A := xy$' means 'We define A to be xy', or 'xy is called A'. This is called *naming* xy with the name A. We take the set to be the basic mathematical concept as most other concepts can be build upon or explained by using sets. A set is a collection of members and it can have both other sets and non-sets as members. Non-sets are basic elements called *urelements*, such as numbers or variables. A set is usually denoted with curly braces, so for example $A := \{0, 1, \{2, 3, 4\}\}$ is a set with *three* members containing the elements 0, 1 and $\{2, 3, 4\}$. Note that $\{2, 3, 4\}$ is an element of A, not a subset. A subset of A would be for example $\{0, \{2, 3, 4\}\}$. A set can be written *extensionally* by naming the members such as $\{-1, 0, 1\}$ or *intensionally* by giving the property that the members must satisfy, such as $\{x | x \in \mathbb{Z} \wedge |x| < 2\}$ where $\mathbb{Z}$ is the set of integers and $|x|$ is the absolute value of x. Notice that these two denote the same set, since they have the same members. This principle of equality is called the *axiom of extensionality*, and it says that two sets are equal if and only if they have the same members. This means that $\{0, 1\}$ and $\{1, 0\}$ are equal, but also $\{1, 1, 1, 1, 0\}$ and $\{0, 0, 1, 0\}$ (all of them have the same members, 0 and 1).[1]

[1] Notice that they also have the same *number of members* or *cardinality*, namely 2.

A set does not remember the order of elements or repetitions of one element. If we have a set that remembers repetitions but not order we have multisets or *bags*, so we have $\{1, 0, 1\} = \{1, 1, 0\}$ but neither is equal to $\{1, 0\}$, we are talking about multisets. The usual way to denote bags to distinguish them from sets is to number the elements, so instead of writing $\{1, 1, 1, 1, 0, 1, 0, 0\}$ we would write {"1" : 5, "0" : 3}. Bags will be very useful to model language via the so-called *bag of words* model as we will see in Chap. 3.

If we care both about the position and repetitions, we write $(1, 0, 0, 1, 1, 1)$. This object is called a *vector*. If we have a vector of variables like $(x_1, x_2, \ldots, x_n)$ we write it as $\mathbf{x}$ or $\mathbf{x}$. The individual x_i, $1 \leq i \leq n$, is called a *component* (in sets they used to be called members), and the number of components is called *the dimensionality of the vector* $\mathbf{x}$.

The terms *tuple* and *list* are very similar to vectors. Vectors are mainly used in theoretical discussions, whereas tuples and lists are used in realizing vectors in programming code. As such, tuples and lists are always named with programming variables such as `myList` or `vectorAsTuple`. So an example of either tuple or list would be `newThing` $:= (11, 22, 33)$. The difference between tuple and a list is that lists are *mutable* and tuples are not. Mutability of a structure means that we can assign a new value to a member of that structure. For example, if we have `newThing` $:= (11, 22, 33)$ and then we do `newThing`$[1] \leftarrow 99$ (to be read 'assign to the *second*[2] item the value of 99'), we get `newThing` $:= (11, 99, 33)$. This means that we have mutated the list. If we do not want to be able to do that, we use a tuple, in which case we cannot modify the elements. We can create a new tuple `newerThing` such that `newerThing`$[0] \leftarrow$ `newThing`$[0]$, `newerThing`$[1] \leftarrow 99$ and `newerThing`$[2] \leftarrow$ `newThing`$[2]$ but this is not *changing* the values, just copying it and composing a new tuple. Of course, if we have an unknown data structure, we can check whether it is a list or tuple by trying to modify some component. Sometimes, we might wish to model vectors as tuples, but we will usually want to model them as lists in our programming codes.

Now we have to turn our attention to functions. We will take a computational approach in their definition.[3] A function is a magical artifact that takes arguments (inputs) and turns them into values (outputs). Of course, the trick with functions is that instead of using magic we must define in them how to get from inputs to outputs, or in other words how to transform the inputs into outputs. Recall a function, e.g. $y = 4x^3 + 18$ or equivalently $f(x) = 4x^3 + 18$, where x is the input, y is the output and f is the function's 'name'. The output y is defined to be the application of f to x, i.e. $y := f(x)$. We are omitting a few things here, but they are not important for this book, but we point the interested reader to [1].

When we think of a function like this, we actually have an instruction (algorithm) of how to transform the x to get the y, by using simpler functions such as addition,

[2]The counting starts with 0, and we will use this convention in the whole book.

[3]The traditional definition uses sets to define tuples, tuples to define relations and relations to define functions, but that is an overly logical approach for our needs in the present volume. This definition provides a much wider class of entities to be considered functions.

multiplication and exponentiation. They in turn can be expressed from simpler functions, but we will not need the proofs for this book. The reader can find in [2] the details on how this can be done.

Note that if we have a function with 2 arguments[4] $f(x, y) = x^y$ and pass in values $(2, 3)$ we get 8. If we pass in $(3, 2)$ we will get 9, which means that functions are order sensitive, i.e. they operate on vector inputs. This means that we can generalize and say that a function always takes a vector as an input, and a function taking an n-dimensional vector is called an n-ary function. This means that we are free to use the notation $f(\mathbf{x})$. A 0-ary function is a function that produces an output but takes in no input. Such a function is called a *constant*, e.g. $p() = 3.14159\ldots$ (notice the notation with the open and closed parenthesis).

Note that we can take a function's argument input vector and add to it the output, so that we have $(x_1, x_2, \ldots, x_n, y)$. This structure is called a *graph* of the function f for inputs $\mathbf{x}$. We will see how we can extend this to all inputs. A function can have parameters and the function $f(x) = ax + b$ has a and b as parameters. They are considered fixed, but we might want to tweak them to get a better version of the function. Note that a function always gives the same result if it is given the same input and you do not change the parameters. By changing the parameters, you can drastically change the output. This is very important for deep learning, since deep learning is a method for automatically tuning parameters which in turn modifies the output.

We can have a set A and we may wish to create a function of x which gives a value 1 to all values which are members of A and 0 to all other values for x. Since this function is different for all sets A, other than this, it always does the same thing, we can give it a name which includes A. We choose the name $\mathbb{1}_A$. This function is called *indicator function* or *characteristic function*, and it is sometimes denoted as χ_A in the literature. This is used for something which we will call *one-hot encoding* in the next chapter.

If we have a function $y = ax$, then the set from which we take the inputs is called the domain of the function, and the set to which the outputs belong is called the codomain of the function. In general, a function does not need to be defined for all members of the domain, and, if it is, it is called a *total* function. All functions that are not total are called *partial*. Remember that a function assigns to every vector of inputs always the same output (provided the parameters do not change). If by doing so the function 'exhausts' the whole codomain, i.e. after assignment there are no members of the codomain which are not outputs of some inputs, the function is called a *surjection*. If on the other hand the function never assigns to different input vectors the same output, it is called an *injection*. If it is both an injection and surjection, it is called a bijection. The set of outputs B given a set of inputs A is called an *image* and denoted by $f[A] = B$. If we look for a set of inputs A given the set of outputs B, we are looking at its inverse image denoted by $f^{-1}[B] = A$ (we can use the same notation for individual elements $f^{-1}(b) = a$).

[4] A function with n-arguments is called an *n-ary function*.

A function f is called *monotone* if for every x and y from the domain (for which the function is defined) the following holds: if $x < y$ then $f(x) \leq f(y)$ or if $x > y$ then $f(x) \geq f(y)$. Depending on the direction, this is called an *increasing* or *decreasing* function, and if we have $<$ instead of $\leq$, it is called *strictly increasing* (or strictly decreasing). A *continuous function* is a function that does not have gaps. For what we will be needing now, this definition is good enough—we are imprecise, but we are sacrificing precision for clearness. We will be returning to this later.

One interesting function is the characteristic function for rational numbers over all real numbers. This function returns 1 if and only if the real number it picked is also a rational number. This function is continuous nowhere. A different function which is continuous in parts but not everywhere is the so-called *step function* (we will mention it again briefly in Chap. 4):

$$step_0(x) = \begin{cases} 1, x > 0 \\ -1, x \leq 0 \end{cases}$$

Note that $step_0$ can be easily generalized to $step_n$ by simply placing n instead of 0. Also, note that the 1 and -1 are entirely arbitrary, so we can put any values instead. A step function that takes in an n-dimensional vector is also sometimes called a *voting function*, but we will keep calling it a step function. In this version, all components of the input vector of the function are added before being compared with the threshold n (the threshold n is called a *bias* in neural network literature). Pay close attention to how we defined the step function with two cases: if a function is defined by cases, it is an important *hint* that the function might not be continuous. It is not always the case (in either way we look at it), but it is a good hint to follow and it is often true.[5]

Before continuing to derivations, we will be needing a few more concepts. If the outputs of the function f approach a value c (and settle in it), we say that the function *converges in c*. If there is no such value, the function is called *divergent*. In most mathematics textbooks, the definitions of convergence are more meticulous, but we will not be needing the additional mathematical finesse in this book, just the general intuition.

An important constant we will use is the *Euler number*, $e = 2.718281828459\ldots$. This is a constant and we will reserve for it the letter e. We will be using the basic numerical operations extensively, and we give a brief overview of their behaviour and notations used here:

- The reciprocal number of x is $\frac{1}{x}$ or equivalently x^{-1}
- The square root of x is $x^{\frac{1}{2}}$ or equivalently $\sqrt{x}$
- The exponential function has the properties: $x^0 = 1$, $x^1 = x$, $x^n \cdot x^m = x^{n+m}$, $(x^n)^m = x^{n \cdot m}$

[5]The ReLU or *rectified linear unit* defined by $\rho(x) = \max(x, 0)$ is an example of a function that is continuous even though it is (usually) defined by cases. We will be using ReLU extensively from Chap. 6 onwards.

- The logarithmic function has the properties: $\log_c 1 = 0$, $\log_c c = 1$, $\log_c(xy) = \log_c x + \log_c y$, $\log_c(\frac{x}{y}) = \log_c x - \log_c y$, $\log_c x^y = y \log_c x$, $\log_x y = \frac{\log_c y}{\log_c x}$, $\log_x x^y = y$, $x^{\log_x y} = y$, $\ln x := \log_e x$).

The last concept we will need before continuing to derivations is the concept of a *limit*. An intuitive definition would be that the limit of a function is a value which the outputs of the function approach but never reach.[6] The trick is that the limit of the function is considered in relation to a change in inputs and it must be a concrete value, i.e. if the limit is ∞ or $-\infty$, we do not call it a limit. Note that this means that for the limit to exist it must be a *finite* value. For example, $\lim_{x \to 5} f(x) = 10$, if we take f to be $f(x) = 2x$. It is of vital importance not to confuse the number 5 which the inputs approach and the limit, 10, which the outputs of the function approach as the inputs approach 5.

The concept of limit is trivial (and mathematically weird) if we think of integer inputs. We shall assume when we think of limits that we are considering real numbers as inputs (where the idea of continuity makes sense). Therefore, when talking about limits (and derivations), the input vectors are real numbers and we want the function to be continuous (but sometimes it might not be). If we want to know a limit of a function, and it is continuous everywhere, we can try to plug in the value to which the inputs approach and see what we get for the output. If there are problems with this, we can either try to simplify the function expression or see what is happening to the pieces. In practice,[7] the problems occur in two way: (i) the function is defined by cases or (ii) there are segments where the outputs are undefined due to a hidden division by 0 for some inputs.

We can now replace our intuitive idea of continuity with a more rigorous definition. We call a function f continuous in a point $x = a$ if and only if the following conditions hold:

1. $f(a)$ is defined
2. $\lim_{x \to a} f(x)$ exists
3. $f(a) = \lim_{x \to a} f(x)$.

A function is called continuous everywhere if and only if it is continuous in all points. Note that all elementary functions are continuous everywhere[8] and so are all

[6] This is why $0.999\cdots \neq 1$.

[7] This is especially true in programming, since when we program we need to approximate functions with real numbers by using functions with rational numbers. This approximation also goes a long way in terms of intuition, so it is good to think about this when trying to figure out how a function will behave.

[8] With the exception of division where the divisor is 0. In this case, the division function is undefined, and therefore the notion of continuity does not have any meaning in this point.

polynomial functions. Rational functions[9] are continuous everywhere except where the value of the denominator is 0. Some equalities that hold for limits are

1. $\lim_{x \to a} c = c$
2. $\lim_{x \to 0+} \frac{1}{x} = \infty$
3. $\lim_{x \to 0-} \frac{1}{x} = -\infty$
4. $\lim_{x \to \infty} \frac{1}{x} = 0$
5. $\lim_{x \to \infty} (1 + \frac{1}{x})^x = e.$

Now, we are all set to continue our journey to differentiation.[10] We can develop a bit of intuition behind derivatives by noting that the derivative of a function can be imagined as the slope of the plot of that function in a given point. You can see an illustration in Fig. 2.1. If a function $f(x)$ (the domain is X) has a derivative in every point $a \in X$, then there exists a new function $g(x)$ which maps all values from X to its derivative. This function is called the *derivative* of f. As $g(x)$ depends on f and x, we introduce the notation $f'(x)$ (Lagrange notation) or, remembering that $f(x) = y$, we can use the notation $\frac{dy}{dx}$ or $\frac{df}{dx}$ (Leibniz notation). We will deliberately use these two notations inconsistently in this book, since some ideas are more intuitive when expressed in one notation, while some are more intuitive in the other. And we want to focus on the underlying mathematical phenomena, not the notational tidiness.

Let us address this in more detail. Suppose we have a function $f(x) = \frac{x}{2}$. The slope of this function can be obtained by selecting two points from it, e.g. $t_1 = (x_1, y_1)$ and $t_2 = (x_2, y_2)$. Without loss of generality, we can assume that t_1 comes before t_2,

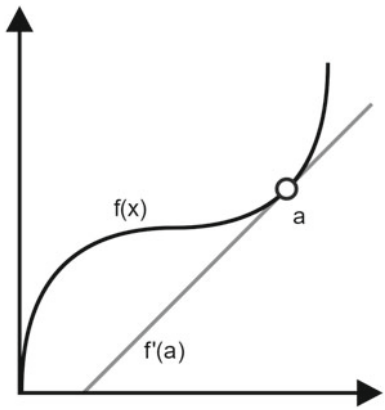


Fig. 2.1 The derivative of $f(x)$ in the point a

[9] Rational functions are of the form $\frac{f(x)}{g(x)}$ where f and g are polynomial functions.
[10] The process of finding derivatives is called 'differentiation'.

i.e. that $x_1 < x_2$ and $y_1 < y_2$. The slope is then equal to $\frac{y_2-y_1}{x_2-x_1}$, which is the ratio of the vertical and horizontal segments. If we restrict our attention to linear functions of the form $f(x) = ax + b$, we can see a couple of things. First, the slope is actually a (you can easily verify this) and it is the same in every point, and second, that the slope of a constant[11] must be 0, and the constant is then b.

Let us take a more complex example such as $f(x) = x^2$. Here, the slope is not the same in every point and by the above calculation we will not be able to get much out of it, and we will have to use differentiation. But differentiation is still just an elaboration of the slope idea. Let us start with the slope formula and see where it takes us when we try to formalize it a bit. So we start with $\frac{y_2-y_1}{x_2-x_1}$. We can denote with h the change in x with which we get x_2 from x_1. This means that the numerator can be written as $f(x+h) - f(x)$, and the denominator is just h by definition of h. The derivative is then *defined* as the limit of that as h approaches 0, or

$$f'(x) = \frac{dy}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{2.1}$$

Let us see how we can get the derivative $f'(x)$ of the function $f(x) = 3x^2$. We will give the rules to calculate the derivative a bit later, and using these rules we would quickly find that $f'(x) = 6x$, but let us see now how we can get this by using only the definition of the derivative:

1. $f(x) = 3x^2$ [initial function]
2. $f'(x) = \lim_{h \to 0} \frac{(f(x+h)-f(x)}{h}$ [definition of the derivative]
3. $f'(x) = \lim_{h \to 0} \frac{(3(x+h)^2-3x^2}{h}$ [we get this by substituting the expression from row 1 in the expression in row 2]
4. $f'(x) = \lim_{h \to 0} \frac{(3(x^2+2xh+h^2)-3x^2}{h}$ [from row 3, by squaring the sum]
5. $f'(x) = \lim_{h \to 0} \frac{(3x^2+6xh+3h^2-3x^2}{h}$ [from row 4, by multiplying]
6. $f'(x) = \lim_{h \to 0} \frac{6xh+3h^2}{h}$ [from 5, cancelling out $+3x^2$ and $-3x^2$ in the numerator]
7. $f'(x) = \lim_{h \to 0} \frac{h(6x+3h)}{h}$ [from 6, by taking out h in the numerator]
8. $f'(x) = \lim_{h \to 0} (6x + 3h)$ [from 7, cancelling out the h in the numerator and denominator]
9. $f'(x) = 6x + 3 \cdot 0$ [from 8, by replacing h with 0 (to which it approaches)]
10. $f'(x) = 6x$ [from 9].

We turn our attention to the rules of differentiation. All of these rules can be derived just as we did with the rules used above, but it is easier to remember the rules than the actual derivations of the rules, especially since the focus in this book is not

[11] Which is a 0-ary function, i.e. a function that gives the same value regardless of the input.

on calculus. One of the most basic things regarding derivatives is that the derivative of a constant is always 0. Also, the derivative of the differentiation variable is always 1, or, in symbols, $\frac{dy}{dx}x = 1$. The constant has to have a slope 0 and a function $f(x) = x$ will have horizontal component equal to the vertical component and the slope will be 1. Also, to get $f(x)$ from $f(x) = ax + b$, a has to be 1 to leave the x and b has to be 0.

The next rule is the so-called *exponential rule*. We have seen this rule derived in the above example: $\frac{d}{dx}a \cdot x^n = a \cdot n \cdot x^{n-1}$. We have placed the a that show how a possible factor behaves. The rules for addition and subtraction are rather straightforward: $\frac{dy}{dx}(k+j) = \frac{dy}{dx}k + \frac{dy}{dx}j$ and $\frac{dy}{dx}(k-j) = \frac{dy}{dx}k - \frac{dy}{dx}j$. The rules for differentiation in the case of multiplication and division are more complex. We give two examples and we leave it to the reader to extrapolate the general form of the rules. If we have $y = x^3 \cdot 10^x$ then $y' = (x^3)' \cdot 10^x + x^3 \cdot (10^x)'$, and if $y = \frac{x^3}{10^x}$ then $y = \frac{(x^3)' \cdot 10^x - x^3 \cdot (10^x)'}{(10^x)^2}$.

The last rule we need is the so-called *chain rule* (not to be confused with the chain rule for exponents). The chain rule says $\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$, for some u. There is a similarity with fractions that goes a long way in terms of intuition.[12] Let us see an example. Let $h(x) = (3-2x)^5$. We can look at this function as if it were two functions: the first is $g(u)$ which gives some number $y = u^5$ (in our case this is $u = 3-2x$), and the second function which just gives u is $f(x) = 3-2x$. The chain rule says that to differentiate y by x (i.e. to get $\frac{dy}{dx}$), we can instead differentiate y by u (which is $\frac{dy}{du}$), u by x ($\frac{du}{dy}$) and simply multiply the two.[13]

To see the chain rule in action, take the function $f(x) = \sqrt{3x^2 - x}$ (i.e. $y = \sqrt{3x^2 - x}$). Then, $f'(x) = \frac{dy}{du} \cdot \frac{du}{dx}$, which means that $y = \sqrt{u}$ and so $\frac{du}{dx} = \frac{1}{2}u^{-\frac{1}{2}}$. On the other hand, $u = 3x^2 - x$, and so $\frac{du}{dx} = 6x - 1$. From this we get $\frac{dy}{du} \cdot \frac{du}{dx} = \frac{1}{2}u^{-\frac{1}{2}} \cdot (6x-1) = \frac{1}{2} \cdot \frac{1}{\sqrt{u}} \cdot (6x-1) = \frac{6x-1}{2\sqrt{u}} = \frac{6x-1}{2\sqrt{3x^2-x}}$.

The chain rule is the soul of backpropagation, which in turn is the heart of deep learning. This is done via function minimization, which we will address in detail in the next section where we will explain gradient descent. To summarize what we said and to add a few simple rules[14] we shall need, we give the following list of rules together with their 'names' and a brief explanation:

- `LD`: Differentiation is linear, so we can differentiate the summands separately and take out the constant factors: $[a \cdot f(x) + b \cdot g(x)]' = a \cdot f'(x) + b \cdot g'(x)$.
- `Rec`: Reciprocal rule $[\frac{1}{f(x)}]' = -\frac{f'(x)}{f(x)^2}$.

[12] The chain rule in Lagrange notation is more clumsy and void of the intuitive similarity with fractions: $h'(x) = f'(g(x))g'(x)$.

[13] Keep in mind that $h(x) = g(f(x)) = (g \circ f)(x) = g(u) \circ f(x)$, which means that h is the *composition* of the functions g and f. It is very important not to mix up compositions of functions like $f(x) = (3-2x)^5$ with an ordinary function like $f(x) = 3 - 2x^5$, or with a product like $f(x) = sinx \cdot x^5$.

[14] These rules are not independent, since both `ChainExp` and `Exp` are a consequence of `CHAINRULE`.

- `Const`: Constant rule $c' = 0$.
- `ChainExp`: Chain rule for exponents $[e^{f(x)}]' = e^{f(x)} \cdot f'(x)$.
- `DerDifVar`: Deriving the differentiation variable $\frac{dy}{dz} z = 1$.
- `Exp`: Exponent rule $[f(x)^n]' = n \cdot f(x)^{n-1} \cdot f'(x)$.
- `CHAINRULE`: Chain rule $\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$ (for some u).

2.2 Vectors, Matrices and Linear Programming

Before we continue, we will need to define one more concept, the *Euclidean distance*. If we have a 2D coordinate system, and have two points $p_1 := (x_1, y_1)$ and $p_2 := (x_2, y_2)$ in it, we can define their distance in space as $d(p_1, p_2) := \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. This distance is called the Euclidean distance and defines the behaviour of the whole space; in a sense, the distance in a space is a fundamental thing upon the whole behaviour of the space behaves. If we use the Euclidean distance when reasoning about space, we will get Euclidean spaces. Euclidean spaces are the most common type: they follow our spatial intuitions. In this book, we will use only Euclidean spaces.

Now, we turn our attention to developing tools for vectors. Recall that an n-dimensional vector $\mathbf{x}$ is $(x_1, \ldots, x_n)$ and that all the individual x_i are called components. It is quite a normal thing to imagine n-dimensional vectors living as points in an n-dimensional space. This space (when fully furnished) will be called a vector space, but we will return to this a bit later. For now, we have only a bunch of n-dimensional vectors from $\mathbb{R}^n$.

Let us introduce the notion of *scalar*. A scalar is just a number, and it can be thought of as a 'vector' from $\mathbb{R}^1$. And n-dimensional vectors are simply sequences of n scalars. We can always multiply a vector by a scalar, e.g. $3 \cdot (1, 4, 6) = (3, 12, 18)$. Vector addition is quite simple. If we want to add two vectors $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$, they must have the same number of components. Then $\mathbf{a} + \mathbf{b} := (a_1 + b_1, \ldots, a_n + b_n)$. For example, $(1, 2, 3) + (4, 5, 6) = (1 + 4, 2 + 5, 3 + 6) = (5, 7, 9)$. This gives us a hint that we must stick with vectors of the same dimensionality (but we will always include scalars even though they are technically 1D vectors). Once we have scalar multiplication and vector addition, we have a *vector space*.[15]

Let us take an in-depth view of the space our vectors live in. For simplicity, we will talk about 3D entities, but anything we will say can be easily generalized to the n-dimensional case. So, to recap, a 3D space is the place where 3D vectors live: they are represented as points in this space. A question can be asked whether there is a minimal set of vectors which 'define' the whole vector universe of 3D vectors. This

[15] We deliberately avoid talking about fields here since we only use $\mathbb{R}$, and there is no reason to complicate the exposition.

question is a bit vague but the answer is yes. If we take three[16] vectors $\mathbf{e}_1 = (1, 0, 0)$, $\mathbf{e}_2 = (0, 1, 0)$ and $\mathbf{e}_3 = (0, 0, 1)$, we can express any vector in this space with the formula:

$$s_1\mathbf{e}_1 + s_2\mathbf{e}_2 + s_3\mathbf{e}_3 \tag{2.2}$$

where s_1, s_2 and s_3 are scalars chosen so that we get the vector we want. This shows how mighty scalars are and how they control everything that happens—they are a kind of aristocracy in the vector realm. Let us turn to an example. If we want to represent the vector $(1, 34, -28)$ in this way, we need to take $s_1 = 1$, $s_2 = 34$ and $s_3 = -28$ and plug them in Eq. 2.2. This equation is called *linear combination*: every vector in a vector field can be defined as a (linear) combination of the vectors $\mathbf{e}_1$, $\mathbf{e}_2$ and $\mathbf{e}_3$, and appropriately chosen scalars. The set $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$ is called *the standard basis* of the 3D vector space (which is usually denoted as $\mathbb{R}^3$).

The reader may notice that we have been talking about the standard basis without defining what a basis is. Let V be a vector space and $B \subseteq V$. Then, B is called a basis if and only if all vectors in B are *linearly independent* (i.e. are not linear combinations of each other) and B is a minimally generating subset of V (i.e. it must be a *minimal*[17] subset which can produce with the help of Eq. 2.2) every vector in V.

We turn our attention to defining the *single most important operation with vectors we will need in this book*, the *dot product*. The dot product of two vectors (which must have the same dimensions) is a scalar. It is defined as

$$\mathbf{a} \cdot \mathbf{b} = (a_1, \ldots, a_n) \cdot (b_1, \ldots, b_n) := \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots a_n b_n \tag{2.3}$$

This means that $(1, 2, 3) \cdot (4, 5, 6) = 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32$. If two vectors have the a dot product equal to zero, they are called *orthogonal*. Vectors also have lengths. To measure the length of a vector $\mathbf{a}$, we compute its L_2 or *Euclidean* norm. The L_2 norm of the vector is defined as

$$||\mathbf{a}||_2 := \sqrt{a_1^2 + a_2^2 + \ldots + a_n^2} \tag{2.4}$$

Bear in mind not to confuse the notation for norms with the notation for the absolute value. We will see more about the L_2 norm in the later chapters. We can convert any vector $\mathbf{a}$ to a so-called *normalized vector* by dividing it with its L_2 norm:

$$\hat{\mathbf{a}} := \frac{\mathbf{a}}{||\mathbf{a}||_2} \tag{2.5}$$

Two vectors are called *orthonormal* if they are normalized and orthogonal. We will be needing these concepts in Chaps. 3 and 9. We not turn our attention to matrices

[16] One for each dimension.

[17] A minimal subset such that a property P holds is a subset (of some larger set) of which we can take no proper subset such that P would still hold.

which are a natural extension of vectors. A matrix is a structure similar to a table as it is made by rows and columns. To understand what a matrix is, take for example the following matrix and try to make some sense of it with what we have already covered when we were talking about vectors:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix}$$

Right away we see a couple of things. First, the entries in the matrix are denoted by a_{jk} and j denotes the row, and k denotes the column of the given entry. A matrix has dimensions similar to a vector, but it has to have two of them. The matrix A is a 4×3 dimensional matrix. Note that this is not the same as a 3×4 dimensional matrix. We can look at a matrix as a vector of vectors (this idea has a couple of formal problems that need to be ironed out, but it is a good intuition). Here, we have two options: It could be viewed as vectors $\mathbf{a_{1x}} = (a_{11}, a_{12}, a_{13})$, $\mathbf{a_{2x}} = (a_{21}, a_{22}, a_{23})$, $\mathbf{a_{3x}} = (a_{31}, a_{32}, a_{33})$ and $\mathbf{a_{4x}} = (a_{41}, a_{42}, a_{43})$ stacked in a new vector $A = (\mathbf{a_{1x}}, \mathbf{a_{2x}}, \mathbf{a_{3x}}, \mathbf{a_{4x}})$ or it could be seen as vectors $\mathbf{a_{x1}} = (a_{11}, a_{21}, a_{31}, a_{41})$, $\mathbf{a_{x2}} = (a_{12}, a_{22}, a_{32}, a_{42})$ and $\mathbf{a_{x3}} = (a_{13}, a_{23}, a_{33}, a_{43})$ which are then bundled together as $A = (\mathbf{a_{x1}}, \mathbf{a_{x2}}, \mathbf{a_{x3}})$.

Either way we look at it something is off since we have to keep track of what is vertical and what is horizontal. It is clear that now need to distinguish a standard, horizontal vector, called a *row vector* (a row of the matrix taken out which is now just a vector), which is a $1 \times n$ dimensional matrix

$$\mathbf{a_h} = (a_1, a_2, a_3, \ldots, a_n) = \begin{bmatrix} a_1 & a_2 & a_3 & \cdots & a_n \end{bmatrix}$$

from a vertical vector called *column vector*, which is a $n \times 1$ dimensional matrix:

$$\mathbf{a_v} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix}$$

We will need an operation to transform row vectors in column vectors and in general, to transform a $m \times n$ dimensional matrix into a $n \times m$ dimensional matrix while keeping the order in both the rows and columns. Such an operation is called a *transposition*, and you can imagine it as having a matrix written down on a transparent sheet of A4 paper in portrait orientation, and then by holding the top-left corner flip it to landscape orientation (and you read the number through the paper). Formally, if we have a $n \times m$ matrix A, we can define another matrix B as the matrix constructed from A by taking each a_{jk} and putting it in place of b_{kj}. B is then called the *transpose* of A and is denoted by $A^\top$. Note that transposing a column vector gives a standard

row vector and vice versa. Transposition is used a lot in deep learning to keep all operations running smoothly and quickly. If we have an $n \times n$ matrix A (called a *square matrix*) for which $A = A^\top$ holds, then such a matrix is called *symmetric*.

Now we turn to operations with matrices. We start with *scalar multiplication*. We can multiply a matrix A by a scalar s by multiplying each entry in the matrix by the scalar:

$$sA = \begin{bmatrix} s \cdot a_{11} & s \cdot a_{12} & s \cdot a_{13} \\ s \cdot a_{21} & s \cdot a_{22} & s \cdot a_{23} \\ s \cdot a_{31} & s \cdot a_{32} & s \cdot a_{33} \\ s \cdot a_{41} & s \cdot a_{42} & s \cdot a_{43} \end{bmatrix}$$

And we note that the multiplication of a matrix and a scalar *is* commutative (matrix by matrix multiplication will not be commutative). If we want to apply a function $f(x)$ to a matrix A, we do it by applying the function to all elements:

$$f(A) = \begin{bmatrix} f(a_{11}) & f(a_{12}) & f(a_{13}) \\ f(a_{21}) & f(a_{22}) & f(a_{23}) \\ f(a_{31}) & f(a_{32}) & f(a_{33}) \\ f(a_{41}) & f(a_{42}) & f(a_{43}) \end{bmatrix}$$

Now, we turn to matrix addition. If we want to add two matrices A and B, they must have the same dimensions, i.e. they must be both $n \times m$, and then we add[18] the corresponding entries. The result will also be a $n \times m$ matrix. To take an example:

$$A + B = \begin{bmatrix} 3 & -4 & 5 \\ -19 & 10 & 12 \\ 1 & 45 & 9 \\ -45 & -1 & 0 \end{bmatrix} + \begin{bmatrix} 4 & -1 & 2 \\ -3 & 10 & 26 \\ 13 & 51 & 90 \\ -5 & 1 & 30 \end{bmatrix} = \begin{bmatrix} 7 & -5 & 7 \\ -22 & 20 & 38 \\ 14 & 96 & 99 \\ -50 & 0 & 30 \end{bmatrix}$$

Now, we turn our attention to matrix multiplication. Matrix multiplication is not commutative, so $AB \neq BA$. To multiply two matrices, they have to have matching dimensions. So if we want to multiply A with B (that is to calculate AB), A has to be $m \times q$ dimensional and b has to be $q \times t$ dimensional. The resulting matrix AB has to be $m \times t$ dimensional. This idea of 'dimensionality agreement' is very important for matrix multiplication to work out. It is a matter of convention, but by taking this convention and saying that this is how matrices are to be multiplied, we will go a long way, and be computationally fast all the time, so it is well worth it.

If we multiply two matrices A and B, we will get the matrix C $(= AB)$ as the result (of the dimensions we specified above). The matrix C consists of elements c_{ij}. For every element c_{ij}, we get it by computing the dot product of two vectors: the row vector i from A and the column vector j from B (the column vector has to be transposed to get a standard row vector). Intuitively, this makes perfect sense: when we have an element c_{km}, k is the row and m is the column, so it is sensible that this

[18] Matrix subtraction works in exactly the same way, only with subtraction instead of addition.

element comes from the k-th row of A and the m-th column of B. An example will make it clear:

$$AB = \begin{bmatrix} 4 & -1 \\ -3 & 0 \\ 13 & 6 \\ -5 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & -4 & 5 \\ 9 & 1 & 12 \end{bmatrix}$$

Let us check the dimensions first: matrix A is 4×2 dimensional, and matrix B is 2×3 dimensional. They have the 2 'connecting' them, and therefore we can multiply these two matrices and we will get a 4×3 dimensional matrix as a result.

$$AB = \begin{bmatrix} 4 & -1 \\ -3 & 0 \\ 13 & 6 \\ -5 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & -4 & 5 \\ 9 & 1 & 12 \end{bmatrix} = \begin{bmatrix} 3 & -17 & 8 \\ -9 & 12 & -15 \\ 93 & -46 & 137 \\ -6 & 21 & -13 \end{bmatrix}$$

We will call the resulting 4×3 dimensional matrix C. Let us show the full calculations of all entries c_{ij}:

- $c_{11} = 4 \cdot 3 + (-1) \cdot 9 = 3$
- $c_{12} = -3 \cdot 3 + 0 \cdot 9 = -9$
- $c_{13} = 13 \cdot 3 + 6 \cdot 9 = 93$
- $c_{14} = -5 \cdot 3 + 1 \cdot 9 = -6$
- $c_{21} = 4 \cdot (-4) + (-1) \cdot 1 = -17$
- $c_{22} = -3 \cdot (-4) + 0 \cdot 1 = 12$
- $c_{23} = 13 \cdot (-4) + 6 \cdot 1 = -46$
- $c_{24} = 5 \cdot (-4) + 1 \cdot 1 = 21$
- $c_{31} = 4 \cdot 5 + (-1) \cdot 12 = 8$
- $c_{32} = -3 \cdot 5 + 0 \cdot 12 = -15$
- $c_{33} = 13 \cdot 5 + 6 \cdot 12 = 137$
- $c_{34} = -5 \cdot 5 + 1 \cdot 12 = -13$

Let us take another example of matrix multiplication:

$$AB = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \cdot \begin{bmatrix} 8 & 9 & 0 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 30 & 36 & 42 \\ 110 & 132 & 114 \end{bmatrix}$$

We show the calculation for all elements of C:

- $c_{11} = 0 \cdot 8 + 1 \cdot 1 + 2 \cdot 4 + 3 \cdot 7 = 30$
- $c_{12} = 0 \cdot 9 + 1 \cdot 2 + 2 \cdot 5 + 3 \cdot 8 = 36$
- $c_{13} = 0 \cdot 0 + 1 \cdot 3 + 2 \cdot 6 + 3 \cdot 9 = 42$
- $c_{21} = 4 \cdot 8 + 5 \cdot 1 + 6 \cdot 4 + 7 \cdot 7 = 110$
- $c_{22} = 4 \cdot 9 + 5 \cdot 2 + 6 \cdot 5 + 7 \cdot 8 = 132$

- $c_{23} = 4 \cdot 0 + 5 \cdot 3 + 6 \cdot 6 + 7 \cdot 9 = 114$

Before continuing, we must define two more classes of matrices. The first one is the *zero matrix*. A zero matrix can be of any size and all of its entries are zeros. Its dimensions will depend on what do we want to do with it, i.e. it will depend on the dimensions of the matrix we want to multiply it with. The second (and much more useful) is a *unit matrix*. A unit matrix is always a square matrix (i.e. both dimensions are the same). It will have the value 1 along the diagonal and all other entries are 0, i.e. $a_{jk} = 1$ if and only if $j = k$ and $a_{jk} = 0$ otherwise. Note that a unit matrix is a symmetric matrix. Note that there is only one unit matrix for every dimension, so we can give it a name, $\mathbf{I}_{n,n}$. Since it is a square matrix (a $n \times n$ matrix), we do not have to specify *both dimensions*, so we can just write $\mathbf{I}_n$. Just to show how they look:

$$\mathbf{I}_1 = \begin{bmatrix} 1 \end{bmatrix}, \mathbf{I}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{I}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}.$$

Now we can define *orthogonality* for matrices. An $n \times n$ square matrix A is called orthogonal if and only if $AA^\top = A^\top A = \mathbf{I}_n$.

Notice that vectors had one dimension, so we talked about n-dimensional vectors. Matrices have 2D parameters, so we talk about $n \times m$ matrices. What if we add an extra dimension? What would be a $n \times k \times j$ dimensional object? Such objects are called *tensors* and behave similarly to matrices. Tensors are an important topic in deep learning but unfortunately are beyond the scope of this book. We point the interested reader to [3].

So far we have talked about derivatives and vectors separately, but it is time to see how they can combine to form the one of the most important structures in deep learning, the *gradient*. We have seen how to compute the derivative of a function of a single variable $f(x)$, but could we extend the notion to multiple variables? Could we get the slope in a point of a mathematical object that needs two variables to be defined? The answer is yes, and we do that by employing partial derivatives. Let us see on an example. Take the simple case of $f(x, y) = (x - y)^2$. First, we must transform it in $x^2 - 2xy + y^2$. Now, we must focus on it as a function of one variable, which means to treat the other one as an unknown constant: $f_y(x) = x^2 - 2xy + y^2$, or even better $f_a(x) = x^2 - 2xa + a^2$. We are now committed to finding *the partial derivative of f with respect to x*. So we are solving $\frac{df}{dx}$ for $f(x) = x^2 - 2xa + a^2$ (or equivalently $f'(x) = x^2 - 2xa + a^2$). Note that we cannot safely use the notation $\frac{dy}{dx}$ but we must write $\frac{df}{dx}$ to avoid confusion. Since differentiation is linear, by the rule `LD` from the previous section we get $\frac{df}{dx}x^2 - 2a\frac{df}{dx}x + \frac{df}{dx}a^2$. By using the exponent rule `Exp` on the first term, the differentiation variable rule `DerDifVar` on the second term and the constant rule `Const` on the third term, we get $2x - 2a + 0$, which simplifies to $2(x - a)$. Let us see what we did: we took the (full) derivative of $f_a(x)$ (with a constant a in place of y), which is the same as taking the *partial derivative*

of $f(x, y)$. In symbols, we calculated $\frac{df_y(x)}{dx}$, and the corresponding partial derivative is denoted as $\frac{\partial f(x,y)}{\partial x}$ and is obtained by re-substituting the variable we took out with the constant we put in. In other words $\frac{\partial f(x,y)}{\partial x} = 2(x - y)$.

Of course, just as $f(x, y)$ has a partial derivative with respect to x, it also has one with respect to y: $\frac{\partial f(x,y)}{\partial y} = 2(y - x)$. So if we have a function f taking as arguments $x_1, x_2, \ldots, x_n$ (or, we can say that f takes an n-dimensional vector), we would have n partial derivatives $\frac{\partial f(x_1,x_2,\ldots,x_n)}{\partial x_1}$, $\frac{\partial f(x_1,x_2,\ldots,x_n)}{\partial x_2}$, …, $\frac{\partial f(x_1,x_2,\ldots,x_n)}{\partial x_n}$. If we store them in a vector and get

$$(\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n})$$

We call this structure the *gradient* of the function $f(\mathbf{x})$ and write it as $\nabla f(\mathbf{x})$. To denote the i-th component of the gradient, we write $\nabla_i f(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial x_i})$. If we have a function f of n variables, it has to live in $n + 1$-dimensional space as an $n + 1$-dimensional surface. This surface in 3D space is called a *plane*, and in four or more dimensions it is called a *hyperplane*. The gradient then is simply a list of slopes in each of the $n + 1$ dimensions.

Building on this idea of a gradient being a list of slopes, let us see how we can find the minimum of an n-ary function using its gradient. Each input component of the function is a coordinate, to which the final function maps an input coordinate (which shows where the hyperplane given those inputs is). Since each component of a gradient is a slope along each of the dimensions of the hyperplane, we can subtract the gradient component from its respective input component and recalculate the function. When we do so and feed the new values to the function, we will get a new output, which is closer to the minimum of the function. This technique is called *gradient descent*, and we will be using it often. In Chap. 4, we will be providing a full calculation for a simple case, and all of our deep learning models will be using it to update their parameters.

Let us see an example of how function minimization with gradient descent looks like. Suppose we have a simple function, $f(x) = x^2 + 1$. We need to find the value of x which shall result with the minimal $f(x)$.[19] From basic calculus, we know that this point will be (0, 1). The gradient of f will have a single component $\nabla f(\mathbf{x}) = (\frac{\partial f(x)}{\partial x})$, corresponding with x.[20] We start by choosing a random starting value for x, let it be $x = 3$. When $x = 3, f(x) = 10$ and $\frac{\partial f}{\partial x} = \frac{df}{dx} = f'(x) = 6$. We take an additional scaling factor of 0.3. This will make us take only 30% of the step along the gradient we would normally take, and it will in turn enable us to be more precise in our quest for minimization. Later, we will call this factor the *learning rate*, and it will be an important part of our models.

[19] To get the actual $f(x)$ we just need to plug in the minimal x and calculate $f(x)$.

[20] In the case of multiple dimensions, we shall do the same calculation for every pair of x_i and $\nabla_i f(\mathbf{x})$.

We will be making a series of steps towards the x which will produce a minimal $f(x)$ (or more precisely, a good approximation of the actual minimal point[21]), we will denote the initial x by $x^{(0)}$, and we will denote all the other xs on the road towards the minimum in a similar fashion. So to get $x^{(1)}$ we calculate $x^{(0)} - 0.3 \cdot f'(x^{(0)})$, or, in numbers, $x^{(1)} = 3 - 0.3 \cdot 6 = 1.2$. Now, we proceed to calculate $x^{(2)} = x^{(1)} - 0.3 \cdot f'(x^{(1)}) = 1.2 - 0.3 \cdot 2.4 = 0.48$. By the same procedure, we calculate $x^{(3)} = 0.19$, $x^{(4)} = 0.07$ and $x^{(5)} = 0.02$ where we stop and call it a day.[22] We could continue to get better and better approximations, but we would have to stop eventually. Gradient descent will take as closer and closer to the value of x for which the function f has the minimal value, which is in our case $x^{(5)} \approx \operatorname{argmin} f(x) = 0$. Note that the minimum of f is actually 1 which we get if we plug in the *argmin* as x in $f(x) = x^2 + 1$. The interested reader may wonder what would happen if we used addition instead of subtraction: then we would be questing for a maximum not a minimum, but all the mechanics of the process would remain the same.

We make a short remark before moving on to statistics and probability. Mathematical knowledge is often considered to be common knowledge and as such it is not cited. That being said, most good math textbooks cite and provide historical remarks about the ideas and theorems proven. As this is not a mathematical book, we will no do that here. We will instead point the reader to other textbooks that do give a historical overview. We suggest that the reader interested in calculus starts her journey with [4], while for linear algebra we recommend [5]. One fantastic book that we believe any deep learning researcher should work her way through is [6], and we strongly recommend it.

2.3 Probability Distributions

In this section, we explore the various concepts from statistics and probability theory which we will be needing for deep learning. We will explore only the bits we will need for deep learning, but we point the interested reader towards two great textbooks, viz. [7][23] and [8].

Statistics is the quintessential data analysis: it analyses a *population* whose members have certain properties. All these terms will be rigorously defined later when we introduce machine learning, but for now we will use an intuitive picture: imagine the population to be the inhabitants of a city, and their properties[24] can be height,

[21] Note that a function can have many local minima or minimal points, but only one global minimum. Gradient descent can get 'stuck' in a local minimum, but our example has only one local minimum which is the actual global minimum.

[22] We stop simply because we consider it to be 'good enough'—there is no mathematical reason for stopping here.

[23] This book is available online for free at `https://www.probabilitycourse.com/`.

[24] Properties are called *features* in machine learning, while in statistics they are called *variables*, which can be quite confusing, but it is standard terminology.

weight, education, foot size, interests, etc. Statistics then analyses the population's properties, such as for example the average height, or which is the most common occupation. Note that for statistical analysis we have to have nice and readable data, but deep learning will not need this.

To find the average height of a population, we take the height of all inhabitants, add them up, and divide them by the number of inhabitants:

$$MEAN(height) := \frac{\sum_{i=1}^{n} height_i}{n} \tag{2.6}$$

The average height is also called *mean* of the height, and we can get a mean for any feature which has numerical values such as weight, body mass index, etc. Features that take numerical values are called *numerical features*. So the mean is a 'numerical middle value', but what can we do when we need a'middle value', for example, the population's occupation? Then, we can use the *mode*, which is a function which returns simply the value which occurs most often, e.g. 'analyst' or 'baker'. Note that the mod can be used for numerical features, but the mode will treat the values 19.01, 19.02 and 19000034 as 'equally different'. This means that if we want to take a meaningful mod, e.g. 'monthly salary', we should round the salary to the nearest thousand, so that 2345 becomes 2000 and 3987 becomes 4000. This process creates the so-called *bins* of data (it aggregates the data), and this kind of data preprocessing is called *binning*. This is a very useful technique since it drastically reduces the complexity of non-numerical problems and often gives a much clearer view of what is happening in the data.

Asides from the mean and the mode, there is a third way to look at centrality. Imagine we have a sequence 1, 2, 5, 6, 10000. With this sequence, the mod is quite useless, since no two values repeat and there is no obvious way to do binning. It is possible to take the mean but the mean is 2002.8, which is a lousy information, since it tells us nothing about *any* part of the sequence.[25] But the reason the mean failed is due to the atypical value of 10000 in the sequence. Such atypical values are called *outliers*. We will be in position to define outliers more rigorously later, but this simple intuition on outliers we have built here will be very useful for all machine learning endeavors. Remember just that the outlier is an atypical value, not necessarily a large value: instead of 10000, we could have had 0.0001, and this would equally be an outlier.

When given the sequence 1, 2, 5, 6, 10000, we would like a good measure of centrality which is not sensitive to outliers. The best-known method is called the *median*. Provided that the sequence we analyse has an odd number of elements, the median of the sequence is the value of the middle element of the sorted sequence.[26] In our case, the median is 5. If we have the sequence 2, 1, 6, 3, 7, the median would be the middle element of the sorted sequence 1, 2, 3, 6, 7 which is 3. We have noted

[25] Note that the mean is equally useless for describing the first four and the last member taken in isolation.

[26] The sequence can be sorted in ascending or descending order, it does not matter.

that we need an odd number of elements in the sequence, but we can easily modify the median a bit to take care of the case when we have an even number of elements: then sort the sequence, the two 'middlemost' elements, and define the median to be the mean of those two elements. Suppose we have 4, 5, 6, 2, 1, 3, then the two elements we need are 3 and 4, and their mean (and the median of the whole sequence) is 3.5. Note that in this case, unlike the case with an odd number of elements, the median is *not* also a member of the sequence, but this is inconsequential for most machine learning applications.

Now that we have covered the measures of central tendency,[27] we turn our attention to the concepts of expected value, bias, variance and standard deviation. But before that, we will need to address basic probability calculations and probability distributions. Let us take a step back and consider what probability is. Imagine we have the simplest case, a coin toss. This process is actually a simple *experiment*: we have a well-defined idea, we know all possible outcomes, but we are waiting to see the outcome of the current coin toss. We have two possible outcomes, heads and tails. The number of all possible outcomes will be important for calculating basic probabilities. The second component we need is how many times the desired outcome happens (out of all times). In a simple coin toss, there are two possibilities, and only one of them is heads, so $\mathbb{P}(heads) = \frac{1}{2} = 0.5$, which means that the probability of heads is 0.5. This may seem peculiar, but let us take a more elaborate example to make it clear. Usually, probability of x is denoted as $P(x)$ or $p(x)$, but we prefer the notation $\mathbb{P}(x)$ in this book, since probability is quite a special property and should not be easily confused with other predicates, and this notation avoids confusion.

