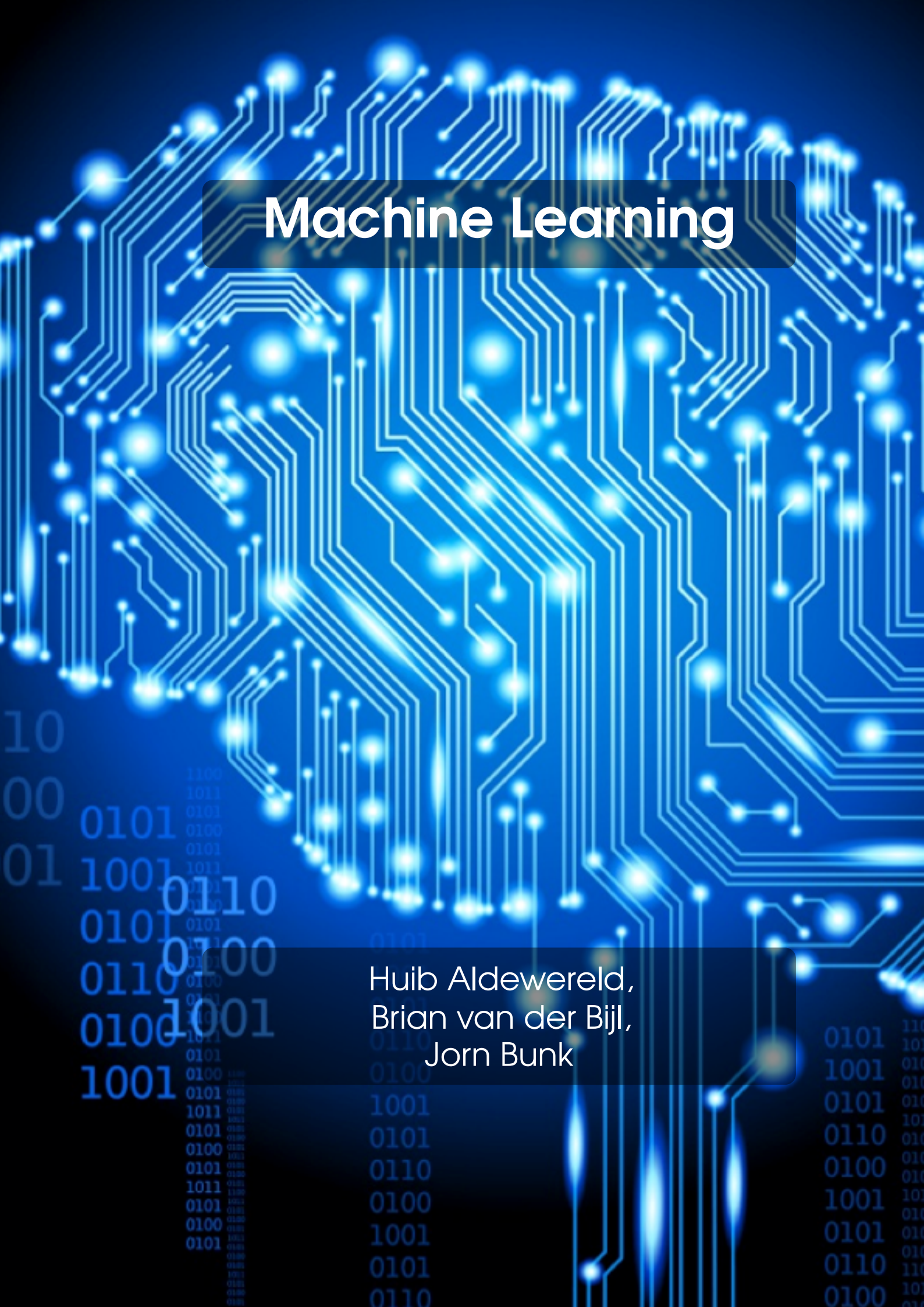
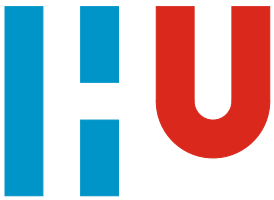