Suppose we have a pair of D6 dice, and we want to know what is the probability of getting a five[28] on them. As before, we will need to calculate $\frac{A}{B}$ where B is the total number of outcomes and A is the time the desired outcome happens. Let us calculate A. We can get five on two D6 dice in the following cases:

1. First die 4, second die 1
2. First die 3, second die 2
3. First die 2, second die 3
4. First die 1, second die 4

So, we can get a five in four cases, and so $A = 4$. Let us calculate B now. We are counting how many outcomes are possible on two D6 dice. If there is a 1 on the first die, there are six possibilities for the second die. If there is a 2 on the first die, we also have six possibilities for the second, and so up to 6 on the first die. This means that there are $6 \cdot 6 = 6^2$ possibilities,[29] and hence $\mathbb{P}(5) = \frac{4}{36} = 0.11$. All simple probabilities are calculated like this by counting the number of times the

[27] This is the 'official' name for the mean, median and mode.

[28] Not 5 on one die or the other, but 5 as in when you need to roll a 5 in Monopoly® to buy that last street you need to start building houses.

[29] In 6^2, the 6 denotes the number of values on each die, and the 2 denotes the number of dice used.

desired outcome will occur and dividing it by the number of all possible outcomes. Please note one interesting thing: if the first die gives a 6 and the second gives a 1, this is one outcome, while if the first gives a 1 and the second gives a 6, this is another outcome. Also, there is only one combination which gives 2, viz. the first die gives a 1 *and* the second die gives a 1.

Now that we have an intuition behind the basic probability calculation,[30] let us turn our attention to probability distributions. A probability distribution is simply a function which tells us how often does something occur. To define the probability distributions, we first need to define what is a *random variable*. A random variable is a mapping from the probability space to a set of real numbers, or in simple words, it is a variable that can take random values. The random variable is usually denoted by X, and the values it takes are usually denoted by x_1, x_2, etc. Note that this 'random' can be replaced by a more specific *probability distribution*, which gives a higher chance for some values to occur (a lower-than-random chance for others). The simple, truly random case is the following: If we have 10 elements in the probability space, a random variable would assign to each the probability of 0.1. This is in fact the first probability distribution called *uniform distribution*, and in this distribution, all members of the probability space get the same value, and that value is $\frac{1}{n}$, where n is the number of elements. We have seen another probability distribution when we analysed the coin toss called the *Bernoulli distribution*. The Bernoulli distribution is the probability distribution of a random variable which takes the value 1 with the probability p and the value 0 with the probability $1 - p$. In our case, $p = \mathbb{P}(heads) = 0.5$, but we could have equally chosen a different p.

To continue, we must define the *expected value*. To build up intuition, we use the two D6 dice example. If we have a single D6 die, we have

$$\mathbb{E}_P[X] = x_1 \cdot p_1 + x_2 \cdot p_2 + \ldots + x_6 p_6, \tag{2.7}$$

where X is the random variable and P is a distribution of X (the xs come from X and ps belong to P). Since there are six outcomes, each one has the probability of $\frac{1}{6}$ this becomes

$$\mathbb{E}_{uniform}[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} \tag{2.8}$$

It seems rather trivial, but if we have two D6 dice, it becomes more complex, because the probabilities become messy, and the distribution is not uniform anymore (recall that the probability of rolling a 5 on two D6 is not $\frac{1}{36}$):

$$\mathbb{E}_{newDistribution}[X] = 2 \cdot \frac{1}{36} + 3 \cdot \frac{2}{36} + 4 \cdot \frac{3}{36} + 5 \cdot \frac{4}{36} + 6 \cdot \frac{5}{36} + 7 \cdot \frac{6}{36} + 8 \cdot \frac{5}{36} + 9 \cdot \frac{4}{36} + 10 \cdot \frac{3}{36} + 11 \cdot \frac{2}{36} + 12 \cdot \frac{1}{36} \tag{2.9}$$

[30] What we called here 'basic probabilities' are actually called *priors* in the literature, and we will be referring to them as such in the later chapters.

But let us see what is happening in the background when we talk about the expected value. We are actually producing an *estimator*,[31] which is a *function* which tells us what to expect in the future. What the future will actually bring is another matter. The 'reality' (also known as probability distribution) is usually denoted often by an uppercase letter from the back of the alphabet such as X, while an estimator for that probability distribution is usually denoted with a little hat over the letter, e.g. $\hat{X}$. The relationship between an estimator and the actual values we will be getting in the future[32] is characterized by two main concepts, the *bias* and the *variance*. The bias of $\hat{X}$ relative to X is defined as

$$BIAS(\hat{X}, X) := \mathbb{E}_P[\hat{X} - X] \tag{2.10}$$

Intuitively, the bias shows by how much the estimator misses the target (on average). A related idea is the *variance*, which tells how wider or narrower are the estimates compared to the actual future values:

$$VAR(\hat{X}) := \mathbb{E}_P[(\hat{X} - \mathbb{E}_P[\hat{X}])^2] \tag{2.11}$$

The *standard deviation* is defined as:

$$STD(\hat{X}) := \sqrt{VAR(\hat{X})} \tag{2.12}$$

Intuitively, the standard deviation keeps the spread information from the variance, but it rescales it to be directly useful.

We return now to probability calculations. We have seen how to calculate a basic probability (prior) like $\mathbb{P}(A)$, but we should develop a calculus for probability. We will provide both the set-theoretic notation and the logical notation in this section, but later we will stick to the less intuitive but standard set-theoretic notation. The most basic equation is the calculation of the joint probability of two independent events:

$$\mathbb{P}(A \cap B) = \mathbb{P}(A \wedge B) := \mathbb{P}(A) \cdot \mathbb{P}(B) \tag{2.13}$$

If we want the probability of two mutually exclusive events, we use

$$\mathbb{P}(A \cup B) = \mathbb{P}(A \oplus B) := \mathbb{P}(A) + \mathbb{P}(B) \tag{2.14}$$

If the events are not necessarily disjoint,[33] we can use the following equation:

$$\mathbb{P}(A \vee B) := \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \wedge B) \tag{2.15}$$

---

[31] All machine learning algorithms are estimators.

[32] Note that ideally we would like an estimator to be a *perfect* predictor of the future in all cases, but this would be equal to having foresight. Scientifically speaking, we have models and we try to make them as accurate as possible, but perfect prediction is simply not on the table.

[33] 'Disjoint' means $A \cap B = \emptyset$.

Finally, we can define the conditional probability of two events. The conditional probability of A given B (or in logical notation, the probability of $B \rightarrow A$) is defined as

$$\mathbb{P}(A|B) = \mathbb{P}(B \rightarrow A) := \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)} \tag{2.16}$$

Now, we have enough definitions to prove Bayes' theorem:

Theorem 2.1 $\mathbb{P}(X|Y) = \frac{\mathbb{P}(Y|X)\mathbb{P}(X)}{\mathbb{P}(Y)}$

Proof By the above definition of conditional probability (Eq. 2.16), we have that $\mathbb{P}(X|Y) = \frac{\mathbb{P}(X \cap Y)}{\mathbb{P}(Y)}$. Now, we must reformulate $\mathbb{P}(X \cap Y)$, and we will also be using the definition of conditional probability. By substituting X for B and Y for A in Eq. 2.16, we get $\mathbb{P}(Y|X) = \frac{\mathbb{P}(Y \cap X)}{\mathbb{P}(X)}$. Since $\cap$ is commutative, this is the same as $\mathbb{P}(Y|X) = \frac{\mathbb{P}(X \cap Y)}{\mathbb{P}(X)}$. Now, we multiply the expression by $\mathbb{P}(X)$ and get $\mathbb{P}(Y|X)\mathbb{P}(X) = \mathbb{P}(X \cap Y)$. We now know what is $\mathbb{P}(X \cap Y)$ and substitutes it in $\mathbb{P}(X|Y) = \frac{\mathbb{P}(X \cap Y)}{\mathbb{P}(Y)}$ to get $\mathbb{P}(X|Y) = \frac{\mathbb{P}(Y|X)\mathbb{P}(X)}{\mathbb{P}(Y)}$, which concludes the proof. □

This is the first and only proof in this book,[34] but we have included it since it is a very important piece of machine learning culture, and we believe that every reader should know how to produce it on a blank piece of paper. If we assume conditional independence of $Y_1, \ldots, Y_n$, then there is also a generalized form of the Bayes' theorem to account for multiple conditions (Y_{all} consists of $Y_1 \wedge \ldots \wedge Y_n$):

$$\mathbb{P}(X|Y_{all}) = \frac{\mathbb{P}(Y_1|X) \cdot \mathbb{P}(Y_1|X) \cdot \ldots \cdot \mathbb{P}(Y_n|X) \cdot \mathbb{P}(X)}{\mathbb{P}(Y_{all})} \tag{2.17}$$

We see in the next chapter how this is useful for machine learning. Bayes' theorem is named after Thomas Bayes, who first proved it, but the result was only published posthumously in 1763.[35] The theorem underwent formalization and the first rigorous formalization was given by Pierre-Simon Laplace in his 1774 Memoir on Inverse probability and later in his *Théorie analytique des probabilités* form 1812. A complete treatment of Laplace's contributions we have mentioned is available in [9,10].

Before leaving the green plains of probability for the desolate mountains of logic and computability, we must address briefly another probability distribution, the *normal* or *Gaussian* distribution. The Gaussian distribution is characterized by the following formula:

$$\frac{1}{\sqrt{2 \cdot VAR \cdot \pi}} e^{-\frac{(x-MEAN)^2}{2 \cdot VAR}} \tag{2.18}$$

[34] There are others, but they are in disguise.

[35] A version of Bayes' original manuscript is available at http://www.stat.ucla.edu/history/essay.pdf.

It is quite a weird equation, but the main thing about the Gaussian distribution is not the elegance of calculation, but rather the natural and nice shape of the graph, which can be used in a number of ways. You can see an illustration of how the Gaussian distribution with mean 0 and standard deviation 1 looks like (see Fig. 2.2a).

The idea behind the Gaussian distribution is that many *natural* phenomena seem to follow it, and in machine learning it is extremely useful for initializing values that are random but at the same time are centred around a value. This value is the mean, and it is usually set to 0, but it can be anything. There is a related concept of a *Gaussian cloud*, which is made by sampling a Gaussian distribution with mean 0 for two values at a time, adding the values to a point with coordinates (x, y) (and drawing the results if one wishes to see it). Visually, it looks like a 'dot' made with the spray paint tool from an old graphical editing program (see Fig. 2.2b).

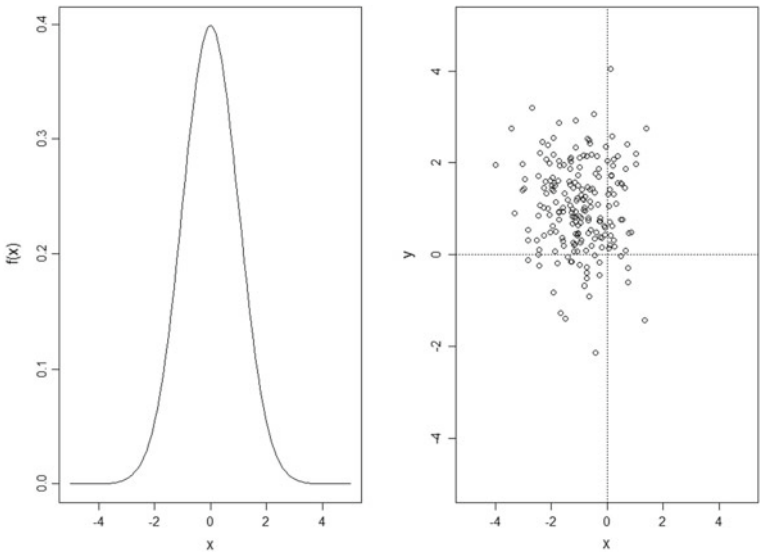


Fig. 2.2 Gaussian distribution and Gaussian cloud

2.4 Logic and Turing Machines

We have already encountered logic in the very beginnings of artificial neural networks, and again with the XOR problem, but we have not really discussed it. Since logic is a highly evolved and mathematical science, an in-depth introduction to logic is far beyond the scope of this book, and we point the reader to [11] or [12], which are both excellent introductions. We are going to give only a very quick tour here, and focus exclusively on the parts which are of direct theoretical and practical significance to deep learning.

Logic is the study of foundations of mathematics, and as such it has to take something to be undefined. This is called a *proposition*. Propositions are represented by symbols $A, B, C, P, Q, \ldots, A_1, B_1, \ldots$. Usually, the first letters are reserved for atomic propositions, while the Ps and Qs are reserved for denoting any proposition, atomic or compound. Compound propositions are built over atomic ones with logical connectives, $\wedge$ ('and'), $\vee$ ('or'), $\neg$ ('not'), $\rightarrow$ ('if…then') and $\equiv$ ('if and only if'). So if A and B are propositions, so is $A \rightarrow (A \vee \neg B)$. All of the connectives are binary, except for negation which is unary. Another important aspect is truth functions. Intuitively, an atomic proposition is assigned either 0 or 1, and a compound proposition gets 0 or 1 depending on whether its components are 0 or 1. So if $t(X)$ is a truth function, $t(A \wedge B) = 1$ if and only if $t(A) = 1$ and $t(B) = 1$, $t(A \vee B) = 1$ if and only if $t(A) = 1$ or $t(B) = 1$, $t(A \rightarrow B) = 0$ if and only if $t(A) = 1$ and $t(B) = 0$, $t(A \equiv B) = 1$ if and only if $t(A) = 1$ and $t(B) = 1$ or $t(A) = 0$ and $t(B) = 0$, and $t(\neg A) = 1$ if and only if $t(A) = 0$. Our old friend, *XOR*, lives here as $XOR(A, B) := A \equiv B$.

The system we described above is called *propositional logic*, and we might want to modify it a bit. Let us briefly address a first modification, *fuzzy logic*. Intuitively, if we allow the truth values to be not just 0 or 1 but actually real values between 0 and 1, we are in fuzzy logic territory. This means that a proposition A (suppose that A means 'This is a steep decline') is not simply 1 ('true'), but can have the value 0.85 ('"kinda" true'). We will be needing this general idea. Connections between fuzzy logic and artificial neural networks form a vast area of active research, but we cannot go in any detail here.

But the main extension of propositional logic is to decompose propositions in properties, relations and objects. So, what was simply A in propositional logic becomes $A(x)$ or $A(x, y, z)$. The x, y, z are then called *variables*, and we need a set of valid objects over which they span, called the *domain*. $A(x, y)$ could mean 'x is above y', and this is then either true or false depending on what we give as x and y. So the main option is to provide two constants c and d which denote some particular members of the domain, say 'lamp' and 'table'. Then $A(c, d)$ is true. But we can also use quantifiers, $\exists$ ('exists') and $\forall$ ('for all') to say that there exists some object which is 'blue', and we write $\exists x B(x)$. This is true if there is any object in the domain which is blue. Same goes $\forall$, and the syntax is the same, but it will be true if all members of the domain are blue. Of course you can also compose sentences like $\exists x(\forall y A(x, y) \wedge \exists z \neg C(x, z))$, the principle is the same.

We can also a quick look at fuzzy first-order logic. Here, we have a predicate P (suppose $P(x)$ means 'x is fragile') and a term c (denoting a flower pot). Then, $t(P(c)) = 0.85$ would mean that the flower pot is 'kinda' fragile. You can look at it from another perspective, as fuzzy sets: take P to be the set of all fragile things, and c then belongs to the fuzzy set P with a degree of 0.85.

One important topic from logic we need to cover is a Turing machine. It is the original simulator of a universal machine from the previously mentioned paper by Alan Turing [13]. The Turing machine has a simple appearance, comprising two parts: a tape and a head. A tape is just an imaginary piece of paper that is infinitely long and is divided into cells. Each cell can either be filled with a single dot (•), with a separator (#) or blank (B). The head can read and memorize a single symbol, write or erase a symbol from a cell on the tape. It can go to any cell of the tape. The idea is that this simple device can compute any function that can be computed at all. In other words, the machine works by getting instructions, and any computable function can be rewritten as instructions for this machine. If we want to compute addition of 5 and 2, we could do it in the following manner:

1. Start by writing the blank on the first cell. Write five dots, the separator and three dots.
2. Return to the first blank.
3. Read the next symbol and if it is a dot, remember it, go right until you find a blank, write the dot there. Else, if the next symbol is a separator return to the beginning and stop.
4. Return to step 2 of this instruction and start over from there.

We conclude with the definition of logic gates. A logic gate is a representation of a logical connective. An AND gate takes two inputs, and if they are both 1, it outputs a 1. An XOR gate is also a gate which gives 1 if a 1 is coming from either side, gives 0 if nothing is coming, and blocks (produces a 0) if both are coming with 1. A special kind of a logic gate is a voting gate. This gate takes not just two but n inputs, and outputs a 1 if more than half of the inputs are 1. A generalization of the voting gate is the *threshold gate* which has a threshold. If T is the threshold, then the threshold gate outputs 1 if more than T inputs are 1 and 0 otherwise. This is the theoretical model of all simple artificial neurons: in terms of theoretical computer science, they are simply threshold logic gates and have the same computational power.

A natural physical interpretation for logic gates is that they are a kind of switch for electricity, where 1 represents current and 0 no current.[36] Most of the things work out (some gates are impossible but they can be obtained as a combination of others), but consider what happens to a negation gate when 0 is coming: it should produce 1, but this eludes our intuitions about currency (if you put two switches on the same

[36]This is not exactly how it behaves, but it is a simplification which is more than enough for our needs.

line and close one, closing the other will not produce a 1). This is a strong case for intuitionistic logic where the rule $\neg\neg P \rightarrow P$ does not hold.

2.5 Writing Python Code

Machine learning today is a process inseparable from computers. This means that any algorithm is written in program code, and this means that we must choose a language. We chose Python. Any programming language is actually just a specification of code. This means to write a program you simply open a textual file, write the correct code and then change the extension of the file from `.txt` into something appropriate. For ANSI C, this is `.c`, and for Python this is `.py`. Remember, a valid code is defined by the given language, but all program code is just text, nothing else, and can be edited by any text editor.[37]

A programming language can be compiled or interpreted. A compiled language is processed by compiling the code, while an interpreted language uses another program called an 'interpreter' as a platform. Python is an interpreted language (ANSI C is a compiled language), and this means we need an interpreter to run Python programs. The usual Python interpreter is available at `python.org`, but we suggest to use Anaconda from `www.continuum.io/downloads`. There are currently two versions of Python, Python 3 and Python 2.7. We suggest to use the latest version of Python, which at the time of writing is Python 3.6. When installing Anaconda, use all the default options except the one that asks you whether you would like to prepend Anaconda to the path. If you are not sure what this means, select 'yes' (the default is 'no'), since otherwise you might end up in a place called 'dependency hell'. There are detailed instructions on how to install Anaconda on the Anaconda web page, and you should consult those.

Once you have Anaconda installed, you must create an Anaconda environment. Open your command prompt (Windows) or terminal (OSX, Linux) and type `conda create -n dlBook01 python=3.5` and hit enter. This creates an Anaconda environment called `dlBook01` with Python 3.5. We need this version for TensorFlow. Now, we must type in the command line `activate dlBook01` and hit enter, which will activate you Anaconda environment (your prompt will change to include the name of the environment). The environment will remain active as long as

[37] Text editors are Notepad, Vim, Emacs, Sublime, Notepad++, Atom, Nano, cat and many others. Feel free to experiment and find the one you like most (most are free). You might have heard of the so-called *IDE*s or *Integrated Development Environments*. They are basically text editors with additional functions. Some IDEs you might know of are Visual Studio, Eclipse and PyCharm. Unlike text editors, most IDEs are not freely available, but there are free versions and trial versions, so you may experiment with them before buying. Remember, there is nothing *essential* an IDE can do but a text editor cannot, but they do offer additional *conveniences* in IDEs. My personal preference is to use Vim.

the command prompt is opened. If you close it, or restart your computer, you must type again `activate dlBook01` and hit enter.

Inside this environment, you should install TensorFlow from https://www.tensorflow.org/install/. After activating your environment, you should write the command `pip install –upgrade tensorflow` and hit enter. If this fails to work, put `pip3 install –upgrade tensorflow` and hit enter. If it still does not work, try to troubleshoot the problem. The usual way to troubleshoot problems is to open the official web page of the application and follow instructions there, and if it fails, try to consult the FAQ section. If you still cannot resolve the issue, try to find the answer on `stackoverflow.com`. If you cannot find a good answer, you can ask the community there for help and usually you will get a response in a couple of hours. The final step is to install Keras. Check `keras.io/#installation` to see whether you need any dependencies and if you are good to go, just type `pip install keras`. If Keras fails to install, consult the documentation on `keras.io`, and if it does not help, it is StackOverflowing time again.

Once you have everything installed, type in the command line `python` and hit enter. This will open the Python interpreter, which will then display a line or two of text, where you should find 'Python 3.5' and 'Anaconda' written. If it does not work, try restarting the computer, and then activate the anaconda environment again and try to write `python` again and see whether this fixes the issue. If it does not, StackOverflow it.

If you manage to open the Python interpreter (with 'Python 3.5' and 'Anaconda…' written), you will have a new prompt looking like >>>. This is the standard Python prompt which will interpret any valid Python code. Try to type in 2+2 and hit enter. Then try '2'+'2' to get '22'. Now try to write `import tensorflow`. It should just write a new prompt with >>>. If it gives you an error, StackOverflow it. Next, do the same thing to verify the Keras installation. Once you have done this, we are done with installation.

Every section of this book will contain a fragmented code. For every section, you should make one file and put the code from that section in that file. The only exceptions from this are the sections in the chapter on Neural Language Models. There the code from both sections should be placed in a single file. Once you save the code to a file, open the command line, navigate to the directory containing the code file (let us call it `myFile.py`), activate the `dlBook01` environment, type in `python myFile.py` and hit enter. The file will execute, print something on the screen and perhaps create some additional files (depending on the code). Notice the difference between the commands `python` and `python myFile.py`. The former opens the Python interpreter and lets you type in code, and the latter runs the Python interpreter on the file you have specified.

2.6 A Brief Overview of Python Programming

In the last section, we have discussed installation of Python, TensorFlow and Keras, as well as how you should make an empty Python file. Now it is time to fill it with code. In this section, we will explore the basic data structures and commands in Python. You can put everything we will be exploring in this section in a single Python file (we will call it `testing.py`). To run it, simply save it, open a command line in the location of the file and type `python testing.py`. We start out by writing the first line of the file:

```
print("Hello, world!")
```

This line has two components, a string (a simple data structure equivalent to a series of words) `"Hello world!"` and the function `print( )`. This function is a *built-in function*, which is a fancy name for a prepackaged function that comes with Python. You can use these functions to define more complex functions, but we will get to that soon. You can find a list and explanation of all the built-in functions at https://docs.python.org/3/library/functions.html. If this or any other link becomes obsolete, simply use a search engine to locate the right web page.

One of the most basic concepts in Python is the notion of *type*. Python has a number of types but the most basic ones we will need are *string* (str), *integers* (int) and *decimals* (float). As we have noted before, strings are words or series of words, ints are simply whole numbers and floats are decimal numbers. Type in `python` in a command line and it will open the Python interpreter. Type in `"1"==1`, an it will return `False`. This relation (==) means 'equal', and we are telling Python to evaluate whether is "1" (a string) equal to 1 (an int). If you put `!=` instead of `==`, which means 'not equal', then Python will return `True`.

The problem is that Python cannot convert an int to a string, or vice versa, but you could try to tell Python `int("1")==1` or `"1"==str(1)` and see what happens. Interestingly, Python *can* convert ints to floats and vice versa, so `1.0==1` evaluates to `True`. Note that the operation + has two meanings, for ints and floats, it is addition, and for strings it is concatenation (sticking two strings together): `"2"+"2"=="22"` returns `True`.

Let us return to our file, `testing.py`. You can use the basic functions to define a more complex one as follows:

```
def subtract_one(my_variable): #this is the first line of code
    return (my_variable - 1)#this is the second line…
print(subtract_one(53))
```

Let us dig into the anatomy of this code, since this is a basis for any more complex Python code. The first line defines (with the command `def`) a new function called `subtract_one` taking a single value referred to as `my_variable`. The line ends with a colon telling Python that it will be given more instructions. The symbol # begins a *comment*, which lasts until the end of the line. A comment is a piece of text inside the Python code file which the interpreter will ignore, and you can put there anything from notes to alternative code.

The second line begins with four ␣. They denote whitespace (the character which the space bar puts in the text, and you see between words of a text). Whitespaces which come in blocks of four are called *indentations*. An alternative way is to use a single tab in place of a block of four whitespaces, but you have to be consistent: if you use whitespaces in one file, then you should use whitespaces throughout that file. In this book, we use whitespaces. After the whitespaces, the line has a `return` command which says to finish the function and return whatever is after the return statement. In our case, the function returns `my_variable - 1` (the parentheses are just to make sure Python does not misunderstand what to bring back from the function). After this, we have a new comment, which the interpreter will ignore, so we may write anything there.

The third line is outside the definition of the function, so it has no indent, and it actually calls the inbuilt function print on our defined function on the value of 53. Notice that without the print, our function would execute, but we would not see anything on the screen since the function does not print anything per se, so we needed to add the print. You can try to modify the defined function so that it prints something, but remember that you need to define first and use after (i.e. a simple copy/paste will not work). This will give you a nice feel of the interaction between `print` and `return`. Every indented whole together with the line preceding the indent (function definition) in Python is called a *block of code*. So far we have seen only the definition block, but other blocks work in the same way. Other blocks include the `for`-loop, the `while`-loop, the `try`-loop, the `if`-statement[38] and several others.

One of the most fundamental and important operations in Python is the variable assignment operation. This is simply placing a value in a new variable. It is done with the command `newVariable =″someString″`. You can use assignments to assign any value to a variable (any string, float, int, list, dictionary–anything), and you can also reuse variables (a variable in this sense is just the name of the variable), but the variable will keep only the most recent assignment value.

Let us revisit strings. Take the string `'testString'`. Python allows to put strings in either single quotes or double quotes `""`, but you must end the string with the same symbol you started it. The empty string is denoted as `″` or `""`, and this is a substring of any string. Try opening the Python interpreter and writing in `"test" in 'testString'`, `"text" in 'testString'`, `"" in "testString"` and even `"" in ""`, and see how it behaves. Try also `len("Deep Learning")` and `len("")`. This is a built-in function which returns the length of an *iterable*. An iterable is a string list, dictionary and any other data structure which has parts. Floats, ints and characters are not iterables, and most other things in Python are.

You can also get substrings of a string. You can first make an assignment of a string to a variable and work with the variable or you can work directly with the string. Write in the interpreter `myVar = "abcdef"`. Now try telling Python `myVar[0]`. This will return the first letter of the string. Why 0? Python starts

[38] Never call this an 'if-loop', since it is simply wrong.

indexing iterables with ints from 0 onwards, and this means that to get the first element of the iterable you need to use the index 0. This also means that each `string` has N-1 values for indices where `N=len(string)`. To get the `f` from `myVar`, you can use `myVar[-1]` (this means 'last element') or a more complex `myVar[(len(myVar)-1)]`. You will always use the -1 variant but it is important to notice that these expressions are equivalent. You can also save a letter from a string to a variable with this notation. Type in `thirdLetter = myVar[2]` to save the `"c"` in the variable. You can also take out substrings like this. Try to type `sub_str = myVar[2:4]` or `sub_str = myVar[2:-2]`. This simply means to take indices from 2 to 4 (or from 2 to -2). This works for any iterable in Python, including lists and dictionaries.

A *list* is a Python data structure capable of holding a wide variety of individual data. A list uses square parentheses to enclose individual values. As an example, `[1,2,3,["c",[1.123,"something"]],1,3,4]` is an example of a list. This list contains another list as one of its elements. Notice also that a list does not omit repeating values and order in the list matters. If you want to add a value of say 1.234 to a list `myList`, just use the function `myList.append(1.234)`. If you need a blank list, just initialize one with a fresh variable, e.g. `newList = []`. You can use both the `len( )` and the index notation we have seen for strings for lists as well. The syntax is the same.[39] Try to initialize blank lists and then adding stuff to them and also to initialize lists as the one we have shown (remember, you must assign a list to a variable to be able to work with it over multiple lines of code, just like a string or number). Also, try finding more methods like `append()` in the official Python documentation or on StackOverflow and play around with them a bit in the test file or the Python interpreter. The main idea is to feel comfortable with Python and to expand your knowledge gradually. Programming is very boring and hard at first, but soon becomes easy and fun if you put in the effort, and it is an extremely valuable skill. Also, do not give up if at first some code does not work: experiment `print()` every part to make sure it connects well and search StackOverflow. If you start coding fulltime, you will be writing code for at most two hours a day, and spend the rest of the time correcting it and debugging it. It is perfectly normal, and debugging and getting the code to work is an essential part of coding, so do not feel bad or give up.

Lists have elements, and you can retrieve an element of a list by using the index of that element. This is the only proper way to do it. There is a different data structure which is like a list, but instead of using an index uses user-defined keywords to fetch elements. This data structure is called a *dictionary*. An example of a dictionary is `myDict={"key_1":"value_1", 1:[1,2,3,4,5], 1.11:3.456, 'c':{4:5}}`. This is a dictionary with four elements (its `len()` is 4). Let us take the first element: it has two components, a *key* (the keyword which fulfills the same role as an index in a list) and a *value* which is the same

[39] In a programming jargon, when we say 'the syntax is the same' or 'you can use a similar syntax' means that you should try to reproduce the same style but with the new values or objects.

as the elements in a list. You can put anything as a value, but there are restrictions on what can be used as a key: only strings, chars, ints and floats—no dictionaries or lists are allowed as keys. Say we want to retrieve the last element of the above dictionary (the one with the key `'c'`). To do so we write `retrieved_value=myDict['c']`. If we want to insert a new element, we cannot use `append()` since we have to specify a key. To insert a new element we simply tell Python `myDict['new_key']='new_value'`. You can use anything you like for the value, but remember the restrictions on keys. You initialize a blank dictionary the same way you would a list, but with curly braces.

We must make a remark. Remember that we said earlier that you can represent vectors with lists. We can also use lists to represent trees (the mathematical structures), but for graphs we need dictionaries. Labelled trees can be represented in a variety of ways but the most common is to use the members of the list to represent the branching. This means that the whole list represents the root, its elements represent the nodes that come after the root, its elements the nodes that come after and so on. This means that `tree_as_list[1][2][3][0][4]` represents a branch, namely the branch you have when you take the second branch from the root, the third branch after that, the fourth after that, the first after that and the fifth after that (remember that Python starts indexing with 0). For a graph, we use the node labels as keys and then in the values we pass on a list containing all nodes which are accessible for the given node. Therefore, if we have an element of the dictionary `3:[1,4]`, means that from the node labelled 3 we can access nodes labelled 1 and 4.

Python has built-in functions and defined functions, but there are a lot of other functions, data structures and methods, and they are available from external libraries. Some of them are a part of the basic Python bundle, like the module `time`, and all you have to do is write `import time` at the beginning of the Python file or when you start the Python interpreter command line. Some of them have to be installed first via `pip`. We have advised you to install Anaconda. Anaconda is simply Python with some of the most common scientific libraries pre-installed. Anaconda has a lot of useful libraries, but we need TensorFlow and Keras on top of that, so we have installed them with `pip`. When we will be writing code, we will import them with lines such as `import numpy as np`, which imports the whole Numpy library (a library for fast computation with arrays), but also assigns `np` as a quick name with which we shall refer to Numpy throughout the current Python file.[40] It is a common omission to leave out an import statement, so be sure to check all import statements you are using.

Let us see another very important block, the `if`-block. The `if`-block is a simple block of code used for forking in the code. This type of block is very simple and self-explanatory, so we proceed to an example:

---

[40] Note that even though the name we assign to a library is arbitrary, there are standard abbreviations used in the Python community. Examples are `np` for Numpy, `tf` for TensorFlow, `pd` for Pandas and so on. This is important to know since on StackOverflow you might find a solution but without the import statements. So if the solution has `np` somewhere in it, it means that you should have a line which imports Numpy with the name `np`.

```
if condition==1:
␣␣␣␣return 1
elif condition==0:
␣␣␣␣print("Invalid input")
else:
␣␣␣␣print("Error")
```

Every `if`-block depends on a statement. In our case, this is the statement that a variable named `condition` has the value 0 or 1 assigned to it. The block then evaluates the statement `condition==1` (to see whether the value in `condition` is equal to 1), and if it is true, it continues to the indented part. We have specified this to be just `return 1`, which means that the output of the whole function where this `if`-block lives will be 1. If the statement `condition==1` is false, Python will continue to the `elif` part. `elif` is just 'else-if', which means that you can give it another statement to check, and we pass in the statement `condition==0`. If this statement evaluates to true, then it will print the string `"Invalid input"`, and return nothing.[41] In an `if`-block, we must have exactly one `if`, either zero or one `else`, and as many `elif` as we like (possibly none). The `else` is here to telly Python what to do if neither of our conditions is met (the two conditions we had are `condition==0` and `condition==1`). Note that the variable name `condition` and the conditions themselves are entirely arbitrary and you can use whatever makes sense for your program. Also, notice that each one of them ends with `:`, and the omission of the colon is a frequent beginner's bug.

The `for`-loop is the main loop in Python used to apply the same procedure to all members of an iterable. Let us see an example:

```
someListOfInts = [0,1,2,3,4,5]
for item in someListOfInts:
␣␣␣␣newvalue = 10*item
␣␣␣␣print(newvalue)
print(newvalue)
```

The first line defines the loop: it has a `for` part which tells Python that it is a `for`-loop, and right after it has a dummy variable which we called `item`. The value of this variable will be changed after each pass and will be subsequently assigned the value `None` after the loop is over. The `someListOfInts` is a list of ints. It is more usual to create a list of ints with the function `range(k,m)`, where `k` is the starting point (it may be omitted, and then it defaults to 0), and `m` is the bound: `range(2,9)` produces the list `[2,3,4,5,6,7,8]`.[42] The indented lines of code do something with every `item`, in our case they multiply them by 10 and print

[41] In Python, technically speaking, every function returns something. If no return command is issued, the function *will* return None which is a special Python keyword for 'nothing'. This a subtle point, but also the cause of many intermediate-level bugs, and therefore it is worth noting it now.

[42] In Python 3, this is no longer *exactly*that list, but this is a minor issue at this stage of learning Python. What you need to know is that you can count on it to behave *exactly* like that list.

them out. The last non-indented line of the code will simply show you the last (and current) value of `newvalue` after the whole `for`-loop. Notice that if you substitute `someListOfInts` in the `for`-loop with `range(0,6)` or `range(6)`, the code will work exactly the same (of course, you can then delete the `someListOfInts = [0,1,2,3,4,5]` line). Feel free to experiment with the `for`-loop, these loops are very important.

We have seen how the `for`-loop works. It takes an iterable (or produces one with the `range()` function), and does something (which is to be specified by the indented block) with the elements from the iterable. There is another loop called the `while`-loop. The `while`-loop does not take an iterable, but a statement, and executes the commands from the indented block as long as the statement is true. This 'as long as the statement is true' is less weird than it sounds, since you want to put a statement which will be modified in the indented block (and whose truth value will change with subsequent passes). Imagine a simple thermostat program told to heat up the room to 20 degrees:

```
room_temperature = 14
while room_temperature != 20:
␣␣␣␣room_temperature = room_temperature + 2
␣␣␣␣print(room_temperature)
```

Notice the fragility of this code. If you put a `room_temperature` of 15, the code will run forever. This shows how careful you must be to avoid possible huge errors that might happen if you change slightly some parameter. This is not a unique feature of `while` loops, and it is a universal programming problem, but here it is very easy to show this pitfall, and how to easily correct it. To correct this bug,[43] you could but `while room_temperature < 20:`, or use a temperature update step of 1 instead of 2, but the former method (< instead of `!=`) is more robust.

In general computer science terminology, a valid Python dictionary is called a JSON object.[44] This may seem weird, but dictionaries are a great way to store information across various applications and languages, and we want other applications not using Python or JavaScript to be able to work with information stored in a JSON. To make a JSON object, write a valid dictionary in a plain text file called `something.json`. You can do it with the following code:

```
employees={"Tom":{"height":176.6}, "Ron":{"height":
180, "skills":["DIY", "Saxophone playing"], "room":12},
"April":"Employee did not fill the form"}
with open("myFile.json", "w") as json_file:
␣␣␣␣json_file.write(str(employees))
```

[43] Notice that the code, as it stands now, does not have this problem, but this is a bug since a problem would arise if the room temperature turns out to be an odd number, and not an even number as we have now.

[44] JSON stands for *JavaScript Object Notation*, and JSONs (i.e. Python dictionaries) are referred to as *objects* in JavaScript.

You can additionally specify a path to the file, so you can write `Skansi/Desktop/myFile.json`. If you do not specify a path, the file will be written in the folder you are currently in. The same holds for opening a file. To open a JSON file, use the following code (you can use the encoding argument when writing or reading the file):

```
with open("myFile.json", 'r', encoding='utf-8') as text:
    for line in text:
        wholeJSON = eval(line)
```

You can modify this code to write any text, not just JSON, but then you need to go through all the lines when opening, and when writing to a file you might want to use `"a"` as the argument so that it appends (the `"w"` just overwrites it). This concludes our brief overview of Python. With a bit of help from the internet and some experimenting, this could be enough to get started without any previous knowledge, but feel free to seek out a beginner's course online since a detailed introduction to Python is beyond the scope of this book. We recommend David Evans' free course on Udacity (www.udacity.com, Introduction to Computer Science), but any other good introductory course will serve the purpose.

References

1. J.R. Hindley, J.P. Seldin, *Lambda-Calculus and Combinators: An Introduction* (Cambridge University Press, Cambridge, 2008)
2. G.S. Boolos, J.P. Burges, R.C. Jeffrey, *Computability and Logic* (Cambridge University Press, Cambridge, 2007)
3. P. Renteln, *Manifolds, Tensors, and Forms: An Introduction for Mathematicians and Physicists* (Cambridge University Press, Cambridge, 2013)
4. R. Courant, J. Fritz, *Introduction to Calculus and Analysis*, vol. 1 (Springer, New York, 1999)
5. S. Axler, *Linear Algebra Done Right* (Springer, New York, 2015)
6. P.N. Klein, *Coding the Matrix* (Newtonian Press, London, 2013)
7. H. Pishro-Nik, *Introduction to Probability, Statistics, and Random Processes* (Kappa Books Publishers, Blue Bell, 2014)
8. D.P. Bertsekas, J.N. Tsitsiklis, *Introduction to Probability* (Athena Scientific, Nashua, 2008)
9. S.M. Stigler, Laplace's 1774 memoir on inverse probability. Stat. Sci. **1**, 359–363 (1986)
10. A. Hald, *Laplace's Theory of Inverse Probability, 1774–1786* (Springer, New York, 2007), pp. 33–46
11. W. Rautenberg, *A Concise Introduction to Mathematical Logic* (Springer, New York, 2006)
12. D. van Dalen, *Logic and Structure* (Springer, New York, 2004)
13. A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem. Proc. Lond. Math. Soc. **42**(2), 230–265 (1936)

Machine Learning Basics

3

Machine learning is a subfield of artificial intelligence and cognitive science. In artificial intelligence, it is divided into three main branches: *supervised learning*, *unsupervised learning* and *reinforcement learning*. Deep learning is a special approach in machine learning which covers all three branches and seeks also to extend them to address other problems in artificial intelligence which are not usually included in machine learning such as knowledge representation, reasoning, planning, etc. In this book, we will cover supervised and unsupervised learning.

In this chapter, we will be providing the general machine learning basics. These are not part of deep learning, but prerequisites that have been carefully chosen to enable a quick and easy grasp of the elementary concepts needed for deep learning. This is far from a complete treatment, and for a more comprehensive treatment we refer the reader to [1] or any other classical machine learning textbook. The reader interested in the GOFAI approach to knowledge representation and reasoning should consult [2]. The first part of this chapter is devoted to supervised learning and its terminology, while the last part is about unsupervised learning. We will not be covering reinforcement learning and we refer the reader to [3] for a comprehensive treatment.

3.1 Elementary Classification Problem

Supervised learning is just classification. The trick is that a vast amount of problems can be seen as classification problems, for example, the problem of recognizing a vehicle in an image can be seen as classifying the image in one of the two classes: 'has vehicle' or 'does not have vehicle'. Same goes for predictions: if we need to

make a portfolio of penny stocks, we can reformulate it to be a classification problem of the form: 'winner! will rise 400% or more' or 'nay, pass'.

Of course the trick is to make a classifier that is good enough. We have two options, either selecting by hand with some property or combination of properties (e.g. is the stock bottoming and making an RSI divergence and trading on a low high for the past two days) or we can remain agnostic about the properties we need and simply say 'look, I have 5000 examples of good ones and 5000 examples of bad ones, feed it to an algorithm and let it decide whether the 10001st is more similar to the good ones or the bad ones in terms of the properties it has'. The latter is the quintessential machine learning approach. The former is known as *knowledge engineering* or *expert system engineering* or (historical term) *hacking*. We will focus on the machine learning approach here.

Let us see what 'classification' means. Imagine that we have two classes of animals, say 'dogs' and 'non-dogs'. In Fig. 3.1, each dog is marked with an X and all 'non-dogs' (you can think of them as 'cats') are marked with an O. We have two properties for them, their length and their weight. Each particular animal has the two properties associated with it and together they form a *datapoint* (a point in space where the axes are the properties). In machine learning, properties are called *features*. The animal can have a *label* or *target* which says what it is: the label might be 'dog'/'non-dog' or simply '1'/'0'. Notice that if we have the problem of multiclass classification (e.g. 'dog', 'cat' and 'ocelot'), we can first perform a 'dog'/'non-dog' classification and then on the 'non-dog' datapoints perform a 'cat'/'non-cat' classification. But this is rather cumbersome and we will develop techniques for multiclass classification which can do it right away without the need to transform it in $n - 1$ binary classifications.

Returning to our Fig. 3.1, imagine that we have three properties, the third being height. Then, we would need a 3D coordinate system or space. In general, if we have n properties, we would need an n-dimensional system. This might seem hard to imagine, but notice what is happening in the 2D versus 3D case and then generalize it: look at the two animals which have the 2D coordinates (38, 7) (it is the overlapping

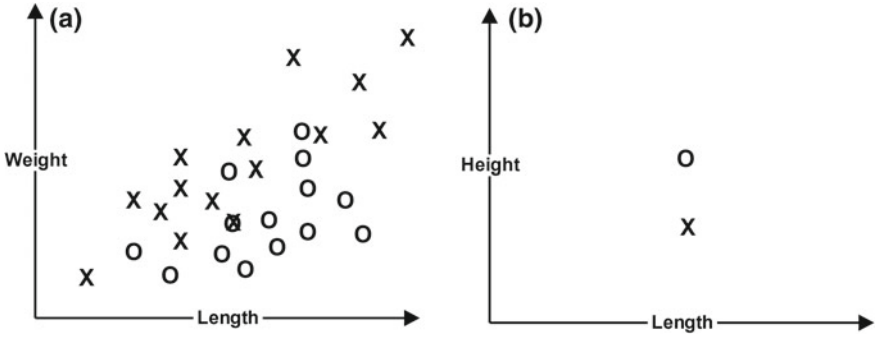


Fig. 3.1 Adding a new dimension

X and O in Fig. 3.1a). We will never be able to distinguish them, and if a new animal were to have this length and weight we would not be able to conclude what it is.

But take a look at the 'top view' in Fig. 3.1b where we have added an axis z: if we were to know that its height (coordinate z) is 20 for one and 30 for the another, we could now easily separate them in this 3D space, but we would need a plane instead of a line if we wanted to draw a boundary between them (and this boundary drawing is actually the essence of classification). The point is that adding a new feature and expanding our graph to a new dimension offers us new ways to separate what was very hard or even impossible in a lower number of dimensions. This is a good intuition to keep while imagining 37-dimensional space: it is the expansion of 36-dimensional space with one extra property that will enable us (hopefully) to better distinguish what we could not distinguish in 36-dimensional space. In a 4D space or higher, this plane is which divides cats and dogs the so-called a *hyperplane* which is one of the most important concepts in machine learning. Once we have the hyperplane which separates the two classes in an n-dimensional space, we know for a new unlabelled datapoint what (probably) is just by looking whether it falls in the 'dog' side or the 'non-dog' side.

Now, the hard part is to draw a good hyperplane. Let us return to the 2D world where we have just a line (but we will keep calling it 'hyperplane' to inculcate the terminology) and look at some examples. Xs and Os represent dogs and cats (labelled datapoints) and little squares represent new unlabelled datapoints. Notice that we have all the properties for these new datapoints, we are just missing a label and we have to find it. We even know how to find it: see on which side of the hyperplane the datapoint is and then add the label which is the label of that side of the hyperplane.[1] Now, we only need to find out how to define the hyperplane. We have one fundamental choice: should we ignore the labelled datapoints and draw the hyperplane by some other method, or should we try to draw the hyperplane so that it *fits* the existing labelled datapoints nicely? The former approach seems to be the epitome of irrationality, while the latter is the machine learning approach.

Let us comment on the different hyperplanes drawn in Fig. 3.2. Hyperplane A is more or less useless. It has a certain appeal since it does separate the datapoints in a manner that on the 'dog' side there are more dogs than non-dogs and on the 'non-dog' side there are more non-dogs. But it seems that we could have done this with no data at all. Hyperplane B is similar, but it has an interesting feature, namely that on the 'non-dog' side all datapoints are non-dogs. If a new datapoint falls here, we would be very confident that it is a cat. On the other side, things are not good. But if we recast this problem in a marketing setting where Os represent people who will most probably buy a product, then a hyperplane like B would provide a very

[1] You may wonder how a side gets a label, and this procedure is different for the various machine learning algorithms and has a number o peculiarities, but for now you may just think that the side will get the label which the majority of datapoints on that side have. This will usually be true, but is not an elegant definition. One case where this is not true is the case where you have only one dog and two cats overlapping (in 2D space) it and four other cats. Most classifiers will place the dog and the two cats in the category 'dog'. Cases like this are rare, but they may be quite meaningful.

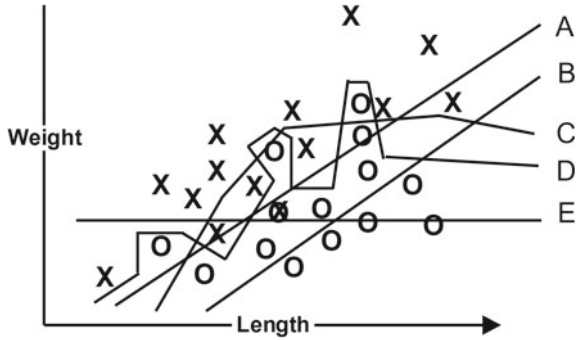


Fig. 3.2 Different hyperplanes

useful separation. Hyperplane E is even worse than hyperplane A, but to define it we just need a threshold on the weight like $weight > 5$. Here, we could quite easily combine it with other parameters and find a better separation by purely logical ways (no arithmetical operations, just relations $<$, $>$ and $=$ and logical connectives $\wedge$, $\vee$, $\neg$). This could offer us the insight on *what the hyperplane means*, since we would know exactly how it behaves and manually tweak it. If we use machine learning for delicate matters (e.g. predicting failures for nuclear reactors), we want to be able to understand the *why*. This is the basis of *decision tree learning* [4], which is a very useful first model when tackling an unknown dataset.[2]

Hyperplane D seems great—it catches all Xs on one side and all Os on the other. Why not use that? Notice how it went out of its way to catch the middle O. We might worry about a hyperplane that provides a perfect fit to the existing data, since there is always some noise[3] in the data, and a new datapoint that falls here might happen to be an X. Think of it this way. If there was no O here, would you still justify the same loop? Probably no. If 25% of the overall Os were here, would that justify a loop like this? Probably yes. So, there seems to be a fuzzy limit of the number of Os we want to see to make such a loop justified. The point is that we want the classifier to be good for new instances, and a classifier that works in 100% of the old cases is probably learning noise along with the important and necessary information from the datapoints. Hyperplane C is a reasonable separation which is quite good and seems to be less concerned with precision than hyperplane C. It is not perfect, but it seems to be capturing a rather general trend in the data.

There is, however, a dose of simplicity in hyperplanes A, B and particularly E we would love to have. Let us see if we can make it happen. What if we use the features we have to create a new one? We have seen we could add a new one like height, but could we just try to build something with what we have? Let us try to plot on the axis z a new feature $\frac{length}{weight}$ (Fig. 3.3, top view). Now, we see that we can

[2] A *dataset* is simply a set of datapoints, some labelled some unlabelled.

[3] Noise is just a name for the random oscillations that are present in the data. They are imperfections that happen and we do not want to learn to predict noise but the elements that are actually relevant to what we want.

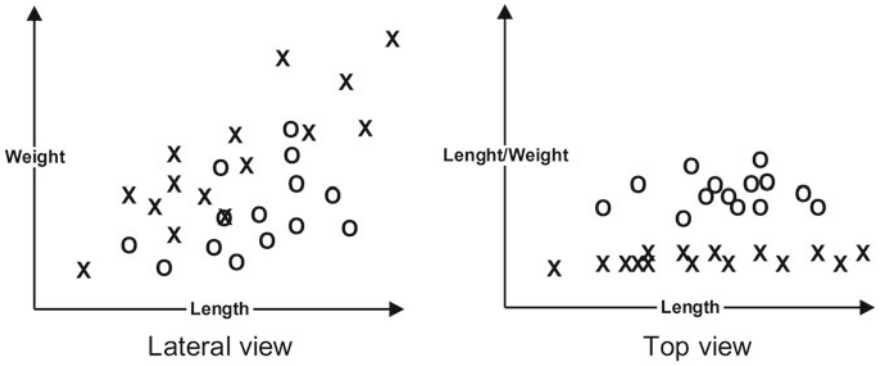


Fig. 3.3 Feature engineering

actually separate the two classes by a simple straight plane in 3D. When it is possible to separate[4] two classes in an n-dimensional space with a 'straight' hyperplane, we say that the classes are *linearly separable*. Usually, one can find a feature which is then added as a new dimension which makes two classes (almost) linearly separable. We can manually add features in which case it is called *feature engineering*, but we would like our algorithms to do it automatically. Machine learning algorithms work by exploiting this idea and they automate the process: they have a linear separator and then they try to find features such that when they are added the classes become linearly separable. Deep learning is no exception, and it is one of most powerful ways to find features automatically. Even though later deep learning will do this for us, to understand deep learning it is important to understand the manual process.

So far we have explored features that are numerical, like height, weight and length. They are specific in two ways. First, order matters: 1 is before 3, 3 is before 14 and we can derive that 1 is before 14. The use of 'before' instead of 'less than' is deliberate. The second thing is that we can add and multiply them. A different kind of feature is an *ordinal feature*. Here, we have the first property of the numerical features 'before' but not the second. Think of the ending positions in a race: the fact that someone is second, someone is third and someone is fourth does not mean that the distance between the second and third is the same as between third and fourth, but the order still holds (second comes before third, and third comes before fourth). If we do not have that either, we are using *categorical features*. Here, we have just the names of the categories and nothing can be inferred from them. An example would be the dog's colour. There are no 'middles' or orders in them, just categories.

Categorical features are very common. Machine learning algorithms cannot accept categorical features as they are and they must be converted. We take the initial table with the categorical feature 'Colour':

[4]It does not have to a perfect separation, a good separation will do.

Length	Weight	Colour	Label
34	7	Black	Dog
59	15	White	Dog
54	17	Brown	Dog
78	28	White	Dog
…	…	…	…

And convert it so that we expand the columns with the initial category names and allow only binary values in those columns which indicate which one of the colours the given dog has. This is called *one-hot encoding*, and it increases the dimensionality[5] of the data but now a machine learning algorithm[6] can process the categorical data. The modified table[7] is

Length	Weight	Brown	Black	White	Label
34	7	0	1	0	Dog
59	15	0	0	1	Dog
54	17	1	0	0	Dog
78	28	0	0	1	Dog
…	…	…	…	…	…

We conclude this section by giving a brief description of all supervised machine learning algorithms in terms of input and output. Every *supervised* machine learning algorithm receives a set of training datapoints and labels (they are row vectors). In this phase, the algorithm creates a hyperplane by adjusting its internal parameters. This phase is called the training phase: it receives as inputs row vectors with corresponding labels (called *training samples*) and does not give any output. Instead, in the training phase, the algorithm simply adjusts its internal parameters (and by doing so creates the hyperplane). The next phase is called the predicting phase. In this phase, the trained algorithm takes in a number of row vectors but this time without labels and creates the labels with the hyperplane (depending on which side of the hyperplane the row vectors end up). The row vectors themselves are simply rows from a table like the one above, so the row vector which corresponds to the training sample in the third line is simply (54, 17, 1, 0, 0, *Dog*). If it were a row vector for which we need to predict a label, it would look the same except it would not have the 'Dog' tag in the end.[8]

[5] Think about how one-hot encoding can boost the understanding of n-dimensional space.

[6] Deep learning is no exception.

[7] Notice that to do one-hot encoding, it needs to make two passes over the data: the first collects the names of the new columns, then we create the columns, and then we make another pass over the data to fill them.

[8] Strictly speaking, these vectors would not look exactly the same: the training sample would be (54,17,1,0,0, Dog), which is a row vector of length 6, and the row vector for which we want to

3.2 Evaluating Classification Results

In the previous section, we have explored the basics of classification and we left the hard part (producing the hyperplane) largely untouched. We will address this in the next section. In this section, we will assume we have a working classifier and we want to see how well it behaves. Take a look at Fig. 3.4.

This image illustrates a classifier named C for classifying Xs. This is the task for this classifier and it is important to keep this in mind at all times. The black line is the hyperplane, and the grey region is what C considers to be the region of X. From the perspective of C, everything inside the grey region *is* X, while everything outside is not an X. We have marked the individual datapoints with X or O depending whether they are in reality an X or O. We can see right away that the reality differs from what C thinks and this is the usual scenario when we have and empirical classification task. Intuitively, we see that the hyperplane makes sense, but we want to define objective classification metrics which can tell us how good a classifier is and, if we have two or more, which classifier is the best.

We can now define the concepts of *true positive*, *false positive*, *true negative* and *false negative*. A true positive is a datapoint for which the classifier says it is an X and it truly is an X. A false positive is a datapoint for which the classifier thinks it is an X but it is an O. A true negative is a datapoint for which the classifier thinks it is *not* and X and in fact it is not, and a false negative is a datapoint for which the classifier thinks it is not an X but in fact it is. In Fig. 3.4, there are five true positives (Xs in the grey), one false positive (the O in the grey), six true negatives (the Os in the white) and two false negatives (the Xs in the white). Remember, the grey area is the area where the classifier C thinks all are Xs and the white area is what the classifier thinks all are Os.

The first and most fundamental classification metric is *accuracy*. Accuracy simply tells us how good is the classifier at sorting Xs and Os. In other words, it is the

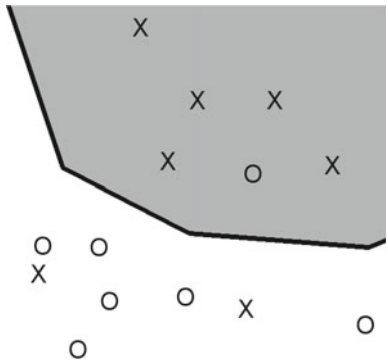


Fig. 3.4 A classifier C for classifying Xs

predict the label would have to be of length 5 (without the last component which is the label), e.g. (47,15,0,0,1).

number of true positives, added to the number of true negatives and divided by the total number of datapoints. In our case, this would be $\frac{5+6}{14} = 0.785714\ldots$ but we will be rounding off to four decimal points.[9]

We might be interested in how good is a classifier at avoiding false alarms. The metric used to calculate this is called *precision*. The precision of a classifier on a dataset is calculated by $\frac{truePositives}{truePositives+falsePositives} = \frac{5}{5+1} = 0.8333$. If we are concerned about missing out and we want to catch as many true Xs we can, we need a different metric called *recall* to measure our success. The recall is calculated by taking $\frac{truePositives}{truePositives+falseNegatives} = \frac{5}{5+2} = 0.7142$.

There is a standard way to display the number of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) in a more visual way and this method is called a *confusion matrix*. For a two-class classification (also known as *binary classification*), the confusion matrix is a 2×2 table of the form:

	Classifier says YES	Classifier says NO
In reality YES	Number of true positives	Number of false negatives
In reality NO	Number of false negatives	Number of true negatives

Once we have a confusion matrix, precision, recall, accuracy and any other evaluation, metric can be calculated directly from it.

The values for all classifier evaluation metrics range from 0 to 1 and can be interpreted as probabilities. Note that there are trivial modifications that can make either the precision or recall reach 100% (but not both at the same time). If we want the precision to be 1, we can simply make a classifier that selects no datapoint, i.e. for each datapoint it should say 'O'. The opposite works for recall: just select all datapoints as Xs, and recall will be 1. This is why we need all three metrics to get a meaningful insight on how good a classifier is and how to compare two classifiers.

Now that we know about evaluation metrics, let us turn to the question of evaluating the classifier performance from a procedural point of view. When faced with a classification task, as noted earlier we have a classification algorithm and a training set. We train the algorithm on the training set and now we are ready to use it for prediction. But where is the evaluation part? The usual strategy is not to use the whole training set for training, but keep a part of it for testing. This is usually 10%, but it can be more or less than that.[10] The 10% we held out and did not train on it is called the *test set*. In the test set, we separate the labels from the other features, so that we have row vectors of the same form we would be getting when predicting. When we have a trained model on the 90% (the training set), we use it to classify the test set, and we compare the classification results with the labels. In this way, we get the necessary information for calculating the precision, recall, and accuracy. This is

[9] If we will be needing more we will keep more decimals, but in this book we will usually round off to four.

[10] It is mostly a matter of choice, there is no objective way of determining how much to split.

called *splitting the dataset in training and testing sets* or simply the *train–test split*. The test set is designed to be a controlled simulation of how well will the classifier behave. This approach is sometimes called *out-of-sample* validation to distinguish it from *out-of-time* validation where the 10% of the data are not chosen randomly from all datapoints, but a time period spanning around 10% of the datapoints is chosen. Out-of-time validation is generally not recommended since there might be seasonal trends in the data which would seriously cripple the evaluation.

3.3 A Simple Classifier: Naive Bayes

In this section, we sketch the simplest classifier we will explore in this book, called the *naive Bayes classifier*. The naive Bayes classifier has been used from at least 1961 [5], but, due to its simplicity, it is hard to pinpoint where research on the applications of Bayes' theorem ends and the research on the naive Bayes classifier begins.

The naive Bayes classifier is based on Bayes' theorem which we saw earlier in Chap. 2 (this accounts for the 'Bayes' in the name), and it makes and additional assumption that all features are conditionally independent from each other (this is why there is 'Naive' in the name). This means that each feature carries 'its own weight' in terms of predictive power: there is no piggy-backing or synergy of features going on. We will rename the variables in the Bayes theorem to give it a more 'machine learning feel':

$$\mathbb{P}(t|f) = \frac{\mathbb{P}(f|t)\mathbb{P}(t)}{\mathbb{P}(f)},$$

where $\mathbb{P}(t)$ is the prior probability[11] of a given target value (i.e. the class label), $\mathbb{P}(f)$ is the prior probability of a feature, $\mathbb{P}(f|t)$ is the probability of the feature f given the target t, and, of course, $\mathbb{P}(t|f)$ is the probability of the target t given only the feature f which is what we want to find.

Recall from Chap. 2 that we can convert Bayes' theorem to accommodate for a (n-dimensional) vector of features, and in that case we have the following formula:

$$\mathbb{P}(t|f_{all}) = \frac{\mathbb{P}(f_1|t) \cdot \mathbb{P}(f_2|t) \cdot \ldots \cdot \mathbb{P}(f_n|t) \cdot \mathbb{P}(t)}{\mathbb{P}(f_{all})}$$

Let us see a very simple example to demonstrate how the naive Bayes classifier works and how it draws its hyperplane. Imagine that we have the following table detailing visits to a webpage:

We first need to convert this into a table with counts (called a *frequency table*, similar to one-hot, but not exactly the same):

Now, we can calculate some basic prior probabilities. The probability of 'yes' is $\frac{9}{13} = 0.6923$. The probability of 'no' is $\frac{4}{13} = 0.3076$. The probability of 'morning'

[11] The prior probability is just a matter of counting. If you have a dataset with 20 datapoints and in some feature there are five values of 'New Vegas' while the others (15 of them) are 'Core region',

Time	Buy
morning	no
afternoon	yes
evening	yes
morning	yes
morning	yes
afternoon	yes
evening	no
evening	yes
morning	no
afternoon	no
afternoon	yes
afternoon	yes
morning	yes

Time	yes	no	TOTAL
morning	3	2	5
afternoon	4	1	5
evening	2	1	3
TOTAL	9	4	13

is $\frac{5}{13} = 0.3846$. The probability of 'afternoon' is $\frac{5}{13} = 0.3846$. The probability of 'evening' is $\frac{3}{13} = 0.2307$. Ok, that takes care of all the probabilities which we can calculate just by counting from the dataset (the so-called 'priors' we addressed in Sect. 2.3 of Chap. 2). We will be needing one more thing but we will get to it.

Imagine now we are given a new case for which we do not know the target label and we must predict it. This new case is the row vector $(morning)$[12] and we want to know whether it is a 'yes' or a 'no', so we need to calculate

$$\mathbb{P}(yes|morning) = \frac{\mathbb{P}(morning|yes)\mathbb{P}(yes)}{\mathbb{P}(morning)}$$

We can plug in the priors $\mathbb{P}(yes) = 0.6923$ and $\mathbb{P}(morning) = 0.3846$ we calculated above. Now, we only need to calculate $\mathbb{P}(morning|yes)$, which is the percentage of times the 'morning' occurs if we restrict ourselves to the rows which have 'yes', which is present 9 times, and out of these, three have also a 'yes', so we have $\mathbb{P}(morning|yes) = \frac{3}{9} = 0.3333$. Taking it all to Bayes' theorem, we have

$$\mathbb{P}(yes|morning) = \frac{\mathbb{P}(morning|yes) \cdot \mathbb{P}(yes)}{\mathbb{P}(morning)} = \frac{0.3333 \cdot 0.6923}{0.3846} = 0.5999$$

[12] If we were to have n features, this would be an n-dimensional row vector such as $(x_1, x_2, \ldots, x_n)$, but now we have only one feature so we have a 1D row vector of the form (x_1). A 1D vector is *exactly the same* as the scalar x_1 but we keep referring to it as a vector to delineate that in the general case it would be an n-dimensional *vector*.

We also know that $\mathbb{P}(no|morning) = 1 - \mathbb{P}(yes|morning) = 0.4$. This means that the datapoint gets the label 'yes', since the value is over 0.5 (we have two classes). In general, if we were to have n classes, $\frac{1}{n}$ is the value over which the probability would have to be.

The diligent reader could say that we could have calculated $\mathbb{P}(yes|morning)$ directly from the table as we did with $\mathbb{P}(morning|yes)$, and this is true. The problem is that we can do it by counting from the table only if there is a single feature, so for the case of multiple features we would have to use calculation we actually used (with the expanded formula for multiple features).

Naive Bayes is a simple algorithm, but it is still very useful for large datasets. In fact, if we adopt a probabilistic view of machine learning and claim that all machine learning algorithms actually learn only $\mathbb{P}(y|\mathbf{x})$, we could say that naive Bayes is *the* simplest machine learning algorithm, since it has only the bare necessities to make the 'flip' from $\mathbb{P}(f|t)$ to $\mathbb{P}(t|f)$ work (from counting to predicting). This is a specific (probabilistic) view of machine learning, but it is compatible with the deep learning mindset, so feel free to adopt it as a pet.

One important thing to remember is that naive Bayes makes the conditional independence assumption.[13] So it cannot handle any dependencies in the features. Sometimes, we might want to be able to model sequences like this, e.g. when the order of the feature matters (we will see this come into play for language modelling or for sequences of events in time), and naive Bayes is unable to do this. Later in the book, we will present deep learning models fully capable of handling this. Before continuing on, notice that the naive Bayes classifier had to draw a hyperplane to be able to classify the new datapoints. Suppose we had a binary classification at hand. Then, naive Bayes expanded the space by one dimension (so the row vectors are augmented to include this value), and that dimension accepts values between 0 and 1. In this dimension, the hyperplane is visible and it passes through the value 0.5.

3.4 A Simple Neural Network: Logistic Regression

Supervised learning is usually divided into two types of learning. The first one is classification, where we have to predict the class. We have seen that already with naive Bayes, we will see it again countless times in this book. The second one is regression where we predict a value, and we will not be exploring regression in this book.[14] In this section, we explore *logistic regression* which is *not* a regression algorithm but a classification algorithm. The reason behind this is that it is considered a regression model in statistics and the machine learning community just adopted it and began using it as a classifier.

[13] That is, the assumption that features are conditionally independent given the target.

[14] Regression problems can be simulated with classification. An example would be if we had to find the proper value between 0 and 1, and we had to round it in two decimals, then we could treat it as a 100-class classification problem. The opposite also holds, and we have actually seen this in the naive Bayes section, where we had to pick a threshold over which we would consider it a 1 and

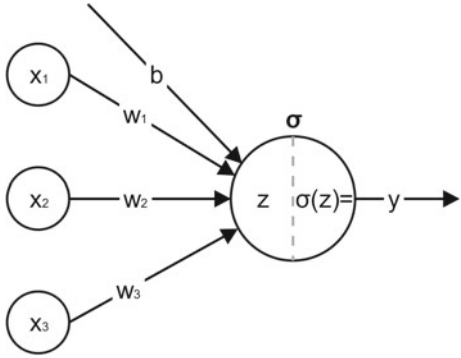


Fig. 3.5 Schematic view of logistic regression

Logistic regression was first introduced in 1958 by D. R. Cox [6], and a considerable amount of research was done both on logistic regression and using logistic regression. Logistic regression is mainly used today for two reasons. First, it gives an interpretation of the relative importance of features, which is nice to have if we wish to build an intuition on a given dataset.[15] The second reason, which is much more important to us, is that the logistic regression is actually a one-neuron neural network.[16]

By understanding logistic regression, we are taking a first and important step towards neural networks and deep learning. Since logistic regression is a supervised learning algorithm, we will have to have the target values for training included in the row vectors for the training set. Imagine that we have three training cases, $\mathbf{x}_A = (0.2, 0.5, 1, 1)$, $\mathbf{x}_B = (0.4, 0.01, 0.5, 0)$ and $\mathbf{x}_C = (0.3, 1.1, 0.8, 0)$. Logistic regression has a much input neurons as it has features in the row vectors,[17] which is in our case 3.

You can see a schematic representation of logistic regression in Fig. 3.5. As for the calculation part, the logistic regression can be divided into two equations:

$$z = b + w_1 x_1 + w_2 x_2 + w_3 x_3,$$

which calculates the *logit* (also known as the weighted sum) and the *logistic* or *sigmoid* function:

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

[15] Afterwards, we may do a bit of feature engineering and use an all-together different model. This is important when we do not have an understanding of the data we use which is often the case in industry.

[16] We will see later that logistic regression has more than one neuron, since each component of the input vector will have to have an input neuron, but it has 'one' neuron in the sense of having a single 'workhorse' neuron.

[17] If the training set consists of n-dimensional row vectors, then there are exactly $n - 1$ features—the last one is the target or label.

If we join them and tidy up a bit, we have simply

$$y = \sigma(b + w_1 x_1 + w_2 x_2 + w_3 x_3)$$

Now, let us comment on these equations. The first equation shows how to calculate the logit from the inputs. The inputs in deep learning are always denoted by x, the output of the neuron is always denoted by y and the logit is denoted by z or sometimes a. The equations above make use of all the notational abuse which is common in the machine learning community, so be sure to understand why the symbols are employed as they are.

To calculate the logit, we need (asides from the inputs) the weights w and the bias b. If you look at the equations, you will notice that everything except the bias and weights is either an input or calculated. The elements which are not given as inputs or are constants like e are called *parameters*. For now, the parameters are the weights and biases, and the point of logistic regression is to *learn* a good vector of weights and a good bias to achieve good classification. This is *the* only learning in logistic regression (and deep learning): finding a good set of weights.

But what are the weights and biases? The weights control how much of each feature from the input we should let in. You can think about them as if they represent percentages. They are not limited to the interval between 0 and 1, but this is a good intuition to have. For weights over 1, you could think of them as 'amplifications'. The bias is a bit more tricky. Historically,[18] it has been called threshold and it behaved a bit differently. The idea was that the logit would simply calculate the weighted sum of the inputs, and if it was above the threshold, the neuron would output a 1, otherwise a 0. The 1 and 0 part was replaced by our equation for $\sigma(z)$, which does not output a sharp 0 or 1, but instead it ranges from 0 to 1. You can see the different plots on Fig. 3.6. Later, in Chap. 4, we will see how to incorporate the bias as one of the weights. For now, it is enough to know that the bias can be absorbed as one of

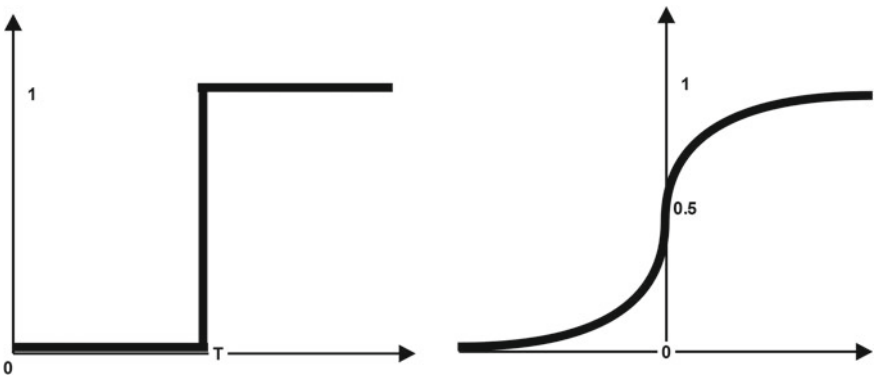


Fig. 3.6 Historic and actual neuron activation functions

[18] Mathematically, the bias is useful to make an offset called the intercept.

the weights so we can forget about the bias knowing it will be taken care of and it will become one of the weights.

Let us make a calculation based on our inputs which will explain the mechanics of logistic regression. We will need a starting value for the weights and bias, and we usually produce this at random. This is done from a gaussian random variable, but to keep things simple, we will generate a set of weights and bias by taking random values between 0 and 1. Now, we would need to pass the input row vectors through one-hot encoding and normalize them, but suppose they already have been one-hot encoded and normalized. So we have $\mathbf{x}_A = (0.2, 0.5, 0.91, 1)$, $\mathbf{x}_B = (0.4, 0.01, 0.5, 0)$ and $\mathbf{x}_C = (0.3, 1.1, 0.8, 0)$ and assume that the randomly generated weight vector is $\mathbf{w} = (0.1, 0.35, 0.7)$ and the bias is $b = 0.66$. Now we turn to our equations, and put in the first input:

$$y_A = \sigma(0.66 + 0.1 \cdot 0.2 + 0.35 \cdot 0.5 + 0.7 \cdot 0.91) = \sigma(1.492) = \frac{1}{1 + e^{-1.492}} = 0.8163$$

We note the result 0.8163 and the actual label 1. Now we do the same for the second input:

$$y_B = \sigma(0.66 + 0.1 \cdot 0.4 + 0.35 \cdot 0.01 + 0.7 \cdot 0.5) = \sigma(1.0535) = \frac{1}{1 + e^{-1.0535}} = 0.7414$$

Noting again the result 0.7414 and label 0. And now we do it for the last input row vector:

$$y_C = \sigma(0.66 + 0.1 \cdot 0.3 + 0.35 \cdot 1.1 + 0.7 \cdot 0.8) = \sigma(1.635) = \frac{1}{1 + e^{-1.635}} = 0.8368$$

Noting again the result 0.8368 and the label 0. It seems quite clear that we did good on the first, but failed to classify the second and third input correctly. Now, we should update the weights somehow, but to do that we need to calculate how lousy we were at classifying. For measuring this, we will be needing an *error function* and we will be using the *sum of squared error* or SSE[19]:

$$E = \frac{1}{2} \sum_n (t^{(n)} - y^{(n)})^2$$

The ts are targets or labels, and the ys are the actual outputs of the model. The weird exponents $(t^{(n)})$ are just indices which range across training samples, so $(t^{(k)})$ would be the target for the kth training row vector. You will see in a moment why

[19]There are other error functions that can be used, but the SSE is one of the simplest.

do we need such weird notation now and a bit later how to dispense with it. Let us calculate our SSE:

$$E = \frac{1}{2}\sum_n (t^{(n)} - y^{(n)})^2 = \tag{3.1}$$

$$= \frac{1}{2}((1 - 0.8163)^2 + (0 - 0.7414)^2 + (0 - 0.8368)^2) = \tag{3.2}$$

$$= \frac{0.0337 + 0.5496 + 0.7002}{2} = \tag{3.3}$$

$$= 0.64175 \tag{3.4}$$

We now update the $\mathbf{w}$ and b by using magic, and get $\mathbf{w} = (0.1, 0.36, 0.3)$ and $b = 0.25$. Later (in Chap. 4), we will see it is actually done by something called the *general weight update rule*. This completes one cycle of weight adjustment. This is colloquially called an *epoch*, but we will redefine this term later in Chap. 4 to make it more precise. Let us recalculate the outputs and the new SSE to see whether the new set of weights is better:

$$y_A^{new} = \sigma(0.25 + 0.1 \cdot 0.2 + 0.36 \cdot 0.5 + 0.3 \cdot 0.91) = \sigma(0.723) = \frac{1}{1 + e^{-0.723}} = 0.6732 \tag{3.5}$$

$$y_B^{new} = \sigma(0.25 + 0.1 \cdot 0.4 + 0.36 \cdot 0.01 + 0.3 \cdot 0.5) = \sigma(0.4436) = \frac{1}{1 + e^{-0.4436}} = 0.6091 \tag{3.6}$$

$$y_C^{new} = \sigma(0.25 + 0.1 \cdot 0.3 + 0.36 \cdot 1.1 + 0.3 \cdot 0.8) = \sigma(0.916) = \frac{1}{1 + e^{-1.635}} = 0.7142 \tag{3.7}$$

$$E^{new} = \frac{1}{2}((1 - 0.6732)^2 + (0 - 0.6091)^2 + (0 - 0.7142)^2) = \tag{3.8}$$

$$= \frac{0.1067 + 0.371 + 0.51}{2} = \tag{3.9}$$

$$= 0.4938 \tag{3.10}$$

We can see clearly that the overall error has decreased. We can continue this procedure a number of times, and the error will decrease, until at one point it will stop decreasing and stabilize. On rare occasions, it might even exhibit chaotic behaviour. This is the essence of logistic regression, and the very core of deep learning—everything we do will be an upgrade or modification of this.

Let us turn our attention to data representation. So far we have used an expanded view of the process so that we may see clearly everything, but let us see how we can make the procedure more compact and computationally faster. Notice that even though a dataset is a set (and the order does not matter), it might make a bit of sense to put $\mathbf{x}_A$, $\mathbf{x}_B$ and $\mathbf{x}_C$ in a vector, since we will be using them one by one (the vector would then simulate a queue or stack). But since they also share the same structures (same features in the same place in each row vector), we might opt for a matrix

to represent the whole training set. This is important in the computational sense as well since most deep learning libraries have somewhere in the background C, and arrays (the programming equivalent of matrices) are a native data structure in C, and computation on them is incredibly fast.

So what we want to do is first turn the n d-dimensional input vectors into and input matrix of the size $n \times d$. In our case, this is a 3×3 matrix:

$$\mathbf{x} = \begin{bmatrix} 0.2 & 0.5 & 0.91 \\ 0.4 & 0.01 & 0.5 \\ 0.3 & 1.1 & 0.8 \end{bmatrix}$$

We will be keeping the targets (labels) in a separate vector, and we have to be extremely careful not to shuffle neither the target vector nor the dataset matrix from this point onwards, since the order of the matrix rows and vector components is the only thing that can join them again. The target vector in our case is $\mathbf{t} = (1, 0, 0)$.

Let us turn our attention to the weights. The bias is a bit of a bother, so we can turn it in one of the weights. To do this, we have to add a single column of 1's as the first column of the input matrix. Notice that this will not be an approximation, but will capture *exactly* the calculation we need to perform. As for the weights, we will be needing as many weights as there are inputs. Also, if we have more than one workhorse neuron, we would need to have that many times the weights, e.g. if we have 5 inputs (5-dimensional input row vectors) and 3 workhorse neurons, we would need 5×3 weights. This 5×3 is deliberate, since we would be using a 5×3 matrix[20] to store it in, since then we could do all the calculations needed for the logit with a simple matrix multiplication. This illustrates something that could be called 'the general deep learning strategy for fast computation': try to do as much work as you can with matrix (and vector) multiplication and transpositions.

Returning to our example, we have three inputs and we add the column of 1's in front of the inputs to make room for the bias in the weight matrix. The new input matrix is now a 3×4 matrix:

$$\mathbf{x} = \begin{bmatrix} 1 & 0.2 & 0.5 & 0.91 \\ 1 & 0.4 & 0.01 & 0.5 \\ 1 & 0.3 & 1.1 & 0.8 \end{bmatrix}$$

Now we can define the weight matrix. It is a 4×1 matrix consisting of the bias followed by weight:

$$\mathbf{w} = \begin{bmatrix} 0.66 \\ 0.1 \\ 0.35 \\ 0.7 \end{bmatrix}$$

[20] Recall that this is not the same as a 3×5 matrix.

This matrix can be equivalently represented as $(0.66, 0.1, 0.35, 0.7)^\top$, but we will use the matrix form for now. Now, to calculate the logit we do simple matrix multiplication of the two matrices, with which we will get a 3×1 matrix in which every row (there is a single value in every row) will represent the logit for each training case (compare this with the previous calculation):

$$\mathbf{z} = \mathbf{xw} = \begin{bmatrix} 1 & 0.2 & 0.5 & 0.91 \\ 1 & 0.4 & 0.01 & 0.5 \\ 1 & 0.3 & 1.1 & 0.8 \end{bmatrix} \cdot \begin{bmatrix} 0.66 \\ 0.1 \\ 0.35 \\ 0.7 \end{bmatrix} = \tag{3.11}$$

$$= \begin{bmatrix} 1 \cdot 0.66 + 0.2 \cdot 0.1 + 0.5 \cdot 0.35 + 0.91 \cdot 0.7 \\ 1 \cdot 0.66 + 0.4 \cdot 0.1 + 0.01 \cdot 0.35 + 0.5 \cdot 0.7 \\ 1 \cdot 0.66 + 0.3 \cdot 0.1 + 1.1 \cdot 0.35 + 0.8 \cdot 0.7 \end{bmatrix} = \tag{3.12}$$

$$= \begin{bmatrix} 1.492 \\ 1.0535 \\ 1.635 \end{bmatrix} \tag{3.13}$$

Now we must only apply the logistic function σ to $\mathbf{z}$. This is done by simply applying the function to each element of the matrix:

$$\sigma(\mathbf{z}) = \begin{bmatrix} \sigma(1.492) \\ \sigma(1.0535) \\ \sigma(1.635) \end{bmatrix} = \begin{bmatrix} 0.8163 \\ 0.7414 \\ 0.8368 \end{bmatrix}$$

We add a final remark. The logistic function is the main component of the logistic regression. But if we treat the logistic regression as a simple neural network, we are not committed to the logistic function. In this view, the logistic function is a *nonlinearity*,[21] i.e. it is the component which enables complex behaviour (especially when we expand the model beyond a single workhorse neuron of the classic logistic regression). There are many types of nonlinearity, and they all have a slightly different behaviour. The logistic regression ranges between 0 and 1. Another common nonlinearity is the *hyperbolic tangent* or $tanh$, which we will denote by τ to enforce a bit of notational consistency. The τ nonlinearity ranges between -1 and 1, and has a similar shape like the logistic function. It is calculated by

$$\tau(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3.14}$$

The choice of which activation function to use in neural networks is a matter of preference, and it is often guided by the results one obtains using them. If, we use the hyperbolic tangent in logistic regression instead of the logistic function, it will still work nicely, but technically that is not logistic regression anymore. Neural networks, on the other hand, are still neural networks regardless of which nonlinearity we use.

[21] In the older literature, this is sometimes called *activation function*.

Fig. 3.7 A single MNIST datapoint

3.5 Introducing the MNIST Dataset

The MNIST dataset is a modification of the National Institute of Standards and Technology of the United States dataset consisting of handwritten digits. The original datasets are described in [7] and the MNIST (*modified NIST*) is a modification of the Special Database 1 and Special Database 3 of the original dataset compiled by Yann LeCun, Corinna Cortes and Christopher J. C. Burges. The MNIST dataset was first used in the paper [8]. Geoffrey Hinton called MNIST 'the fruit fly of machine learning'[22] since a lot of research in machine learning was performed on it and it is quite versatile for a number of simple tasks. Today, MNIST is available from a variety of sources, but the 'cleanest' source is probably Kaggle where the data is kept in a simple CSV file,[23] accessible by any software with ease. In Fig. 3.7 (image taken from [9]), we can see an example of a MNIST digit.

MNIST images are 28 by 28 pixels in greyscale, so the value for each pixel ranges between 0 (white) and 255 (black). This is different from the usual greyscale where 0 is black and 255 is white, but the community thought it might make more sense since it can be stored in less space this way, but this is a minor point today for a dataset of the size of MNIST.

There is one issue here, to which we will return at the very end of the book. The problem is that all currently available supervised machine learning algorithms can only process vectors as inputs: no matrices, graphs, trees, etc. This means that whatever we are trying to do, we have to find a way to put in vector form and transform *all* of our inputs in n-dimensional vectors. The MNIST dataset consists of 28 by 28 images, so, in essence, the inputs are matrices. Since they are all of the same size, we can transform them in 784-dimensional vectors.[24] We could do this by simply 'reading' them as we would a written page: left to right, after the row of pixels ends, move to the leftmost part of the next line and continue again. By doing this, we have transformed a 28×28 matrix into a 784-dimensional vector. This is a rather simple transformation (note that it only works if all input samples are of the same size), and if we want to learn graphs and trees, we have to have a vector representation of them. We will return to this as an open problem at the very end of this book.

There is one additional point we want to make here. MNIST consists of greyscale images. What could we do if it was RGB? Recall that an RGB image consists of three

[22] See http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec1.pdf.
[23] Available at https://www.kaggle.com/c/digit-recognizer/data.
[24] The interested reader may look up the details in Chap. 4 of [10].

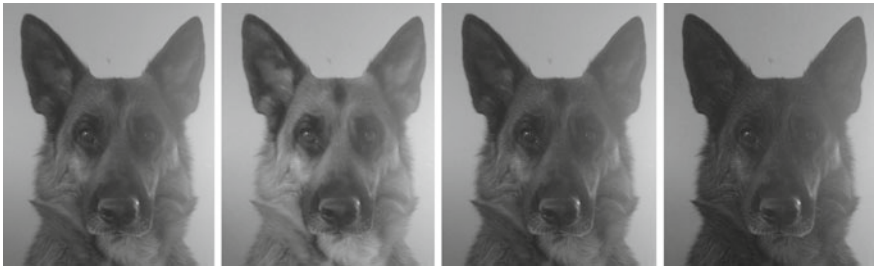


Fig. 3.8 Greyscale for all colours, red channel, green channel and blue channel

component 'images' called *channels*: red, green and blue. They are joined to form the complete (colour) image. We could print these in colour (each pixel of the red channel would have a value from 0 to 255 to denote how much red is in it), but we have actually converted the colours to greyscale without noticing (see Fig. 3.8). It might seem weird to represent the red channel as grey, but that is *exactly* what a computer does. The *name* of the channel image is 'red' but the values in pixels are between 0 and 255, which is, *computationally speaking*, grey. This is because an RGB pixel is simply three values from 0 to 255. The first one is *called* 'red', but computationally it is red just because it is in the first place. There is no intrinsic 'redness' or qualia in it. If we were to display the pixels without providing the other two components, 0 will be interpreted as black and 255 as white, making it a greyscale. In other words, an RGB image would have a pixel with the value (34, 67, 234), but if we separate a channel by taking only the red component 34 we would get a greyscale. To get the 'redness' in the display, we must state it as (34, 0, 0) and keep it as an RGB image. And the same goes for green and blue. Returning to our initial question, if we were processing RGB images would have several options:

- Average the components to produce and average greyscale representation (this is the usual way to create greyscale images from RGB).
- Separate the channels and form three different datasets and train three classifiers. When predicting, we take the average of their result as the final result. This is an example of a *committee* of classifiers.
- Separate the channels in distinct images, shuffle them and train a single classifier on all of them. This approach would be essentially *dataset augmentation*.
- Separate the channels in distinct images, train three instances of the same classifier on each (same size and parameters), and then use a fourth classifier to make the final call. This is the approach that leads to *convolutional neural networks* which we will explore in detail in Chap. 6.

Each of these approaches has its merit, and depending on the problem at hand, any of them can be a good choice. You can consider other options, deep learning has an exploratory element to it, and an unorthodox method which contributes to accuracy will be appreciated.

3.6 Learning Without Labels: K-Means

We now turn our attention to two algorithms for unsupervised learning, the *K-means* and the *PCA*. We will briefly address PCA in the next section (especially the intuition behind it), but we will be returning to it in Chap. 9 where we will be giving the technical details. PCA represents a branch of unsupervised learning called *distributed representations*, and it is one of the most important topics in deep learning today, and PCA is the simplest algorithm for building distributed representations.[25] Another branch of unsupervised learning which is conceptually simpler is called *clustering*. The goal of clustering is to assign all datapoints to clusters which (hopefully) capture their similarity in n-dimensional space. K-means is the simplest clustering algorithm, and we will use it to illustrate how a clustering algorithm works.[26]

But before we proceed to K-means, let us comment briefly what is unsupervised learning. *Unsupervised learning is learning without labels or targets*. Since unsupervised learning is usually the last of the three areas to be defined (supervised and reinforcement being the other two), there is a tendency to put everything which is not supervised or reinforcement learning in unsupervised learning. This is a very broad definition, but it is very interesting, since it begs the cognitive question of how we learn without feedback, and is learning without feedback actually learning or is it a different phenomenon? By exploring unsupervised learning, we are dwelling deep in cognitive modelling and this makes this an exciting and colourful area.

Let us demonstrate how K-means works. K-means is a clustering algorithm, which means it will produce clusters of data. Producing clusters actually means assigning a cluster name to all datapoints so that similar datapoints share the same cluster name. The usual cluster names are '1', '2', '3', etc. Assume we have two features so that we work in 2D space. In unsupervised learning, we do not have a training and testing set, but all datapoints we have are 'training' datapoints, and we build the clusters (which will define the hyperplane) from them. The input row vectors do not have a label; they consist only of features.

The K-means algorithm takes as an input the number of centroids to be used. Each centroid will define a cluster. At the very start of the algorithm, the centroids are placed in a random location in the datapoint vector space. K-means has two phases, one called 'assign' and the another 'minimize' forming a cycle, and it repeats this cycle a number of times.[27] During the assign phase, each datapoint is assigned to the nearest centroid in terms of Euclidean distance. During the 'minimize' phase, centroids are moved in a direction that minimizes the sum of the distance of all datapoints assigned to it.[28] This completes a cycle. The next cycle begins by dis-

[25] But PCA itself is not that simple to understand.

[26] K-means (also called the Lloyd-Forgy algorithm) was first proposed by independently by S. P. Lloyd in [16] and E. W. Forgy in [17].

[27] Usually, in a predefined number of times, there are other tactics as well.

[28] Imagine that a centroid is pinned down and connected to all its datapoints with rubber bands, and then you unpin it from the surface. It will move so that the rubber bands are less tense in total (even though individual rubber bands may become *more* tense).

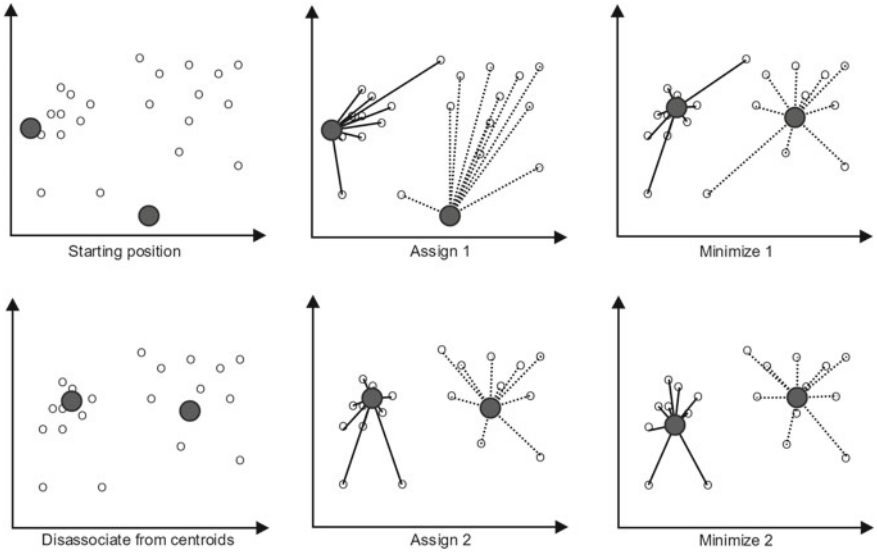


Fig. 3.9 Two complete cycles of K-means with two centroids

associating all datapoints from centroids. Centroids stay where they are, but a new assignment phase begins, which may make a different assignment than the previous one. You can see this in Fig. 3.9. After the end of the cycles, we have a hyperplane ready: when we get a new datapoint, it will be assigned to the closest centroid. In other words, it will get the name of the closest centroid as a label.

In the usual setting, we do not have labels when using clustering (and we do not need them for unsupervised learning). The evaluation metrics we discussed in the previous sections are useless without labels since we cannot calculate the true positives, false positives, true negatives and false negatives. It can happen that we have access to labels but prefer to use clustering, or that we will obtain the true labels at a later time. In such case, we may evaluate the results of clustering as if they were classification results, and this is called *external evaluation of clustering*. A detailed exposition of using classification evaluation metrics for the external evaluation of clustering is given in [11].

But sometimes we do not have any labels and we must work without them. In such cases, we can use a class of evaluation metrics called *internal evaluation of clustering*. There are several evaluation metrics, but the Dunn coefficient [12] is the most popular. The main idea is to measure how dense the clusters are in n-dimensional space. So for each cluster[29] C the Dunn coefficient is calculated by

$$D_C = \frac{min\{d(i,j)|i,j \in Centroids\}}{d^{in}(C)} \tag{3.15}$$

[29]Recall that a cluster in K-means is a region around a centroid separated by the hyperplane.

Here, $d(i, j)$ is the Euclidean distance between centroids i and j and $d^{in}(C)$ is the intra-cluster distance which is taken to be the distance:

$$d^{in}(C) = max\{d(x, y)|x, y \in C\}, \tag{3.16}$$

where C is the cluster for which we calculate the Dunn coefficient. The Dunn coefficient is calculated for each cluster and the quality of each cluster can be assessed by it. The Dunn coefficient can be used to evaluate different clusterings by taking the average of the Dunn coefficients for each cluster in both clusterings[30] and then comparing them.

3.7 Learning Different Representations: PCA

The data we used so far has *local representations*. If the value of a feature named 'Height' is 180, then that piece of information about that datapoint (we could even say 'that property of the entity') exists only there. A different column 'Weight' contains no information on height. Such representations of the properties of the entities that we are describing as features of a datapoint are called *local representations*. Notice that the fact that the object has some height does put a constraint on weight. This is not a hard constraint but more of an 'epistemic shortcut': if we know that the person is 180 cm tall, then they will probably have around 80 kg. Individual persons may vary, but in general we could make a relatively decent guess of the person's weight just by knowing their height. This phenomenon is called *correlation* and it is a tricky phenomenon. If two features are highly correlated, they are very hard to tell apart. Ideally, we would want to find a transformation of the data which has weird features, but that are not correlated. In this representation, we would have a feature 'Argh' which captures the underlying component[31] by which we were able to deduce the weight from the height, and leave 'Haght' and 'Waght' as the part which is left in 'Height' and 'Weight' after 'Argh' was removed from them. Such representations are called *distributed representations*.

Building distributed representations by hand is hard, and yet this is the essence of what artificial neural networks do. Every layer builds its own distributed representation and this facilitates learning (this is perhaps the very essence of deep learning—learning many layers of distributed representations). We will show the simplest method of building a meaningful distributed representation, but we will write all the mathematical details of it only in Chap. 9. It is quite hard, and this is why we want deep learning to build such things for us. This method of building distributed representations is called the *principal component analysis* or PCA for short. In this chapter, we will provide a bird's-eye view of PCA and we will give all

[30] We have to use the same number of centroids in both clusterings for this to work.
[31] These features are known as *latent variables* in statistics.

the details in Chap. 9.[32] PCA has the following form:

$$Z = XQ, \tag{3.17}$$

where X is the input matrix, Z is the transformed matrix and Q is the 'tool-matrix' with which we do the transformation. If X is an $n \times d$ matrix, Z should also be $n \times d$. This gives us our first information about Q: it has to be a $d \times d$ matrix for the multiplication to work. We will show how to find the appropriate Q in Chap. 9. In the remainder of this section, we will introduce the intuition behind PCA as a whole and some of the elements needed to build Q. We will also describe in detail what do we want PCA to do and for what we want to be able to use it.

In general terms, PCA is used to preprocess the data. This means that it has to transform the data before the data is fed in a classifier, to make it more digestible. PCA is helpful for preprocessing in a couple of ways. We have seen above that we will use it to build distributed representations of data to eliminate correlation. We will also be able to use PCA for dimensionality reduction. We have seen how dimensions can expand with one-hot encoding and manual feature engineering. When we make distributed representations with artificial features such as 'Argh', 'Haght' and 'Waght', we would like to be able to order them in terms of informativity, so that we can discard the uninformative features. Informativity is just variance[33]: if a feature varies more, it carries more information.[34] This is something we want our Z to be like: the feature that has the most variance should be in the first column, the one with the second most variance in the second column of Z and so on.

To illustrate how the variance can change with simple transformations, see Fig. 3.10 where we have a simple case of six 2D datapoints. The part A of Fig. 3.10 illustrates the starting position. Note that the variance along the x coordinate is relatively small: the projections of the datapoints on the x axis are tightly packed together. The variance along the y axis is better, and the y coordinates are further apart. But we can do even better. Take a look at the part B of Fig. 3.10: we have obtained this by rotating the coordinate system a bit. Notice that all data stays the same and we are changing our representation of the data, i.e. the axes (which correspond to features). The new 'coordinate system' is actually, mathematically speaking, just a different basis for the points in this 2D vector space. You are not changing the points (i.e. 2D vectors), but the 'coordinate system' they live in. You are actually not even changing the coordinate system, but simply the basis of the vector space. The question of how to do this mathematically is actually the same as asking how to find a matrix Q such that it behaves in this way, and we will answer this in Chap. 9. Along the axes, we have plotted the distance between the first and last datapoint coordinates, which may be seen as a 'graphical proxy' for variance. In the B part of

[32] One of the reasons for this is that we have not yet developed all the tools we need to write out the details now.

[33] See Chap. 2.

[34] And if a feature is always the same, it has a variance of 0 and it carries no information useful for drawing the hyperplane.

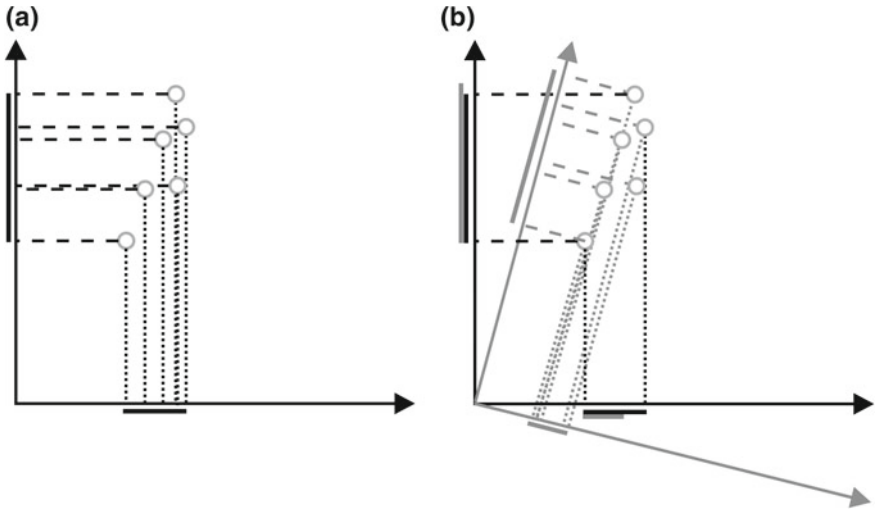


Fig. 3.10 Variance under rotation of the coordinate system

Fig. 3.10, we have compared the black (original coordinate system) with the grey (transformed) variance side-by-side (next to the black coordinate system). Notice that the variance along the y axis (the axis which had more variance in the original system) has increased, while the variance on the x axis (the axis which had less variance in the original system) has actually decreased.