Machine Learning
Huib Aldewereld,
Brian van der Bijl,
Jorn Bunk

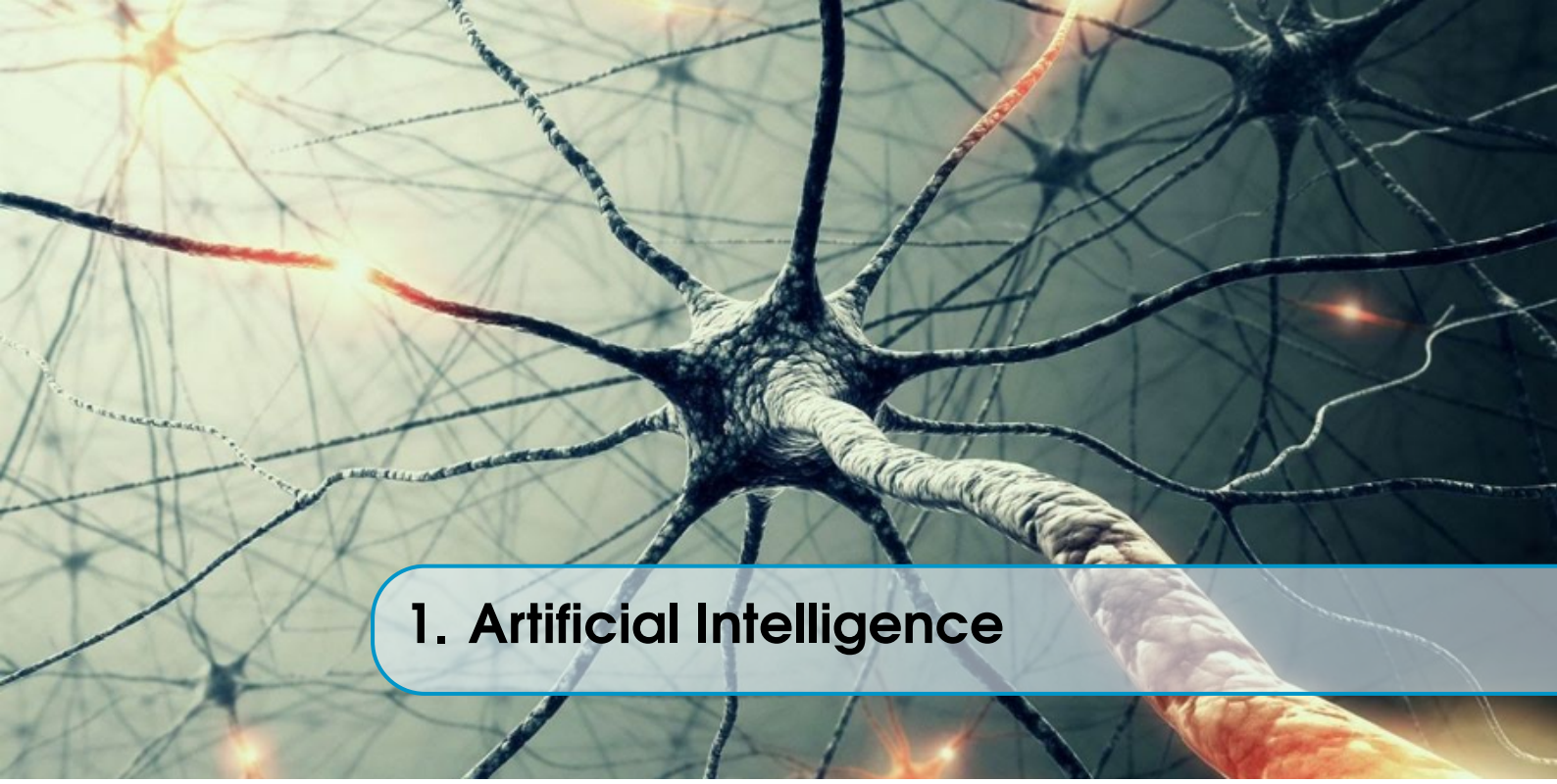


This reader was verified by:

Huib Aldewereld	hogeschooldocent
Brian van der Bijl	trainee
Jorn Bunk	docent
Leo van Moergestel	hogeschoolhoofddocent

First version: October 2017
Current version: February 7, 2022

Contents



1. Artificial Intelligence

In this chapter we explore the history of Artificial Intelligence (AI), to understand what AI is. Man's urge to think about and build artificial life (let alone artificial intelligence) is almost as old as man itself. Proof of this can be found in, for instance, ancient literature; for example, think of Hephaestos' golden robots or Pygmalion's Galatea.