Before continuing, let us make a final remark about PCA and preprocessing. One of the most fundamental problems with any kind of data is that it is noisy. Noise can be defined as everything except relevant information. If our dataset has enough training samples, then it should have non-random information and random noise. They are usually mixed up in features. But if we can build a distributed representation, this means we can extract as separate features the parts which have more variance and part which have less variance; we could assume that noise (which is random) has low variance (it is 'equally random' everywhere), whereas information has high variance. Suppose we have used PCA on a 20-dimensional input matrix. Then, we can keep the first 10 new features and by doing so we have eliminated a lot of noise (low variance features) by eliminating only a little bit of information (since they are low variance features—not 'no variance' features).

PCA has been around a long time. It was first discovered by Karl Pearson of the University College London [13] in 1901. Since then variants of the PCA went by many names, and often there were subtle differences. The details of the relations between various variants of the PCA are interesting, but unfortunately they would require a whole book to explore, and consequently are beyond the scope of this volume.

3.8 Learning Language: The Bag of Words Representation

So far we have addressed numerical features, ordinal features and categorical features. We have seen how to do one-hot encoding for categorical features. We have not addressed a whole field, namely natural language processing. We refer the reader to [14] or [15] for a thorough introduction to natural language processing. In this section, we will see how to process language by using one of the simplest models, the *bag of words*.

Let us first define a couple of terms for natural language processing. A *corpus* is a whole collection of texts we have. A corpus can be decomposed into *fragments*. Fragments can be single sentences, paragraphs or multi-page documents. Basically, a fragment is something we wish to treat as a training sample. If we are analysing clinical documents, each patient admission document might be one fragment; if we are analysing all PhD theses from a major university, each 200-page thesis is one fragment; if we are analysing sentiment on social media, each user comment is one fragment; and so on. A bag of words model is made by turning each word from the *corpus* in a feature and in each row, under that word, counting how many times the word occurs in that *fragment*. Clearly, the order of the words is lost by creating a bag of words.

The bag of words model is one of the main ways to convert language in features to be fed to a machine learning algorithm, and only deep learning has good alternatives to it as we shall see in Chaps. 6, 7 and 8. Other machine learning methods use the bag of words or variations[35] almost exclusively, and for many language processing tasks, the bag of words is a great language model even in deep learning. Let us see how the bag of words works in a simple social media dataset[36]:

User	Comment	Likes
S. A	you dont know	22
F. F	as if you know	13
S. A	i know what i know	9
P. H	i know	43

We need to convert the column 'Comment' into a bag of words. The columns 'User' and 'Likes' are left as they are for now. To create a bag of words from the comments, we need to make two passes. The first just collects all the words that occur and turns them into features (i.e. collects the unique words and creates the columns from them) and the second writes in the actual values:

[35] An example of an expansion of the basic bag of words model is a bag of n-grams. An n-gram is a n-tuple consisting of n words that occur next to each other. If we have a sentence 'I will go now', the set of its 2-grams will be $\{(`I', `will'), (`will', `go'), (`go', `now')\}$.

[36] For most language processing tasks, especially tasks requiring the use of data collected from social media, it makes sense to convert all text to lowercase first and get rid of all commas apostrophes and non-alphanumerics, which we have already done here.

User	you	dont	know	as	if	i	what	Likes
S. A	1	1	1	0	0	0	0	22
F. F	1	0	1	1	1	0	0	13
S. A	0	0	2	0	0	2	1	9
P. H	0	0	1	0	0	1	0	43

Now, we have the bag of words of the column 'Comment' and we need to do one-hot encoding on the column 'User' before being able to feed the dataset in a machine learning algorithm. We do this as we have explained earlier and get the final input matrix:

S. A	F. F	P. H	you	dont	know	as	if	i	what	Likes
1	0	0	1	1	1	0	0	0	0	22
0	1	0	1	0	1	1	1	0	0	13
1	0	0	0	0	2	0	0	2	1	9
0	0	1	0	0	1	0	0	1	0	43

This example shows the difference between one-hot encoding and the bag of words. In one-hot, each row has only 1 or 0 and, moreover, it must have exactly one 1. This means that it can be represented rather compactly by noting only the column number where it is 1. Take the fourth example in the upper column: we know everything for the one-hot part by simply noting '3' as the column number, which takes less space than writing '0,0,1'. The bag of words is different. Here, we take the word count for each fragment, which can be more than 1. Also, we need to use the bag of words on the entire dataset which means that we have to encode the training and test set together. This means that words that occur only in the test set will have 0 in the whole training set. Also, note that since most classifiers require that all samples have the same dimensionality (and feature names), when we will use the algorithm to predict, we will have to toss away any new word which is not in the trained model to be able to feed the data to the algorithm.

What they both have in common is that they expand the dimensions considerably and almost everywhere they will have the value 0. When we encode data like this we say, we have a *sparse encoding*. This means that a lot of features will be meaningless and that we want our classifier to dismiss them as soon as possible. We will see later how some techniques like PCA and L_1 regularization can be useful when confronted with a dataset which is sparsely encoded. Also, notice how we use the expansions of dimensions of the space to try to capture 'semantics' by counting words.

References

1. R. Tibshirani, T. Hastie, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd edn. (Springer, New York, 2016)
2. F. van Harmelen, V. Lifschitz, B. Porter, *Handbook of Knowledge Representation* (Elsevier Science, New York, 2008)
3. R.S. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, Cambridge, 1998)
4. J.R. Quinlan, Induction of decision trees. Mach. Learn. **1**, 81–106 (1986)
5. M.E. Maron, Automatic indexing: an experimental inquiry. J. ACM **8**(3), 404–417 (1961)
6. D.R. Cox, The regression analysis of binary sequences (with discussion). J. Roy. Stat. Soc. B (Methodol.) **20**(2), 215–242 (1958)
7. P.J. Grother, NIST special database 19: handprinted forms and characters database (1995)
8. Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
9. M.A. Nielsen, *Neural Networks and Deep Learning* (Determination Press, 2015)
10. P.N. Klein, *Coding the Matrix* (Newtonian Press, London, 2013)
11. I. Färber, S. Günnemann, H.P. Kriegel, P. Kroöger, E. Müller, E. Schubert, T. Seidl, A. Zimek. On using class-labels in evaluation of clusterings, in *MultiClust: Discovering, Summarizing, and Using Multiple Clusterings*, ed. by X.Z. Fern, I. Davidson, J. Dy (ACM SIGKDD, 2010)
12. J. Dunn, Well separated clusters and optimal fuzzy partitions. J. Cybern. **4**(1), 95–104 (1974)
13. K. Pearson, On lines and planes of closest fit to systems of points in space. Phil. Mag. **2**(11), 559–572 (1901)
14. C. Manning, H. Schütze, *Foundations of Statistical Natural Language Processing* (MIT Press, Cambridge, 1999)
15. D. Jurafsky, J. Martin, *Speech and Language Processing* (Prentice Hall, New Jersey, 2008)
16. S. P. Lloyd, Least squares quantization in PCM. *IEEE Transactions on Information Theory*, **28**(2), 129–137 (1982)
17. E. W. Forgy, Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, **21**(3), 768–769 (1965)

Feedforward Neural Networks 4

4.1 Basic Concepts and Terminology for Neural Networks

Backpropagation is the core method of learning for deep learning. But before we can start exploring backpropagation, we must define a number of basic concepts and explain their interactions. Deep learning is machine learning with deep artificial neural networks, and the goal of this chapter explains how shallow neural networks work. We will also refer to shallow neural networks as *simple feedforward neural networks*, although the term itself should be used to refer to any neural network which does not have a feedback connection, not just shallow ones. In this sense, a convolutional neural network is also a feedforward neural network but not a shallow neural network. In general, deep learning consists of fixing the problems which arise when we try to add more layers to a shallow neural network. There are a number of other great books on neural networks. The book [1] offers the reader a rigorous treatment with most of the mathematical details written out, while the book [2] is more geared towards applications, but gives an overview of some connected techniques that we have not explored in this volume such as the Adaline. The book [3] is a great book written by some of the foremost experts in deep learning, and this book can be seen as a natural next step after completing the present volume. One final book we mention, and this book is perhaps the most demanding, is [4]. This is a great book, but it will place serious demands on the reader, and we suggest to tackle it after [3]. There are a number of other excellent books, but we offered here our selection which we believe will best augment the material covered in the present volume.

Any neural network is made of simple basic elements. In the last chapter, we encountered a simple neural network without even knowing it: the logistic regression. A shallow artificial neural network consists of two or three layers, anything more than that is considered deep. Just like a logistic regression, an artificial neural network has an input layer where inputs are stored. Every element which holds an input is called a 'neuron'. The logistic regression then has a single point where all inputs are

directed, and this is its output (this is also a neuron). The same holds for a simple neural network, but it can have more than one output neuron making the output layer. What is different from logistic regression is that a 'hidden' layer may exist between the input and output layer. Depending on the point of view, we can think of a neural network being a logistic regression with not one but multiple workhorse neurons, and then after them, a final workhorse neuron which 'coordinates' their results, or we could think of it as a logistic regression with a whole layer of workhorse neurons squeezed between the inputs and the old workhorse neuron (which was already present in the logistic regression). Both of these views are useful for developing intuition on neural networks, and keep this in mind in the remainder of this chapter, since we will switch form one view to the other if it becomes convenient.

The structure of a simple three-layer neural network shown in Fig. 4.1. Every neuron of one layer is connected to all neurons of the next layer, but it gets multiplied by a so-called *weight* which determines how much of the quantity from the previous layer is to be transmitted to a given neuron of the next layer. Of course, the weight is not dependent on the initial neuron, but it depends on the initial neuron–destination neuron pair. This means that the link between say neuron N_5 and neuron M_7 has a weight w_k while the link between the neurons N_5 and M_3 has a different weight, w_j. These weights can happen to have the same value by accident, but in most cases, they will have different values.

The flow of information through the neural network goes from the first-layer neurons (input layer), via the second-layer neurons (hidden layer) to the third-layer neurons (output neurons). We return now to Fig. 4.1. The input layer consists of three neurons and each of them can accept one input value, and they are represented by variables x_1, x_2, x_3 (the actual input values will be the values for these variables). Accepting input is the *only* thing the first layer does. Every neuron in the input

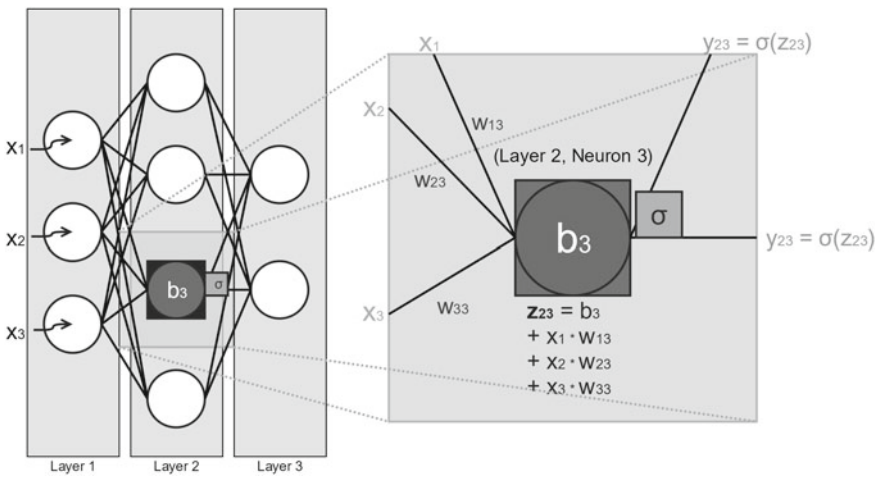


Fig. 4.1 A simple neural network

layer can take a single output. It is possible to have less input values than input neurons (then you can hand 0 to the unused neurons), but the network cannot take in more input values than it has input neurons. Inputs can be represented as a sequence $x_1, x_2, \ldots, x_n$ (which is actually the same as a *row vector*) or as a *column vector* $\mathbf{x} := (x_1, x_2, \ldots, x_n)^\top$. These are different representations of the same data, and we will always choose the representation that makes it easier and faster to compute the operations we might need. In our choice of data representation, we are not constrained by anything else but computational efficiency.

As we already noted, every neuron from the input layer is connected to every neuron from the hidden layer, but neurons of the same layer are not interconnected. Every connection between neuron j in layer k and neuron m in layer n has a weight denoted by w_{jm}^{kn}, and, since it is usually clear from the context which layers are concerned, we may omit the superscript and write simply w_{jm}. The weight regulates how much of the initial value will be forwarded to a given neuron, so if the input is 12 and the weight to the destination neuron is 0.25, the destination will receive the value 3. The weights can decrease the value, but they can also increase it since they are not bound between 0 and 1.

Once again we return to Fig. 4.1 to explain the zoomed neuron on the right-hand side. The zoomed neuron (neuron 3 from layer 2) gets the input which is the sum of the products of the inputs from the previous layer and respective weights. In this case, the inputs are x_3, x_2 and x_3, and the weights are w_{13}, w_{23} and w_{33}. Each neuron has a modifiable value in it, called the *bias*, which is represented here by b_3, and this bias is added to the previous sum. The result of this is called the *logit* and traditionally denoted by z (in our case, z_{23}).

Some simpler models[1] simply give the logit as the output, but most models apply a nonlinear function (also called a *nonlinearity* or *activation function* and represented by 'S' in Fig. 4.1) to the logit to produce the output. The output is traditionally denoted with y (in our case the output of the zoomed neuron is y_{23})[2] The nonlinearity can be generically refered to as $S(x)$ or by the name of the given function. The most common function used is the *sigmoid* or *logistic* function. We have encountered this function before, when it was the main function in logistic regression. The logistic function takes the logit z and returns as its output $\sigma(z) = \frac{1}{1+e^{-z}}$. The logistic function 'squashes' all it receives to a value between 0 and 1, and the intuitive interpretation of its meaning is that it calculates the probability of the output given the input.

A couple of remarks. Different layers may have different nonlinearities which we shall see in the later chapters, but all neurons of the same layer apply the same nonlinearity to its logits. Also, the output of a neuron is the same value in every direction it sends it. Returning to the zoomed neuron in Fig. 4.1, the neuron sends y_{23} in to directions, and both of them are the same value. As a final remark, following Fig. 4.1 again, note that the logits in the next layer will be calculated in the same manner. If we take, for example z_{31}, it will be calculated as $z_{31} = b_{31} + w_{11}^{23} y_{21} +$

[1] These models are called *linear neurons*.

[2] From linear neurons we still want to use the same notation but we set $y_{23} := z_{23}$.

$w_{21}^{23} y_{22} + w_{31}^{23} y_{23} + w_{41}^{23} y_{24}$. The same is done for z_{32}, and then by applying the chosen nonlinearity to z_{31} and z_{32} we obtain the final output.

4.2 Representing Network Components with Vectors and Matrices

Let us recall the general shape of a $m \times n$ matrix (m is the number of rows and n is the number of columns):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}$$

Suppose we need to define with matrix operations the process sketched in Fig. 4.2.

In Chap. 3 we have seen how to represent the calculations for logistic regression with matrix operators. We follow the same idea here but for simple feedforward neural networks. If we want the input to follow the vertical arrangement as it is in the picture, we can represent it as a column vector, i.e. $\mathbf{x} = (x_1, x_2)^\top$. The Fig. 4.2 also offers us the intermediate values in the network, so we can verify each step of our calculation. As explained in the earlier chapters, if A is a matrix, the matrix entry in the j row and k column is denoted by $A_{j,k}$ or by A_{jk}. If we want to 'switch' the j and k, we need the transpose of the matrix A denoted $A^\top$. So for all entries in the matrices A and $A^\top$ the following holds: A_{jk} has the same value as $A_{kj}^\top$, i.e. $A_{jk} = A_{kj}^\top$. When representing operations in neural networks as vectors and matrices, we want to minimize the use of transpositions (since each one of them has a computational cost), and keep the operations as natural and simple as possible. On the other hand, matrix transposition is not that expensive, and it is sometimes better to keep things intuitive rather than fast. In our case, we will want to represent a weight w which connects the

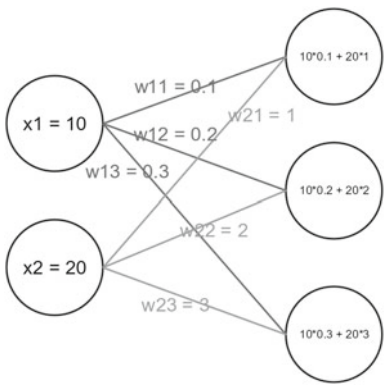


Fig. 4.2 Weights in a network

second neuron in layer 1 and the third neuron in layer 2 with a variable named w_{23}. We see that the index retains information on which neurons in the layers are connected, but one might ask where do we store the information which layers are in question. The answer is very simple, that information is best stored in the matrix name in the program code, e.g. `input_to_hidden_w`. Note that we can call a matrix by its 'mathematical name', e.g. $\mathbf{u}$ or by its 'code name' e.g. `hidden_to_output_w`. So, following Fig. 4.2 we write the weight matrix connecting the two layers as:

$$\begin{bmatrix} w_{11}(=0.1) & w_{12}(=0.2) & w_{13}(=0.3) \\ w_{21}(=1) & w_{22}(=2) & w_{23}(=3) \end{bmatrix}$$

Let us call this matrix $\mathbf{w}$ (we can add subscripts or superscripts to its name). Using matrix multiplication $\mathbf{w}^\top\mathbf{x}$ we get a 3×1 matrix, namely the column vector $\mathbf{z} = (21, 42, 63)^\top$.

With this we have described, alongside the structure of the neurons and connections the forwarding of data through the network which is called *the forward pass*. The forward pass is simply the sum of calculations that happen when the input travels through the neural network. We can view each layer as computing a function. Then, if $\mathbf{x}$ is the input vector, $\mathbf{y}$ is the output vector and f_i, f_h and f_o are the overall functions calculated at each layer, respectively,(products, sums and nonlinearities), we can say that $\mathbf{y} = f_o(f_h(f_i(\mathbf{x})))$. This way of looking at a neural network will be very important when we will address the correction of weights through backpropagation.

For a full specification of a neural network we need:

- The number of layers in a network
- The size of the input (recall that this is the same as the number of neurons in the input layer)
- The number of neurons in the hidden layer
- The number of neurons in the output layer
- Initial values for weights
- Initial values for biases

Note that the neurons are not objects. They exist as entries in a matrix, and as such, their number is necessary for specifying the matrices. The weights and biases play a crucial role: the whole point of a neural network is to find a good set of weights and biases, and this is done through training via *backpropagation*, which is the reverse of a forward pass. The idea is to measure the error the network makes when classifying and then modify the weight so that this error becomes very small. The remainder of this chapter will be devoted to backpropagation, but as this is the most important subject in deep learning, we will introduce it slowly and with numerous examples.

4.3 The Perceptron Rule

As we noted before, the learning process in the neurons is simply the modification or update of weights and biases during training with backpropagation. We will explain the backpropagation algorithm shortly. During classification, only the forward pass is made. One of the early learning procedures for artificial neurons is known as *perceptron learning*. The perceptron consisted of a *binary threshold neuron* (also known as binary threshold units) and the perceptron learning rule and altogether looks like a modified logistic regression. Let us formally define the binary threshold neuron:

$$z = b + \sum_i w_i x_i$$

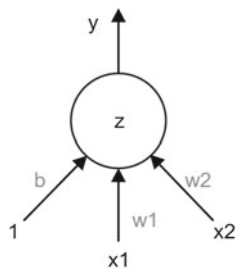
$$y = \begin{cases} 1, z \geq 0 \\ 0, \text{otherwise} \end{cases}$$

Where x_i are the inputs, w_i the weights, b is the bias and z is the logit. The second equation defines the decision, which is usually done with the nonlinearity, but here a binary step function is used instead (hence the name). We take a digression to show that it is possible to absorb the bias as one of the weights, so that we only need a weight update rule. This is displayed in Fig. 4.3: to absorb the bias as a weight, one needs to add an input x_0 with value 1 and the bias is its weight. Note that this is *exactly* the same:

$$z = b + \sum_i w_i x_i = w_0 x_0 (= b) + w_1 x_1 + w_2 x_2 + \ldots$$

According to the above equation, b could either be x_0 or w_0 (the other one must be 1). Since we want to change the bias with learning, and inputs never change, we must treat it as a weight. We call this procedure *bias absorption*.

Fig. 4.3 Bias absorption



The perceptron is trained as follows (this is the perceptron learning rule[3]):

1. Choose a training case.
2. If the predicted output matches the output label, do nothing.
3. If the perceptron predicts a 0 and it should have predicted a 1, add the input vector to the weight vector
4. If the perceptron predicts a 1 and it should have predicted a 0, subtract the input vector from the weight vector

As an example, take the input vector to be $\mathbf{x} = (0.3, 0.4)^\top$ and let the bias be $b = 0.5$, the weights $\mathbf{w} = (2, -3)^\top$ and the target[4] $t = 1$. We start by calculating the current classification result:

$$z = b + \sum_i w_i x_i = 0.5 + 2 \cdot 0.3 + (-3) \cdot 0.4 = -0.1$$

As $z < 0$, the output of the perceptron is 0 and should have been 1. This means that we have to use clause (3) from the perceptron rule and add the input vector to the weight vector:

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) + (\mathbf{x}, 1) = (2, -3, 0.5) + (0.3, 0.4, 1) = (2.3, -2.6, 1.5)$$

If adding handcrafted features is not a option, the perceptron algorithm is very limited. To see a simple problem that Minsky and Papert exposed in 1969 [5], consider that each classification problem can be understood as a query on the data. This means that we have a property we want the input to satisfy. Machine learning is just a method for defining this complex property in terms of the (numerical) properties present in the input. A query then retrieves all the input points satisfying this property. Suppose we have a dataset consisting of people and their height and weight. To return only those higher than say 175 cm, one would make a query of the form `select * from table where cm>175`. If we, on the other hand, only have jpg files of mugshots with the black and white meter behind the faces, then we would need a classifier to determine the people's height and then sort them accordingly. Note that this classifier would not use numbers, but rather pixels, so it might find people of e.g. 155 cm similar to those of height 175, but not those of 165, since the black and white parts of the background are similar. This means that the machine learning algorithm learns 'similar' in terms of the information representation it is given: what might seem similar in terms of numbers might not be similar in terms of pixels and vice versa. Consider the numbers 6 and 9: visually they are close (just rotate one to get

[3] Formally speaking, all units using the perceptron rule should be called perceptrons, not just binary threshold units.

[4] The target is also called expected value or true label, and it is usually denoted by t.

the other) but numerically they are not. If the representation given to an algorithm is in pixels, and it can be rotated,[5] the algorithm will consider them the same.

When classifying, the machine learning algorithm (and perceptrons are a type of machine learning algorithms) selects some datapoints as belonging to a class and leaves the other out. This means that some of them get the label 1 and some get the label 0, and this learned partitioning hopefully captures the underlying reality: that the datapoints labelled 1 really are 'ones' and the datapoints labelled 0 really are 'zeros'. A classic query in logic and theoretical computer science is called *parity*. This query is done over binary strings of data, and only those with an equal number of ones and zeros are selected and given the label 1. Parity can be relaxed so it considers only strings of length n, then we can formally name it $\texttt{parity}_n(x_0, x_1, \ldots, x_n)$, where each x_i is a single binary digit (or *bit*). $\texttt{parity}_2$ is also called XOR and it is also a logical function called exclusive disjunction. XOR takes two bits and returns 1 if and only if there is the same amount of 1 and 0, and since they are binary strings, this means that there is one 1 and one 0. Note that we can equally use the logical equivalence which has the resulting 0 and 1 exchanged, since they are just names for classes and do not carry much more meaning. So XOR gives the following mapping: $(0, 0) \mapsto 0$, $(0, 1) \mapsto 1$, $(1, 0) \mapsto 1$, $(1, 1) \mapsto 0$.

When we have XOR as the problem (or any instance of parity for that matter), the perceptron is unable to learn to classify the input so that they get the correct labels. This means that a perceptron that has two input neurons (for accepting the two bits for XOR) cannot adjust its two weights to separate the 1 and 0 as they come in the XOR. More formally, if we denote by w_1, w_2 and b the weights and biases of the perceptron, and take the following instance of parity $(0, 0) \mapsto 1$, $(0, 1) \mapsto 0$, $(1, 0) \mapsto 0$ i $(1, 1) \mapsto 1$, we get four inequalities:

1. $w_1 + w_2 \geq b$,
2. $0 \geq b$,
3. $w_1 < b$,
4. $w_2 < b$

The inequality (a) holds since if $(x_1 = 1, x_2 = 1) \mapsto 1$, and we can get 1 as an output only if $w_1x_1 + w_2x_2 = w_1 \cdot 1 + w_2 \cdot 1 = w_1 + w_2$ is greater or equal b, which means $w_1 + w_2 \geq b$.

The inequality (b) holds since if $(x_1 = 0, x_2 = 0) \mapsto 1$, and we can get 1 as an output only if $w_1x_1 + w_2x_2 = w_1 \cdot 0 + w_2 \cdot 0 = 0$ is greater or equal b, which means $0 \geq b$.

The inequality (c) holds since if $(1, 0) \mapsto 0$, then $w_1x_1 + w_2x_2 = w_1 \cdot 1 + w_2 \cdot 0 = w_1$, and for the perceptron to give 0, w_1 has to be less than the bias b, i.e. $w_1 < b$.

[5] As a simple application, think of an image recognition system for security cameras, where one needs to classify numbers seen regardless of their orientation.

The inequality (d) is derived in a similar fashion to (c). By adding (a) and (b) we get $w_1 + w_2 \geq 2b$, and by adding (c) and (d) we get $w_1 + w_2 < 2b$. It is easy to see that the system of inequalities has no solution.

This means that the perceptron, which was claimed to be a contendant for general artificial intelligence could not even learn logical equality. The proposed solution was to make a 'multilayered perceptron'.

4.4 The Delta Rule

The main problem with making the 'multilayered perceptron' is that it is unknown how to extend the perceptron learning rule to work with multiple layers. Since multiple layers are needed, the only option seemed to be to abandon the perceptron rule and use a different rule which is more robust and capable of learning weights accross layer. We already mentioned this rule—*backpropagation*. It was first discovered by Paul Werbos in his PhD thesis [6], but it remained unnoticed. It was discovered for the second time by David Parker in 1981, who tried to get a patent but he subsequently published it in 1985 [7]. The third and the last time it was discovered independently by Yann LeCun in 1985 [8] and by Rumelhart, Hinton and Williams in 1986 [9].

To see what we want to archive, let us consider an example[6] imagine that each day we buy lunch at the nearby supermarket. Every day our meal consists of a piece of chicken, two grilled zucchinis and a scoop of rice. The cashier just gives us the total amount, which varies each day. Suppose that the price of the components does not very over time and that we can weight the food to see how much we have. Note that one meal will not be enough to deduce the prices, since we have three of them[7] and we do not know which component is responsible in what proportion for a total price increase in one euro.

Notice that the price per kilogram is actually similar to the neural network weight. To see this think of how you would find the price per kilogram of the meal components: you make a guess on the prices per kilogram for the components, multiply with the quantity you got today and compare their sum to the price you have actually paid. You will see that you are off by e.g. 6€. Now you must find out which components are 'off'. You could stipulate that each component is off by 2€ and then readjust your stipulated price per kilogram by the 2€ and wait for you next meal to see whether it will be better now. Of course you could have also stipulated that the components are off by 3, 2, 1€ respectively, and either way, you would have to wait for your next meal with your new price per kilograms and try again to see whether you will be off by a lesser or greater amount. Of course, you want to correct your

[6] This is a modified version of an example given by Geoffrey Hinton.

[7] For example, if we only buy chicken, then it would be easy to get the price of the chicken analytically as $total = price \cdot quantity$, and we get $price = \frac{total}{quantity}$.

estimations so that you are off by less and less as the meals pass, and hopefully, this will lead you to a good approximation.

Note that there exists a *true* price per kilogram but we do not know it, and our method is trying to discover it just by measuring how much we miss the total price. There is a certain 'indirectness' in this procedure and this is highly useful and the essence of neural networks. Once, we find our good approximations, we will be able to calculate with appropriate precision the total price of all of our future meals, without needing to find out the actual prices.[8]

Let us work a bit more this example. Each meal has the following general form:

$$total = ppk_{chicken} \cdot quant_{chicken} + ppk_{zucchini} \cdot quant_{zucchini} + ppk_{rice} \cdot quant_{rice}$$

where *total* is the total price, the *quant* is the quantity and the *ppk* is the price per kilogram for each component. Each meal has a total price we know, and the quantities we know. So each meal places a *linear constraint* on the *ppk*-s. But with only this we cannot solve it. If we plug in this formula our initial (or subsequenlty corrected) 'guesstimate'[9] we will get also the predicted value, and by comparing it with the true (target) total value we will also get an error value which will tell us by how much we missed. If after each meal we miss by less, we are doing a great job.

Let us imagine that the true price is $ppk_{chicken} = 10$, $ppk_{zucchini} = 3$, and $ppk_{rice} = 5$. Let us start with a guesstimate of $ppk_{chicken} = 6$, $ppk_{zucchini} = 3$, and $ppk_{rice} = 3$. We know we bought 0.23 kg of chicken, 0.15 kg of zucchini and 0.27 kg of rice and that we paid 3€ in total. By multiplying our guessed prices with the quantities we get 1.38, 0.45 and 0.81, which totals to 2.64, which is 0.35 less than the true price. This value is called the *residual error*, and we want to minimize it over the course of future iterations (meals), so we need to distribute the residual error to the *ppk*-s. We do this simply by changing the *ppk*-s by:

$$\Delta ppk_i = \frac{1}{n} \cdot quant_i(t - y)$$

where $i \in \{chicken, zucchini, rice\}$, n is the cardinality (number of elements) of this set (i.e. 3), $quant_i$ is the quantity of i, t is the total price and y is the predicted total price. This is known as the *delta rule*. When we rewrite this in standard neural network notation it looks like:

$$\Delta w_i = \eta x_i(t - y)$$

[8] In practical terms this might seem far more complicated than simply asking the person serving you lunch the price per kilogram for components, but you can imagine that the person is the soup vendor from the soup kitchen from the TV show Seinfeld (116th episode, or S07E06).

[9] A guessed estimate. We use this term just to note that for now, we should keep things intuitive an not guess an initial value of, e.g. 12000, 4533233456, 0.0000123, not because it will be impossible to solve it, but because it will need much more steps to assume a form where we could see the regularities appear.

where w_i is a weight, x_i is the input and $t - y$ is the residual error. The η is called the *learning rate*. Its default value should be $\frac{1}{n}$, but there is no constraint placed on it so values like 10 are perfectly ok to use. In practice, however, we want the values for η to be small, and usually of the form 10^{-n}, meaning 0.1, 0.01, etc., but values such as 0.03 or 0.0006 are also used. The learning rate is an example of a *hyperparameter*, which are parameters in the neural network which cannot be learned like regular parameters (like weights and biases) but have to be adjusted by hand. Another example of a hyperparameter is the hidden layer size.

The learning rate controls how much of the residual error is handed down to the individual weights to be updated. The proportional distribution of $\frac{1}{n}$ is not that important if the learning rate is close to that number. For example, if $n = 90$ it is virtually the same if one uses the proportional learning rate of $\frac{1}{90}$ or a learning rate of 0.01. From a practical point of view, it is best to use a learning rate close to the proportional learning rate or smaller. The intuition behind using a smaller learning rate than the proportional is to update the weights only a bit in the right direction. This has two effects: (i) the learning takes longer and (ii) the learning is much more precise. The learning takes longer since with a smaller learning rate each update make only a part of the change needed, and it is more precise since it is much less likely to be overinfluenced by one learning step. We will make this more clear later.

4.5 From the Logistic Neuron to Backpropagation

The delta rule as defined above works for a simple neuron called the *linear neuron*, which is even simpler than the binary threshold unit:

$$y = \sum_i w_i x_i = \mathbf{w}^\top \mathbf{x}$$

To make the delta rule work, we will be needing a function which should measure if we got the result right, and if not, by how much we missed. This is usually called an *error function* or *cost function* and traditionally denoted by $E(x)$ or by $J(x)$. We will be using the *mean squared error*:

$$E = \frac{1}{2} \sum_{n \in train} (t^{(n)} - y^{(n)})^2$$

where the ($t^{(n)}$ denotes the target for the training case n (same for ($y^{(n)}$, but this is the prediction). The training case n is simply a training example, such as a single image or a row in a table. The mean squared error sums the error across all the training cases n, and after that we will update the weights. The natural choice for measuring how far were we from the bullseye would be to use the absolute value as a measure of distance that does not depend on the sign, but the reason behind choosing the square of the difference is that by simply squaring the difference we get a measure similar

to absolute values (albeit larger in magnitude, but this is not a problem, since we want to use it in relative, not absolute terms), but we will get as a bonus some nice properties to work with down the road.

Let us see, how we can derive the delta rule from the SSE to see that they are the same.[10] We start with the above equation defining the mean squared error and differentiate E with respect to w_i and get:

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_n \frac{\partial y^{(n)}}{\partial w_i} \frac{dE^{(n)}}{dy^{(n)}}$$

The partial derivatives are here just because we have to consider a single w_i and treat all others as constants, but the overall behaviour apart from that is the same as with ordinary derivatives. The above formula tells us a story: it tells us that to find out how E changes with respect to w_i, we must find out how $y^{(n)}$ changes with respect to w_i and how E changes with respect to $y^{(n)}$. This is a nice example of the chain rule of derivations in action. We explored the chain rule in the second chapter but we will give a cheatsheet for derivations shortly so you do not have to go back. Informally speaking, the chain rule is similar to fraction multiplication, and if one recalls that a shallow neural network is a structure of the general form $\mathbf{y} = f_o(f_h(f_i(\mathbf{x})))$, it is easy to see that there will be a lot of places to use the chain rule, especially as we go on to deep learning and add more layers.

We will explain the derivations shortly. The above equation means the weight updates are proportional to the error derivations in all training cases added together:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} = \sum_n \eta x_i^{(n)} (t^{(n)} - y^{(n)})$$

Let us proceed to the actual derivation. We will be deriving the result for a logistic neuron (also called a *sigmoid* neuron), which we have already presented before, but we will define it once more:

$$z = b + \sum_i w_i x_i$$

$$y = \frac{1}{1 + e^{-z}}$$

Recall that z is the logit. Let us absorb the bias right away, so we do not have to deal with it separately. We will calculate the derivation of the logistic neuron with respect to the weights, and the reader can adapt the procedure to the simpler linear neuron if she likes. As we noted before, the chain rule is your best friend for obtaining derivations, and the 'middle variable' of the chain rule will be the logit. The first

[10] Not in the sense that they are the same formula, but that they refer to the same process and that one can be derived from the other.

part $\frac{\partial z}{\partial w_i}$ which is equal to x_i since $z = \sum_i w_i x_i$ (we absorbed the bias). By the same argument $\frac{\partial z}{\partial x_i} = w_i$.

The derivation of the output with respect to the logit is a simple expression ($\frac{dy}{dz} = y(1-y)$) but is not easy to derive. Let us restate the derivation rules we use[11]

- `LD`: Differentiation is linear, so we can differentiate the summands separately and take out the constant factors: $[f(x)a + g(x)b]' = a \cdot f'(x) + b \cdot g'(x)$
- `Rec`: Reciprocal rule $[\frac{1}{f(x)}]' = -\frac{f'(x)}{f(x)^2}$
- `Const`: Constant rule $c' = 0$
- `ChainExp`: Chain rule for exponents $[e^{f(x)}]' = e^{f(x)} \cdot f'(x)$
- `DerDifVar`: Deriving the differentiation variable $\frac{dy}{dz} z = 1$
- `Exp`: Exponent rule $[f(x)^n]' = n \cdot f(x)^{n-1} \cdot f'(x)$

We can now start deriving $\frac{dy}{dz}$. We start with the definition for y, i.e. with

$$\frac{dy}{dz} \frac{1}{1+e^{-z}}$$

From this expression by application of the `Rec` rule we get

$$-\frac{\frac{dy}{dz}(1+e^{-z})}{(1+e^{-z})}$$

From this by applying `LD` we get

$$-\frac{\frac{dy}{dz}1 + \frac{dy}{dz}e^{-z}}{(1+e^{-z})^2}$$

On the first summand in the numerator, we apply `Const` and it becomes 0, and on the second we apply `ChainExp` and it becomes $e^{-z} \cdot \frac{dy}{dz}(-z)$, and so we have

$$-\frac{e^{-z} \cdot \frac{dy}{dz}(-z)}{(1+e^{-z})^2}$$

By applying `LD` to the constant factor -1 implicit with z we get

$$-\frac{-1 \cdot \frac{dy}{dz} z \cdot e^{-z}}{(1+e^{-z})^2}$$

[11] For the sake of easy readability, we deliberately combine Newton and Leibniz notation in the rules, since some of them are more intuitive in one, while some of them are more intuitive in the second. We refer the reader back to Chap. 1 where all the formulations in both notations were given.

which by `DerDifVar` becomes

$$-\frac{-1 \cdot e^{-z}}{(1+e^{-z})^2}$$

We tidy up the signs and get

$$\frac{e^{-z}}{(1+e^{-z})^2}$$

Therefore,

$$\frac{dy}{dz} = \frac{e^{-z}}{(1+e^{-z})^2}$$

Let us factorize the right-hand side in two factors which we will call A and B:

$$\frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}}$$

It is obvious that $A = y$ from the definition of y. Let us turn our attention to B:

$$\frac{e^{-z}}{1+e^{-z}} = \frac{(1+e^{-z})-1}{1+e^{-z}} = \frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}} = 1 - \frac{1}{1+e^{-z}} = 1 - y$$

Therefore $A = y$ and $B = 1 - y$, and $\frac{dy}{dz} = A \cdot B$, from which follows that

$$\frac{dy}{dz} = y(1-y)$$

Since we have $\frac{\partial z}{\partial w_i}$ and $\frac{dy}{dz}$ with the chain rule we get

$$\frac{\partial y}{\partial w_i} = x_i y(1-y)$$

The next thing we need is $\frac{dE}{dy}$.[12] We will be using the same rules for this derivation as we did for $\frac{dy}{dz}$. Recall that $E = \frac{1}{2}(t^{(n)} - y^{(n)})^2$, but we will use the version $E = \frac{1}{2}(t-y)^2$ which is focused on a single target value t and a single prediction y.

Therefore, we need to find

$$\frac{dE}{dy}[\frac{1}{2}(t-y)^2]$$

[12] Strictly speaking, we would need $\frac{\partial E}{\partial y^{(n)}}$ but this generalization is trivial and we chose the simplification since we wanted to improve readability.

By applying LD we get

$$\frac{1}{2}\frac{dE}{dy}(t-y)^2$$

By applying Exp we get

$$\frac{1}{2}\cdot 2\cdot(t-y)\cdot\frac{dE}{dy}(t-y)$$

Simple cancellation yields

$$(t-y)\cdot\frac{dE}{dy}(t-y)$$

With LD we get

$$(t-y)\cdot\frac{dE}{dy}t\cdot\frac{dE}{dy}y$$

Since t is a constant, its derivative is 0 (rule Const), and since y is the differentiation variable, its derivative is 1 (DerDiVar). By tidying up the expression we get $(t-y)(0-1)$ and finally, $-1\cdot(t-y)$.

Now, we have all the elements for formulating the learning rule for the logistic neuron using the chain rule:

$$\frac{\partial E}{\partial w_i}=\sum_n\frac{\partial y^{(n)}}{\partial w_i}\frac{\partial E}{\partial y^{(n)}}=-\sum_n x_i^{(n)}y^{(n)}(1-y^{(n)})(t^{(n)}-y^{(n)})$$

Note that this is very similar to the delta rule for the linear neuron, but it has also $y^{(n)}(1-y^{(n)})$ extra: this part is the slope of the logistic function.

4.6 Backpropagation

So far we have seen how to use derivatives to learn the weights of a logistic neuron, and without knowing it we have already made excellent progress with understanding backpropagation, since backpropagation is actually the same thing but applied more than once to 'backpropagate' the errors through the layers. The logistic regression (consisting of the input layer and a single logistic neuron), strictly speaking, did not need to use backpropagation, but the weight learning procedure described in the previous section actually *is* a simple backpropagation. As we add layers, we will not have more complex calculations, but just a large number of those calculations. Nevertheless, there are some things to watch out for.

We will write out all the necessary details for backpropagation for the feedforward neural networks, but first, we will build up the intuition behind it. In Chap. 2 we have explained gradient descent, and we will revisit some of the concepts here as

needed. *Backpropagation of errors is basically just gradient descent*. Mathematically speaking, backpropagation is:

$$w_{updated} = w_{old} - \eta \nabla E$$

where w is the weigh, η is the learning rate (for simplicity you can think of it just being 1 for now) and E is the cost function measuring overall performance. We could also write it in computer science notation as a rule that assigns to w a new value:

$$w \leftarrow w - \eta \nabla E$$

This is read as 'the new value of w is w minus $\eta \nabla E$'. This is not circular,[13] since it is formulated as an assignment ($\leftarrow$), not a definition ($=$ or $:=$). This means that first, we calculate the right-hand side, and then we assign to w this new value. Notice that if were to write out this mathematically, we would have a recursive definition.

We may wonder whether we could do weight learning in a more simple manner, without using derivatives and gradient descent.[14] We could try the following approach: select a weight w and modify it a bit and see if that helps. If it does, keep the change. If it makes things worse, then change it in the opposite direction (i.e. instead of adding the small amount from the weight, subtract it). If this makes it better keep the change. If neither change improves the final result, we can conclude that w is perfect as it is and move to the next weight v.

Three problems arise right away. First, the process takes a long time. After the weight change, we need to process at least a couple of training examples for each weight to see if it is better or worse than before. Simply speaking, this is a computational nightmare. Second, by changing the weights individually we will never find out whether a combination of them would work better, e.g. if you change w or v separately (either by adding the small amount or subtracting to one or the other), it might make the classification error worse, but if you were to change them by adding a small amount to both of them it would make things better. The first of these problems will be overcome by using gradient descent,[15] while the second will be only partially resolved. This problem is usually called *local optima*.

The third problem is that near the end of learning, changes will have to be small, and it is possible that the 'small change' our algorithm test will be too large to successfully learn. Backpropagation also has this problem, and it is usually solved by using a dynamic learning rate which gets smaller as the learning progresses.

---

[13] A definition is circular if the same term occurs in both the *definiendum* (what is being defined) and *definiens* (with which it is defined), i.e. on both sides of $=$ (or more precisely of $:=$) and in our case this term could be w. A recursive definition has the same term on both sides, but on the defining side (definiens) it has to be 'smaller' so that one could resolve the definition by going back to the starting point.

[14] If you recall, the perceptron rule also qualifies as a 'simpler' way of learning weights, but it had the major drawback that it cannot be generalized to multiple layers.

[15] Although it must be said that the whole field of deep learning is centered around overcoming the problems with gradient descent that arise when using it in deep networks.

If we formalize this approach we will get a method called *finite difference approximation*[16]:

1. Each weight w_i, $1 \leq i \leq k$ is adjusted by adding to it a small constant ε (e.g. whose value is 10^{-6}) and the overall error (with only w_i changed) is evaluated (we will denote this by E_i^+)
2. Change back the weight to its initial value w_i and subtract ε from it and reevaluate the error (this will be E_i^-)
3. Do this for all weights w_j, $\leq j \leq k$
4. Once finished, the new weights will be set to $w_i \leftarrow w_i - \frac{E_i^+ - E_i^-}{2\varepsilon}$

The finite difference approximation does a good job in approximating the gradient, and nothing more than elementary arithmetic is used. If we recall what a derivative is and how it is defined from Chap. 2, the finite difference approximation makes sense even in terms of the 'meaning' of the procedure. This method can be used to build up the intuition how weight learning proceeds in full backpropagation. However, most current libraries which have tools for automatic differentiation perform gradient descent in a fraction of the time it would take to compute the finite difference approximation. Performance issues aside, the finite difference approximation would indeed work in a feedforward neural network.

Now, we turn to backpropagation. Let us examine what happens in the hidden layer of the feedforward neural network. We start with randomly initialized weights and biases, multiply them with the inputs, add them together, and take them through the logistic regression which "flattens" them to a value between 0 and 1, and we do that one more time. At the end, we get a value between 0 and 1 from the logistic neuron in the output layer. We can say that everything above 0.5 is 1 and below is 0. But the problem is that if the network gives a 0.67 and the output should have been 0, we know only the error the network produced (the function E), and we should use this. More precisely, we want to measure how E changes when the w_i change, which means that we want to find the derivative of E with regard to the activities of the hidden layer. We want to find all the derivatives at the same time, and for this, we use vector and matrix notations and, consequently, the gradient. Once we have the derivatives of E with regard to the hidden layer activity, we will easily compute the changes for the weights themselves.

We will address the procedure illustrated in Fig. 4.4. To keep the exposition as clear as possible, we will use only two indices, as if each layer has only one neuron. In the following section, we shall expand this to a fully functional feedforward neural network. As illustrated in Fig. 4.4 we will use the subscripts o for the output layer and h for the hidden layer. Recall that z is the logit, i.e. everything except the application of the nonlinearity.

[16] Cf. G. Hinton's Coursera course, where this method is elaborated.

As we have

$$E = \frac{1}{2} \sum_{o \in Output} (t_o - y_o)^2$$

the first thing we need to do is turn the difference between the output and the target value into an error derivation. We have done this already in the previous sections of this chapter:

$$\frac{\partial E}{\partial y_o} = -(t_o - y_o)$$

Now, we need to reformulate the error derivative with regard to y_o into an error derivative with regard to z_o. For this, we use the chain rule:

$$\frac{\partial E}{\partial z_o} = \frac{\partial y_o}{\partial z_o} \frac{\partial E}{\partial y_o} = y_o(1 - y_o) \frac{\partial E}{\partial y_o}$$

Now we can calculate the error derivative with respect to y_h:

$$\frac{\partial E}{\partial y_h} = \sum_o \frac{dz_o}{dy_h} \frac{\partial E}{\partial z_o} = \sum_o w_{ho} \frac{\partial E}{\partial z_o}$$

These steps we made from $\frac{\partial E}{\partial y_o}$ to $\frac{\partial E}{\partial y_h}$ are the heart of backpropagation. Notice that now we can repeat this to go through as many layers as we want. There will be a catch though, but for now is all good. A few remarks about the above equation. From the previous section, when we addressed the logistic neuron we know that $\frac{dz_o}{dy_h} = w_{ho}$. Once, we have $\frac{\partial E}{\partial z_o}$ it is very simple to get the error derivative with regard to the weights:

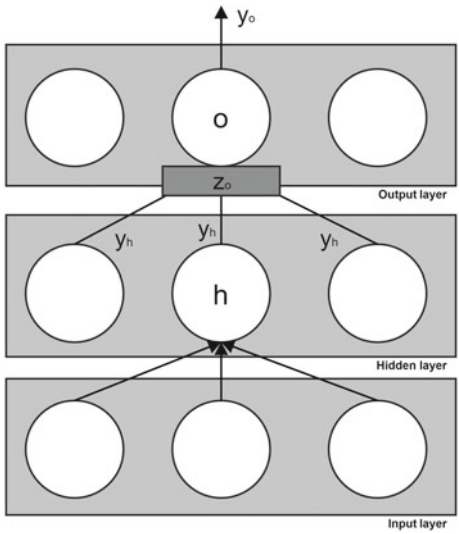


Fig. 4.4 Backpropagation

$$\frac{\partial E}{\partial w_{ho}} = \frac{\partial z_o}{\partial w_{ho}} \frac{\partial E}{\partial z_o} = y_i \frac{\partial E}{\partial z_j}$$

The rule for updating weights is quite straightforward, and we call it the *general weight update rule*:

$$w_i^{new} = w_i^{old} + (-1)\eta \frac{\partial E}{\partial w_i^{old}}$$

The η is the learning rate and the factor -1 is here to make sure we go towards minimizing E, otherwise we would be maximizing it. We can also state it in vector notation[17] to get rid of the indices:

$$\mathbf{w}^{new} = \mathbf{w}^{old} - \eta \nabla E$$

Informally speaking, the learning rate controls by how much we should update. There are a couple of possibilities (we will discuss the learning rate in more detail later):

1. Fixed learning rate
2. Adaptable global learning rate
3. Adaptable learning rate for each connection

We will address these issues in more detail later, but before that, we will show a detailed calculation for error backpropagation in a simple neural network, and in the next section, we will code the network. The remainder of this chapter is probably the most important part of the whole book, so be sure to go through all the details.

Let us see a working example[18] of a simple and shallow feedforward neural network. The network is represented in Fig. 4.5. Using the notation, the starting weights and the inputs specified in the image, we will calculate all the intricacies of the forward pass and backpropagation for this network. Notice the enlarged neuron D. We have used this to illustrate, where the logit z_D is and how it becomes the output of D (y_D) by applying to it the logistic function σ.

We will assume (as we did previously) that all the neurons have a logistic activation function. So we need to do a forward pass, a backpropagation, and a second forward pass to see the decrease in the error. Let us briefly comment on the network itself. Our network has three layers, with the input and hidden layers consisting of two neurons, and the output error which consists of one neuron. We have denoted the layers with capital letters, but we have skipped the letter E to avoid confusing it with the error function, so we have neurons named A, B, C, D and F. This is not usual.

[17] We must then use the gradient, not individual partial derivatives.

[18] This is a modified version of the example by Matt Mazur available at https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

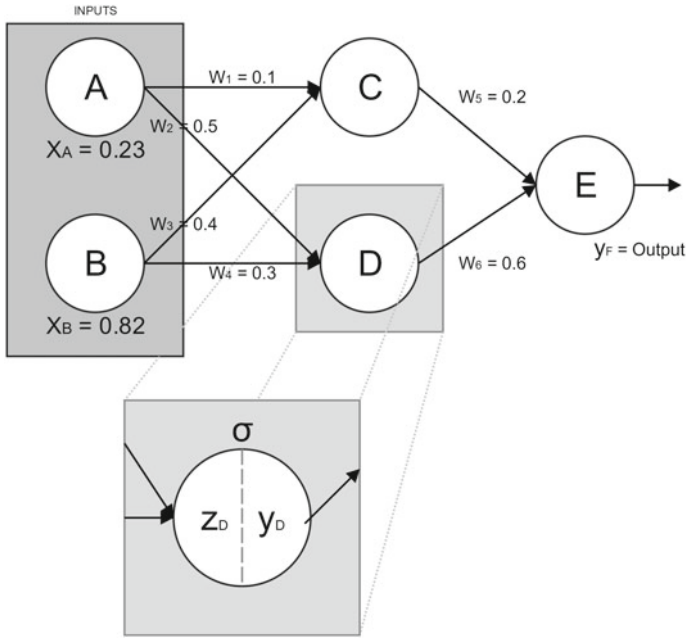


Fig. 4.5 Backpropagation in a complete simple neural network

The usual procedure is to name them by referring to the layer and neuron in the layer, e.g. 'third neuron in the first layer' or '1, 3'. The input layer takes in two inputs, the neuron A takes in $x_A = 0.23$ and the neuron B takes in $x_B = 0.82$. The target for this training case (consisting of x_A and x_B) will be 1. As we noted earlier, the hidden and output layers have the logistic activation function (also called *logistic nonlinearity*), which is defined as $\sigma(z) = \frac{1}{1+e^{-z}}$.

We start by computing the forward pass. The first step is to calculate the outputs of C and D, which are referred to as y_C and y_D, respectably:

$$y_C = \sigma(0.23 \cdot 0.1 + 0.82 \cdot 0.4) = \sigma(0.351) = 0.5868$$

$$y_D = \sigma(0.23 \cdot 0.5 + 0.82 \cdot 0.3) = \sigma(0.361) = 0.5892$$

And now we use y_C and y_D as inputs to the neuron F which will give us the final result:

$$y_F = \sigma(0.5868 \cdot 0.2 + 0.5892 \cdot 0.6) = \sigma(0.4708) = 0.6155$$

Now, we need to calculate the output error. Recall that we are using the mean squared error function, i.e. $E = \frac{1}{2}(t - y)^2$. So we plug in the target (1) and output (0.6155) and get:

$$E = \frac{1}{2}(t-y)^2 = \frac{1}{2}(1-0.6155)^2 = 0.0739$$

Now we are all set to calculate the derivatives. We will explain how to calculate w_5 and w_3 but all other weights are calculated with the same procedure. As back-propagation proceeds in the opposite direction that the forward pass, calculating w_5 is easier and we will do that first. We need to know how the change in w_5 affects E and we want to take those changes which minimize E. As noted earlier, the chain rule for derivatives will do most of the work for us. Let us rewrite what we need to calculate:

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial y_F} \cdot \frac{\partial y_F}{\partial z_F} \cdot \frac{\partial z_F}{\partial w_5}$$

We have found the derivatives for all of these in the previous sections so we will not repeat their derivations. Note that we need to use partial derivatives because every derivation is made with respect to an indexed term. Also, note that the vector containing all partial derivatives (for all indices i) is the gradient. Let us address $\frac{\partial E}{\partial y_F}$ now. As we have seen earlier:

$$\frac{\partial E}{\partial y_F} = -(t-y_F)$$

In our case that means:

$$\frac{\partial E}{\partial y_F} = -(1-0.6155) = -0.3844$$

Now we address $\frac{\partial y_F}{\partial z_F}$. We know that this is equal to $y_F(1-y_F)$. In our case this means:

$$\frac{\partial y_F}{\partial z_F} = y_F(1-y_F) = 0.6155(1-0.6155) = 0.2365$$

The only thing left to calculate is $\frac{\partial z_F}{\partial w_5}$. Remember that:

$$z_F = y_C \cdot w_5 + y_D \cdot w_6$$

By using the rules of differentiation (derivatives of constants (w_6 is treated like a constant) and differentiating the differentiation variable) we get:

$$\frac{\partial z_F}{\partial w_5} = y_C \cdot 1 + y_D \cdot 0 = y_C = 0.5868$$

We take these values back to the chain rule and get:

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial y_F} \cdot \frac{\partial y_F}{\partial z_F} \cdot \frac{\partial z_F}{\partial w_5} = -0.3844 \cdot 0.2365 \cdot 0.5868 = -0.0533$$

We repeat the same process[19] to get $\frac{\partial E}{\partial w_6} = -0.0535$. Now, all we have to do is use these values in the general weight update rule[20] (we use a learning rate, $\eta = 0.7$):

$$w_5^{new} = w_5^{old} - \eta \frac{\partial E}{\partial w_5} = 0.2 - (0.7 \cdot 0.0533) = 0.2373$$

$$w_6^{new} = 0.6374$$

Now we can continue to the next layer. But an important note first. We will be needing a value for w_5 and w_6 to find the derivatives of w_1, w_2, w_3 and w_4, and we will be using the old values, not the updated ones. We will update the whole network when we will have all the updated weights. We proceed to the hidden layer. What we need to now is to find the update for w_3. Notice that to get from the output neuron F to w_3 we need to go across C, so we will be using:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial y_C} \cdot \frac{\partial y_C}{\partial z_C} \cdot \frac{\partial z_C}{\partial w_3}$$

The process will be similar to $\frac{\partial E}{\partial w_3}$, but with a couple of modifications. We start with:

$$\frac{\partial E}{\partial y_C} = \frac{\partial z_F}{\partial y_C} \frac{\partial E}{\partial z_F} = w_5 \frac{\partial E}{\partial z_F} = w_5 \frac{\partial y_F}{\partial z_F} \cdot \frac{\partial E}{\partial y_F} = 0.2 \cdot 0.2365 \cdot (-0.3844) = 0.2 \cdot (-0.0909) = -0.0181$$

Now we need $\frac{\partial y_C}{\partial z_C}$:

$$\frac{\partial y_C}{\partial z_C} = y_C(1 - y_C) = 0.5868 \cdot (1 - 0.5868) = 0.2424$$

And we also need $\frac{\partial z_C}{\partial w_3}$. Recall that $z_C = x_1 \cdot w_1 + x_2 \cdot w_2$, and therefore:

$$\frac{\partial z_C}{\partial w_3} = x_1 \cdot 0 + x_2 \cdot 1 = x_2 = 0.82$$

Now we have:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial y_C} \cdot \frac{\partial y_C}{\partial z_C} \cdot \frac{\partial z_C}{\partial w_3} = -0.0181 \cdot 0.2424 \cdot 0.82 = -0.0035$$

[19] The only difference is the step for $\frac{\partial z_F}{\partial w_5}$, where there is a 0 now for w_5 and a 1 for w_6.

[20] Which we discussed earlier, but we will restate it here: $w_k^{new} = w_k^{old} - \eta \frac{\partial E}{\partial w_k}$.

Using the general weight update rule we have:

$$w_3^{new} = 0.4 - (0.7 \cdot (-0.0035)) = 0.4024$$

We use the same steps (across C) to find $w_1^{new} = 0.1007$. To get w_2^{new} and w_4^{new} we need to go across D. Therefore we need:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial y_D} \cdot \frac{\partial y_D}{\partial z_D} \cdot \frac{\partial z_D}{\partial w_3}$$

But we know the procedure, so:

$$\frac{\partial E}{\partial y_D} = w_6 \cdot \frac{\partial E}{\partial z_F} = 0.6 \cdot (-0.0909) = -0.0545$$

$$\frac{\partial y_C}{\partial z_C} = y_D(1 - y_D) = 0.5892(1 - 0.5892) = 0.2420$$

And:

$$\frac{\partial z_D}{\partial w_2} = 0.23$$

$$\frac{\partial z_D}{\partial w_4} = 0.82$$

Finally, we have (remember we have the 0.7 learning rate):

$$w_2^{new} = 0.5 - 0.7 \cdot (-0.0545 \cdot 0.2420 \cdot 0.23) = 0.502$$

$$w_4^{new} = 0.3 - 0.7 \cdot (-0.0545 \cdot 0.2420 \cdot 0.82) = 0.307$$

And we are done. To recap, we have:

- $w_1^{new} = 0.1007$
- $w_2^{new} = 0.502$
- $w_3^{new} = 0.4024$
- $w_4^{new} = 0.307$
- $w_5^{new} = 0.2373$
- $w_6^{new} = 0.6374$
- $E^{old} = 0.0739$

We can now make another forward pass with the new weights to make sure that the error has decreased:

$$y_C^{new} = \sigma(0.23 \cdot 0.1007 + 0.82 \cdot 0.4024) = \sigma(0.3531) = 0.5873$$

$$y_D^{new} = 0.5907$$

$$y_F^{new} = \sigma(0.5873 \cdot 0.2373 + 0.5907 \cdot 0.6374) = \sigma(0.5158) = 0.6261$$

$$E^{new} = \frac{1}{2}(1 - 0.6261)^2 = 0.0699$$

Which shows that the error has decreased. Note that we have processed only one training sample, i.e. the input vector $(0.23, 0.82)$. It is possible to use multiple training samples to generate the error and find the gradients (mini-batch training[21]), and we can do this a number of times and each repetition is called an *iteration*. Iterations are sometimes erroneously called *epochs*. The two terms are very similar and we can consider them synonyms for now, but quite soon we will need to delineate the difference, and we will do this in the next chapter.

An alternative to this would be to update the weights after every single training example.[22] This is called *online learning*. In online learning, we process a single input vector (training sample) per iteration. We will discuss this in the next chapter in more detail.

In the remainder of this chapter, we will integrate all the ideas we have presented so far in a fully functional feedforward neural network, written in Python code. This example will be fully functional Python 3.x code, but we will write out some things that could be better left for a Python module to do.

Technically speaking, in anything but the most basic setting, we shall not use the SSE, but its variant, the *mean squared error* (MSE). This is because we need to be able to rewrite the cost function as the average of the cost functions SSE_x for individual training samples x, and we therefore define $MSE := \frac{1}{n}\sum_x SSE_x$.

4.7 A Complete Feedforward Neural Network

Let us see a complete feedforward neural network which does a simple classification. The scenario is that we have a webshop selling books and other stuff, and we want to know whether a customer will abandon a shopping basket at checkout. This is why we are making a neural network to predict it. For simplicity, all the data is just numbers. Open a new text file, rename it to `data.csv` and write in the following:

[21] Or full-batch if we use the whole training set.

[22] Which is equal to using a mini-batch of size 1.

```
includes_a_book,purchase_after_21,total,user_action
1,1,13.43,1
1,0,23.45,1
0,0,45.56,0
1,1,56.43,0
1,0,44.44,0
1,1,667.65,1
1,0,56.66,0
0,1,43.44,1
0,0,4.98,1
1,0,43.33,0
```

This will be our dataset. You can actually substitute this for anything, and as long as values are numbers, it will still work. The target is the `user_action` column, and we take 1 to mean that the purchase was successful, and 0 to mean that the user has abandoned the basket. Notice that we are talking about abandoning a shopping basket, but we could have put anything in, from images of dogs to bags of words. You should also make another CSV file named `new_data.csv` that has the same structure as `data.csv`, but without the last column (`user_action`). For example:

```
includes_a_book,purchase_after_21,total
1,0,73.75
0,0,64.97
1,0,3.78
```

Now let is continue to the Python code file. All the code in the remainder of this section should be placed in a single file, you can name it `ffnn.py`, and placed in the same folder as `data.csv` and `new_data.csv`. The first part of the code contains the import statements:

```
import pandas as pd
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense
TARGET_VARIABLE ="user_action"
TRAIN_TEST_SPLIT=0.5
HIDDEN_LAYER_SIZE=30
raw_data = pd.read_csv("data.csv")
```

The first four lines are just imports, the next three are hyperparameters. The `TARGET_VARIABLE` tells Python what is the target variable we wish to predict. The last line opens the file `data.csv`. Now we must make the train–test split. We have a hyperparameter that currently leaves 0.5 of the datapoints in the training set, but you can change this hyperparameter to something else. Just be careful since we have a tiny dataset which might cause some problems if the split is something like 0.95. The code for the train–test split is:

```
mask = np.random.rand(len(raw_data)) < TRAIN_TEST_SPLIT
tr_dataset = raw_data[mask]
te_dataset = raw_data[~mask]
```

The first line here defines a random sampling of the data to be used to get the train–test split and the next two lines select the appropriate sub-dataframes from the original Pandas dataframe (a dataframe is a table-like object, very similar to an Numpy array but Pandas focuses on easy reshaping and splitting, while Numpy focuses on fast computation). The next lines split both the train and test dataframes into labels and data, and then convert them into Numpy arrays, since Keras needs Numpy arrays to work. The process is relatively painless:

```
tr_data = np.array(raw_data.drop(TARGET_VARIABLE,
axis=1))
tr_labels = np.array(raw_data[[TARGET_VARIABLE]])
te_data = np.array(te_dataset.drop(TARGET_VARIABLE,
axis=1))
te_labels = np.array(te_dataset[[TARGET_VARIABLE]])
```

Now, we move to the Keras specification of a neural network model, and its compilation and training (fitting). We need to compile the model since we want Keras to fill in the nasty details and create arrays of appropriate dimensionality of the weight and bias matrices:

```
ffnn = Sequential()
ffnn.add(Dense(HIDDEN_LAYER_SIZE, input_shape=(3,),
activation="sigmoid"))
ffnn.add(Dense(1, activation="sigmoid"))
ffnn.compile(loss="mean_squared_error", optimizer=
"sgd", metrics=['accuracy'])
ffnn.fit(tr_data, tr_labels, epochs=150, batch_size=2,
verbose=1)
```

The first line initializes a new sequential model in a variable called `ffnn`. The second line specifies both the input layer (to accept 3D vectors as single data inputs), and the hidden layer size which is specified at the beginning of the file in the variable `HIDDEN_LAYER_SIZE`. The third line will take the hidden layer size from the previous layer (Keras does this automatically), and create an output layer with one neuron. All neurons will be having sigmoid or logistic activation functions. The fourth line specifies the error function (MSE), the optimizer (stochastic gradient descent) and which metrics to calculate. It also compiles the model, which means that it will assemble all the other stuff that Python needs from what we have specified. The last line trains the neural network on `tr_data`, using `tr_labels`, for 150 epochs, taking two samples in a mini-batch. `verbose=1` means that it will print the accuracy and loss after each epoch of training. Now we can continue to analyze the results on the test set:

```
metrics = ffnn.evaluate(te_data, te_labels, verbose=1)
print("%s: %.2f%%" % (ffnn.metrics_names[1],
metrics[1]*100))
```

The first line evaluates the model on `te_data` using `te_labels` and the second prints out accuracy as a formatted string. Next, we take in the `new_data.csv` file which simulates new data on our website and we try to predict what will happen using the `ffnn` trained model:

```
new_data = np.array(pd.read_csv("new_data.csv"))
results = ffnn.predict(new_data)
print(results)
```

References

1. M. Hassoun, *Fundamentals of Artificial Neural Networks* (MIT Press, Cambridge, 2003)
2. I.N. da Silva, D.H. Spatti, R.A. Flauzino, L.H.B. Liboni, S.F. dos Reis Alves, *Artificial Neural Networks: A Practical Course* (Springer, New York, 2017)
3. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, Cambridge, 2016)
4. G. Montavon, G. Orr, K.R. Müller, *Neural Networks: Tricks of the Trade* (Springer, New York, 2012)
5. M. Minsky, S. Papert, *Perceptrons: An Introduction to Computational Geometry* (MIT Press, Cambridge, 1969)
6. P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences* (Harvard University, Cambridge, 1975)
7. D.B. Parker, Learning-logic. Technical Report-47 (MIT Center for Computational Research in Economics and Management Science, Cambridge, 1985)
8. Y. LeCun, Une procédure d'apprentissage pour réseau a seuil asymmetrique. Proc. Cogn. **85**, 599–604 (1985)
9. D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning internal representations by error propagation. Parallel Distrib. Process. **1**, 318–362 (1986)

Modifications and Extensions to a Feed-Forward Neural Network

5

5.1 The Idea of Regularization

Let us recall the ideas of variance and bias. If we have two classes (denoted by X and O) in a 2D space and the classifier draws a very straight line we have a classifier with a high bias. This line will generalize well, meaning that the classification error for the new points (test error) will be very similar to the classification error for the old points (training error). This is great, but the problem is that the error will be too large in the first place. This is called *underfitting*. On the other hand, if we have a classifier that draws an intricate line to include every X and none of the Os, then we have high variance (and low bias), which is called *overfitting*. In this case, we will have a relatively low training error a much larger testing error.

Let us take and abstract example. Imagine that we have the task of finding orcas among other animals. Then our classifier should be able to locate orcas by using the properties that are common to *all* orcas but not present in other animals. Notice that when we said 'all' we wanted to make sure we are identifying the species, not a subgroup of the specie: e.g. having a blue tag on the tail might be something that some orcas have, but we want to catch only those things that all orcas have and no other animal has. A 'species' in general is called a *type* (e.g. orcas), whereas an individual is called a *token* (e.g. the orca Shamu). We want to find a property that defines the type we are trying to classify. We call such a property a *necessary property*. In case of orcas this might be simply the property (or query):

$$\texttt{orca}(x) := \texttt{mammal}(x) \wedge \texttt{livesInOcean}(x) \wedge \texttt{blackAndWhite}(x)$$

But, sometimes it is not that easy to find such a property. Trying to find such a property is what a supervised machine learning algorithm does. So the problem might be rephrased as trying to find a complex property which defines a type as best as possible (by trying to include the biggest possible number of tokens and try

to include only the relevant tokens in the definition). Therefore, overfitting can be understood in another way: our classifier is so good that we are not only capturing the necessary properties from our training examples, but also the non-necessary or *accidental properties*. So, we would like to capture all the properties which we need, but we want something to help us stop when we begin including the non-necessary properties.

Underfitting and overfitting are the two extremes. Empirically speaking, we can really go from high bias and low variance to high variance and low bias. Want to stop at a point in between, and we want this point to have better-than-average generalization capabilities (inherited from the higher bias), and a good fit to the data (inherited from high variance). How to find this 'sweet spot' is the art of machine learning, and the received wisdom in the machine learning community will insist it is best to find this by hand. But it is not impossible to automate, and deep learning, wanting to become a contender for artificial intelligence, will automate as much as possible. There is one approach which tries to automate our intuitions about overfitting, and this idea is called *regularization*.

Why are we talking about overfitting and not underfitting? Remember that if have a very high bias we will end up with a linear classifier, and linear classifiers cannot solve the XOR or similar simple problems. What we want then is to significantly lower the bias until we have reached the point after which we are overfitting. In the context of deep learning, after we have added a layer to logistic regression, we have said farewell to high bias and sailed away towards the shores of high variance. This sounds very nice, but how can we stop in time? How can we prevent overfitting. The idea of regularization is to add a regularization parameter to the error function E, so we will have

$$E^{improved} := E^{original} + RegularizationTerm$$

Before continuing to the formal definitions, let us see how we can develop a visual intuition on what regularization does (Fig. 5.1).

The left-hand side of the image depicts the classical various choices of hyperplanes we usually have (bias, variance, etc.). If we add a regularization term, the effect will be that the error function will not be able to pinpoint the datapoints *exactly*, and the

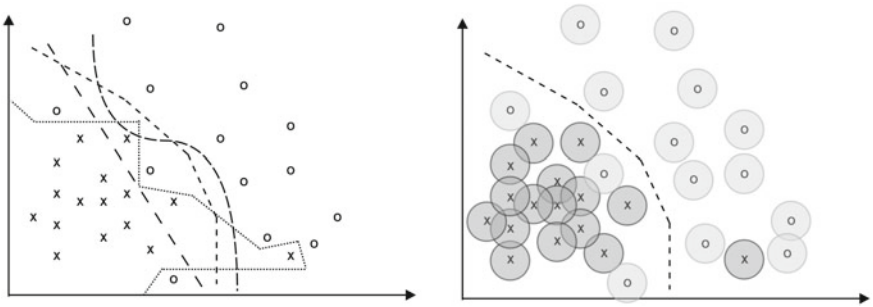


Fig. 5.1 Intuition about regularization

effect will be similar to the points becoming actually little circles. In this way, some of the choices for the hyperplane will simply become impossible, and the one that are left will be the ones that have a good "neutral zone" between Xs and Os. This is not the exact explanation of regularization (we will get to that shortly) but an intuition which is useful for informal reasoning about what regularization does and how it behaves.

5.2 L_1 and L_2 Regularization

As we have noted earlier, regularization means adding a term to the error function, so we have:

$$E^{improved} := E^{original} + RegularizationTerm$$

As one might guess, adding different regularization terms give rise to different regularization techniques. In this book, we will address the two most common types of regularization, L_1 and L_2 regularization. We will start with L_2 regularization and explore it in detail, since it is more useful in practice and it is also easier to grasp the connections with vector spaces and the intuition we developed in the previous section. Afterwards we will turn briefly to L_1 and later in this chapter we will address dropout which is a very useful technique unique to neural networks and has effects similar to regularization.

L_2 regularization is known under many names, 'weight decay', 'ridge regression', and 'Tikhonov regularization'. L_2 regularization was first formulated by the Soviet mathematician Andrey Tikhonov in 1943 [1], and was further refined in his paper [2]. The idea of L_2 regularization is to use the L_2 or *Euclidean norm* for the regularization term.

The L_2 norm of a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is simply $\sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$. The L_2 norm of the vector $\mathbf{x}$ can be denoted by $L_2(\mathbf{x})$ or, more commonly, by $||\mathbf{x}||_2$. The vector used is the weights of the final layer, but a version using all weights in the network can also be used (but in that case, our intuition will be off). So now we can rewrite the *preliminary* L_2-regularized error function as:

$$E^{improved} := E^{original} + ||\mathbf{w}||_2$$

But, in the machine learning community, we usually do not use the square root, so instead of $||\mathbf{w}||_2$ we will use the square of the L_2 norm, i.e. $(||\mathbf{w}||_2)^2 = ||\mathbf{w}||_2^2$ which is actually just $\sum_w w^2$. We will also want to add a hyperparameter to be able to adjust how much of the regularization we want to use (called the *regularization parameter* or *regularization rate*, and denoted by λ), and divide it by the size of the

batch used (to account for the fact that we want it to be proportional), so the final L_2-regularized error function is:

$$E^{improved} := E^{original} + \frac{\lambda}{n}||\mathbf{w}||_2^2 = E^{original} + \frac{\lambda}{n}\sum_{w_i in w_o} w_i^2$$

Let us work a bit on the explanation[1] what L_2 regularization does. The intuition is that during the learning procedure, smaller weights will be preferred, but larger weights will be considered if the overall decrease in error is significant. This explains why it is called 'weight decay'. The choice of λ determines how much will small weights be preferred (when λ is large, the preference for small weights will be great). Let us work through a simple derivation. We start with our regularized error function:

$$E^{new} = E^{old} + \frac{\lambda}{n}\sum_{w} w^2$$

By taking the partial derivatives of this equation we get:

$$\frac{\partial E^{new}}{\partial w} = \frac{\partial E^{old}}{\partial w} + \frac{\lambda}{n}w$$

Taking this back to the general weight update rule we get:

$$w^{new} = w^{old} - \eta \cdot (\frac{\partial E^{old}}{\partial w} + \frac{\lambda}{n}w)$$

One might wonder whether this would actually make the weights converge to 0, but this is not the case, since the first component $\frac{\partial E^{old}}{\partial w}$ will increase the weights if the reduction in error (this part controls the unregularized error) is significant.

We can now proceed to briefly sketch L_1 regularization. L_1 regularization, also known as 'lasso' or 'basis pursuit denoising' was first proposed by Robert Tibshirani in 1996 [4]. L_1 regularization uses the absolute value instead of the squares:

$$E^{improved} := E^{original} + \frac{\lambda}{n}||\mathbf{w}||_1 = E^{original} + \frac{\lambda}{n}\sum_{w_i in w_o} |w_i|$$

Let us compare the two regularizations to expose their peculiarities. For most classification and prediction problems, L_2 is better. However, there are certain tasks where L_1 excels [5]. The problems where L_1 is superior are those that contain a lot of irrelevant data. This might be either very noisy data, or features that are not informative, but it can also be sparse data (where most features are irrelevant because

[1] We will be using a modification of the explanation offered by [3]. Note that this book is available online at http://neuralnetworksanddeeplearning.com.

they are missing). This means that there are a number of useful applications of L_1 regularization in signal processing (e.g. [6]) and robotics (e.g. [7]).

Let us try to develop an intuition behind the two regularizations. The L_2 regularization tries to push down the square of the weights (which does not increase linearly as the weights increase), whereas L_1 is concerned with absolute values which is linear, and therefore L_2 will quickly penalize large weights (it tends to concentrate on them). L_1 regularization will make much more weights slightly smaller, which usually results in many weights coming close to 0. To simplify the matter completely, take the plots of the graphs $f(x) = x^2$ and $g(x) = |x|$. Imagine that those plots are physical surfaces like bowls. Now imagine putting some points in the graphs (which correspond to the weights) and adding 'gravity', so that they behave like physical objects (tiny marbles). The 'gravity' corresponds to gradient descent, since it is a move towards the minimum (just like gravity would push to a minimum in a physical system). Imagine that there is also friction, which corresponds to the idea that E does not care anymore about the weights that are already very close to the minimum. In the case of $f(x)$, we will have a number of points around the point $(0, 0)$, but a bit dispersed, whereas in $g(x)$ they would be very tightly packed around the $(0, 0)$ point. We should also note that two vectors can have the same L_1 norm but different L_2 norms. Take $\mathbf{v}_1 = (0.5, 0.5)$ and $\mathbf{v}_2 = (-1, 0)$. Then $||\mathbf{v}_1||_1 = |0.5| + |0.5| = 1$ and $||\mathbf{v}_2||_1 = |-1| + |0| = 1$, but $||\mathbf{v}_1||_2 = \sqrt{0.5^2 + 0.5^2} = \frac{1}{\sqrt{2}}$ and $||\mathbf{v}_2||_2 = \sqrt{1^2 + 0^2} = 1$.

5.3 Learning Rate, Momentum and Dropout

In this section, we will revisit the idea of the learning rate. The learning rate is an example of a *hyperparameter*. The name is quite unusual, but there is actually a simple reason behind it. Every neural network is actually a function which assigns to a given input vector (input) a class label (output). The way the neural network does this is via the operations it performs and the parameters it is given. Operations include the logistic function, matrix multiplication, etc., while the parameters are all numbers which are not inputs, viz. weights and biases. We know that the biases are simply weights and that the neural network finds a good set of weights by backpropagationg the errors it registers. Since operations are always the same, this means that all of the learning done by a neural network is actually a search for a good set of weight, or in other words, it is simply adjusting its parameters. There is nothing more to it, no magic, just weight adjusting. Now that this is clear, it is easy to say what a hyperparameter is. A hyperparameter is any number used in the neural network which cannot be learned by the network. An example would be the learning rate or the number of neurons in the hidden layer.

This means that learning cannot adjust hyperparameters, and they have to be adjusted manually. Here machine learning leans heavily towards art, since there is no scientific way to do it, it is more a matter of intuition and experience. But despite the fact that finding a good set of hyperparameters is not easy, there is a standard

procedure how to do it. To do this, we must revisit the idea of splitting the data set in a training set and a testing set. Suppose we have kept 10% of the datapoints for testing, and the rest we wanted to use as the training set. Now we will take another 10% of datapoints from the training set and call it a *validation set*. So we will have 80% of the datapoints in the training set for training, 10% we use for a validation set, and 10% we keep for a test set. The idea is to train on the train set with a given set of hyperparameters and test it on the validation set. If we are not happy, we re-train the network and test the validation set again. We do this until we get a good classification. Then, and only then we test on the test set to see how it is doing.

Remember that a low train error and a high test error is a sign of overfitting. When we are just training and testing (with no hyperparameter tuning), this is a good rule to stick to. But if we are tuning hyperparameter, we might get overfitting to both the training and validation set, since we are changing the hyperparameters *until* we get a small error on the validation set. If the errors can become misleadingly small since the classifier learns the noise of the training set, and we manually change the hyperparameters to suit the noise of the validation set. If, after this, there is proportionately small error on the test set, we have a winner, otherwise it is back to the drawing board. Of course, it is possible to alter the sizes of the train, validation and test sets, but these are the standard starting values (80%, 10% and 10% respectively).

We return to the learning rate. The idea of including a learning rate was first explicitly proposed in [8]. As we have seen in the last chapter, the learning rate controls how much of the update we want, since the learning rate is part of the general weight update rule, i.e. it comes into play in the very end of backpropagation. Before turning to the types of the learning rate, let us explore why the learning rate is important in an abstract setting.[2] We will construct an abstract model of learning by generalizing the idea with the parabola we proposed in the previous section. We need to expand this to three dimensions just so we can have more than one way to move. The overall shape of the 3D surface we will be using is like a bowl (Fig. 5.2).

Its lateral view is given by the axes x and y (we do not see z). Seen from the top (axes x and z visible, axis y not visible), it looks like a circle or ellipse. When we 'drop' a point at (x_k, z_k), it will get the value y_k from the curve at the coordinates (x_k, z_k). In other words, it will be as if we drop the point and it falls towards the bowl and stops as soon as it meets the surface of the bowl (imagine that our point is a sticky object, like a chewing gum). We drop it at a precise (x_k, z_k) (this is the 'top view'), we do not know the final 'height' of the sticky object, but we will measure it when it falls to the side of the bowl.

The gradient is like gravity, and it tries to minimize y. If we want to continue our analogy, we must make a couple of changes to the physical world: (i) we will not have sticky objects all the time (we needed them to explain how can we get the y of a point if we only have (x, z)), but little marbles which turn to sticky objects when they have finished their move (or you may think that they 'freeze'), (b) there is no

[2] We take the idea for this abstraction from Geoffrey Hinton's courses.

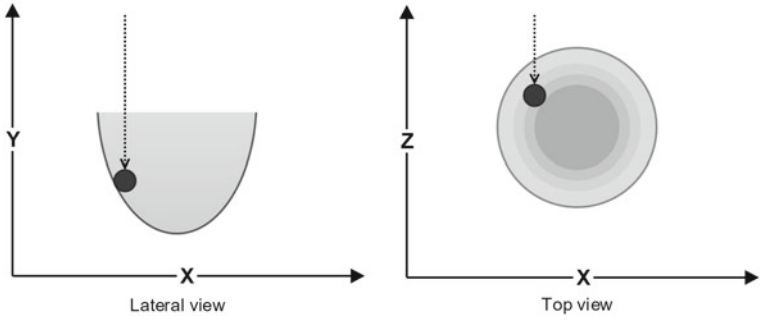


Fig. 5.2 Gradient bowl

friction or inertia and, perhaps the most counterintuitive, (c) our gravity is similar to physical gravity but different.

Let us explain (c) in more detail. Suppose we are looking from the top, so we see only axes x and z and we drop a marble. We want our gravity to behave like physical gravity in the sense that it will automatically generate the direction the marble has to move (looking from the top, the x and z view) so that it moves along the curvature of the bowl which is, hopefully, the direction of the bottom of the bowl (the global minimum value for y).

We want it to be different to physical gravity so that the amount of movement in this direction is not determined by the exact position of the minimum for y, i.e. it does not settle in the bottom but may move on the other side of the bowl (and remains there as if it became a sticky object again). We leave the amount of movement unspecified at the moment, but assume it is rarely the *exact* amount needed to reach the actual minimum: sometimes it is a bit more and it overshoots and sometimes is a bit less and it fails to reach it. One very important point has to be made here: the curvature 'points' at the minimum, but we are following the curvature at the point we currently are, and not the minimum. In a sense, the marble is extremely 'short-sighted' (marbles usually are): it sees only the current curvature and moves along it. We will know we have found the minimum when we have the curvature of 0. Note that in our example we have an 'idealized bowl', which has only one point where the curvature is 0, and that is the global minimum for y. Imagine how many more complex surfaces there might be where we cannot say that the point of curvature 0 is the global minimum, but also note that if we could have a transformation which transforms any of these complex surfaces into our bowl we would have a perfect learning algorithm.

Also, we want to add a bit of imprecision, so imagine that the direction of our gravity is the 'general direction' of the curvature of the bowl—sometimes a bit to the left, sometimes a bit to the right of the minimum, but only on rare occasions follows precisely the curvature.

Now we have the perfect setting for explaining learning in the abstract sense. Each epoch of learning is one move (of some amount) in the 'general direction' of the curvature of the bowl, and after it is done, it sticks where it is. The second epoch

'unfreezes' the situation, and again the general direction towards of the curvature is followed. this second move might either be the continuation of the first, or a move in an almost opposite direction if the marble overshot the minimum (bottom). The process can continue indefinitely, but after a number of epochs the moves will be really small and insignificant, so we can either stop after a predetermined number of epochs or when the improvement is not significant.[3]

Now let us return to the learning rate. The learning rate controls how much of the amount of movement we are going to take. A learning rate of 1 means to make the whole move, and a learning rate of 0.1 means to make only 10% of the move. As mentioned earlier, we can have a global learning rate or parametrized learning rate so that it changes according to certain conditions we specify such as the number of epochs so far, or some other parameter.

Let us return a bit to our bowl. So far we had a round bowl, but imagine we have a shallow bowl of the shape of an elongated ellipse (Fig. 5.3). If we drop the marble near the narrow middle, we will have almost the same situation as before. But if we drop it on the marble at the top left portion, it will move along a very shallow curvature and it will take a very large number of epochs to find its way towards the bottom of the bowl. The learning rate can help here. If we take only a fraction of the move, the direction of the curvature for the next move will be considerably better than if we move from one edge of a shallow and elongated bowl to the opposing edge. It will make smaller steps but it will find a good direction much more quickly.

This leaves us with discussing the typical values for the learning rate η. The values most often used are 0.1, 0.01, 0.001, and so on. Values like 0.03 will simply get lost and behave very similarly to the closest logarithm, which is 0.01 in case of 0.03.[4] The learning rate is a hyperparameter, and like all hyperparameters it has to be tuned on the validation set. So, our suggestion is to try with some of the standard values for a given hyperparameter and then see how it behaves and modify it accordingly.

We turn our attention now to an idea similar to the learning rate, but different called *momentum*, also called *inertia*. Informally speaking, the learning rate controls

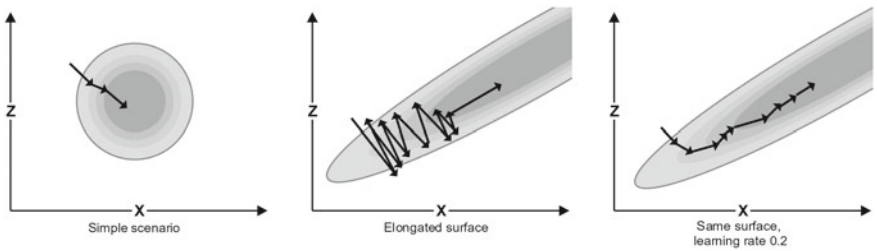


Fig. 5.3 Learning rate

[3]This is actually also a technique which is used to prevent overfitting called *early stopping*.

[4]You can use the learning rate to force a gradient explosion, so if you want to see gradient explosion for yourself try with an η value of 5 or 10.

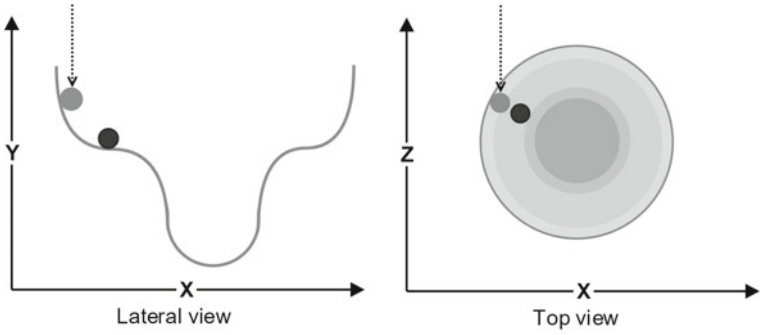


Fig. 5.4 Local minimum

how much of the move to keep in the present step, while momentum controls how much of the move from the previous step to keep in the current step. The problem which momentum tries to solve is the problem of *local minima*. Let us return to our idea with the bowl but now let us modify the bowl to have local minima. You can see the lateral view in Fig. 5.4. Notice that the learning rate was concerned with the 'top' view whereas the momentum addresses problems with the 'lateral' view.

The marble falls down as usual (depicted as grey in the image) and continues along the curvature, and stops when the curvature is 0 (depicted by black in the image). But the problem is that the curvature 0 is not necessarily the global minimum, it is only local. If it were a physical system, the marble would have momentum and it would fall over the local minimum to a global minimum, there it would go back and forth a bit and then it would settle. Momentum in neural networks is just the formalization of this idea. Momentum, like the learning rate is added to the general weight update rule:

$$w_i^{new} = w_i^{old} - \eta \frac{\partial E}{\partial w_i^{old}} + \mu(|w_i^{old} - w_i^{older}|)$$

Where w_i^{new} is the current weight to be computed, w_i^{old} is the previous value of the weight and w_i^{older} was the value of the weight before that. μ is the *momentum rate* and ranges from 0 to 1. It directly controls how much of the previous change in weight we will keep in this iteration. A typical value for μ is 0.9, and should be adjusted usually to a value between 0.10 and 0.99. Momentum is as old as the last discovery of backpropagation, and it was first published in the same paper by Rumelhart, Hinton and Williams [9].

There is one final interesting technique for improving the way neural networks learn and reduce overfitting, named *dropout*. We have chosen to define regularization as adding a regularization term to the cost function, and according to this definition dropout is not regularization, but it does lower the gap between the training error and the testing error, and consequently it reduces overfitting. One could define regularization to be any technique that reduces this spread, and then dropout would be a regularization technique. One could call dropout a 'structural regularization' and

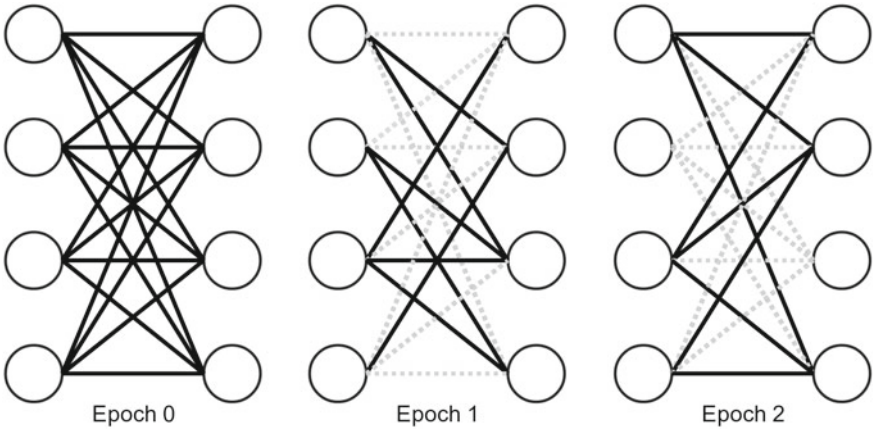


Fig. 5.5 Dropout with $\pi = 0.5$

the L_1 and L_2 regularizations 'numerical regularizations', but this is not standard terminology and we will not be using it.

Dropout was first explained in [10], but one could find more details about it in [11] and especially [12]. Dropout is a surprisingly simple technique. We add a dropout parameter π ranging from 0 to 1 (to be interpreted as a probability), and in each epoch every weight is set to zero with a probability of π (Fig. 5.5). Returning to the general weight update rule (where we need a w_k^{old} for calculating the weight updates), if in epoch n the weight w_k was set to zero, the w_k^{old} for epoch $n+1$ will be the w_k from epoch $n-1$. Dropout forces the network to learn redundancies so it is better in isolating the necessary properties of the dataset. A typical value for π is 0.2, but like all other hyperparameters it has to be tuned on the validation set.

5.4 Stochastic Gradient Descent and Online Learning

So far in this book, we have been a bit clumsy with one important question[5]: how does backpropagation work from a 'bird's-eye view'. We have been avoiding this question to avoid confusion until we had enough conceptual understanding to address it, and now we know enough to state it clearly. Backpropagation in the neural network works in the following way: we take one training sample at a time and pass it through the network and record the squared error for each. Then we use it to calculate the mean (squared) error. Once we have the mean squared error, we backpropagate it using gradient descent to find a better set of weights. Once we are done, we have

[5]We have been clumsy around several things, and this section is intended to redefine them a bit to make them more precise.

finished one epoch of training. We may do this for as many epochs we want. Usually, we want to continue either for a fixed number of epochs or stop it if it does not help with decreasing the error anymore.

What we have used when explaining backpropagation was a training set of size 1 (a single example). If this is the whole training set (a weirdly small training set), this would be an example of (full) gradient descent (also called *full-batch learning*). We could however think of it as being a subset of the training set. When using a randomly selected subset of from the training set of the size n, we say we use *stochastic gradient descent* or *minibatch learning* (with batch size n). Learning with a minibatch of size 1 is called *online learning*. Online learning can be either 'stationary' with fixed training set and then selecting randomly[6] one by one, or simply giving new training samples as they come along.[7] So we could think of our example backpropagation from the last chapter as an instance of online learning.

Now we are also in position to introduce a terminological finesse we have been neglecting until now. An *epoch* is one complete forward and backward pass over the *whole* training set. If we divide the training set of the size 10000 in 10 minibatches,[8] then one forward and one backward pass over a batch is called one *iteration*, and ten iterations (the size of the minibatch) is one *epoch*. This will hold only if the samples are divided as we stated in the footnote. If we use a random selection of training samples for the minibatch, then ten iterations will not make one epoch. If, on the other hand, we shuffle the training set and then divide it, then ten iterations will make one epoch and the forces fighting for order in the universe will be triumphant once more.

Stochastic gradient descent is usually much quicker to converge, since by random sampling we can get a good estimate of the overall gradient, but if the minimum is not pronounced (the bowl is too shallow) it tends to compound the problems we have seen previously in Fig. 5.3 (the middle part) in the previous section. The intuitive reason behind it is that when we have a shallow curvature and sample the surface randomly we will be prone to losing the little amount of information about the curvature that we had in the beginning. In such cases, full gradient descent couple with momentum might be a good choice.

---

[6] We could use also a non-random selection. One of the most interesting ideas here is that of learning the simplest instances first and then proceeding to the more tricky ones, and this approach is called *curriculum learning*. For more on this see [13].

[7] This is similar to *reinforcement learning*, which is, along with supervised and unsupervised learning one of the three main areas of machine learning, but we have decided against including it in this volume, since it falls outside of the the idea of a *first* introduction to deep learning. If the reader wishes to learn more, we refer her to [14].

[8] Suppose for the sake of clarification it is non-randomly divided: the first batch contains training samples 1 to 1000, the second 1001 to 2000, etc.

5.5 Problems for Multiple Hidden Layers: Vanishing and Exploding Gradients

Let us return to the calculation of the fully functional feed-forward neural network from the last chapter. Remember it was a neural network with the configuration (2, 2, 1), meaning it has two input neurons, two hidden neurons[9] and one output neuron. Let us revisit the weight updates we calculated:

- $w_1^{old} = 0.1, w_1^{new} = 0.1007$
- $w_2^{old} = 0.5, w_2^{new} = 0.502$
- $w_3^{old} = 0.4, w_3^{new} = 0.4024$
- $w_4^{old} = 0.3, w_4^{new} = 0.307$
- $w_5^{old} = 0.2, w_5^{new} = 0.2373$
- $w_6^{old} = 0.6, w_6^{new} = 0.6374$

Just by looking at the amount of the weight update you might notice that two weights have been updated with a significantly larger amount than the other weights. These two weights (w_5 and w_6) are the weights connecting the output layer with the hidden layer. The rest of the weights connect the input layer with the hidden layer. But why are they larger? The reason is that we had to backpropagate through few layers, and they remained larger: backpropagation is, structurally speaking, just the chain rule. The chain rule is just multiplication of derivatives. And, derivatives of everything we needed[10] have values between 0 and 1. So, by adding layers through which we had to backpropagate, we needed to multiply more and more 0 to 1 numbers, and this generally tends to become very small very quickly. And this is without regularization, with regularization it would be even worse, since it would prefer small weights at all times (since the weight updates would be small because of the derivatives, there would be little chance of the unregularized part to increase the weights). This phenomena is called *vanishing gradient*.

We could try to circumvent this problem by initializing the weights to a very large value and hope that backpropagation will just chip them to the correct value.[11] In this case, we might get a very large gradient which would also hinder learning since a step in the direction of the gradient would be the right direction but the magnitude of the step would take us farther away from the solution than we were before the step. The moral of the story is that usually the problem is the vanishing gradient, but

[9] A single hidden layer with two neurons in it. It it was (3, 2, 4, 1) we would know it has two hidden layer, the first one with two neurons and the second one with four.

[10] Ok, we have used the adjusted the values to make this statement true. Several of the derivatives we need will become a value between 0 and 1 soon, but it the sigmoid derivatives are mathematically bound between 0 and 1, and if we have many layers (e.g. 8), the sigmoid derivatives would dominate backpropagation.

[11] If the regular approach was something like making a clay statue (removing clay, but sometimes adding), the intuition behind initializing the weights to large values would be taking a block of stone or wood and start chipping away pieces.

if we change radically our approach we would be blown in the opposite direction which is even worse. Gradient descent, as a method, is simply too unstable if we use many layers through which we need to backpropagate.

To put the importance of the the vanishing gradient problem, we must note that the vanishing gradient is *the* problem to which deep learning is the solution. What truly defines deep learning are the techniques which make possible to stack many layers and yet avoid the vanishing gradient problem. Some deep learning techniques deal with the problem head on (LSTM), while some are trying to circumvent it (convolutional neural networks), some are using different connections than simple neural networks (Hopfield networks), some are hacking the solution (residual connections), while some have been using weird neural network phenomena to gain the upper hand (autoencoders). The rest of this book is devoted to these techniques and architectures. Historically speaking, the vanishing gradient was first identified by Sepp Hochreiter in 1991 in his diploma thesis [15]. His thesis advisor was Jürgen Schmidhuber, and the two will develop one of the most influential recurrent neural network architectures (LSTM) in 1997 [16], which we will explore in detail in the following chapters. An interesting paper by the same authors which brings more detail to the discussion of the vanishing gradient is [17].