Aside 1.1 — Pygmalion's Galatea.
Pygmalion is a sculptor who is totally disgusted by a group of prostitutes and swears of all women. Instead, he decides to sculpt his ideal woman out of ivory. As Pygmalion makes the statue so beautiful, he actually falls in love with it. At the festival of Aphrodite (the goddess of love), he prays that the goddess will give him a wife just like his statue. She decides to do him one better and actually turns his statue to life. The statue becomes a real woman, and she and Pygmalion get married and have two children.

The story of Pygmalion has inspired many artist through the centuries. At some

point in time, later authors have given the statue the name of Galatea or Galathea.

Variants of the theme are apparent also in the story of *Pinocchio*, the final scene of Shakespeare's *The Winter's Tale* and Bernard Shaw's play *Pygmalion* (later adapted as musical and film, *My Fair Lady*).

Based largely on (**ovid**) and (**pygmalion**). Picture by (**pygmalion**).

But also, man has been interested in understanding what intelligence means. What it means to be intelligent, why humans are intelligent (and why animals are not), and what it means to think and understand. Philosophical discussions about thinking (cognition) and reasoning date back as far as Aristotle's investigations in rational reasoning and syllogisms, which is the basis of the logics frameworks we still use today. This reasoning about the formalisation of computability of rational thought (through, e.g., Liebniz, Frege, Russel, and Gödel, see Aside 1.2) has captivated scientists for many ages.

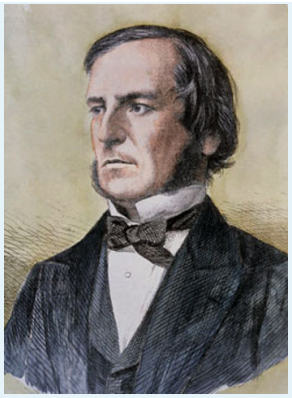
Aside 1.2 — *Calculemus!* – A history of formal reasoning.
The search for a formal, mathematical, system for reasoning started with the Syllogisms of Aristotle. The famous example: *"If all men are mortal, and Socrates is a man, one can derive that Socrates is mortal"*.

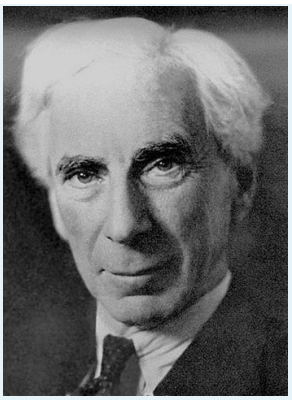
This idea of formal reasoning was further enriched over the years by philosophers like Descartes and Hobbes, until the German philosopher Leibniz dreamt of a formal apparatus that would allow one to precisely determine the truth of things by means of calculation – *calculus ratiocinator*.

> "The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so that we can find our error at a glance, and when there are disputes among persons, we can simply say: '*Let us calculate [**calculemus**], without further ado, to see who is right*''.

Leibniz's idea was further formalised in the algebraic laws of George Boole, who therewith created a first formal language of reasoning, which was further enhanced over the ninth and twentieth century by Gottlob Frege (Begriffschrift) and Bertrand Russel (Principia Mathematica). Finally, through twentied-century philosophers like David Hilbert, Kurt Gödel, Alan Turing and Alonso Church we reached a formal language of reasoning that lay the foundations of mathematical reasoning as was (and largely still is) commonly exploited in research on Artificial Intelligence.


Gottfried Leibniz


George Boole


Bertrand Russel

Based largely on (**logicomix**). Pictures by Wikipedia.

It is therefore not unexpected that when man was able to make machinery that could compute (more efficiently than man itself), people started to wonder whether machines could actually think. That is, can machines be intelligent? This marks the true beginning of AI as a field of research.