We make a final remark before continuing to the second part of this book. We have chosen what we believe to be the most popular and influential neural architectures, but there are many more and many more will be discovered. The aim of this book is not to provide a comprehensive view of everything there is or will be, but to help the reader acquire the knowledge and intuition needed to pursue research-level deep learning papers and monographs. This is not a final tome about deep learning, but a first introduction which is necessarily incomplete. We made a serious effort to include a range of neural architectures which will demonstrate to the reader the vast richness and fulfilling diversity of this amazing field of cognitive science and artificial intelligence.

References

1. A.N. Tikhonov, On the stability of inverse problems. Dokl. Akad. Nauk SSSR **39**(5), 195–198 (1943)
2. A.N. Tikhonov, Solution of incorrectly formulated problems and the regularization method. Sov. Math. **4**, 1035–1038 (1963)
3. M.A. Nielsen, *Neural Networks and Deep Learning* (Determination Press, 2015)
4. R. Tibshirani, Regression shrinkage and selection via the lasso. J. Roy. Stat. Soc. Ser B (Methodol.) **58**(1), 267–288 (1996)
5. A. Ng, Feature selection, L1 versus L2 regularization, and rotational invariance, in *Proceedings of the International Conference on Machine Learning* (2004)
6. D.L. Donoho, Compressed sensing. IEEE Trans. Inf. Theory **52**(4), 1289–1306 (2006)
7. E.J. Candes, J. Romberg, T. Tao, Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information. IEEE Trans. Inf. Theory **52**(2), 489–509 (2006)

8. J. Wen, J.L. Zhao, S.W. Luo, Z. Han, The improvements of BP neural network learning algorithm, in *Proceedings of 5th International Conference on Signal Processing* (IEEE Press, 2000), pp. 1647–1649
9. D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning internal representations by error propagation. Parallel Distrib. Process. **1**, 318–362 (1986)
10. G.E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Improving neural networks by preventing co-adaptation of feature detectors (2012)
11. G.E. Dahl, T.N. Sainath, G.E. Hinton, Improving deep neural networks for LVCSR using rectified linear units and dropout, in *IEEE International Conference on Acoustic Speech and Signal Processing* (IEEE Press, 2013), pp. 8609–8613
12. N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**, 1929–1958 (2014)
13. Y. Bengio, J. Louradour, R. Collobert, J. Weston, Curriculum learning, in *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, New York, NY, USA*, (ACM, 2009), pp. 41–48
14. R.S. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, Cambridge, 1998)
15. S. Hochreiter, Untersuchungen zu dynamischen neuronalen Netzen, Diploma thesis, Technische Universität Munich, 1991
16. S. Hochreiter, J. Schmidhuber, Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
17. S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, in *A Field Guide to Dynamical Recurrent Neural Networks*, ed. by S.C. Kremer, J.F. Kolen (IEEE Press, 2001)

Convolutional Neural Networks 6

6.1 A Third Visit to Logistic Regression

In this chapter, we explore convolutional neural networks, which were first invented by Yann LeCun and others in 1998 [1]. The idea which LeCun and his team implemented was older, and built up on the ideas of David H. Hubel and Torsten Weisel presented in their 1968 seminal paper [2] which won them the 1981 Nobel prize in Physiology and Medicine. They explored the animal visual cortex and found connections between activities in a small but well-defined area of the brain and activities in small regions of the visual field. In some cases, it was even possible to pinpoint exact neurons that were in charge of a part of the visual field. This led them to the discovery of the *receptive field*, which is a concept used to describe the link between parts of the visual fields and individual neurons which process the information.

The idea of a receptive field completes the third and final component we need to build convolutional neural networks. But what were the other two part we have? The first was a technical detail: flattening images (2D arrays) to vectors. Even though most modern implementations deal readily with arrays, under the hood they are often flattened to vectors. We adopt this approach in our explanation since it has less handwaiving, and enables the reader to grasp some technical details along the way. You can see an illustration of flattening a 3 by 3 image in the top of Fig. 6.1. The second component is the one that will take the image vector and give it to a single workhorse neuron which will be in charge of processing. Can you figure out what can we use?

If you said 'logistic regression', you were right! We will however be using a different activation function, but the structure will be the same. A convolutional neural network is a neural network that has one or more convolutional layers. This is not a hard definition, but a quick and simple one. There will be architectures using

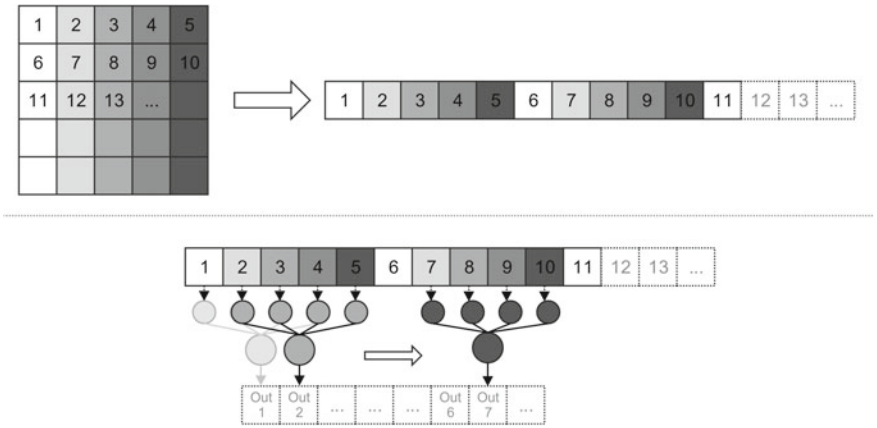


Fig. 6.1 Building a 1D convolutional layer with a logistic regression

convolutional layers which will not be called 'convolutional neural networks'.[1] So now we have to describe what a convolutional layer is.

A convolutional layer takes an image[2] and a small logistic regression with e.g. input size 4 (these sizes are usually 4 or 9, sometimes 16) and passes the logistic regression over the whole image. This means that the first input consists of components 1–9 of the flattened vector, the second input are the components 2–10, the third are components 3–11, and so on. You can see an overview of the process in the bottom of Fig. 6.1. This process creates an output vector which is smaller than the overall input vector, since we start at component 1, but take four components, and produce a single output. The end result is that if we were to move along a 10-dimensional vector with the logistic regression (this logistic regression is called *local receptive field* in convolutional neural networks), we would produce a 7-dimensional output vector (see the bottom of Fig. 6.1). This type of convolutional layer is called a *1D convolutional layer* or a *temporal convolutional layer*. It does not have to use a time series (it can use any data, since you can flatten out any data), but the name is here to distinguish it from a classical 2D convolutional layer.

We can take also a different approach and say we want the output dimension to be same as the input, but then our 4-dimensional local receptive field would have to start at input at 'cells' $-1, 0, 1, 2$ and then continue to $0, 1, 2, 3$, and so on, finishing at $9, 10, 11$ (you can draw it yourself to see why we do not need to go to 12). Putting

[1] Yann LeCun once told in an interview that he prefers the name 'convolutional network' rather than 'convolutional neural network'.

[2] An image in this sense is any 2D array with values between 0 and 255. In Fig. 6.1 we have numbered the positions, and you may think of them as 'cell numbers', in the sense that they will contain some value, but the number on the image denotes only their order. In addition, note that if we have e.g. 100 by 100 RGB images, each image would be a 3D array (tensor) with dimensions (100, 100, 3). The last dimension of the array would hold the three channels, red, green and blue.

in $-1, 0$ and 11 components to get the output vector to have the same size as the input vector is called *padding*. The additional components usually get values 0, but it sometimes makes sense to take either the values of the first and last components of the image or the average of all values. The important thing when padding is to think how not to trick the convolutional layer in learning regularities of the padding. Padding (and some other concepts we discussed) will become much more intuitive when we switch from flattened vectors to non-flattened images. But before we continue, one final comment. We moved the local receptive field one component at a time, but we could move it by two or more. We could even experiment with dynamically changing by how much we move, by moving quicker around the ends and slower towards the centre of the vector. The parameter which says by how many components we move the receptive field between taking inputs is called the *stride* of the convolutional layer.

Let us review the situation in 2D, as if we did not flatten the image into a vector. This is the classical setting for convolutional layers, and such layers are called *2D convolutional layers* or *planar convolutional layers*. If we were to use 3D, we would call it *spatial*, and for 4D or more *hyperspatial*. In the literature is common to refer to the 2D convolutional layer as 'spatial', but this makes one's spider sense tingle.

The logistic regression (local perceptive field) inputs now should be also 2D, and this is the reason why we most often use 4, 9 and 16, since they are squares of 2 by 2, 3 by 3 and 4 by 4 respectively. The stride now represents a move of this square on the image, staring from left, going to the right and after it is finished, one row down, move all the way to the left without scanning, and start scanning from left to right (you can see the steps of this process on the top part of Fig. 6.2). One thing that becomes obvious is that now we will get less outputs. If we use a 3 by 3 local

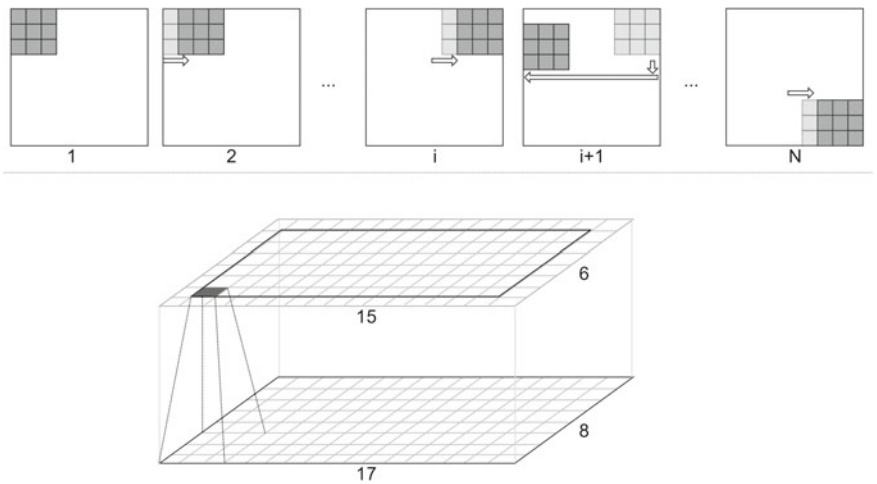


Fig. 6.2 2D Convolutional layer

receptive field to scan a 10 by 10 image, as the output from the local receptive field we will get an 8 by 8 array (see bottom part of Fig. 6.2). This completes a convolutional layer.

A convolutional neural network has multiple layers. Imagine a convolutional neural network consisting of three convolutional layers and one fully connected layer. Suppose it will be processing an image of size 10 and that all three layers have a local receptive field of 3 by 3. Its task is to decide whether a picture has a car in it or not. Let us see how the network works.

The first layer takes a 10 by 10 image, produces an output (it has randomly initialized weights and bias) of size 8 by 8, which is then given to the second convolutional layer (which has its own local receptive field with randomly initialized weights and biases but we have decided to have it also 3 by 3), which produces an output of size 6 by 6, and this is given to the third layer (which has a third local receptive field). This third convolutional layer produces a 4 by 4 image. We then flatten it to a 16-dimensional vector and feed it to a standard fully-connected layer which has one output neuron and uses a logistic function as its nonlinearity. This is actually another logistic regression in disguise, but it could have had more than one output neuron, and then it would not be a proper logistic regression, so we call it a *fully-connected layer of size 1*. The input layer size is not specified and it is assumed to be equal to the output of the previous layer. Then, since it uses the logistic function, it produces an output between 0 and 1 and compares its output to the image label. The error is calculated and backpropagated, and this is repeated for every image in the dataset which completes the training of the network.

Training a convolutional layer means training the local receptive fields of the layers (and weights and biases of fully-connected layers). It has a single bias, and small number of weights (equal to the number of units in the local receptive field). In this respect, it is just like a small logistic regression, and that is what makes convolutional networks quick to train–they have only a small number of parameters to learn. The main structural difference between a logistic regression and a local receptive field is that in a local receptive field we can use any activation function and in logistic regression we are supposed to use the logistic function (if we want to call it 'logistic regression'). The activation function which is most often used is the *rectified linear unit* or ReLU. A ReLU of x is simply the maximal value of 0 and x, meaning that it will return a 0 if the input is negative or the raw input otherwise. In symbols:

$$\rho(x) = max(x, 0) \tag{6.1}$$

Padding in 2D is simply a 'frame' of n pixels around the image. Note that it does not make much sense to use a padding of say 3 (pixels) if we use only a 3 by 3 local receptive field, since it will only go one pixel over the image border.

6.2 Feature Maps and Pooling

Now that we know how a convolutional neural network works, we can use a trick. Recall that a convolutional layer scans a 10 by 10 image with an e.g. 3 by 3 local receptive field (9 weights, 1 bias) and builds a new 8 by 8 'image' as the output. Imagine also that the image has three channels for colours. How would you process an image with three channels? A natural answer is to run over the same receptive field (which has trainable but randomly initialized weights and bias). This is a good strategy. But what if we invert it, and instead of using one local receptive field over three channels, we want to use five local receptive fields over one channel? Remember that a local receptive field is defined by its size and by its weights and bias. The idea here is to keep the same size but initialize the other receptive fields with different weights and biases.

This means that when they scan a 10 by 10 3-channel image, they will construct 15 8 by 8 output images. These images are called *feature maps*. It is like having an 8 by 8 image with 15 channels. This is very useful since only one feature map which learns a good representation (e.g. eyes and noses on pictures of dogs) will boost considerably the overall accuracy of the network[3] (suppose that the task for the whole network was to classify images of dogs and various non-dog objects (i.e. detecting a dog in an image)).

One of the main ideas here is that a 10 by 10 3-channel image turns into an 8 by 8 15-channel image. The input image was transformed into a smaller but deeper object, and this will happen in every convolutional layer.[4] Getting the image smaller, means packing the information in a more compact (but deeper) representation. In our quest for compactness, we may add a new layer after or before a convolutional layer. This new layer is called a *max-pooling* layer. The max-pooling layer takes a pool size as a hyperparameter, usually 2 by 2. It then processes its input image in the following way: divide the image in 2 by 2 areas (like a grid), and take from each four-pixel pool the pixel with the maximal value. Compose these pixels into a new image, with the same order as the original image. A 2 by 2 max-pooling layer produces an image that is half the size of the original image (it does not increase the channel number). Of course, instead of the maximum, a different pixel selection or creation can be devised, such as the average of the four pixels, the minimum, and so on.

The idea behind max-pooling is that important information in a picture is seldom contained in adjacent pixels (this accounts for the 'pick-one-out-of-four' part), and it is often contained in darker pixels (this accounts for using the max). You may notice right away that this is a very strong assumption which may not be generally valid. It must be said that max-pooling is rarely used on images themselves (although it can be used), but rather on learned feature maps, which are images but they are very

[3] Here you might notice how important is weight initialization. We do have some techniques that are better than random initialization, but to find a good weight initialization strategy is an important open research problem.

[4] If using padding we will keep the same size, but still expand the depth. Padding is useful when there is possibly important information on the edges of the image.

peculiar images. You can try to modify the code in the section below to print out feature maps which come out of a convolutional layer.[5] You can think of max-pooling in terms of decreasing the screen resolution. In general, if you recognize a dog on a 1200 by 1600 image, you will probably recognize him on a grainer 600 by 800 image.

Usually a convolutional neural network is composed of a convolutional layer followed by a max-pooling layer, followed by a convolutional layer, and so on. As the image goes through the network, after a number of layers, we get a small image with a lot of channels. Then we can flatten this to a vector and use a simple logistic regression at the end to extract which parts are relevant for our classification problem. The logistic regression (this time with the logistic function) will pick out which parts of the representation will be used for classification and create a result which will be compared with the target and then the error will be backpropagated. This forms a complete convolutional neural network. A simple but fully functional convolutional network with four layers is shown in Fig. 6.3.

Why are convolutional neural networks easier to train? The answer is in the number of parameters used. A five-layer deep fully connected neural network for MNIST has a lot of weights,[6] through which we need to backpropagate. A five-layer convolutional network (containing only convolutional layers) with all receptive fields of 3 by 3 has 45 weight and 5 biases. Notice that this configuration can be used for arbitrarily large images: we do not have to expand the input layer (which is a convolutional layer in our case), but we will need more convolutional layers then to shrink the image. Even if we add feature maps, the training of each feature map is independent of the other, i.e. we can train it in parallel. This makes the process not only computationally fast, but we can also split it across many processors. By contrast, to backpropagate errors

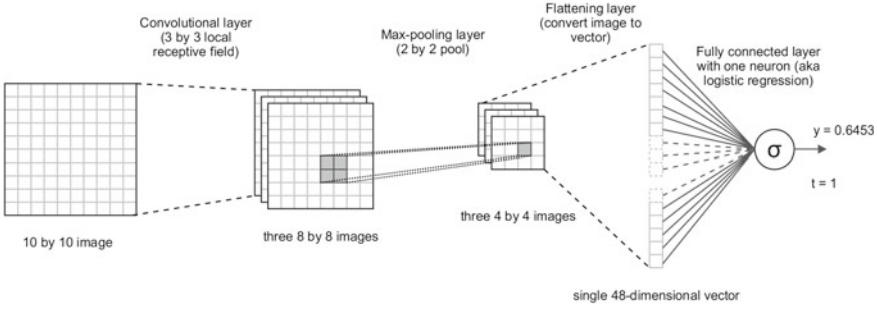


Fig. 6.3 A convolutional neural network with a convolutional layer, a max-pooling layer, a flattening layer and a fully connected layer with one neuron

[5] You have everything you need in this book to get the array (tensor) with the feature maps, and even to squash it to 2D, but you might have to search the Internet to find out how to visualize the tensor as an image. Consider it a good (but advanced) Python exercise.

[6] If it has 100 neurons per layer, with only one output neuron, that makes the total of $784 \cdot 100 + 100 \cdot 100 + 100 \cdot 100 + 100 \cdot 1 = 98500$ parameters, and that is without the biases!.

through a regular feed-forward fully connected network is highly sequential, since we need to have the derivatives of the outer layers to compute the derivatives of the inner layers.

6.3 A Complete Convolutional Network

We now show a complete convolutional neural network in Python. We are using the library Keras, which gives us the ability to build neural networks from components, without having to worry too much about dimensionality. All the code here should be placed in one Python file and then executed in the terminal or command prompt. There are other ways to run Python code, and feel free to experiment with them—nothing will break. The first part of the code which should be placed in the file handles the imports from Keras and Numpy:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
from keras.datasets import mnist
(train_samples, train_labels), (test_samples, test_labels) = mnist.load_data()
```

You might notice the we are importing MNIST from the Keras repository. The last line of this code loads training samples, training labels, test samples and test labels in four different variables. Most of the code in this Python file will actually be used for formatting (or preprocessing) MNIST data to meed the demands which it must fulfill to be fed into a convolutional neural network. The next part of the code processes the MNIST images:

```
train_samples = train_samples.reshape(train_samples.shape [0], 28, 28, 1)
test_samples = test_samples.reshape(test_samples.shape [0], 28, 28, 1)
train_samples = train_samples.astype('float32')
test_samples = test_samples.astype('float32')
train_samples = train_samples/255
test_samples = test_samples/255
```

First notice that the code is actually duplicated: all operations are performed on both the training set and the testing set, and we will comment only one (we will talk about the training set), the other one functions in the same manner. The first line of this block of code reshapes the array which holds MNIST. The result of this reshaping is a (60000, 28, 28, 1)-dimensional array.[7] The first dimension is simply the number of samples, the second and the third are here to represent the 28

[7] Which is, mathematically speaking, a tensor.

by 28 dimension of the images, and the last one is the channel. It could be RGB, but MNIST is in greyscale, so this might seem redundant, but the whole point of reshaping the array (the initial dimension was (60000, 28, 28)) was actually to add the final dimension with 1 component in it. The reason behind this is that as we progress through convolutional layers, feature maps will be added in this direction, so we need to prepare the tensor to be able to accept it. The third row declares the entries in the array to be of type `float32`. This simply means that they are to be treated as decimal numbers. Python would to his automatically, but Numpy, which speeds up computation drastically, needs type declarations, so we have to put this line in. The fifth line normalizes array entries from a range of 0 to 255 to a range between 0 and 1 (to be interpreted as the percentage of grey in a pixel). That takes care of the samples, now we must preprocess the labels (digits from 0 to 9) with one hot encoding. We do this with the following code:

```
c_train_labels = np_utils.to_categorical(train_labels, 10)
c_test_labels = np_utils.to_categorical(test_labels, 10)
```

With that we are finished preprocessing the data and we may continue to build the actual convolutional neural network. The following code specifies the layers:

```
convnet = Sequential()
convnet.add(Convolution2D(32, 4, 4, activation='relu', input_shape=(28,28,1)))
convnet.add(MaxPooling2D(pool_size=(2,2)))
convnet.add(Convolution2D(32, 3, 3, activation='relu'))
convnet.add(MaxPooling2D(pool_size=(2,2)))
convnet.add(Dropout(0.3))
convnet.add(Flatten())
convnet.add(Dense(10, activation='softmax'))
```

The first line of this block of code creates a new blank model, and the rest of the lines here fill the network specification. The second line adds the first layer, in this case it is a convolutional layer, which has to produce 32 feature maps, has ReLU as the activation function and has a 4 by 4 receptive field. For the first layer, we also have to specify the input dimensions for each training sample that we will be giving it. Notice that Keras takes the first dimension of an array to represent individual training samples and chops up (parses) the dataset along it, so we do not need to worry about giving a (65600, 28, 28, 1) tensor instead of a (60000, 28, 28, 1) after we have specified that it takes `input_shape=(28, 28, 1)`, but the code will crash if we give it a (60000, 29, 29, 1) or even a (60000, 28, 28) dataset. The third row defines a max pooling layer with a pool size of 2 by 2. The next line specifies a third layer, which is a convolutional layer, this time with a receptive field of 3 by 3. Here we do not have to specify the input dimensions, Keras will do that for us. Following that we have another max pooling layer, also with a pool size of 2 by 2.

After this we have a dropout 'layer'. This is not a real layer, but only a modification of the connections between the previous and the following layer. The connections are modified to include a dropout rate of 0.3 for all connections. The next line flattens the

tensor. This is a generalized version of the process which we described for translating fixed-size matrices into a vector,[8] only here it is generalized for arbitrary tensors.

The flattened vector is then fed into the final layer (the final line of code in this block) which is a standard fully-connected feed-forward layer,[9] accepting as many inputs as there are components in the flattened vector, and outputting 10 values (10 output neurons), where each of them will represent one digit and it will output the respective probability. Which of them represents which digit is actually defined only by the order we had when we did one-hot encoding of the labels.

The softmax activation function used in the final layer is a version of the logistic function for more than two classes, but we will describe it in the later chapters, for now just think of it as a logistic function for many classes (we have one class for each label 0–9). Now we have a model specified, and we must compile it. Compiling a model means that Keras can now deduce and fill in all the necessary details we did not specify such as the input size for the second convolutional layer, or the dimensionality of the flattened vector. The next line of code compiles the model:

```
convnet.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])
```

Here we can see that we have specified the training method to be `'sgd'` which is stochastic gradient descent, with MSE as the error function. We have also asked the Keras to calculate the accuracy when training. The next line of code trains the compiled model:

```
convnet.fit(train_samples, c_train_labels, batch_size=32, nb_epoch=20, verbose=1)
```

This line of code trains the model using `train_samples` as training samples and `c_train_labels` as training labels. It also uses a batch size of 32 and trains for 20 epochs. The 'verbose' flag is set to 1 which means that it will print out details of training. And now we continue to the final part of the code which prints the accuracy and makes predictions from what it has learned for a new set of data:

```
metrics = convnet.evaluate(test_samples, c_test_labels, verbose=1)
print()
print("%s: %.2f%%" % (convnet.metrics_names[1], metrics[1]*100))
predictions = convnet.predict(test_samples)
```

The last line is important. Here we have put `test_samples`, but if you want to use it for predictions, you should put some fresh samples here, bearing in mind that they have to have *exactly* the same dimensions as `test_samples` asides from the first dimension, which holds individual training samples and along which Keras parses the dataset. The variable `predictions` will have exactly the same dimensionality as `c_test_labels` asides from the first dimension, but the first dimension of `test_samples` and `c_test_labels` will be the same (since they are predicted labels for *that* set of samples). You can add a line to the end saying `print(predictions)` to see the actual predictions, or

[8] Remember how we can convert a 28 by 28 matrix into a 784-dimensional vector.
[9] Keras calls them 'Dense'.

`print(predictions.shape)` to see the dimensionality of the array stored in `predictions`. These 29 lines of code (or 30 if you added one of the last ones) form a fully functional convolutional network.

6.4 Using a Convolutional Network to Classify Text

Even though the standard setting for a convolutional neural network is pattern recognition in images, convolutional neural networks can also be used to classify text. A standard approach is to use characters instead of words as primitives, and then try to map a representation of text on a character level to a higher level idea like positive or negative sentiment. This is very interesting since it allows to do a considerable amount of language processing from raw text, without any fancy feature engineering or a knowledge-heavy logical system—just learning from the letters used. In this section, we explore the now classical paper by Xiang Zhang, Junbo Zhao and Yann LeCun titled *Character-level Convolutional Networks for Text Classification* [3]. The paper itself is considerably more rich than what we present here, but we will be showing the bare bones of the approach that the authors used. We do this to help the reader to understand how to read research papers, and we strongly encourage the reader to download a copy of the paper from `arxiv.org/abs/1509.01626` and compare the text with what we write here. There will be a couple more sections like this, all with the same aim, to help the student understand papers we consider to be especially interesting. Of course, there are many more seminal and interesting papers, but we had to pick only a couple, but we encourage the reader to find more and work through them by herself.

The paper *Character-level Convolutional Networks for Text Classification* uses convolutional neural networks to classify text. One of the tasks the authors explore is the Amazon Review Sentiment Analysis. This is the most widely used sentiment analysis dataset, and it is available from a variety of sources, perhaps the best one being https://www.kaggle.com/bittlingmayer/amazonreviews. You will need a bit of formatting to get it to run, and getting this to work will be a great data wrangling exercise. Every line in these files has a review together with a label at the beginning. Two samples from the raw file are (you can conclude which label is which, there are only these two):

```
__label__1 Waste of money!

__label__2 Great book for travelling Europe:
```

The authors use a couple of architectures, and we focus on the larger one. The network uses 1D convolutional layers. Note that here we will have an example of a 1D convolutional layer processing a $m \times n$ matrix rather than a vector. This is the same as processing a vector, since the 1D convolutional layer will behave in the same way, except it will take all m rows in a pass instead of a single one as it would if it

were a vector. The 'width' of the local receptive field remains a hyperparameter, as does the stride. The stride here is 1 throughout the paper.

The first layer of the network used in the paper is of size 1024, with a local receptive field (called 'kernel' in the paper) of 7, followed by a pooling layer with a pool of size 3. This all is called 'layer 1' in the paper. The authors consider pooling to be a part of the convolutional layer, which is ok, but Keras treats pooling as a separate layer, so we will re-enumerate the layers here so that the reader can recreate them in Keras. The third and fourth level are the same as the first and second. The fifth, sixth, seventh and eighth layer are the same as the first layer (they are convolutional layers with no pooling), the ninth layer is a max pooling layer with a pool of 3 (i.e. it is like the second layer). The tenth layer is a flattening layer, and the eleventh and twelfth layers are fully-connected layers of size 2048. The final layer's size depends on the number of classes used. For sentiment this is 'positive' and 'negative', so we may use a logistic function with a single output neuron (all other layers use ReLUs). If we were to have more classes, we would use softmax, but we will do this in the later chapters. There are also two dropout layers between the three fully-connected layers and special weight initializations, but we ignore them here.

So now we have explained the task, shown you where to find the dataset with the data and labels, and explored the network architecture. What is left to do is to see how to feed the data to the network, and for this, we need encoding. The encoding is the trickiest part of this paper.[10]

Let us see how the authors encode the text. We have already noted that they use a character based approach, so we have to specify which characters to use, i.e. which we shall leave in the text and which we will remove. The authors substitute all uppercase letters for lower ones, and keep all the 26 letters of the English alphabet as valid characters. In addition, they keep the ten digits and 33 other characters (including brackets, $, #, etc.). They total to 69. They keep also the new line character, often denoted as $\backslash n$. This is the character that the Enter or Return key produces when hit. You do not see it directly, but the computer produces a new line. This means that the vocabulary size is 69, and we shall denote this by M.

The length of the particular review as a string is denoted by L. The review (without the label part) will be one-hot-encoded (aka 1-of-M encoding) using characters, but there is a twist. To make the system behave like human memory, every string is reversed, so `Waste of money!` will become `!yenom fo etsaW`. To see a complete example, imagine we have only allowed a, b, c, d and S as allowed characters,[11] where the S simply represents whitespace, since leaving it as a space would probably cause confusion (and we have used the _for Python code indentation). Suppose the text of the review is 'abbaScadd', and $L_{final} = 7$. First, the reverse it to 'ddacSabba', and then cut it to have a length of 7, to get 'ddacSab'. Then we use one hot encoding to get an M by L_{final} matrix to represent this input sample:

[10] Trivially, every paper will have a 'trickiest part', and it is your job to learn how to decode this part, since it is often the most important part of the paper.

[11] Since the whole alphabet will not fit on a page, but you can easily imagine how it will expand to the normal English alphabet.

a	0	0	1	0	0	1	0
b	0	0	0	0	0	0	1
c	0	0	0	1	0	0	0
d	1	1	0	0	0	0	0
S	0	0	0	0	1	0	0

If on the other hand we had the review 'bad' and $L_{final} = 7$, we would first reverse it to 'dab' and then put it in the left of the M by L_{final} matrix and pad the rest of the columns with zeros:

a	0	1	0	0	0	0	0
b	0	0	1	0	0	0	0
c	0	0	0	0	0	0	0
d	1	0	0	0	0	0	0
S	0	0	0	0	0	0	0

But for a convolutional neural networks, all input matrices must have the same dimension, so we have an L_{final}. All inputs for which $L > L_{final}$ are clipped to L_{final} and all of the inputs for which $L_{final} > L$ are padded by adding enough zeros to the right side to make their length exactly L_{final}. This is why the authors used the reversing, so that we loose only the more remote information at the beginning when clipping, and not the more recent one at the end.

We might ask how to make a Keras-friendly dataset from these? The first task is to view them as a tensor. This just means to collect all of the M by L_{final} matrices and add a third dimension along which they will be 'glued'. This simply means if we have 1000 M by L_{final} matrices, that we will make one M by L_{final} by 1000 tensor. Depending on the implementation you will use, it might make sense to make a 1000 by M by L_{final} tensor. Now initialize this tensor (a 3D Numpy array) with all zeros, and devise a function which will put a 1 where it should be. Try to write Keras code which implements this architecture. As always, if you get stuck, StackOverflow it. If you have never done anything similar before, it might take you even a week[12] to get it to work, even though the end result does not have many lines of code. This is a great exercise in deep learning, so don't skip it.

References

1. Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)

[12] A couple of hours each day—not a literal week.

2. D.H. Hubel, T.N. Wiesel, Receptive fields and functional architecture of monkey striate cortex. J. Physiol. **195**(1), 215–243 (1968)
3. X. Zhang, J. Zhao, Y. LeCun, Character-level convolutional networks for text classification, in *Advances in Neural Information Processing Systems 28, NIPS* (2015)

Recurrent Neural Networks 7

7.1 Sequences of Unequal Length

Let us take bird's eye view of things. Feedforward neural networks can process vectors, and convolutional neural networks can process matrices (which are translated into vectors). How would we process sequences of unequal length? If we are talking about, e.g. images of different sizes, then we could simply re-scale them to match. If we have a 800 by 600 image and a 1600 by 1200, it is obvious we can simply resize one of the images. We have two options. The first option is to make the bigger picture smaller. We could do this in two ways: either by taking the average of four pixels, or by max-pooling them. On the other hand, we can similarly make the image bigger by interpolating pixels. If the images do not scale nicely, e.g. one is 800 by 600 an the other is 800 by 555, we can simply expand the image in one direction. The deformations made will not affect the image processing since the image will retain most of the shapes. A case where it would affect the neural network would be if we were to build a classifier to discriminate between ellipses and circles and then resize the images, since that would make circles look like ellipses. Note, that if all matrices, we analyse are of the same size they can be represented by long vectors, as we have seen in the section on MNIST. If they vary in size, we cannot encode them as vectors and keep the nice properties since the rows would be of different lengths. If all images are 20 by 20, then we can translate them in a vector of size 400. This means that the second pixel in the third row of the image is the 43 component of the 400-dimensional vector. If we have two images one 20 by 20 and one 30 by 30, then the 43rd component of the ?-dimensional vector (suppose for a second that we can fit a dimensionality here somehow), would be the second pixel in the third row of the first image and the thirteenth pixel of the second row of the second image. But, the real problem is how to fit vectors of different dimensions (400 and 300) in a neural network. Everything we have seen so far, needs a fixed-dimensional vectors.

The problem of varying dimensionality can be seen as the problem of learning sequences of unequal length, and audio processing is a nice example of how we might need this, since various audio clips are necessarily of different lengths. We could in theory just take the longest and then make all others of the same length as that one, but this is waste in terms of the space needed. But there is a deeper problem here. Silence is a part of language, and it is often used for communicating meaning, so a sound clip with some content labeled with the label 1 in the training set might be correct, but if add 10 s of silence at the beginning or the end of the clip, the label 1 might not be appropriate anymore, since the clip with the silence may have a different meaning. Think about irony, sarcasm and similar phenomena.

So the question is what we can do? The answer is that we need a different nerural network architecture than we have seen before. Every neural network we have seen so far has connections which push the information forward, and this is why we have called them 'feedforward neural networks'. It will turn out that by having connections that feed the output back into a layer as inputs, we can process sequences of unequal length. This makes the network deep, but it does share weights so it partly avoids the vanishing gradient problem. Networks that have such feedback loops are called *recurrent neural networks*. In the history of recurrent neural networks, there is an interesting twist. As soon as the idea of the perceptron did not seem good, the idea of making a 'multi-layer perceptron' seemed natural. Remember that this idea was theoretical and predated backpropagation (which was widely accepted after 1986), which means that no one was able to make it work back then. Among the theoretical ideas explored was adding a single layer, adding multiple layers and adding feedback loops, which are all natural and simple ideas. This was before 1986.

Since backpropagation was not yet available, J. J. Hopfield introduced the idea of Hopfield networks [1], which can be thought of the first successful recurrent neural networks. We will explore them in detail in Chap. 10. They were specific since they were different from what we consider today to be recurrent neural networks. The most important recurrent neural networks are the *long short-term memory* networks or *LSTM*s which were invented in 1997 by Hochreiter and Schmidhuber [2]. To this day, they remain the most widely used recurrent neural networks and are responsible for many state-of-the-art results in various fields, from speech recognition to machine translation. In this chapter, we will focus on developing the necessary concepts to explain the LSTM in detail.

7.2 The Three Settings of Learning with Recurrent Neural Networks

Let us return a bit to the naive Bayes classifier. As we saw in Chap. 3, the naive Bayes classifier calculates $\mathbb{P}(target|features)$ after we calculate $\mathbb{P}(feature1|target)$, $\mathbb{P}(feature2|target)$, etc., from the dataset. This is how the naive Bayes classifier works, but all classifiers (supervised learning algorithms) try to calculate $\mathbb{P}(target|features)$ or $\mathbb{P}(\mathbf{t}|\mathbf{x})$ in some way. Recall that any predicate $\mathbb{P}$ such that

(i) $\mathbb{P}(A) \geq 0$, (ii) $\mathbb{P}(\Omega) = 1$, where Ω is the possibility space and (iii) for all disjoint A_n, $n \in \mathbb{N}$, $\mathbb{P}(\bigcup_{n=1}^{\infty} A_n) = \sum_{n=1}^{\infty} \mathbb{P}(A_n)$ is a probability predicate. Moreover, it is *the* probability predicate (try to work out the why by yourself).

Taking the probabilistic interpretation to analyze the machine learning algorithms from a bird's-eye perspective, we could say that what a *supervised* machine learning algorithm does is calculate $\mathbb{P}(\mathbf{t}|\mathbf{x})$ (where $\mathbf{x}$ denotes an input vector, and $\mathbf{t}$ denotes the target vector[1]). This is the *classic setting*, simple supervised learning with labels.

Recurrent neural networks can learn in this standard setting by simply digesting a lot of labelled sequences and then they predict the label of each finished sequence. An example might be classifying audio clips according to emotions. But recurrent neural networks are capable of much more. They can also learn from sequences with multiple labels. Imagine an industrial robotic arm that we wish to train to perform a task. It has a multitude of sensors and it has to learn directions (for simplicity suppose we have only four, North, South, East and West). The training set is then produced with movement sequences, each consisting of a string of directions, e.g. $x_1 N x_2 N x_3 W x_4 E x_5 W x_6 W$ or just $x_1 N x_2 W$. Notice how different this is from what we have seen before. Here we have a sequence of sensor data (x_i) and movements (N, E, S or W, we will denote them by D). Notice that it would be a very bad idea to break up the sequences in xD pieces, since a movement of the form $xNxN$ might happen most often when broken, it might make sense only in the beginning of the sequence (e.g. as a 'get out of the dock' command) and in any other case it would be disastrous. Sequences cannot be broken, and it is not enough to know the previous state to be able to predict the next. The idea that the next state depends only on the current is known as the *Markov assumption*, and one of the greatest strengths of the recurrent neural networks is that they do not need to make the Markov assumption—they can model more complex behaviour. This means that the recurrent network learns from uneven sequences whose parts are labelled and it creates a bunch of labels when it predicts over an unknown vector. This we will call *sequential setting*.

There is a third setting which is an evolved form of the sequential setting and we can call it the *predict-next setting*. This setting does not need labels at all and it is commonly used for natural language processing. Actually, it has labels, but they are implicit. The idea is that for every input sequence (sentence), the recurrent network breaks it down to subsequences and use the next word as the target. We will need special tokens for the start and end of the sentence, which we must put in manually, and we denote them here by \$ ('start') and & ('end'). If we have a sentence 'All I want for Christmas is you', then we first have to transform it into '\$ all I want for Christmas is you &'.[2] Then the sentence is broken into inputs and targets, which we will denote as ('input string','target'):

---

[1] In machine learning literature, it is common to find the notation $\hat{y}$, which denotes the results from the predictor, and y is kept for denoting target values. We have used a different notation, more common to deep learning, where y denotes the outputs from the predictor, and t is used to denote actual values or targets.

[2] Notice which capital letters we kept and try to conclude why.

- (‘$’,‘all’)
- (‘$ all’,‘I’)
- (‘$ all I’,‘want’)
- (‘$ all I want’, ‘for’)
- (‘$ all I want for’, ‘Christmas’)
- (‘$ all I want for Christmas’, ‘is’)
- (‘$ all I want for Christmas is’, ‘you’)
- (‘$ all I want for Christmas is you’, ‘&’).

Then, the recurrent network will learn how to return the most likely next word after hearing a word sequence. This means that the recurrent network is learning a probability distribution from the inputs, i.e. $\mathbb{P}(\mathbf{x})$, which actually makes this unsupervised learning, since there are no targets. Targets here are synthesized from the inputs.

Note that we will usually want to limit how many words we want to look back (i.e. the word-wise length of the ‘input string’ part). Notice that this is actually quite a big deal since this can be seen as a question answering capability, which is the basis of the Turing test, and this is a step towards not just a useful tool, but also towards general AI. But, we have to make one tiny adjustment here. Notice that if the recurrent network learns which is the most probable word following a sequence, it might become repetitive. Imagine that we have the following five sentences in the training set:

- ‘My name is Cassidy’
- ‘My name is Myron’
- ‘My name is Marcus’
- ‘My name is Marcus’
- ‘My name is Marcus’.

Now, the recurrent neural network would conclude that $\mathbb{P}(Marcus) = 0.6$, $\mathbb{P}(Myron) = 0.2$ and $\mathbb{P}(Cassidy) = 0.2$. So when given a sequence ‘My name is’ it would always pick ‘Marcus’ since it has the highest probability. The trick here is not to let it pick the one with the highest probability, but rather the recurrent neural network should build a probability distribution for every input sequence with the individual probabilities of all outcomes and then randomly sample it. The result will be that in 60% of the time it will give ‘Marcus’ but sometimes it will also produce ‘Myron’ and ‘Cassidy’. Note that this actually solves quite a bit of problems which might arise. If it were not so, we would have always the same response to the same sequences of words. Now that we have given a quick black box view, it is time to dig deep into the mechanics of recurrent neural networks.

7.3 Adding Feedback Loops and Unfolding a Neural Network

Let us now see how recurrent neural networks work. Remember the vanishing gradient problem? There we have seen that adding layers one after the other would severely cripple the ability to learn weights by gradient descent, since the movements would be really small, sometimes even rounded to zero. Convolutional neural networks solved this problem by using a shared set of weights, so learning even little by little is not a problem since each time the same weights get updated. The only problem is that convolutional neural networks have a very specific architecture making them best suited for images and other limited sequences.

Recurrent neural networks work not by adding new layers to a simple feedforward neural network, but by adding recurrent connections on the hidden layer. Figure 7.1a shows a simple feedforward neural netwok and Fig. 7.1b shows how to add recurrent connections to the simple feedforward neural network from Fig. 7.1a. The *outputs* from a given layer are denoted by $\mathbf{I}$, $\mathbf{O}$ and $\mathbf{H}$ for the simple feedforward network, and by $\mathbf{H}_1, \mathbf{H}_2, \mathbf{H}_3, \mathbf{H}_4, \mathbf{H}_5, \ldots$ when we add recurrent connections. The weights in the simple feedforward network are denoted by $\mathbf{w}_x$ (input-to-hidden) and $\mathbf{w}_o$ (hidden-to-output). It is very important not to confuse *multiple outputs from a hidden layer with multiple hidden layers*, since a layer is actually defined in terms of weights, i.e. each layer has its own set of weights, and here all $\mathbf{H}_n$ share the same set of weights, viz. $\mathbf{w}_h$. Figure 7.1c is exactly the same as Fig. 7.1b with the only difference being that we condensed the individual neurons (circles) into vectors (rectangles), which we have been doing since Chap. 3 in our calculations, but now we do it on the visual display as well. Notice that to add the recurrent connection, we had to add a set of weights, $\mathbf{w}_h$, to the calculation and this is all that is needed to add recurrence to the network.

Note that the recurrent neural network can be unfolded so that the recurrent connections are all specified. Figure 7.2a shows the previous network and Fig. 7.2 shows how to unfold the recurrent connections. Figure 7.2c is the same as Fig. 7.2b but with

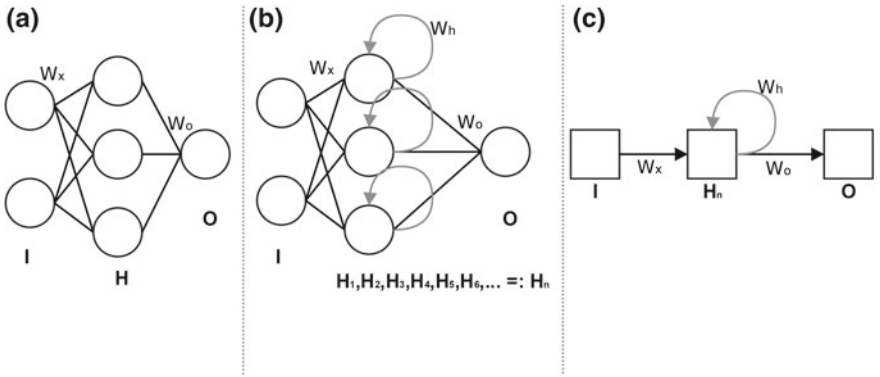


Fig. 7.1 Adding recurrent connections to a simple feedforward neural network

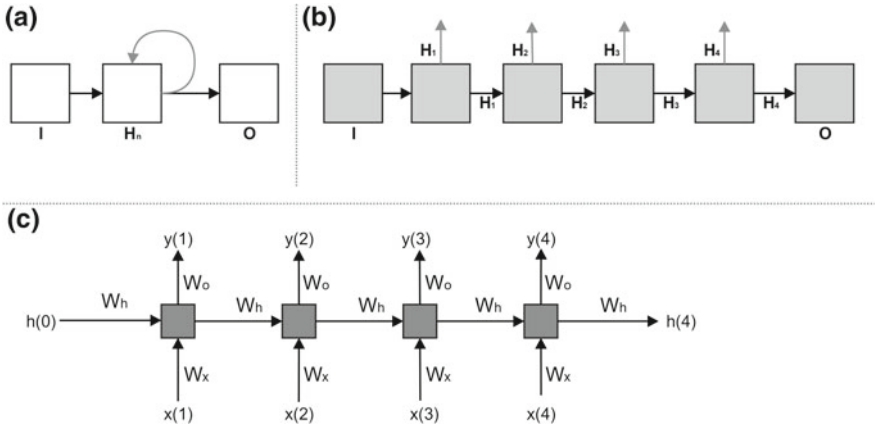


Fig. 7.2 Unfolding a recurrent neural network

the proper and detailed notation used in the recurrent neural network literature, and we will focus on this representation for commenting on the fly how a recurrent neural network works. The next section will use the sub-image C of Fig. 7.2 for reference, and this will be our standard notation for the rest of the chapter.[3]

7.4 Elman Networks

Let us comment on the Fig. 7.2c. $\mathbf{w}_x$ represent input weights, $\mathbf{w}_h$ represent the recurrent connection weights and the $\mathbf{w}_o$ the hidden-to-output weights. The xs are inputs, and the ys are outputs, just like before. But here we have an additional sequential nature, which tries to capture time. So $x(1)$ is the first input, and later it gets $x(2)$ and so on. The same holds of outputs. If we have the classic setting, we would only be using $x(1)$ (to give the input vector) and $y(4)$ to catch the (overall) output. But for the sequential and predict-next settings, we would be using all xs and ys.

Notice that unlike the situation we had in simple feedforward networks, here we also have the h, and they represent the inputs for the recurrent connection. We need something to start with, and we can generate $h(0)$ by simply setting all its entries to 0. We give an example calculation where it can be seen how to calculate all elements and it will be much more insightful than giving a piece by piece calculation. By f, we will be denoting a nonlinearity, and you can think of it as the *logistic function*. A bit later we will see a new nonlinearity called *softmax*, which can be used here and has natural fit with recurrent neural networks. So, the recurrent neural network

[3]We used the shades of grey just to visually denote the gradual transition to the proper notation.

calculates the output y at a final time t. The calculation can be unfolded to the following recursive structure (which makes it clear why we need $h(0)$):

$$y(t) = f(\mathbf{w}_o^\top h(t)) = \tag{7.1}$$
$$= f(\mathbf{w}_o^\top f(\mathbf{w}_h^\top h(t-1) + \mathbf{w}_x^\top x(t))) = \tag{7.2}$$
$$= f(\mathbf{w}_o^\top f(\mathbf{w}_h^\top f(\mathbf{w}_h^\top h(t-2) + \mathbf{w}_x^\top x(t-1)) + \mathbf{w}_x^\top x(t))) = \tag{7.3}$$
$$= f(\mathbf{w}_o^\top f(\mathbf{w}_h^\top f(\mathbf{w}_h^\top f(\mathbf{w}_h^\top h(t-3) + \mathbf{w}_x^\top x(t-2)) + \mathbf{w}_x^\top x(t-1)) + \mathbf{w}_x^\top x(t))). \tag{7.4}$$

We can make this more readable by condensing it to two equations:

$$h(t) = f_h(\mathbf{w}_h^\top h(t-1) + \mathbf{w}_x^\top x(t)) \tag{7.5}$$
$$y(t) = f_o(\mathbf{w}_o^\top h(t)), \tag{7.6}$$

where f_h is the nonlinearity of the hidden layer, and f_o is the nonlinearity of the output layer, which are not necessarily the same function, but they can be the same if we want. This type of recurrent neural network is called *Elman networks* [3], after the linguist and cognitive scientist Jeffrey L. Elman.

If we change the $h(t-1)$ for $y(t-1)$ in Eq. 7.5, so that it becomes as follows:

$$h(t) = f_h(\mathbf{w}_h^\top y(t-1) + \mathbf{w}_x^\top x(t)). \tag{7.7}$$

We obtain a *Jordan network* [4], which are named after the psychologist, mathematician and cognitive scientist Michael I. Jordan. Both Elman and Jordan networks are known in the literature as *simple recurrent networks* (SRN for short). Simple recurrent networks are seldom used in applications today, but they are the main teaching method for explaining recurrent networks before running in the much more complex LSTMs, which are the main recurrent architecture used today. It is very easy to look down on SRNs today, but when they were first proposed, it became the first model that could operate on words of a text without having to rely on an 'alien' representation such as the bag of words or n-grams. In a sense, those representations seemed to suggest that language processing is something very foreign to a computer, since people do not use anything like the Bag of words for understanding language. The SRN made a decisive move towards the language processing as word sequence processing paradigm we have today, and made the whole process much closer to human intelligence. Consequently, SRNs should be considered a milestone in AI, since they have made that crucial step: what previously seemed impossible was now conceivable. But a couple of years later, a stronger architecture would come and take over all practical applications, but this strength comes with a price: LSTMs are much slower to train than SRNs.

7.5 Long Short-Term Memory

In this section, we will give a graphical illustration of the workings of the *long short-term memory* (LSTM), and the interested reader should have no problem in coding LSTMs from scratch just by following our explanation and the accompanying images. All images in the current section on LSTMs are reproduced from Christopher Olah's blog.[4] We follow the same notation as is used there (except from a couple of minor details), and we omit the weights in Fig. 7.3 to simply exposition, but we will add them when addressing individual components of the LSTM in the later images. Since we know from Eq. 7.5 that $y(t) = f_o(\mathbf{w}_o \cdot h(t))$ (f_o is the nonlinearity of choice for the output layer), in this chapter $y(t)$ is the same as $h(t)$, but we still point to the places, where $h(t)$ is to be multiplied by $\mathbf{w}_o$ to get $y(t)$ by simply noting $y(t) = h(t)$. This is really not that important from a purely formal point of view, but we hope to be more clear by holding a place for $y(t)$.

Figure 7.3 shows a bird's-eye perspective on LSTMs and compares them to SRNs. One thing that can be seen right away is that SRNs have one link from one unit to the next (it is the flow of $h(t)$), whereas the LSTMs have the same $h(t)$ but also $C(t)$. This $C(t)$ is called *the cell state*, an this is the main flow of information through the LSTMs. Figuratively speaking, the cell state is the 'L', the 'T' and the 'M' from 'LSTM', i.e. it is the *long-term memory* of the model. Everything else that happens is just different filters to decide what should be kept or added to the cell state. The

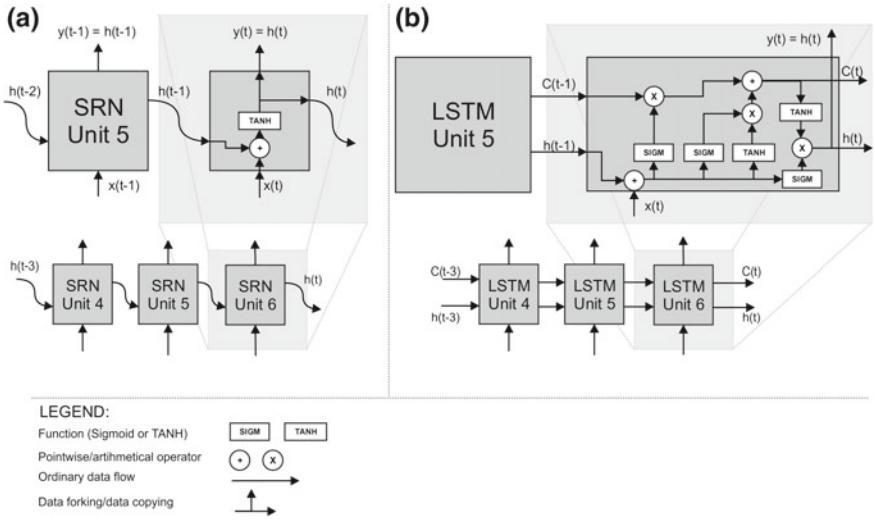


Fig. 7.3 SRN and LSTM units zoomed

[4] http://colah.github.io/posts/2015-08-Understanding-LSTMs/, accessed 2017-03-22.

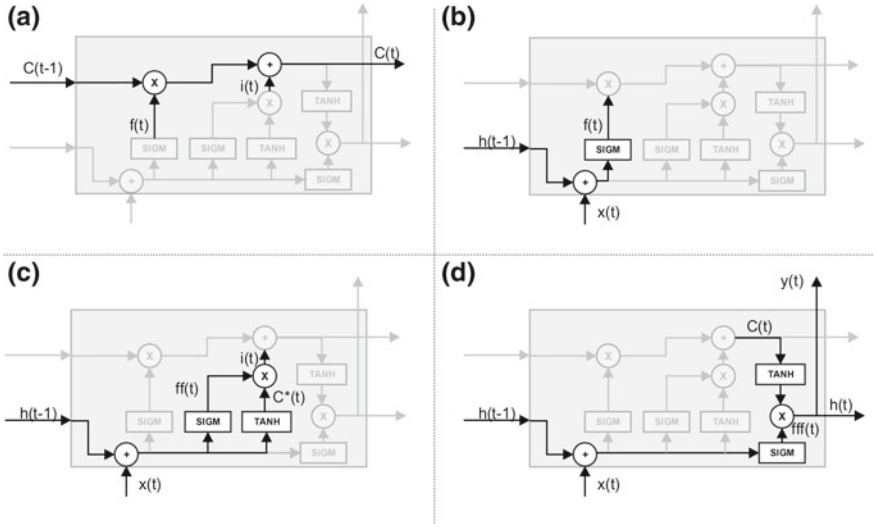


Fig. 7.4 Cell state (**a**), forget gate (**b**), input gate (**c**) and output gate (**d**)

cell state is emphasized on Fig. 7.4a (for now you should ignore the $f(t)$ and $i(t)$ on the image, you will see how they are calculated in a couple of paragraphs).

The LSTM adds or removes information from the cell with so-called *gates*, and these make up the rest of the unit in an LSTM. The gates are actually very simple. They are a combination of addition, multiplication and nonlinearities. The nonlinearities are used simply to 'squash' information. The logistic or sigmoid function (denoted as SIGM in the images) is used to 'squash' information to values between 0 and 1, and the hyperbolic tangent (denoted as TANH in the images) is used to 'squash' the information to values between -1 and 1. You can think of it in the following way: SIGM makes a fuzzy 'yes'/'no' decision, while TANH makes a fuzzy 'negative'/'neutral'/'positive' decision. They do nothing else except this.

The first gate is the *forget gate*, which is emphasized in Fig. 7.4b. The name 'gate' comes from analogies with the logic gates. The forget gate at unit t is denoted by $f(t)$, and is simply $f(t) := \sigma(\mathbf{w}_f(x(t)+h(t-1)))$. Intuitively, it controls how much of the weighted raw input and weighted previous hidden state is to be *remembered*. Note that the σ is the symbol for the logistic function.

Regarding weights, there are different approaches, but we consider the most intuitive to be the one which breaks up $\mathbf{w}_h$ into several different weights, $\mathbf{w}_f$, $\mathbf{w}_{ff}$, $\mathbf{w}_C$ and $\mathbf{w}_{fff}$.[5] The point to remember is that there are different ways to look at the weights and some of them try to keep the same names as they had in simpler models, but the most natural approach for deep learning is to think of an architecture as composed

[5]Notice that we are not quite precise here and that the $\mathbf{w}_f$ in the LSTMs is actually the same as $\mathbf{w}_x$ in the SRN and not a component of the old $\mathbf{w}_h$.

of basic 'building blocks' to be assembled together like LEGO® bricks, and then each block should have its own set of weights. All of the weight in a complete neural network are trained together with backpropagation and the joint training actually makes a neural network a connected whole (like each LEGO brick normally has its own studs to connect to other bricks to make a structure).

The next gate (emphasized in Fig. 7.4c), called the *input gate*, is a bit more complex. It basically decides on what to put in the cell state. It is composed of another forget gate (which we unimaginatively denote with $ff(t)$) but with different weights, but it also has an additional module which creates candidates to be added to the cell state. The $ff(t)$ can be thought of as a saving mechanism, which controls how much of the input we will save to the cell state. In symbols:

$$ff(t) := \sigma(\mathbf{w}_{ff}(x(t) + h(t-1))), \tag{7.8}$$

$$i(t) := ff(t) \cdot C^*(t). \tag{7.9}$$

What we are missing is a calculation for the candidates (denoted by $C^*(t)$). Calculating the candidates is pretty easy: $C^*(t) := \tau(\mathbf{w}_C \cdot (x(t) + h(t-1)))$, where τ is the symbol for the hyperbolic tangent or $tanh$. We are using the hyperbolic tangent here to squash the results to values which range between -1 and 1. Intuitively, the negative part of the range (-1 to 0) can be seen as a way to get quick 'negations', so that even opposites would be considered to get, for example a quick processing of linguistic antonyms.

As we have seen before, an LSTM unit has three outputs: $C(t)$, $y(t)$ and $h(t)$. We have all we need to compute the current cell state $C(t)$ (this calculation is shown in Fig. 7.4a):

$$C(t) := f(t) \cdot C(t-1) + i(t). \tag{7.10}$$

Since $y(t) = g_o(\mathbf{w}_o \cdot h(t))$ (where g_o is a nonlinearity of choice), all that is left is to compute $h(t)$. To compute $h(t)$, we will need a third copy of the forget gate ($fff(t)$), which will have the task of deciding which parts of the inputs and how much of it to include in $h(t)$:

$$fff(t) := \sigma(\mathbf{w}_{fff}(x(t) + h(t-1))). \tag{7.11}$$

Now, the only thing left for a complete *output gate* (whose result is actually not $o(t)$ but $h(t)$), we need to multiply the $fff(t)$ by the current cell state squashed between -1 and 1:

$$h(t) := fff(t) \cdot \tau(C(t)). \tag{7.12}$$

And now finally, we have the complete LSTM. Just a quick final remark: the $fff(t)$ can be thought of as a 'focus' mechanism which tries to say what is the most important part of the cell state. You might think of $f(t)$, $ff(t)$ and $fff(t)$, but the idea is that they all participate in different parts and as such, we hope they will take on the mechanism we want ('remember from last unit', 'save input' and 'focus on this part of the cell state' respectively). Remember that this is only our wild hope, we

have no way to 'force' this interpretation on the LSTM other than with the sequence of calculations or flow of information we have chosen to use. This means that these interpretations are metaphorical, and only if we have made a one-in-a-million lucky guesstimate will these mechanisms actually coincide with the mechanisms in the human brain.

The LSTMs have been first proposed by Hochreiter and Schmidhuber in 1997 [2], and they have become one of the most important deep architectures for natural language processing, time series analysis and many other sequential tasks. Today one of the best reference books on recurrent neural networks is [5], and we highly recommend it for any reader that wishes to specialize in these amazing architectures.

7.6 Using a Recurrent Neural Network for Predicting Following Words

In this section, we give a practical example of a simple recurrent neural network used for predicting next words from a text. This sort of task is highly flexible, since it allows not just predictions but also question answering—the (single word) answer is simply the next word in the sequence. The example we use is a modification of an example from [6], with ample comments and explanations. Some portions of the original code have been modified to make the code easier to understand. As we explained in the previous section, this is a working Python 3 code, but you will need to install all dependencies. You should also be able to follow the ideas from the code on chapter, but to see the subtleties, one needs to have the actual code on the computer.[6] We start by importing the Python libraries and we will be needing:

```
from keras.layers import Dense, Activation
from keras.layers.recurrent import SimpleRNN
from keras.models import Sequential
import numpy as np
```

The next thing is to define hyperparameters:

```
hidden_neurons = 50
my_optimizer ="sgd"
batch_size = 60
error_function = "mean_squared_error"
output_nonlinearity = "softmax"
cycles = 5
epochs_per_cycle = 3
context = 3
```

[6]Which you can get either from the book's GitHub repository, or by typing in all the code in this section in one simple file (.txt) and rename it to change its extension to .py.

Let us take a minute and see what we are using. The variable `hidden_neurons` simply states how many hidden units are we going to use. We use Elman units here, so this is the same as the number of feedback loops on the hidden layer. The variable `optimizer` defines which Keras optimizer we are going to use, and in this case it is the stochastic gradient descent, but there are others,[7] and we recommend to experiment with several optimizers just to get a feel. Note that `"sgd"` is a Keras name for it, so you must type it exactly like this, not `"SGD"`, nor `"stochastic_GD"`, nor anything similar. The `batch_size` simply says how many examples we will use for a single iteration of the stochastic gradient descent. The variable `error_function = "mean_squared_error"` tells Keras to use the MSE we have been using before.

But now we come to the activation function `output_nonlinearity`, and we see something we have not seen before, the *softmax* activation function or nonlinearity, with its Keras name `"softmax"`. The softmax function is defined as

$$\zeta(z_j) := \frac{e^{z_j}}{\sum_{n=1}^{N} e^{z_k}}, j = 1, \ldots, N. \tag{7.13}$$

The softmax is quite a useful function: it basically transforms a vector $\mathbf{z}$ with arbitrary real values to a vector with values ranging from 0 to 1, and they are such that they all add up to 1. This is why the softmax is very often used in the final layer of a deep neural network used for multiclass classification[8] to get the output which can be a probability proxy for the classes. It can be shown that if the vector $\mathbf{z}$ has only two components, z_0 and z_1 (which would simulate binary classification) would reduce *exactly* to the logistic function classification, only with the weight being $\mathbf{w}_\sigma = \mathbf{w}_{\zeta 1} - \mathbf{w}_{\zeta 0}$. We can now continue to the next part of the SRN code, bearing in mind that the rest of the parameters we will comment when they become active in the code:

```
def create_tesla_text_from_file(textfile="tesla.txt"):
    clean_text_chunks = []
    with open(textfile, 'r', encoding='utf-8') as text:
        for line in text:
            clean_text_chunks.append(line)
    clean_text = ("".join(clean_text_chunks)).lower()
    text_as_list = clean_text.split()
    return text_as_list
text_as_list = create_tesla_text_from_file()
```

This part of the code opens a plain text file `tesla.txt`, which will be used for training and predicting. This file should be encoded in utf-8 or the `utf-8` in the

[7] There is a full list on https://keras.io/optimizers/.

[8] Where we have more than two classes. Note that in binary classification were we have two classes, say A and B, we actually do a classification (with, for e.g. the logistic function in the output layer) in only one of them and get a probability score p_A. The probability score of B is then calculated as $1 - p_A$.

code should be changed to reflect the appropriate file encoding. Note that most text editors today distinguish 'file encoding' (actual encoding of the file) from 'encoding' (the encoding used to display text for that file in the editor). This approach will work for files that are about 70% the size of the available RAM on the computer you are using. Since we are talking about plain text files, having an 16GB machine and a 10GB file will work out well, and 10GB is a lot of plain text (just for comparison, the whole English Wikipedia with metadata and page history in plain text has a size of 14GB). For larger datasets, we would take a different approach, namely to separate the big file into chunks and consider them batches, and feed them one by one, but the details of such big data processing are beyond the scope of this book.

Notice that when Python opens and reads a file, it returns it line by line, so we are actually accumulating these lines in a list called `clean_text_chunks`. We then glue all of these together in one big string called `clean_text`, and then cut them into individual words and store it in the list called `text_as_list`, and this is what the whole function `create_tesla_text_from_file(textfile="tesla.txt")` returns. The part `(textfile="tesla.txt")` means that the function `create_tesla_text_from_file()` expects an argument (which is refered to as `textfile`) but we have provided a default value `"tesla.txt"`. This means that if we give a file name, it will use that, otherwise it will se `"tesla.txt"`. The final line `text_as_list = create_tesla_text_from_file()` calls the function (with the default file name), and stores what the function has returned in the variable `text_as_list`. Now, we have all of our text in a list, where each individual element is a word. Notice that there may be repetitions of words here, and that is perfectly fine, as this will be handled by the next part of the code:

```
distinct_words = set(text_as_list)
number_of_words = len(distinct_words)
word2index = dict((w, i) for i, w in enumerate(distinct_words))
index2word = dict((i, w) for i, w in enumerate(distinct_words))
```

The `number_of_words` simply counts the number of words in the text. The `word2index` creates a dictionary with unique words as keys and their position in the text as values, and `index2word` does the exact opposite, creates a dictionary where positions are keys and words are values. Next, we have the following:

```
def create_word_indices_for_text(text_as_list):
    input_words = []
    label_word = []
    for i in range(0,len(text_as_list) - context):
        input_words.append((text_as_list[i:i+context]))
        label_word.append((text_as_list[i+context]))
    return input_words, label_word
input_words,label_word = create_word_indices_for_text(text_as_list)
```

Now, it gets interesting. This is a function which creates a list of input words and a list of label words from the original text, which has to be in the form of a list of

individual words. Let us explain a bit of the idea. Suppose we have a tiny text '*why would anyone ever eat anything besides breakfast food?*'. Then we want to make an 'input'/'label' structure for predicting the next word, and we do this by decomposing this sentence into an array:

Input word 1	Input word 2	Input word 3	Label word
why	would	anyone	ever
would	anyone	ever	eat
anyone	ever	eat	anything
ever	eat	anything	besides
eat	anything	besides	breakfast
anything	besides	breakfast	food?

Note that we have used three input words and declared the next one the label, and then shifted for one word and repeated the process. How many input words we use is actually defined by the hyperparameter `context`, and can be changed. The function `create_word_indices_for_text(text_as_list)` takes a text in the form of the list, creates the input words list and the label word list and returns them both. The next part of the code is

```
input_vectors = np.zeros((len(input_words), context, number_of_words), dtype=np.int16)
vectorized_labels = np.zeros((len(input_words), number_of_words), dtype=np.int16)
```

This code produces 'blank' tensors, populated by zeros. Note that the term 'matrix' and 'tensor' come from mathematics, where they are objects that work with certain operations, and are distinct. Computer science treats them both as multidimensional *arrays*. The difference is that computer science places the focus on their structure: if we iterate along one dimension, all elements along that dimension (properly called 'axis') have the same shape. The type of entries in the tensors will be `int16`, but you can change this as you wish.

Let us discuss tensor dimensions a bit. The tensor `input_vectors` is technically called a *third-order tensor*, but in reality this is just a 'matrix' with three dimensions, or simply a 3D array. To understand the dimensionality of the `input_vectors` tensor note that first we have three words (i.e. a number of words defined by `context`) to make a one-hot encoding of. Notice that we are technically using a one-hot encoding and not a bag of words, since we have only kept distinct words from the text. Since we have a one-hot encoding, this would expand a second dimension. This takes care of the `context` and `number_of_words` dimensions of the tensor, and third one (in the code it is the first one, `len(input_words)`) is actually here just to bundle all inputs together, like we had a matrix holding all input vectors in the previous chapters. The `vectorized_labels` is the same, only here we do not have three or n words specified by the variable `context`, but only a single one, the label word, so we need one less dimension in the tensor. Since we have initialized two blank tensors, we need something to put the 1s in the appropriate places, and the next part of the code does just that which is as follows:

```
for i, input_w in enumerate(input_words):
    for j, w in enumerate(input_w):
        input_vectors[i, j, word2index[w]] = 1
        vectorized_labels[i, word2index[label_word[i]]] = 1
```

It is a bit hard, but try to figure out for yourself how this code 'crawls' the tensors and puts the 1s where they should be.[9] Now, we have cleared all the messy parts, and the next part of the code actually specifies the complete simple recurrent neural network with Keras functions.

```
model = Sequential()
model.add(SimpleRNN(hidden_neurons, return_sequences=False,
input_shape=(context,number_of_words), unroll=True))
model.add(Dense(number_of_words))
model.add(Activation(output_nonlinearity))
model.compile(loss=error_function, optimizer=my_optimizer)
```

Most of the things that can be tweaked here are actually placed in the hyperparameters. No change should be done in this part, except perhaps add a number of new layers, which is done by duplicating the line or lines specifying the layer, in particular the second line, *or* the third *and* fourth lines. The only thing left to do is to see how well does the model work, and what does it produce as output. This is done by the final part of the code which is as follows:

```
   for cycle in range(cycles):
    print("> - <" * 50)
    print("Cycle: %d" % (cycle+1))
    model.fit(input_vectors, vectorized_labels, batch_size = batch_size,
epochs = epochs_per_cycle)
    test_index = np.random.randint(len(input_words))
    test_words = input_words[test_index]
    print("Generating test from test index %s with words %s:" % (test_index,
test_words))
    input_for_test = np.zeros((1, context, number_of_words))
    for i, w in enumerate(test_words):
        input_for_test[0, i, word2index[w]] = 1
    predictions_all_matrix = model.predict(input_for_test, verbose = 0)[0]
    predicted_word = index2word[np.argmax(predictions_all_matrix)]
    print("THE COMPLETE RESULTING SENTENCE IS: %s %s" % ("".join(test_words),
predicted_word))
    print()
```

This part of the code trains and tests the complete SRN. Testing would usually be predicting a part of data we held out (test set) and then measuring accuracy. But here

[9] This is perhaps the single most challenging task in this book, but do not skip it since it will be extremely useful for a good understanding, and it is just four lines of code.

we have the predict-next setting, which does not have labels, so we have to adopt a different approach. The idea is to train and test in a *cycle*. A cycle is composed of a training session (with a number of epochs) and then we generate a test sentence from the text and see whether the word which the network gives makes sense when placed after the words from the text. This completes one cycle. These cycles are cumulative, and sentences will become more and more meaningful after each successive cycle. In the hyperparameters we have specified that we will train for 5 cycles, each having 3 epochs.

Let us make a brief remark on what we have done. For computational efficiency, most tools used for the predict-next make use of the *Markov assumption*. Informally, the Markov assumption means that we simplify a probability which would have to consider all steps from the beginning of time, $\mathbb{P}(s_n|s_{n-1}, s_{n-2}, s_{n-3}, \ldots)$, to a probability which just considers the previous step $\mathbb{P}(s_n|s_{n-1})$. If a system takes this computational detour it is said to 'use the Markov assumption'. If a process turns out to be such that it really does not matter anything but the preceding state in time, it is said to be a *Markov process*. Language production is not a Markov process. Suppose you are a classifier and you have a 'training' sentence: 'We need to remember what is important in life: friends, waffles, work. Or waffles, friends, work. Does not matter, but work is third'. If it were a Markov process, and you could make the Markov assumption without a big loss in functionality, you would be needing just one word and you could tell which one follows. If you have 'Does', you can tell that in you training set, after this it always comes 'not', and you would be right. But if you were given 'work', you would have more trouble, but you could get away with a probability distribution. But what if you did not have a predict-next setting, but your task was to identify when the speaker got confused (i.e. when you try to dig into meaning). Then, you would need all of the previous words for comparison. At many times you can cut corners a bit and make the Markov assumption for non-Markov processes and get away with it, but the point is that unlike many other machine learning algorithms, recurrent neural networks do no *have to make* the Markov assumption, since they are fully capable of handling many time steps, not just the last one.

There is one last thing we need to comment before leaving recurrent neural networks, an this is how backpropagation works. Backpropagation in recurrent neural networks is called *backpropagation through time* (BPTT). In our code, we did not have to worry about backpropagation since TensorFlow, which is the default backend for Keras calculated the gradients for us automatically, but let us see what is happening under the hood. Remember that the goal in backpropagation is to calculate the gradients of the error E with respect to $\mathbf{w}_x$, $\mathbf{w}_h$ and $\mathbf{w}_o$.

When we we were talking of the MSE and SSE error functions, we have seen that we resort to summing up the errors, and that this is good enough for machine learning. We can also just sum up the gradients for each training sample at a given point in time:

$$\frac{\partial E}{\partial w_i} = \sum_t \frac{\partial E_t}{\partial w_i}. \tag{7.14}$$

Let us see how this works in a whole example. Say, we want to calculate the gradient for E_2:

$$\frac{\partial E_2}{\partial \mathbf{w}_o} = \frac{\partial E_2}{\partial \mathbf{y}_2}\frac{\partial \mathbf{y}_2}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial \mathbf{w}_o}. \tag{7.15}$$

This means that for $\mathbf{w}_o$ the time component plays no part. As expected, for $\mathbf{w}_h$ ($\mathbf{w}_x$ is similar) it is a bit different which is as follows:

$$\frac{\partial E_2}{\partial \mathbf{w}_h} = \frac{\partial E_2}{\partial \mathbf{y}_2}\frac{\partial \mathbf{y}_2}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{w}_h}. \tag{7.16}$$

But remember that $h_2 = f_h(\mathbf{w}_h\mathbf{h}_1 + \mathbf{w}_x\mathbf{x}_2)$ which means the whole expression depends on $\mathbf{h}_1$, so if we want the derivative with respect to $\mathbf{w}_h$ we cannot treat it as a constant. The proper way to do it is to split the last term into a sum as follows:

$$\frac{\partial \mathbf{h}_2}{\partial \mathbf{w}_h} = \sum_{i=0}^{2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_i}\frac{\partial \mathbf{h}_i}{\partial \mathbf{w}_h}. \tag{7.17}$$

So, except for the summation, backpropagation through time is exactly the same as standard backpropagation. This simplicity of calculation is actually the reason why SRNs are more resistant to the vanishing gradient than a feedforward network with the same number of hidden layers. Let us address a final issue. The error function we have previously used was MSE, and this is a valid choice for regression and binary classification. A better choice for multi-class classification is the *cross-entropy error function*, which is defined as

$$CE = -\frac{1}{n}\sum_{i \in currBatch}(t_i \ln y_i + (1 - y_i)\ln(1 - y_i)). \tag{7.18}$$

Where t is the target, y is the classifier outcome, i is the dummy variable which iterates over the current batch targets and outputs, and n is the number of all samples in the batch. The cross-entropy error function is derived from the log-likelihood, but this derivation is rather tedious and beyond our needs so we skip it. The cross-entropy is a more natural choice of error functions, but it is less straightforward to understand conceptually, so we used the MSE throughout this book, but you will want to use the CE for all multiclass classification tasks. The Keras code is `loss=categorical_crossentropy`, but feel free to browse all loss functions https://keras.io/losses/, you might be surprised to find some functions

which we will discuss in a different context can also be used as a loss or error function in neural network training. In fact, finding or defining a good loss function is often a very important part of getting a good accuracy with a deep learning model.

References

1. J.J. Hopfield, Neural networks and physical systems with emergent collective computational abilities. Proc. Nat. Acad. Sci. U.S.A **79**(8), 2554–2558 (1982)
2. S. Hochreiter, J. Schmidhuber, Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
3. J.L. Elman, Finding structure in time. Cogn. Sci. **14**, 179–211 (1990)
4. M.I. Jordan, Attractor dynamics and parallelism in a connectionist sequential machine, in *Proceedings of the 26th Annual International Conference on Machine Learning, Erlbaum, NJ, USA* (Cognitive Science Society, 1986), pp. 531–546
5. A. Graves, *Supervised Sequence Labelling with Recurrent Neural Networks* (Springer, New York, 2012)
6. A. Gulli, S. Pal, *Deep Learning with Keras* (Packt publishing, Birmingham, 2017)

Autoencoders

8

8.1 Learning Representations

In this and the next chapter we turn our attention to unsupervised deep learning, also known as learning distributed representations or representation learning. But first we need to fill in a blank we had from Chap. 3. There we discussed PCA as a form of learning distributed representations, and formulated the problem as finding $Z = XQ$, where all features have been decorrelated. Here we will calculate the matrix Q. We will need to have a *covariance matrix* of X. The covariance matrix of a given matrix shows the entries of the original matrix. The covariance of two random variables X and Y is defined as $COV(X, Y) := \mathbb{E}((X - \mathbb{E}(X))(Y - \mathbb{E}(Y)))$ and show how two random variables change together. Remember that with a bit of hand waving everything relating to data can be thought of as a random variable. Also, with a bit more of hand waving, for a random variable X we may think of $\mathbb{E}(X) = MEAN(X)$.[1] This will only hold if the distribution of X is uniform, but it can be helpful from a practical perspective even when it is not, especially since in machine learning we will probably have some optimization somewhere so we can be a bit sloppy.

The attentive reader may notice that $\mathbb{E}(X)$ was actually a vector, while $MEAN(X)$ is a single value, but we will use something called *broadcasting* to make it right again. Broadcasting a value v into an n-dimensional vector $\mathbf{v}$ means simply to put the same v in every component of $\mathbf{v}$, or simply:

$$broadcast(v, n) = \underbrace{(v, v, v, \ldots, v)}_{n} \tag{8.1}$$

[1] The expected value is actually the weighted sum, which can be calculated from a frequency table. If 3 out of five students got the grade '5', and the other two got a grade '3', $\mathbb{E}(X) = 0.6 \cdot 5 + 0.4 \cdot 3$.

We will denote the covariance matrix of the matrix X as $\Xi(X)$. This is not a standard notation, but (unlike the standard notation C or Σ) this notation will avoid confusion, since we are using the standard notations in a different sense in this book. To address the covariance matrix more formally, if we have a column vector $\mathbf{X} = (X_1, X_2, \ldots, X_d)^\top$ populated with random variables, the covariance matrix Ξ_X (which can also be denoted as Ξ_{ij}) can be defined as $\Xi_{ij} = COV(X_i, X_j) = \mathbb{E}((X_i - \mathbb{E}(X_i))(X_j - \mathbb{E}(X_j)))$, or if we write the whole $d \times d$ matrix:

$$\Xi_{\mathbf{X}} = \begin{bmatrix} \mathbb{E}((X_1 - \mathbb{E}(X_1))(X_1 - \mathbb{E}(X_1))) & \cdots & \mathbb{E}((X_1 - \mathbb{E}(X_1))(X_d - \mathbb{E}(X_d))) \\ \mathbb{E}((X_2 - \mathbb{E}(X_2))(X_1 - \mathbb{E}(X_1))) & \cdots & \mathbb{E}((X_2 - \mathbb{E}(X_2))(X_d - \mathbb{E}(X_d))) \\ \vdots & \ddots & \vdots \\ \mathbb{E}((X_d - \mathbb{E}(X_d))(X_1 - \mathbb{E}(X_1))) & \cdots & \mathbb{E}((X_d - \mathbb{E}(X_d))(X_d - \mathbb{E}(X_d))) \end{bmatrix} \tag{8.2}$$

It should now be clear that the covariance matrix actually measures 'self'-covariance, i.e. covariance between its own elements. Let us see what properties does a matrix $\Xi(X)$ have. First, it must be symmetric, since the covariance of X with Y is the same as the covariance of Y with X. $\Xi(X)$ is also a *positive-definite matrix*, which means that the scalar $v^\top Xz$ is positive for every non-zero vector v.

Let us turn to a slightly different topic, *eigenvectors*. Eigenvectors of a $d \times d$ matrix A are vectors whose *direction* does not change (but the length does) when they are multiplied by A. It can be proved that there are exactly d of them. How to find the eigenvectors is the hard part, and there are number of approaches, and one of the more popular ones is gradient descent. Since all numerical libraries can find eigenvectors for us, we will not go into details.

So the eigenvectors when multiplied by a matrix A do not change direction, only the length. It is common practice to normalize the eigenvectors and denote them by $\mathbf{v}_i$. This change of length is called the *eigenvalue*, usually denoted by λ_i. This actually gives rise to a fundamental property of eigenvectors and eigenvalues of a matrix, namely $A\mathbf{v}_i = \lambda_i \mathbf{v}_i$

Once we have the $\mathbf{v}$s and λs, we start by arranging the lambdas in descending order:

$$\lambda_1 > \lambda_2 > \ldots > \lambda_d$$

This also creates an arrangement in the corresponding eigenvectors $\mathbf{v}_1$, $\mathbf{v}_2$, ..., $\mathbf{v}_d$ (note that each of them is of the form $\mathbf{v}_i = (v_i^{(1)}, v_i^{(2)}, \ldots, v_i^{(d)})$, $1 \leq i \leq d$) since there is a one to one correspondence between them and the eigenvalues, so we can simply 'copy' the order of the eigenvalues on the eigenvectors. We create a $d \times d$ matrix with the eigenvectors as columns which are sorted with ordering of the the corresponding eigenvalues (in the last step we are simply renaming the entries to

follow the usual matric entry naming conventions):

$$V = (\mathbf{v}_1^\top, \mathbf{v}_2^\top, \ldots, \mathbf{v}_d^\top) = \begin{bmatrix} v_1^{(1)} & v_2^{(1)} & \cdots & v_d^{(1)} \\ v_1^{(2)} & v_2^{(2)} & \cdots & v_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ v_1^{(d)} & v_2^{(d)} & \cdots & v_d^{(d)} \end{bmatrix} = \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1d} \\ v_{21} & v_{22} & \cdots & v_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ v_{d1} & v_{d2} & \cdots & v_{dd} \end{bmatrix}$$

We now create a blank matrix of zeros (size $d \times d$) and put the lambdas in descending order on the diagonal. We call this matrix Λ:

$$V = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_d \end{bmatrix}$$

With this, we turn to the *eigendecomposition of a matrix*. We need to have a symmetric matrix A and then its eigendecomposition is:

$$A = V \Lambda V^{-1} \tag{8.3}$$

The only condition is that all eigenvectors $\mathbf{v}_i$ are linearly independent. Since Ξ is a symmetrical matrix with linearly independent eigenvectors, we can use the eigendecomposition to get the following equations which hold for any covariance matrix Ξ:

$$\Xi = V \Lambda V^{-1} \tag{8.4}$$

$$\Xi V = V \Lambda \tag{8.5}$$

Since V is orthonormal,[2] we also have $V^\top V = I$. Now we are ready to return to $Z = XQ$. Let us take a look at the transformed data Z. We can express the covariance of Z as the covariance of X multiplied by Q:

$$\Xi_Z = \frac{1}{d}((Z - MEAN(Z))^\top (Z - MEAN(Z))) = \tag{8.6}$$

$$= \frac{1}{d}((XQ - MEAN(X)Q)^\top (XQ - MEAN(X)Q)) = \tag{8.7}$$

$$= \frac{1}{d} Q^\top (X - MEAN(X))^\top (X - MEAN(X)) Q = \tag{8.8}$$

$$= Q^\top \Xi_X Q \tag{8.9}$$

[2] We omit the proof but it can be found in any linear algebra textbook, such as e.g. [1].

We now have to choose a matrix Q so that we get what we want (correlation zero and features ordered according to variance). We simply chose $Q := V$. Then we have:

$$\Xi_Z = V^\top \Xi_X V = V^\top V \Lambda = \Lambda \tag{8.10}$$

Let us see what we have achieved. All elements except the diagonal elements of Ξ_Z are zero, which means that the only correlation left in Z is along the diagonal. This is the covariance of a variable with itself, which is actually the variance we have encountered earlier, and the matrix is ordered in descending variance ($VAR(X_i) = COV(X_i, X_i) = \lambda_i$). This is everything we wanted. Note that we have done PCA for the 2D case with matrices but the same ideas hold for tensors. More on the principal component analysis can be found in [2].

So we have seen how we can create a different representation of the same data such that the features it is described with have a covariance of zero, and are sorted by variance. In doing so we have created a distributed representation of the data, since a column named 'height' does not exist anymore, and we have synthetic columns. The point here is that we can build various distributed representations, but we have to know what constraint we want the final data to obey. If we want this constraint to be left unspecified and we want to specify it not directly but by feeding examples, then we will have to employ a more general approach. This is the approach that leads us to autoencoders, which offer a surprising generality across many tasks.

8.2 Different Autoencoder Architectures

An autoencoder is a three-layered feed-forward neural network. They have one peculiarity: the targets $\mathbf{t}$ are actually the same values as inputs $\mathbf{x}$, which means that the task of the autoencoder is simply to recreate the inputs. So autoencoders are a form of unsupervised learning. This entails that the output layer has to have the same number of neurons as the input layer. This is all that is needed for a feed-forward neural network to be called an *autoencoder*. We can call this version the 'plain vanilla autoencoder'. There is a problem right away for plain vanilla autoencoders. If there are at least as many neurons in the hidden layer layer as there are in the input and output layer, the autoencoder is in danger of learning the identity function. This leads to a constraint, namely that there have to be less neurons in the hidden layer than in the input and output layers. We can call autoencoders which satisfy this property *simple autoencoders*. The outputs of the hidden layer of a fully trained autoencoder constitute a distributed representation, similar to PCA, and, as with PCA, this representation can be fed to a logistic regression or a simple feed-forward neural network as input and it will produce much better results than the regular representation.

But we can take another path, which is called *sparse autoencoders*. Let us say we constrain the number of neurons on the hidden layers to be at most double the number of neurons in the input layer, but we add a heavy dropout of e.g. 0.7. Then, we will have for each iteration less hidden neurons than input neurons, but at the same

time we will produce a large hidden layer vector. This large hidden layer vector is a (very large) distributed representation. What is happening here intuitively speaking is that simple autoencoders make a compact distributed representation, which is a different representation of the input. This makes it more easy for a simple neural network to digest it and process it, resulting in higher accuracy. Sparse autoencoders digest the inputs in the same way, but in addition, they learn redundancies and offer a more 'dilluted' and bigger vector, which is even simpler to process well. Recall how the hyperplane works in multiple dimensions and this will make sense. There is a different way to define sparse autoencoders, via a sparsity rate, which forces the activations below a certain threshold to be considered zero, it is similar to our approach.

We can also make the autoencoder's job harder, by inserting some noise into the input. This is done by creating a copy of the input with inserted random numbers at a fixed amount, e.g. on randomly chosen 10% of the input. The targets are a copy of the inputs without noise. These autoencoders are called *denoising autoencoders*. If we add explicit regularization, we obtain a flavour of autoencoders known as *contractive autoencoders*. Figure 8.1 offers an illustration of the various types of autoencoders. There are many other types of autoencoders, but they are more complex and fall outside the scope of this book. We point the interested reader to [3].

All of the autoencoders are used to preprocess data for a simple feed-forward neural network. This means that we have to get the preprocessed data from the autoencoder. This data is not the output of the whole autoencoder, but the output of the middle (hidden) layer, which is the layer that does the donkey work.

Let us address a technical issue. We have seen but not formally introduced the concept of a *latent variable*. A latent variable is a variable which lies in the background and is correlated with one or many 'visible' variables. We have seen an example in Chap. 3 when we addressed PCA in an informal manner, and we had synthetic properties behind 'height' and 'weight'. These are a prime example of a latent variable. When we hypothesize a latent variable (or create it), we postulate we have a probability distribution to define it. Note that it is a philosophical question whether we *discover* or *define* latent variables, but it is clear that we want our latent variables (the defined ones) to follow as closely as possible the latent variables in nature (the ones that we measure or discover). A distributed representation is a probability dis-

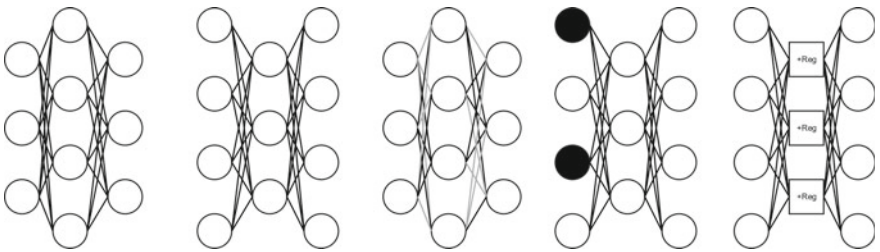


Fig. 8.1 Plain vanilla autoencoder, simple autoencoder, sparse autoencoder, denoising autoencoder, contractive autoencoder

tribution of latent variables which hopefully are the objective latent variables and learning will conclude when they are very similar. This means that we have to have a way of measuring similarities between probability distributions. This is usually done via the Kullback-Leibler divergence, which is defined as:

$$\mathbb{KL}(P, Q) := \sum_{n=1}^{N} P(n) \log \frac{P(n)}{Q(n)} \tag{8.11}$$

where P and Q are two probability distributions. Notice that $\mathbb{KL}(P, Q)$ is not symmetric (it will change if you change the P and Q). Traditionally, the Kullback-Liebler divergence is denoted as D_{KL}, but the notation we used is more consistent with the other notation in the book. There are a number of sources which provide more detail, but we will refer the reader to [3]. Autoencoders are a relatively old idea, and they were first proposed by Dana H. Ballard in 1987 [4]. Yann LeCun [5] also considered similar structures independently from Ballard. A good overview of the many types of autoencoders and their functionality can be found in [6] as an introduction to the stacked denoising autoencoders which we will reproduce in the next section.

8.3 Stacking Autoencoders

If autoencoders seem like LEGO bricks, you have the right intuition, and in fact they may be stacked together, and then they are called *stacked autoencoders*. But keep in mind that the real result of the autoencoder is not in the output layer, but the activations in the middle layer, which are then taken and used as inputs in a regular neural network. This means that to stack them we need not simply stick one autoencoder after the other, but actually combine their middle layers as shown in Fig. 8.2. Imagine that we have two simple autoencoders of size (13, 4, 13) and

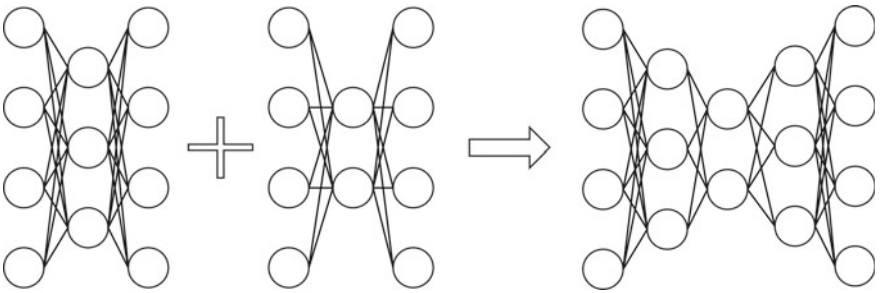


Fig. 8.2 Stacking a (4, 3, 4) and a (4, 2, 4) autoencoder resulting in a (4, 3, 2, 3, 4) stacked autoencoder

(13, 7, 13). Notice that if they want to process the same data they have to have the same input (and output) size. Only the middle layer or autoencoder architecture may vary. For simple autoencoders, they are stacked by creating a 13, 7, 4, 7, 13 stacked autoencoder. If you think back on what the autoencoder does, it makes sense to create a natural bottleneck. For other architectures, it may make sense to make a different arrangement. The real result of the stacked autoencoder is again the distributed representation built by the middle layer. We will be stacking denoising autoencoders following the approach of [6] and we present a modification of the code available at `https://blog.keras.io/building-autoencoders-in-keras.html`. The first part of the code, as always, consists of import statements:

```
from keras.layers import Input, Dense
from keras.models import Model
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
```

The last line of code loads the MNIST dataset from the Keras repositories. You could do this by hand, but Keras has a built-in function that lets you load MNIST into Numpy[3] arrays. Note that the Keras function returns two pairs, one consists of train samples and train labels (both as Numpy arrays of 60000 rows), and the second consisting of test samples and test labels (again, Numpy arrays, but this time of 10000 rows). Since we do not need labels, we load them in the _ *anonymous* variable, which is basically a trash can, but we need it since the function needs to return two pairs and if we do not provide the necessary variables, the system will crash. So we accept the values and dump them in the variable _. The next part of the code preprocesses the MNIST data. We break it down in steps:

```
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
noise_rate = 0.05
```

This part of the code turns the original values ranging from 0 to 255 to values between 0 and 1, and declares their Numpy types as *float32* (decimal number with a precision of 32). It also introduces a noise rate parameter, which we will be needing shortly.