In the following, we give a brief overview of the history of AI, from the inception of the field by Alan Turing, and the definition of the field of research at the Dartmouth Conference in 1956, up to what we see as AI nowadays.

1.1 A brief history of Artificial Intelligence

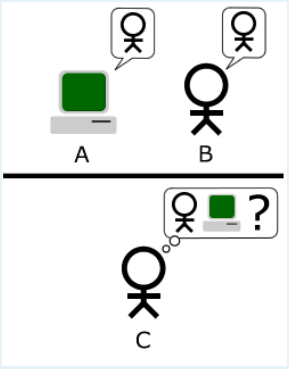
Birth (1952 – 1956)

In the 1930s and 1940s, a handful of scientists from, e.g., mathematics, psychology, engineering, economics, and political science, began to discuss the possibility of creating artificial brains. Discoveries in the, for instance, the field of neuroscience showed that the brain consisted of an electrical network of neurons that fire in all-or-nothing pulses. This started the interest in connectionism. Researching digital signals and creating (analogue) circuitry to simulate brain activity further sparked the interest of creating electrical brains.

pitts:mcculloch:1943 analysed networks of idealised artificial neurons and showed that they might perform simple logical functions. They were the first to describe what researchers would call a *neural network*. One of the students of Pitts and McCulloch was Marvin Minsky, who would later built the first neural network machine, and would become one of the founding fathers of Artificial Intelligence.

In 1950 Alan Turing published a landmark paper (**turing:1950**) in which he speculated about the possibility of thinking machines. Turing's work on the formalisation of computation, which has lead to the creation of the first (digital) computer and subsequently the field of computer science, also introduced the possibilities to think further about creating artificial intelligence. Turing noted that it was difficult to define what "thinking" means and devised a test to verify whether a machine could actually think; the Turing Test (see aside 1.3). A simplified version of this test allowed Turing to argue convincingly that a "thinking machine" was at least *plausible*.

Aside 1.3 — Turing Test.
The Turing Test is a test of a machine's ability to exhibit external intelligent behaviour equivalent to, or indistinguishable from, that of a human. Turing proposed that a human evaluator (C in the picture) would judge natural language conversations between a human and a machine that is designed to generate human-like responses. The evaluator would be aware that one of the two partners in conversation is a machine, and all participants would be separated from each other (they cannot see one another). Interaction would be limited to a text-only channel.



If the evaluator cannot reliably tell the machine from the human (Turing originally suggested that the machine would convince a human 30% of the time after five minutes of conversation), the machine is said to have passed the test.

Picture by (**turingtest**).

With access to digital computers, scientists instinctively recognised that a machine that could manipulate numbers could also manipulate symbols and that the manipulation of symbols could well be the essence of human thought. This was a new approach to creating thinking machines.

In 1956, John McCarthy and Marvin Minsky organised a workshop at the Dartmouth college to describe "every aspect of learning or any other feature of intelligence" with the purpose of creating machine simulations. The 1956 Dartmouth Conference was the moment that AI gained its name, its mission, and its major players, and it is widely accepted as the birth of AI.

The golden years (1956 – 1974)

The Dartmouth Conference spurred tremendous research in the field: computers were winning at checkers, solving word problems in algebra, proving logical statements in English. Programs were developed that were to most people, simply astonishing.

Many basic AI programs used the same basic algorithm. To achieve some goal (like winning a game, or proving a theorem), they proceed step by step towards it. Many advances were made in search algorithms too apply this 'reasoning as search' paradigm. For instance, **newell:simon:1959** captured a general version of this algorithm in a program called "General Problem Solver". Other programs searched through goals and sub goals to plan actions, like the STRIPS system developed at Stanford University.

An important goal of AI research is to allow computers to communicate in natural languages like English. Early successes were made with programs that could solve high school algebra word problems and semantic nets to represent the relationships between concepts. ELIZA (**weizenbaum:1976**) could carry out conversation that were so realistic that users were occasionally fooled in believing that the machine was actually intelligent. ELIZA, however, had no idea what she was talking about, she simply gave canned responses and repeated back (somewhat rephrased) what was said

to her (including spelling and grammatical errors).

By the middle of the 1960s research was heavily funded by the American Department of Defence. AI founders were optimistic about the future: Herbert Simon predicted, "machines will be capable, within twenty years, of doing any work a man can do". Marvin Minsky wrote, "within a generation [...] the problem of creating 'artificial intelligence' will substantially be solved." (**minsky**)

First AI winter (1974 – 1980)

In the seventies, however, numerous problems arose for AI, that led to an almost complete shut-down of all research related to AI. The capabilities of AI programs, in those years, were still limited. Even the most impressive ones could only handle trivial versions of problems that they were supposed to solve. All the programs were, in some sense, "toys".

Several fundamental limits arose that AI research at the time could not overcome. Although some of these limits would be overcome in later decades, others still trouble the field to this day. They are the following:

- **Limited computer power** – Computers at that time were rather simple (at least compared to what we have available nowadays), with only limited computing power and memory at hand, only toy-like representations of problems could be solved. For example, work on natural language processing (by Ross Quillian) was demonstrated on a vocabulary of merely *twenty* words, because that was all that would fit in memory. It was argued that as a certain threshold would be crossed, some hundred times the capabilities that were available then, AI would really take off.
- **Intractability and combinatory explosion** – many of the problems that AI researchers were trying to solve have an exponential complexity. That means that the additional amount of time required to solve a larger version of the problem grows with exponential speed. The combination of inputs, constraints and bounds of the problem lead to an combinatory explosion (the number of possibilities grows very rapidly when increasing the size of the problem), leading to problems that cannot be solved in 'a reasonable amount of time' (that is, the calculations are possible to perform, but given the computational power available (at the time), it would take aeons before an answer would be produced). For example, calculating a solution to Tic-Tac-Toe ('Drie-op-een-rij') would require one to search (naïvely) through $3^9 = 19,683$ positions (there are three states (empty, cross, circle) for every nine cells). Consider then that the game of chess has 64 positions, 32 pieces and many thousands of possible moves, and is thus still considered 'unsolvable'.
- **Common-sense knowledge and reasoning** – many important artificial intelligence applications, like vision or natural language processing, rely on enormous amounts of information about the world; world-knowledge or common-sense knowledge. Humans gather a large amounts of implicit knowledge (knowledge that we are not really aware of) about the how the world functions ('if I drop an egg, it will fall to the ground, and break'). A computer does not have this information, and in the 1970 this presented large problems, since there

were no databases large enough to even store all the information, let alone any knowledge on how to obtain/learn all that information.

- **Moravec's paradox** – computers had proven themselves very capable in proving theorems and solving geometry problems, since they require computation capabilities in which computers excel (crunching numbers); however, a supposedly simple task like recognizing a face or crossing a room without bumping into anything is extremely difficult to them. This contradiction is known as Moravec's paradox.
- **Frame and qualifications problems** – AI researchers (like John McCarthy) who used logic discovered that the formal apparatus that was supposed to operationalise human reasoning had some basic flaws. For one, if one wanted to make deductions (calculations) about anything, everything related to that had to be encoded in the logic (frame problem). Logics cannot make deductions about facts or knowledge that has not been a priori encoded in the model. Moreover, logics could not represent ordinary deductions involving planning or default reasoning (human reasoning is not black-and-white; true or false, but uses 'defaults' if not enough knowledge is available; e.g. 'Birds can fly', 'Tweety is a bird, therefore it can fly... unless Tweety is a penguin'). New logics had to be developed (like non-monotonic logics and modal logics) to try to solve these problems.