```
x_train_noisy = x_train + noise_rate * np.random.normal
(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_rate * np.random.normal
(loc=0.0, scale=1.0, size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0.0, 1.0)
x_test_noisy = np.clip(x_test_noisy, 0.0, 1.0)
```

This part of the code introduces the noise into a copy of the data. Note that the `np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)`

[3]Numpy is the Python library for handling arrays and fast numerical computations.

introduces a new array, of the size of the `x_train` array populated with a Gaussian random variable with `loc=0.0` (which is actually the mean), and a `scale=1.0` (which is the standard deviation). This is then multiplies with the noise rate and added to the data. The next two rows actually make sure that all the data is bound between 0 and 1 even after the addition. We can now reshape our arrays which are currently (60000, 28, 28) and (10000, 28, 28) into (60000, 784) and (10000, 784) respectively. We have touched upon this idea when we have first introduced MNIST, and now we can see the code in action:

```
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
x_train_noisy = x_train_noisy.reshape((len(x_train_noisy), np.prod(x_train_noisy.shape[1:])))
x_test_noisy = x_test_noisy.reshape((len(x_test_noisy), np.prod(x_test_noisy.shape[1:])))
assert x_train_noisy.shape[1] == x_test_noisy.shape[1]
```

The first four rows reshape the four arrays we have, and the final row is a test to see whether the sizes of the noisy train and test vectors are the same. Since we are using autoencoders, this has to be the case. If they are somehow not the same, the whole program will crash here. It might seem strange to want to crash the program on purpose, but in this way we actually gain control, since we know where it has crashed, and by using as many tests as we can, we can quickly debug even very complex codes. This ends the preprocessing part of the code, and we continue to build the actual autoencoder:

```
inputs = Input(shape=(x_train_noisy.shape[1],))
encode1 = Dense(128, activation='relu')(inputs)
encode2 = Dense(64, activation='tanh')(encode1)
encode3 = Dense(32, activation='relu')(encode2)
decode3 = Dense(64, activation='relu')(encode3)
decode2 = Dense(128, activation='sigmoid')(decode3)
decode1 = Dense(x_train_noisy.shape[1], activation='relu')(decode2)
```

This offers a different view from what we are used to, since now we manually connect the layers (you can see the layer sizes, 128, 64, 32, 64, 128). We have added different activations just to show their names, but you can freely experiment with different combinations. What is important here to notice is that the input size and the output size are both equal to `x_train_noisy.shape[1]`. Once we have the layers specified, we continue to build the model (feel free to experiment with different optimizers[4] and error functions[5]):

```
autoencoder = Model(inputs, decode1)
autoencoder.compile(optimizer='sgd', loss='mean_squared_error',metrics=['accuracy'])
autoencoder.fit(x_train,x_train,epochs=5,batch_size=256,shuffle=True)
```

[4]Try'adam'.
[5]Try'binary_crossentropy'.

You should also increase the number of epochs once you get the code to work. Finally we get to the last part of the autoencoder code when we evaluate, predict and pull out the weight of the deepest middle layer. Note that when we print all the weight matrices, the right weight matrix (the result of the stacked autoencoder) is the first one where the dimensions start to increase (in our case (32, 64)):

```
metrics = autoencoder.evaluate(x_test_noisy, x_test, verbose=1)
print()
print("%s:%.2f%%" % (autoencoder.metrics_names[1], metrics[1]*100))
print()
results = autoencoder.predict(x_test)
all_AE_weights_shapes = [x.shape for x in autoencoder.get_weights()]
print(all_AE_weights_shapes)
ww=len(all_AE_weights_shapes)
deeply_encoded_MNIST_weight_matrix = autoencoder.get_weights()[int((ww/2))]
print(deeply_encoded_MNIST_weight_matrix.shape)
autoencoder.save_weights("all_AE_weights.h5")
```

The resulting weight matrix is stored in the variable `deeply_encoded_MNIST_weight_matrix`, which contains the trained weights for the middlemost layer of the stacked autoencoder, and this should afterwards be fed to a fully connected neural network together with the labels (the ones we dumped). This weight matrix is a distributed representation of the original dataset. A copy of all weights is also saved for later use in a H5 file. We have also added a variable `results` to make predictions with the autoencoder, but this is mainly used for assessing autoencoder quality, and not for actual predictions.

8.4 Recreating the Cat Paper

In this section, we recreate the idea presented in the famous 'cat paper', with the official title *Building High-level Features Using Large Scale Unsupervised Learning* [7]. We will present a simplification to better delineate the subtleties of this amazing paper. This paper became famous since the authors made a neural network which was capable of learning to recognize cats just by watching YouTube videos. But what does that mean? Let us take a step back. The 'watching' means simply that the authors sampled frames from 10 million YouTube videos, and took a number of 200 by 200 images in RGB. Now, the tricky part: what does it mean to 'recognize a cat'? Surely it could mean that they build a classifier which was trained on images of cats and then it classified cats. But the authors did not do this. They gave the network an unlabelled dataset, and then tested it against images of cats from ImageNet (negative samples were just random images not containing cats). The network was trained by learning to reconstruct inputs (it means that the number of output neurons is the same

as the number of input neurons), which makes it an autoencoder. Result neurons are found in the middle part of the autoencoder. The network had a number of result neurons (let us say there are 4 of them for simplicity), and they noticed that the activations of those neurons formed a pattern (activations are sigmoid so they range from 0 to 1). If the network was classifying something similar to what it has seen (cats), it formed a pattern, e.g. neuron 1 was 0.1, neuron 2 was 0.2, neuron 3 was 0.5 and neuron 4 was 0.2. If it got something it did not know about, neuron 1 would get 0.9, and the others 0. In this way, an implicit label generation was discovered.

But the cat paper presented another cool result. They asked the network what was in the videos, and the network drew the face of a cat (as the tech media formulated it). But what does that mean? It means that they took the best performing 'cat finder' neuron, in our case neuron 3, and found the top 5 images it recognized as cats. Suppose the cat finder neuron had activations of 0.94, 0.96, 0.97, 0.95 and 0.99 for them. They then combined and modified this image (with numerical optimization, similar to gradient descent) to find a new image such that given neuron gets the activation 1. Such image was a drawing of a cat face. It may seem like science fiction, but if you think about it, it is not that unusual. They picked the best cat recognizer neuron, and then selected top 5 images it was most confident of. It is easy to imagine that these were the clearest pictures of cat faces. It then combined them, added a little contrast, and there you have it—an image which produced the activation of 1 in that neuron. And it was an image of a cat different from any other image in the dataset. The neural network was set loose to watch YouTube videos of cats (without knowing it was looking at cats), and once prompted to answer what it was looking at, the network drew a picture of a cat.

We scaled down a bit, but the actual architecture used was immense: 16000 computer cores (your laptop has 2 or 4), and the network was trained over three days. The autoencoder had over 1 billion trainable parameters, which is still only a fraction of the number of synapses in the human visual cortex. The input images were a 200 by 200 by 3 tensors for training, and for testing 32 by 32 by 3. The authors used a receptive field of 18 by 18 similar to the convolutional networks, but the weights were not shared across the image but each 'tile' of the field had its own weights. The number of feature maps used was 8. After this, there was a pooling layer using L2 pooling. L2 pooling takes a region (e.g. 2 by 2) in the same way as max-pooling, but instead of outputting the max of the inputs, it squares all inputs, adds them, and then takes the square root of it and presents this as the output.

The overall autoencoder has three parts, all of them are of the same architecture. A part takes the input, applies the receptive field (no shared weights), and then applies L2 pooling, and finally a transformation known as local contrast normalization. After this part is finished, there are two more exactly the same. The whole network is trained with asynchronous SGD. This means that there are many SGDs working at once over different parts, and have a central weights repository. At the beginning of each phase, every SGD asks the repository for the update on weights, optimizes them a bit, and

then sends them back to the repository so that other instances running asynchronous SGD can use them. The minibatch size used was 100. We omit the rest of the details, and refer the reader to the original paper.

References

1. S. Axler, *Linear Algebra Done Right* (Springer, New York, 2015)
2. R. Vidal, Y. Ma, S. Sastry, *Generalized Principal Component Analysis* (Springer, London, 2016)
3. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, Cambridge, 2016)
4. D.H. Ballard, Modular learning in neural networks, in *AAAI-87 Proceedings* (AAAI, 1987), pp. 279–284
5. Y. LeCun, *Modeles connexionnistes de l'apprentissage (Connectionist Learning Models)* (Université P. et M. Curie (Paris 6), 1987)
6. P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol, Stacked denoising autoencoders: learning useful representations in a deep network with a local denoising criterion. J. Mach. Learn. Res. **11**, 3371–3408 (2010)
7. Q.V. Le, M.A. Ranzato, R. Monga, M. Devin, K. Chen, G.S. Corrado, J. Dean, A.Y. Ng, Building high-level features using large scale unsupervised learning, in *Proceedings of the 29th International Conference on Machine Learning. ICML* (2012)

Neural Language Models

9

9.1 Word Embeddings and Word Analogies

Neural language models are distributed representations of words and sentences. They are learned representations, meaning that they are numerical vectors. A *word embedding* is any method which converts words in numbers, and therefore, any learned neural language model is a way of obtaining word embeddings. We use the term 'word embedding' to denote a very concrete numerical representation of a certain word or words, represent 'Nowhere fast' as $(1, 0, 0, 5.678, -1.6, 1)$. In this chapter, we focus on the most famous of the neural language models, the Word2vec model, which learns vectors which represent words with a simple neural network.

This is similar to the predict-next setting for recurrent neural networks, but it gives an added bonus: we can calculate word distances and have similar words only a short distance away. Traditionally, we can measure the distances of two words as strings with the *Hamming distance* [1]. For measuring the Hamming distance, two strings have to be of the same length and the distance is simply the number of characters that are different. The Hamming distance between the words 'topos' and 'topoi' is 1, while the distance between 'friends' and 'fellows' is 5. Note that the distance between 'friends' and '0r$8MMs' is also 5. It can easily be normalized to a percentage by dividing it by the words' length. You can probably see already how this would be a useful but very limited technique for processing language.

The Hamming distance is the simplest method from a wide variety of string similarity measures collectively known as *string edit distance metrics*. More evolved forms such as Levenshtein distance [2] or Jaro–Winkler [3,4] distance can compare strings of different lengths and penalize differently various errors, such as insertion, deletion or edit. All of these are measures of a word by the form of the word. They would be useless in comparing 'professor' and 'teacher', since they would never recognize the similarity in meaning. This is why, we want to embed a word in a

vector in a way which will convey information about the meaning of the word (i.e. its use in our language).

If we represent words as vectors, we need to have a distance measure between vectors. We have touched upon this idea a number of times before, but we can now introduce the notion of the *cosine similarity* of vectors. A good overview of cosine similarity is given in [5]. Cosine similarity of two n-dimensional vectors $\mathbf{v}$ and $\mathbf{u}$ is given by:

$$\mathbb{CS}(\mathbf{v}, \mathbf{u}) := \frac{\mathbf{v} \cdot \mathbf{u}}{||\mathbf{v}|| \cdot ||\mathbf{u}||} = \frac{\sum_{i=1}^{n} v_i u_i}{\sqrt{\sum_{i=1}^{n} v_i^2} \sqrt{\sum_{i=1}^{n} u_i^2}} \tag{9.1}$$

Where v_i and u_i are components of $\mathbf{v}$ and $\mathbf{u}$, and $||\mathbf{v}||$ and $||\mathbf{u}||$ denote the norms of the vectors $\mathbf{v}$ and $\mathbf{u}$ respectively. The cosine similarity ranges from 1 (equal) to -1 (opposite), and 0 means that there is no correlation. When using the bag of words, one-hot encoding s or similar word embeddings the cosine similarity ranges from 0 to 1, since the vectors representing fragments do not contain negative components. This means that 0 takes the meaning of 'opposite' in such contexts.

We will now continue to show the Word2vec neural language model [6]. In particular, we will address the questions of what input does it need, what will it give as an output, does it have parameters to tune it and how can we use it in a complete system, i.e. how does it interact with other components of a bigger system.

9.2 CBOW and Word2vec

The Word2vec model can be built with two different architectures, the skip-gram and the Word2vec. Both of these are actually shallow neural networks with a twist. To see the difference, we will use the sentence 'Who are you, that you do not know your history?'. First, we clean the sentence from uppercase and interpunction. Both architectures use the context of the word (the words around it) as well as the word itself. We must define in advance how large will the context be. For the sake of simplicity, we will be using a context of size 1. This means that the context of a word consists of one word before and one word after. Let us break or sentence into word and context pairs:

We have already noted that both versions of the Word2vec are learned models, and this means they must learn something. The skip-gram model learns to predict a word from the context given the middle word. This means that if we give the model 'are' it should predict 'who', if we give it 'know' it should predict 'not' or 'your'. The CBOW version does the opposite, assuming the context to be 1, it takes two words[1] from the context (we will call them $c1$ and $c2$) and uses it to predict the middle or main word (which we will denote by m).

[1] If the context were 2, it would take 4 words, two before the main word and two after.

Context	Word
‘are’	‘who’
‘who’, ‘you’	‘are’
‘are’, ‘that’	‘you’
‘you’, ‘you’	‘that’
‘that’, ‘do’	‘you’
‘you’, ‘not’	‘do’
‘do’, ‘know’	‘not’
‘not’, ‘your’	‘know’
‘know’, ‘history’	‘your’
‘your’	‘history’

The production of the word embeddings is structurally quite similar to autoencoders. To make the network which produces the embeddings, we use a shallow feedforward network. The input layer will receive word index vectors, so we will need as many input neurons as there are unique words in the vocabulary. The number of hidden neurons is called *embedding size* (suggested values range between 100 and 1000, which is considerably less than the vocabulary size even for modest datasets), and the number of output neurons is the same as input neurons. The input to hidden connections are linear, i.e. they have no activation function, and the hidden to output have softmax activations. The weights of the input to hidden are the deliverables of the model (similar to the autoencoder deliverables), and this matrix contains as rows the individual word vectors for a particular word. One of the easiest methods of extracting the proper word vector is to multiply this matrix by the word index vector for a given word. Note that these weights are trained with backpropagation in the usual way. Figure 9.1 offers an illustration of the whole process. If something is unclear, we ask the reader to fill out the details for herself by using what we have previously covered in this book—there should be no problem in doing this.

Before continuing to the code for the CBOW Word2vec, we must correct a historical mistake. The idea behind Word2vec is that the meaning of a given word is determined by a context, which is usually defined as the way the word is used in a language. Most deep learning textbooks (including the official TensorFlow documentation on Word2vec) attribute this idea to a paper from 1954 by Harris [7], and note that he idea came to be known in linguistics as the distributional hypothesis in 1957 [8]. This is actually wrong. The first time this idea was proposed was in Wittgenstein’s *Philosophical Investigations* in 1953 [9], and since ordinary language philosophy and philosophical logic (the area of logic dealing mainly with language formalization) played a major role in the history of natural language processing, the historical merit must be acknowledged and attributed correctly.

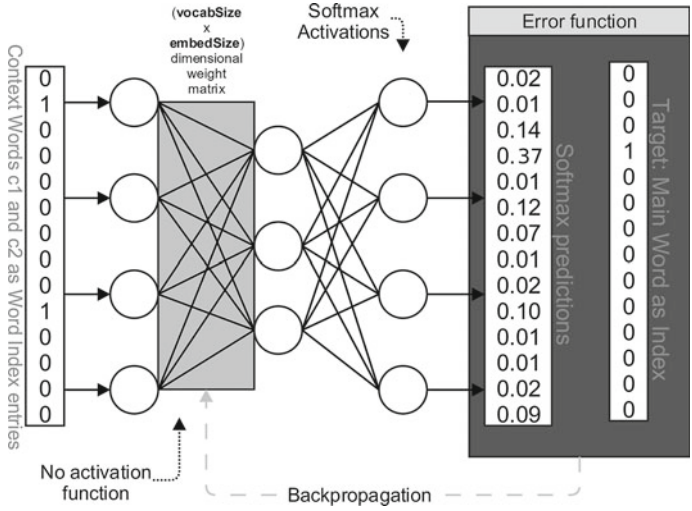


Fig. 9.1 CBOW Word2vec architecture

9.3 Word2vec in Code

In this and the next section, we give an example of a CBOW Word2vec implementation. All the code in these two sections should be placed in one Python file, since it is connected. We start with the usual imports and hyperparameters:

```
from keras.models import Sequential
from keras.layers.core import Dense
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
text_as_list=["who","are","you","that","you","do","not","know","your","history"]
embedding_size = 300
context = 2
```

The `text_as_list` can hold any text, so you can put here your text, or use the parts of the code from the recurrent neural network which parse a text file into a list of words. The embedding size is the size of the hidden layer (and, consequently, that the word vectors will have). The context is the number of words before and after the given word which will be used this. If the context is 2, this means we will use two words before the main word and two words after the main word to create the inputs (the main word will be the target). We continue to the next block of code which is exactly the same as the same part of code for recurrent neural networks:

```
distinct_words = set(text_as_list)
number_of_words = len(distinct_words)
```

```
word2index = dict((w, i) for i, w in enumerate(distinct_words))
index2word = dict((i, w) for i, w in enumerate(distinct_words))
```

This code creates word and index dictionaries in both ways, one where the word is the key and the index is the value and another one where the index is the key and the word is the value. The next part of the code is a bit tricky. It creates a function that produces two lists, one is a list of main words, and the other is a list of context words for a given word (it is a list of lists):

```
def create_word_context_and_main_words_lists(text_as_list):
    input_words = []
    label_word = []
    for i in range(0,len(text_as_list)):
        label_word.append((text_as_list[i]))
        context_list = []
        if i >= context and i<(len(text_as_list)-context):
            context_list.append(text_as_list[i-context:i])
            context_list.append(text_as_list[i+1:i+1+context])
            context_list = [x for subl in context_list for x in subl]
        elif i<context:
            context_list.append(text_as_list[:i])
            context_list.append(text_as_list[i+1:i+1+context])
            context_list = [x for subl in context_list for x in subl]
        elif i>=(len(text_as_list)-context):
            context_list.append(text_as_list[i-context:i])
            context_list.append(text_as_list[i+1:])
            context_list = [x for subl in context_list for x in subl]
        input_words.append((context_list))
    return input_words, label_word
input_words,label_word = create_word_context_and_main_words_lists(text_as_list)
input_vectors = np.zeros((len(text_as_list), number_of_words), dtype=np.int16)
vectorized_labels = np.zeros((len(text_as_list), number_of_words), dtype=np.int16)
for i, input_w in enumerate(input_words):
    for j, w in enumerate(input_w):
        input_vectors[i, word2index[w]] = 1
        vectorized_labels[i, word2index[label_word[i]]] = 1
```

Let us see what this block of code does. The first part is the definition of a function that takes in a list of words and returns two lists. One is a copy of that list of words (named `label_word` in the code), and the second is `input_words`, which is a list of lists. Each list in the list carries the words from the context of the corresponding word in `label_word`. After the whole function is defined, it is called on the variable `text_as_list`. After that two matrices to hold the word vectors corresponding to the two lists are created with zeros, and the final part of the code updates the corresponding parts of the matrices with 1, to make a final model of the context for inputs and of the main word for the target. The next part of the code initializes and trains the Keras model:

```
word2vec = Sequential()
word2vec.add(Dense(embedding_size, input_shape=(number_of_words,), activation=
"linear", use_bias=False))
word2vec.add(Dense(number_of_words, activation="softmax", use_bias=False))
word2vec.compile(loss="mean_squared_error", optimizer="sgd", metrics=['accuracy'])
word2vec.fit(input_vectors, vectorized_labels, epochs=1500, batch_size=10, verbose=1)
metrics = word2vec.evaluate(input_vectors, vectorized_labels, verbose=1)
print("%s: %.2f%%" % (word2vec.metrics_names[1], metrics[1]*100))
```

The model follows closely the architecture we presented in the last section. It does not use biases since we will be taking out the weights and we do not want any information to be anywhere else. The model is trained for 1500 epochs and you may want to experiment with these. If one wants to make a skip-gram model instead, one should just interchange these matrices, so the part that says `word2vec.fit(input_vectors, vectorized_labels, epochs =1500, batch_size=10, verbose=1)` should be changed to `word2vec.fit(vectorized_labels, input_vectors, epochs=1500, batch_size=10, verbose=1)` and you will have a skip-gram. Once we have this, we just take out the weights with the following code:

```
word2vec.save_weights("all_weights.h5")
embedding_weight_matrix = word2vec.get_weights()[0]
```

And we are done. The first line of this code returns the word vectors for all the words, in the form of a `number_of_words`×`embedding_size` dimensional array, and we can pick the appropriate row to get the vector for that word. The first line saves all the weights in the network to a H5 file. You can do several things with word2vec and for all of them we need these weights. First, we may just learn weights from scratch, as we did with our code. Second, we might want to fine-tune a previously learned word embedding (suppose it was learned from Wikipedia data), and in that case, we want to load previously saved weights in a copy of the original model and train it on new texts that are perhaps more specific and more closely connected with e.g. legal texts. The third way we may use word vectors is to simply use them instead of one-hot encoded words (or a Bag of Words), and feed them in another neural network which has the task of e.g. predicting sentiment.

Note that the H5 file contains all the weights of the network, and we want to use just the weight matrix from the first layer,[2] and this matrix is fetched by the last line of code and named `embedding_weight_matrix`. We will be using `embedding_weight_matrix` in the code in the next section (which should be in the same file as the code of this section).

[2]If we were to save and load from a H5 file, we would be saving ans loading all the weights in a new network of the same configuration, possibly fine-tuning them and then taking out just the weight matrix with the same code we used here.

9.4 Walking Through the Word-Space: An Idea That Has Eluded Symbolic AI

Word vectors are a very interesting type of word embeddings, since they allow much more than meets the eye. Traditionally, reasoning is viewed as a symbolic concept which ties together various relations of an object or even various relations of various objects. Objects, and symbols denoting them, have been seen as logically primitive. This means that they were defined, and as such void of any content other than that which we explicitly placed in them. This has been a dogma of the logical approach to artificial intelligence (GOFAI) for decades. The main problem is that rationality was equated with intelligence, and this meant that the higher faculties, where the one that embodied intelligence. Hans Moravec [10] discovered that higher faculties (such as chess playing and theorem proving) where in fact easier than recognizing cats on unlabelled photos, and this caused the AI community to rethink the previously accepted concept of intelligence, and with it ideas of *low faculty reasoning* became interesting.

To explain what low faculty reasoning is we turn to an example. If you consider two sentences 'a tomato is a vegetable' and 'a tomato is a suspension bridge', you might conclude that they are both false, and you would technically be right. But most people (and intelligent animals) endorse an idea of fuzziness which takes into account the degree of wrongness. You are less wrong by uttering 'a tomato is a vegetable' than 'a tomato is a suspension bridge'. Note also that these are not sentences of natural phenomena, but sentences about linguistic classification and the social conventions on language use. You are not referring to objects (except for 'tomato'), but to classes defined by descriptions (composed of properties) or examples (which share to a degree a number of common properties). Notice that you are using singular terms in all three cases, and the only symbolic part is '_is a_', which is irrelevant.

If an agent were locked in a room and given only books in a foreign language to read, we would consider her intelligent if she would be able to find patters, such as a word which denote places and words that denote people. So if she would classify two sentences 'Luca frequenta la scuola elementare Pedagna' and 'Marco frequenta la scuola elementare Zolino' as being similar, she would display a certain degree of intelligence. She might even go so far to say that in this context 'Luca' is to 'Pedagna' as 'Marco' is to 'Zolino'. If she was given a new sentence 'Luca vive in Pedagna', she might infer the sentence 'Marco vive in Zolino', and she might hit it spot on. The question of semantically similar terms very quickly became a question of reasoning.

We can actually find similarities of terms in our datasets an even reason with them in this fashion using Word2vec. To see how, let us return to our code. The following code goes immediately after the code from the last section (in the same Python file). We will use the `embedding_weight_matrix` to find an interesting way to measure word similarities (actually word vector clusterings) and to calculate and reason with words with the help of word vectors. To do this, we first run

`embedding_weight_matrix` through PCA and keep just the first two dimensions,[3] and then simply draw the results to a file:

```
pca = PCA(n_components=2)
pca.fit(embedding_weight_matrix)
results = pca.transform(embedding_weight_matrix)
x = np.transpose(results).tolist()[0]
y = np.transpose(results).tolist()[1]
n = list(word2index.keys())
fig, ax = plt.subplots()
ax.scatter(x, y)
for i, txt in enumerate(n):
    ax.annotate(txt,(x[i],y[i]))
plt.savefig('word_vectors_in_2D_space.png')
plt.show()
```

This produces Fig. 9.2. Note that we need a significantly larger dataset than our nine word sentence to be able to learn similarities (and to see them in the plot), but you can experiment with different datasets using the parser we used with recurrent neural networks.

Reasoning with word vectors is also quite straightforward. We need to take the corresponding vectors from `embedding_weight_matrix` and do simple arithmetic with them. They are all of the same dimensionality, which means it is quite easy to add and subtract them. Let *w2v(someword)* denote the trained word embedding

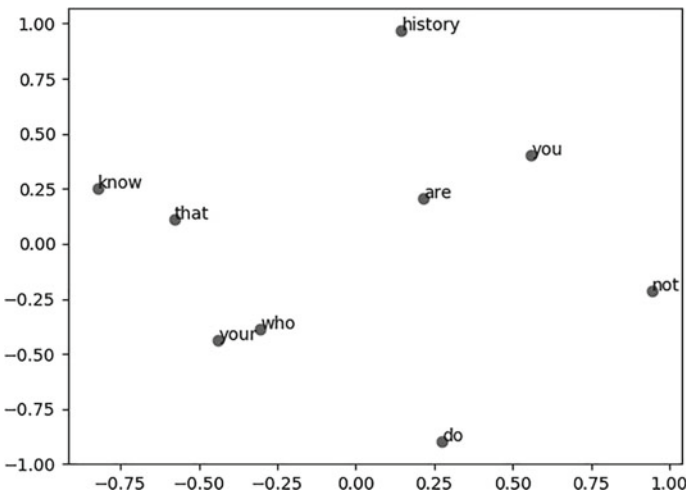


Fig. 9.2 Word similarity clusters in transformed 2D space

[3]More precisely: to transform the matrix into a decorrelated matrix whose columns are arranged in descending variance and then keep the first two columns.

for the word ‘someword’. To recreate the classic example, take *w2v(king)*, subtract from it *w2v(man)* add to it *w2v(woman)* and the result would be near *w2v(queen)*. The same holds even if we use PCA to transform the vectors and keep just the first two or three components, although it is sometimes more distorted. This depends on the quality and size of the dataset, and we suggest the reader to try to make a script which does this over a large dataset as an exercise.

References

1. R.W. Hamming, Error detecting and error correcting codes. Bell Syst. Tech. J. **29**(2), 147–160 (1950)
2. V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals. Sov. Phys. Dokl. **10**(8), 707–710 (1966)
3. M.A. Jaro, Advances in record linkage methodology as applied to the 1985 census of tampa florida. J. Am. Stat. Assoc. **84**(406), 414–420 (1989)
4. W.E. Winkler, String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage, in *Proceedings of the Section on Survey Research Methods* (American Statistical Association, 1990), pp. 354–359
5. A. Singhal, Modern information retrieval: a brief overview. Bull. IEEE Comput. Soc. Tech. Comm. Data Eng. **24**(4), 35–43 (2001)
6. T. Mikolov, T. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, in *ICLR Workshop* (2013), arXiv:1301.3781
7. Z. Harris, Distributional structure. Word **10**(23), 146–162 (1954)
8. J.R. Firth, A synopsis of linguistic theory 1930–1955, in *Studies in Linguistic Analysis* (Philological Society, 1957), pp. 1–32
9. L. Wittgenstein, *Philosophical Investigations* (MacMillan Publishing Company, London, 1953)
10. H. Moravec, *Mind Children: The Future of Robot and Human Intelligence* (Harvard University Press, Cambridge, 1988)

An Overview of Different Neural Network Architectures

10

10.1 Energy-Based Models

Energy-based models are a specific class of neural networks. The simplest energy model is the *Hopfield Network* dating back from the 1980s [1]. Hopfield networks are often thought to be very simple, but they are quite different from what we have seen before. The network is made of neurons, and all of these neurons are connected among them with weights w_{ij} connecting neurons n_i and n_j. Each neuron has a threshold associated with it, and we denote it by b_i. All neurons have 1 or -1 in them. If you want to process and image, you can think of -1 as white and 1 as black (no shades of grey here). We denote the inputs we place in neurons by x_i. A simple Hopfield network is shown in Fig. 10.1a.

Once a network is assembled, the training can start. The weights are updated by the following rule, where n denotes an individual training sample:

$$w_{ij} = \sum_{n=1}^{N} x_i^{(n)} x_j^{(n)} \tag{10.1}$$

Then we compute activations for each neuron:

$$y_i = \sum_j w_i j x_j \tag{10.2}$$

There are two possibilities on how to update weights. We can either do it synchronously (all weights at the same time) or asynchronously (one by one, this is the standard way). In Hopfield networks there is no recurrent connections, i.e. $w_{ii} = 0$ for all i, and all connections are symmetric, i.e. $w_{ij} = w_{ji}$. Let us see how the simple Hopfield Network shown in Fig. 10.1b processes the simple 1 by 3 pixel 'images' in Fig. 10.1c, which we represent by vectors $\mathbf{a} = (-1, 1, -1)$, $\mathbf{b} = (1, 1, -1)$ and

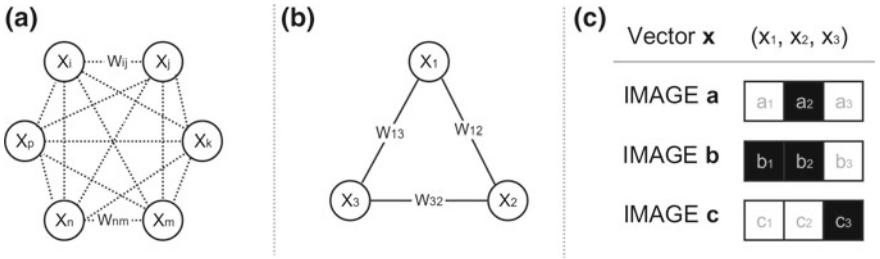


Fig. 10.1 Hopfield networks

$\mathbf{c} = (-1, -1, 1)$. Using the equation above, we calculate the weight updates with the update equation:

$$w_{11} = w_{22} = w_{33} = 0$$

$$w_{12} = a_1 a_2 + b_1 b_2 + c_1 c_2 = -1 \cdot 1 + 1 \cdot 1 + (-1) \cdot (-1) = 1$$

$$w_{13} = -1$$

$$w_{23} = -3$$

Hopfield networks have a global measure of success, similar to the error function of regular neural networks, called the *energy*. Energy is defined for each stage of network training as a single value for the whole network. It is calculated as:

$$ENE = -\sum_{i,j} w_{ij} y_i y_j + \sum_i b_i y_i \tag{10.3}$$

The as learning progresses, ENE either stays the same or diminishes, and this is how Hopfield networks reach local minima. Each local minimum is a memory of some training samples. Remember logical functions and logistic regression? We needed two input neurons and one output neurons for conjunction and disjunction, and an additional hidden one for XOR. We need three neurons in Hopfield networks for conjunction and disjunction and four for XOR.

The next model we briefly present are *Boltzmann machines* first presented in 1985 [2]. At first glance, they are very similar to Hopfield networks, but have input neurons and hidden neurons as well, which are all interconnected with weights. These weights are non-recurrent and symmetrical. A sample Boltzmann machine is displayed in Fig. 10.2a. Hidden units are initialized at random, and they build a hidden representation to mimic the inputs. These form two probability distributions, which can be compared with the Kullback-Leibler divergence $\mathbb{KL}$. The main goal then becomes clear, calculate $\frac{\partial \mathbb{KL}}{\partial w}$, and backpropagate it.

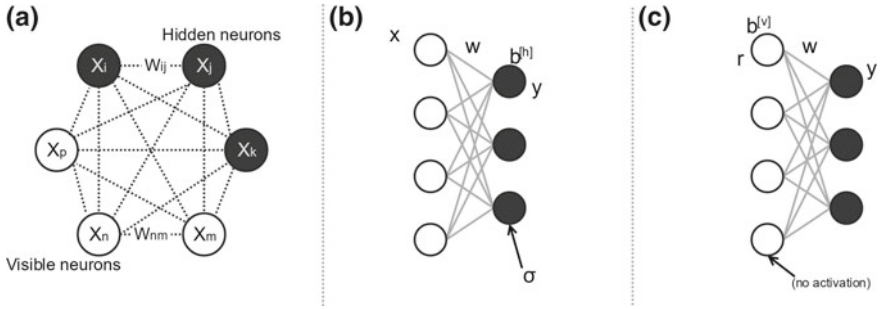


Fig. 10.2 Boltzmann machines and restricted Boltzmann machines

We turn to a subclass of Boltzmann machines, called *restricted Boltzmann machines* (RBM) [3]. Structurally speaking, restricted Boltzmann machines are just Boltzmann machines where there are no connections between neurons of the same layer (hidden to hidden and visible to visible). This seems like a minor point, but this actually makes it possible to use a modification of the backpropagation used in feed-forward networks. The restricted Boltzmann machine therefore has two layers, a visible, and a hidden. The visible layer (this is true for Boltzmann machines in general) is the place where we put in inputs and read out outputs. Denote the inputs with x_i, the biases of the hidden layer with $b_j^{[h]}$. Then, during the forward pass (see Fig. 10.2b), the RBM calculates $\mathbf{y} = \sigma(\mathbf{x}^\top \mathbf{w} + \mathbf{b}^{[h]})$. If we were to stop here, RBMs would be similar to autoencoders, but we have a second phase, the *reconstruction* (see Fig. 10.2c). During the reconstruction, the $\mathbf{y}$ are fed to the hidden layer and then passed to the visible layer. This is done by multiplying them with the same weights, and adding another set of biases, i.e. $\mathbf{r} = \mathbf{y}^\top \mathbf{w} + \mathbf{b}^{[v]}$. The difference between $\mathbf{x}$ and $\mathbf{r}$ is measured with $\mathbb{KL}$ and then this error is used in backpropagation. RBMs are fragile, and every time one gets a nonzero reconstruction, this is a good sign. Boltzmann machines are similar to logical *constraint satisfaction solvers*, but they focus on what Hinton and Sejnowski called 'weak constraints'. Notice that we moved away quite a bit from the energy function, and well back into standard neural network territory.

The final architecture we will briefly discuss is *deep belief networks* (DBN), which are just stacked RBMs. They were introduced in [4] and in [5]. They are conceptually similar to stacked autoencoders, but they can be trained with backpropagation to be generative models, or with *contrastive divergence*. In this setting, they may be even used as classifiers. Contrastive divergence is simply an algorithm that efficiently approximates the gradients of the log-likelihood. A discussion on contrastive divergence is beyond the scope of this book, but we point the interested reader to [6] and [7]. For a discussion about the cognitive aspects of energy-based models, see [8].

10.2 Memory-Based Models

The first memory-based model we will explore are *neural Turing-machines* (NTM) first proposed in [9]. Remember how a Turing-machine works: you have a read-write head and a tape which acts as a memory. The Turing-machine then is given a function in the form of an algorithm and it computes that function (takes in the given inputs and outputs the result). The neural Turing-machine is similar, but the point is to have all components trainable, so that they can do soft computation, and they should also learn how to do it well.

The neural Turing-machine acts similarly to an LSTM. It takes input sequences and outputs sequences. If we want it to output a single result, we just take the last component and discard everything else. The neural Turing-machine is built upon an LSTM, and can be seen as an architecture extending the LSTM similarly how LSTMs builds upon simple recurrent networks.

A neural Turing-machine has several components. The first one is called a *controller*, and a controller is simply an LSTM. Similar to an LSTM, the neural Turing-machine has a temporal component, and all elements are indexed by t, and the state of the machine at time t takes as inputs components calculated at $t-1$. The controller takes in two inputs: (i) raw inputs at time t, i.e. $\mathbf{x}_t$ and (ii) results of the previous step, $\mathbf{r}_t$. The neural Turing-machine has another major component, the memory, which is just a tensor denoted by M_t (it is usually just a matrix). Memory is not an input to the controller, but it is an input to the step t of the whole neural Turing-machine (the input is M_{t-1}).

The structure of a complete neural Turing-machine is shown in Fig. 10.3, but we have omitted the details.[1] The idea is that the whole neural Turing-machine should be expressed as tensors, and trainable by gradient descent. To enable this, all crisp concepts from regular Turing-machines are fuzzified, so that there is no single memory location which is accessed in separation, but all memory locations are accessed to a certain degree. But in addition to the fuzzy part, the amount of the accessed memory is also trainable, so it changes dynamically.

To reiterate: the neural Turing-machine has an LSTM (controller) which receives the outputs from the previous step, and a fresh vector of inputs, and uses them and a memory matrix to produce outputs and everything is trainable. But how do the components work? Let us now work our way from the memory upward. We will be needing three vectors, all of which the controller will produce: add vector $\mathbf{a_t}$, erase vector $\mathbf{e_t}$, and weighting vector $\mathbf{w_t}$. They are similar but used for different purposes. We will be coming back to them later to explain how they are produced.

Let us see how the memory works. The memory is represented by a matrix (or possibly higher order tensor) M_t. Each row in this matrix is called a *memory location*. If there are n rows in the memory, the controller produces a weighting vector of size n

[1] For a fully detailed view, see the blog entry of one of the creators of the NTM, https://medium.com/aidangomez/the-neural-turing-machine-79f6e806\penalty-\@Mc0a1.

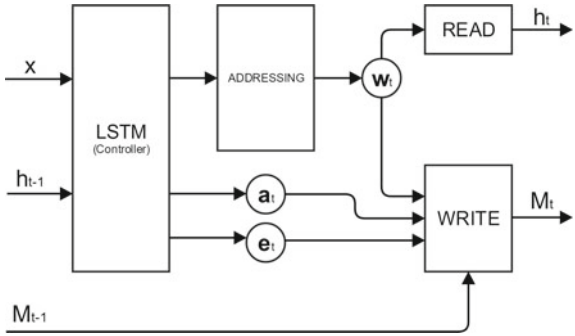


Fig. 10.3 Neural Turing-machines

(components range from 0 to 1) which indicates how much of each of those locations to take in consideration. This can be a crisp access to a one or several locations or a fuzzy access to those locations. Since this vector is trainable, it is almost never crisp. This is the reading operation, defined simply as the Hadamard product (pointwise multiplication) of m by n matrix M_t and B, where B is obtained by transposing the m-dimensional row vector $\mathbf{w_t}$, and then broadcasting its values (just copying this column $n-1$ times) to match the dimensions of M_t.

The neural Turing-machine will now write. It *always* reads and writes, but sometimes it writes very similar values, so we have the impression that the content is not changed. This is important since it is a common source of confusion thinking that the NTM makes a *decision* whether to (over)write or not. It does not make this decision (it does not have a separate decision mechanism), it always performs the writing, but sometimes the value written is the same as the old value.

The write operation itself is composed by two components: (i) the erase component, and (ii) add component. The *erase* operation resets the components of a memory location to zero only if both the weighting vector $\mathbf{w}_t$ component for that location and the erase vector $\mathbf{e}_t$ component are both 1. In symbols: $\hat{M}_t = M_{t-1} \cdot (\mathbf{I} - \mathbf{w}_t \cdot \mathbf{e}_t)$, where $\mathbf{I}$ is a row vector of 1s, and all products are Hadamard or pointwise, so these multiplications are commutative. To take care of the dimensions, transpose and broadcast as needed. The *add* operation performs exactly the same taking in $\hat{M}_t$ instead of M_{t-1}, but by using the equation: $M_t = \hat{M}_t + \mathbf{w}_t \cdot \mathbf{a}_t$). Remember, the way these things work is the same, they are all operations on trainable components–there is no intrinsic difference, only operations and trainable differences. We now have to connect the two parts, and this is done by addressing. Addressing is the part which describes how the weighting vectors $\mathbf{w}_t$ are produced. It is a relatively complex procedure involving a number of components, and we refer the reader to the original paper [9] for details. What is important to note is that neural Turing-machines have location-based addressing and content-based addressing.

A second memory-based model, much simpler and equally powerful is the *memory networks* (MemNN) introduced in [10]. The idea is to extend LSTM to make the long term dependency memory better. Memory networks have several components, and aside from the memory, all of them are neural networks, which makes memory

networks even more aligned with the spirit of connectionism than neural Turing-machines, while retaining all the power. The components of the memory network are:

- Memory (M): An array of vectors
- Input feature map (**I**): converts the input into a distributed representation
- Updater (**G**): decides how to update the memory given the distributed representation passed in by **I**
- Output feature map (**O**): receives the input distributed representation and finds supportive vectors from memory, and produces an output vector
- Responder (**R**): Additionally formats the output vectors given by **O**

Their connections are illustrated in Fig. 10.4. All of these components except memory are functions described by neural networks and hence trainable. In a simple version, **I** would be word2vec, **G** would simply store the representation in the next available memory slot, **R** would modify the output by replacing indexes with words and adding some filler words. **O** is the one that does the hard work. It would have to find a number of supporting memories (a single memory scan and update is called a *hop*[2]), and then find a way of 'bundling' them with what **I** has forwarded. This 'bundling' is simple matrix multiplication, of the input and the memory, but with also some additional learned weights. This is how it always should be in connectionists models: just adding, multiplying and weights. And the weights are where the magic happens. A fully trainable complex memory network is presented in [11].

One problem that both neural Turing-machines and memory networks have in common is that they have to use segmented vector-based memory. It would be interesting to see how to make a memory-based model with a continuous memory, perhaps with encoding vectors in floats. But a word of warning, even plain-vanilla memory networks have a lot more trainable parameters than LSTMs, and training could take a lot of time, so one of the major challenges in memory models mentioned in [11] is how to reuse parameters in various components, which would speed up learning. Memory networks memory addressing is only content-based.

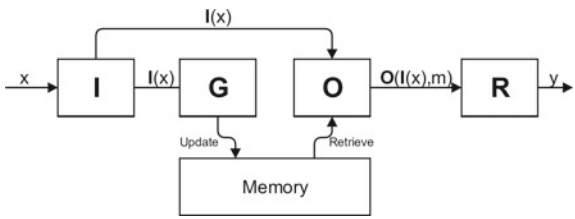


Fig. 10.4 Memory networks

[2]By default, memory networks make one hop, but it has been shown that multiple hops are beneficial, especially in natural language processing.

10.3 The Kernel of General Connectionist Intelligence: The bAbI Dataset

Despite their colourful past, neural networks today are a recognized subfield of AI, and deep learning is making a run for the whole AI. A natural question arises, how can we evaluate neural networks as an AI system, and it seems that the old idea of the Turing test is coming back. Fortunately, there is a dataset of toy tasks called bAbI [12], which was made with the idea of it becoming a kernel for general AI: Any agent hoping to be recognized as general AI should be able to pass all the toy tasks in the bAbI dataset. The bAbI dataset is one of the most important general AI tasks to be confronted with a purely connectionistic approach.

The tasks in the dataset are expressed in natural language, and there are twenty categories of them. The first category addresses single supporting fact, and it has samples that try to capture a simple repetition of what was already stated like the example produced 'Mary went to the bathroom. John moved to the hallway. Mary travelled to the office. Where is Mary?. The next two tasks introduce more supporting facts, i.e. more actions by the same person. The next task focuses on learning and resolving relations, like being given 'the kitchen is north of the bathroom. What is north of the bathroom?. A similar but considerably more complex task is Task 19 (Path finding): 'the kitchen is north of the bathroom. How to get from the kitchen to the bathroom?'. It is the 'flipping' that adds to the complexity. Also, here the task is to produce directions (with multiple steps), where in the relation resolution the network just had to produce the resolvent.

The next task addresses binary answer questions in natural language. Another interesting task is called 'counting', and the information given contains a single agent picking up and dropping stuff. The network has to count how many items he has in his hands at the end of the sequence. The next three tasks are based on negation, conjunction and using three-valued answering ('yes', 'no', 'maybe'). The tasks which address coreference resolution follow. Then come the tasks for time reasoning, positional reasoning and size reasoning (resembling Winograd sentences[3]), and tasks dealing with basic syllogistic deduction and induction. The last task is to resolve the agent's motivation.

The authors of the dataset tested a number of methods against the data, but the results for plain (non-tweaked) memory networks[10] are the most interesting, since they represent what a pure connectionist approach can achieve. We reproduce the list of accuracies for plain memory networks [12], and refer the reader to the original paper for other results.

[3]Winograd sentences are sentences of a particular form, whare the computer should resolve the coreference of a pronoun. They were proposed as an alternative to the Turing test, since the turing test has some deep flaws (deceptive behaviour is encouraged), and it is hard to quantify its results and evaluate it on a large scale. Winograd sentences are sentances of the form 'I tried to put the book in the drwer but it was too [big/small]', and they are named after Terry Winograd who first considered them in the 1970s [13].

1. Single supporting fact: 100%
2. Two supporting facts: 100%
3. Three supporting facts: 20%
4. Two argument relations: 71%
5. Three argument relations: 83%
6. Yes-no questions: 47%
7. Counting: 68%
8. Lists: 77%
9. Simple negation: 65%
10. Indefinite knowledge: 59%
11. Basic coreference: 100%
12. Conjunction: 100%
13. Compound coreference: 100%
14. Time reasoning: 99%
15. Basic deduction: 74%
16. Basic induction: 27%
17. Positional reasoning: 54%
18. Size reasoning: 57%
19. Path Finding: 0%
20. Agent's motivations: 100%

These results point at a couple of things. First, it is amazing how well memory networks address coreference resolution. It is also remarkable how well the memory network performs on pure deduction. But the most interesting part is how the problems arise from inference-heavy tasks where deduction has to be applied to obtain the result (as opposed to basic deduction, where the emphasis is on form). The most representative of these tasks are path finding and size reasoning. We find it interesting since memory networks have a memory component, but not a component for reasoning, and it would seem that memory is more helpful in form-based reasoning such as deduction. It is also interesting that the *tweaked* memory network jumped to 100% on induction but dropped to 73% on deduction. The question on how to get a neural network to reason seems to be of paramount importance in getting past these benchmarks made by memory networks.

References

1. J.J. Hopfield, Neural networks and physical systems with emergent collective computational abilities. Proc. Nat. Acad. Sci. U.S.A **79**(8), 2554–2558 (1982)
2. D.H. Ackley, G.E. Hinton, T. Sejnowski, A learning algorithm for boltzmann machines. Cogn. Sci. **9**(1), 147–169 (1985)
3. P. Smolensky, Information processing in dynamical systems: foundations of harmony theory, in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, ed. by D.E. Rumelhart, J.L. McClelland, the PDP Research Group, (MIT Press, Cambridge)

4. G.E. Hinton, S. Osindero, Y.-W. Teh, A fast learning algorithm for deep belief nets. Neural Comput. **18**(7), 1527–1554 (2006)
5. Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, Greedy layer-wise training of deep networks, in *Proceedings of the 19th International Conference on Neural Information Processing Systems* (MIT Press, Cambridge, 2006), pp. 153–160
6. Y. Bengio, Learning deep architectures for AI. Found. Trends Mach. Learn. **2**(1), 1–127 (2009)
7. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, Cambridge, 2016)
8. W. Bechtel, A. Abrahamsen, *Connectionism and the Mind: Parallel Processing, Dynamics and Evolution in Networks* (Blackwell, Oxford, 2002)
9. A. Graves, G. Wayne, I. Danihelka, Neural turing machines (2014), arXiv:1410.5401
10. J. Weston, S. Chopra, A. Bordes, Memory networks, in *ICLR* (2015), arXiv:1410.3916
11. S. Sukhbaatar, A. Szlam, J. Weston, End-to-end memory networks (2015), arXiv:1503.08895
12. J. Weston, A. Bordes, S. Chopra, A.M. Rush, B. van Merriënboer, A. Joulin, T. Mikolov, Towards ai-complete question answering: A set of prerequisite toy tasks, in *ICLR* (2016), arXiv:1502.05698
13. T. Winograd, *Understanding Natural Language* (Academic Press, New York, 1972)

Conclusion

11

11.1 An Incomplete Overview of Open Research Questions

We conclude this book with a list of open research questions. A similar list, from which we have borrowed some of the problems we present here, can be found in [1]. We were hoping to compile a diverse list to show how rich and diverse research in deep learning can be. The problems we find most intriguing are:

1. Can we find something else than gradient descent as a basis for backpropagation? Can we find something as an alternative to backpropagation as a whole for weight updates?
2. Can we find new and better activation functions?
3. Can reasoning be learned? If so, how? If not, how can we approximate symbolic processes in connectionist architectures? How can we incorporate planning, spatial reasoning and knowledge in artificial neural networks? There is more here than meets the eye, since symbolic computation can be approximated with solutions to purely numerical expressions (which can then be optimized). A good nontrivial example is to represent $A \to B, A \vdash B$ with $\frac{B}{A} \cdot A = B$. Since it seems that a numerical representation of logical connectives can be found quite easily, can a neural network find and implement it by itself?
4. There is a basic belief that deep learning approaches consisting of many layers of nonlinear operations correspond to the idea of re-using many subformulas in symbolic systems. Can this analogy be formalized?
5. Why are convolutional networks easy to train? This is of course connected with the number of parameters, but they are still easier to train than other networks with the same number of parameters.
6. Can we make a good strategy for self-taught learning, where training samples are found among unlabelled samples, or even actively sought by an autonomous agent?

7. The approximation of the gradient is good enough for neural networks, but it is currently computationally less efficient than symbolic derivation. For humans, it is much easier to guess a number that is close to a value (e.g. a minimum) than to compute the exact number. Can we find better algorithms for computing approximate gradients?
8. An agent will be faced with an unknown future task. Can we develop a strategy so that it is expecting it and can start learning right away (without forgetting the previous tasks)?
9. Can we prove theoretical results for deep learning which use more than just formalized simple networks with linear activations (threshold gates)?
10. Is there a depth of deep neural networks which is sufficient to reproduce all human behaviour? If so, what would we get by producing a list of human actions ordered by the number of hidden layers a deep neural network needs to reproduce the given action? How would it relate to the Moravec paradox?
11. Do we have a better alternative than simply randomly initializing weights? Since in neural networks everything is in the weights, this is a fundamental problem.
12. Are local minima a fact of life or only an inherent limitation of the presently used architectures? It is known that by adding hand-crafted features helps, and that deep neural networks are capable of extracting features themselves, but why do they get stuck? Curriculum learning helps a lot in some cases, and we can ask whether the curriculum is necessary for some tasks?
13. Are models that are hard to interpret probabilistically (such as stacked autoencoders, transfer learning, multi-task learning) interpretable in other formalisms? Perhaps fuzzy logic?
14. Can deep networks be adapted to learn from trees and graphs, not just vectors?
15. The human cortex is not always feed-forward, it is inherently recurrent, and there is recurrence in most cognitive tasks. Are there cognitive tasks which are learnable only by feed-forward or only by recurrent networks?

11.2 The Spirit of Connectionism and Philosophical Ties

Connectionism today is more alive and vibrant than ever. For the first time in the history of AI, connectionism, under its present name of 'deep learning', is trying to take over GOFAI's central position, and reasoning is the only major cognitive ability that remains largely unconquered. Whether this is a final wall which can never be breached, or just a matter of months, is hard to tell. Artificial neural networks as a research area almost died out a couple of times during similar quests. They were always the underdog, and perhaps this is the most fascinating part. They finally became an important part of AI and Cognitive Science, and today (in part thanks to marketing) they have an almost magical appeal.

A sculptor has to have two things to make a masterpiece: a clear and precise idea what to make, and the skill and tools to make it. Philosophy and mathematics are the two oldest branches of science, old as civilization itself, and most of science

can be seen as a gradual transition from philosophy to mathematics. This can chart one's way in any scientific discipline, and this is especially true of connectionism: whenever you feel without ideas, reach for philosophy, and when you feel you do not have the tools, reach for mathematics. A little research in both can build an astounding career in any branch of science, and neural networks are no exception here.

This book ends here, and if you feel it has been a fantastic journey, then I am happy. This is only the beginning of your path to deep learning. I strongly encourage you to seek out knowledge[1] and never settle for the status quo. Always dismiss when someone says 'why are you doing this, this does not work' or 'you are not qualified to do this' or 'this is not relevant to your field' and continue to research and do your very best. A proverb I like very much[2] goes: *Every day, write something new. If you do not have anything new, write something old. If you do not have anything old, read something*. At one point, someone with a new brilliant mind will make a breakthrough. It will be hard, and there will be a lot of resistance, and the resistance will take weird forms. But try to find solace in this: neural networks are a symbol of struggle, the struggle of pulling yourself up from rock-bottom, falling again and again, and finally reaching the stars against all odds. The life of the father of neural networks was an omen of all the future struggles. So, remember the story of Walter Pitts, the philosophical logician, the teenager who hid in the library to read the *Principa*, the student who tried to learn from the best, the person who walked out of life straight into the annals of history, the inconsolable man who tried to redeem the world with logic. Let his story be an inspiration.

Reference

1. Y. Bengio, Learning deep architectures for AI. Found. Trends Mach. Learn. **2**(1), 1–127 (2009)

[1] Books, journal articles, Arxiv, Coursera, Udacity, Udemy, etc—there is a vast universe of resources out there.

[2] I do not know whose proverb it is, but I do know it was someone's, and I would be very grateful if a reader who knows the author contacts me.

Index