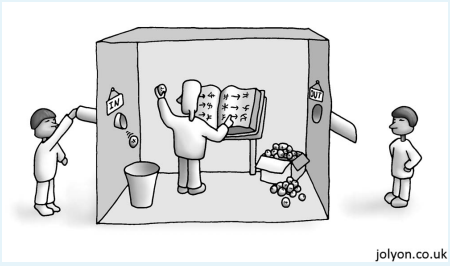
Due to the lack of progress in AI, funding agencies (such as British Government, DARPA and the National Research Council) became frustrated and cut off almost all funding for undirected research in AI. To make matters worse, AI received many critiques even from researchers within the field, that AI might not be as promising as expected and that intelligent machines might not be possible after all. Philosophers like John Searle argued that Turing's proposition of a thinking machine was impossible (see aside 1.4), and that symbolic reasoning (i.e., using logic) was never going to create intelligence. A book by **minsky:papert** showed the severe limitations of what perceptrons (early neural networks devised by Frank Rosenblatt) could do, and showed that the predictions by Rosenblatt where grossly exaggerated. This halted the research in neural networks for nearly 10 years.

Aside 1.4 — Chinese room.
The Chinese Room argument, a thought-experiment by John Searle in (**searle**), holds that a program cannot give a computer a "mind", "understanding", or "consciousness", regardless of how intelligent or human-like the program may make the computer behave. It is a direct attack on Turing's proposition earlier, that computers can be understood as intelligent, if they behave intelligently.

The argument is as follows. Suppose that artificial intelligence has succeeded in constructing a computer that behaves as if it understands Chinese. It takes Chinese characters as input, and by following the instructions of a computer program, produces other Chinese characters, which it presents as output. Let's assume that the machine performs convincingly such that it would pass a Turing Test; it convinces a human Chinese speaker that the program is itself a Chinese speaker. The questions Searle then poses are: does this machine literally understand

Chinese? Or is it merely simulating an understanding of Chinese? The former he calls ***strong AI***, the latter he calls ***weak AI***.

Now, let's replace the computer with Searle himself. He is locked into a room with a book with the English version of the program run on the computer, and enough pencils, paper, erasers, and filing cabinets to execute the program by hand. Searle could receive the Chinese characters through a slot in the door, process them by means of the book (the program's instructions), and produce other Chinese characters as output. If the computer would have passed the Turing Test, so would Searle.



Searle asserts that his role and that of the computer are essentially the same. As he does not speak a word of Chinese, he is unable to understand anything of the conversation. Therefore, he argues, it follows that the computer would not be able to understand either.

Searle argues that, without "understanding", we cannot describe what the machine is doing as "thinking" and, since it does not think, it does not have a "mind" in anything like the normal sense of the word. Therefore, he concludes that *strong AI is false*.

While the image of AI had gotten a severe dent in the 1970s, some major advances where made in the fields of logics (introduction of non-monotonic logics, and the basis for logic programming through ProLog) and representation (using McCarthy's 'frames', which later became the roots of inheritance in object-oriented programming). The real value of these discoveries, however, where largely only noticed later, while the failures of AI were all the more prominent.

Rise of expert systems (1980 – 1987)

In the 1980s a form of AI program called *expert systems* was adopted by corporations around the world and knowledge became the focus of mainstream AI. Expert systems are programs that can answer questions or solve problems about a specific domain of knowledge, using logical rules derived from experts' knowledge. One of the earliest was the MYCIN program (1972), which diagnosed infectious blood diseases. This demonstrated the feasibility of the approach.

Expert systems restricted themselves to a specific domain of knowledge, thus avoiding the common-sense problems, and their simple design made it relatively easy for programs to be built and then modified once they were in place. AI, for a first, had proven itself *useful*, something which it had failed to do so far.

The power of expert systems came from the expert knowledge they contained. This was part of a new direction in AI research that had gained traction in throughout the 70s. The great lesson from the 1970s was that intelligent behaviour depended on dealing with knowledge, sometimes quite detailed knowledge, of a domain where a given tasks lay. Therefore, knowledge based systems and knowledge engineering became the major focus of AI research in the 1980s.

Connectionism saw a revival, when in 1982 John Hopfield was able to prove that a form of neural network (now called a "Hopfield net") could learn and process information in a completely new way. Around the same time, David Rumelhart popularized a new learning method called "backpropagation". The new field was unified and inspired by the appearance of Parallel Distributed Processing, and by 1990, neural networks would have become commercially successful, when used as the engines driving programs like optical character recognition (OCR) and speech recognition.

The bust: second AI winter (1987 – 1993)

The business community's fascination with AI rose and fell in the 80s in the classic pattern of an economic bubble. In the late 80s and early 90s, AI suffered again a series of financial setbacks. The collapse was, however, largely in the *perception* of AI by government agencies and investors, as the field continued to make advances despite the criticism.

The market for expensive specialised AI hardware to run symbolic programs was overtaken by the advances made in desktop computers. The field once again took a few hits from its own researchers, with Rodney Brooks and Hans Moravec as its main opponents. These researchers from the field of robotics argued for a different approach to AI, where the intelligence is looked from an embodied perspective; that is, situated in the environment, and (intuitively) interacting with it, instead of being a mere thinking machine.

Aside 1.5 — *Elephants don't play chess* (Embodied AI).
In a 1990 paper titled *"Elephants Don't Play Chess"* (**brooks1990elephants**) robotics researcher Rodney Brooks took a direct aim at the physical symbol system hypothesis, arguing that symbols are not always the best way to intelligence since "the world is its own best model. It is always exactly up to date. It always has every detail there is to be known. The trick is to sense it appropriately and often enough."

Brooks was wrong.
Picture by Ethiriel Photography.

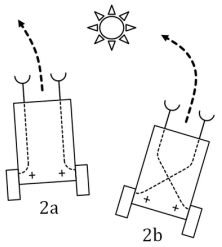
In the 80s and 90s, many cognitive scientists also rejected the symbol processing model of the mind and argued that the body was essential for reasoning. This lead to the fields of behaviour based AI.

In contrast to classic AI, where robots takes a set of steps to solve problems (typically, sensing, reasoning, acting), behaviour based AI rather relies on adapt-

ability. Brooks showed that with a much simpler architecture ant-like behaviour could be achieved that appeared natural in intelligence (though limited), whereas classic AI's reasoning approach resulted in slow and clumsy robots.

A well-known milestone in the field of behaviour based AI is the work by Valentino Braitenberg, who showed that by clever wiring between sensors and motors a complex-appearing behaviour (such as fear and love) could be created.

An example of the Braitenberg machine exhibiting fear (2a) and love (2b) of light.
Picture by Wikipedia.



Nouvelle AI (1993 – 2001)

The field of AI, now more than half a century old, finally achieved some of its oldest goals. It began to be used successfully throughout the technology industry, though somewhat behind the scenes. Some of the success was due to increasing computer power and some was achieved by focussing on specific isolated problems. Inside the field there was little agreement on the reasons for AI's failure to fulfil the dream of human level intelligence, which lead to a fragmentation of AI into competing subfields focussed on particular problems or approaches, sometimes even under new names that disguised the tarnished pedigree of "artificial intelligence". AI was both more cautious and more successful than it had ever been.

On May 11 1997, Deep Blue managed to beat the reigning world chess champion Garry Kasparov, becoming the first successful chess playing computer. The super computer was a specialised version of a framework developed by IBM and was capable of processing a stunning 200,000,000 moves per second. In 2005, a Stanford robot won the DARPA Grand Challenge by driving autonomously for 131 miles along a unrehearsed desert trail. Two years later, a team from CMU won the DARPA Grand Challenge by autonomously navigating 55 miles in an urban environment (adhering to traffic hazards and all traffic laws). In Februari 2011, IBM's Watson won the Jeopardy! quiz show, defeating two Jeopardy champions by a significant margin.

All of these successes were not due to some revolutionary new paradigm, but mostly on the tedious application of engineering skill and on the tremendous computing power of computers these days. The dramatic increase in power, measured by Moore's law, slowly by surely overcame the fundamental problem of having enough "raw computer power".

A new paradigm called "intelligent agents" became widespread during the 90s, based on earlier "divide and conquer" modular approaches proposed by AI researchers. An intelligent agent is a system that perceives its environment and takes actions to maximise its chance of success. By this definition, simple programs that solve specific problems are intelligent agents, as are human beings and organisations of human beings. The paradigm is a generalisation of some earlier definition of AI: it goes

beyond studying human intelligence; it studies all kinds of intelligence. This brought other fields, such as decision theory and probability into AI research.

Algorithms originally developed by AI researchers began to appear as parts of larger systems. AI had solved a lot of very difficult problems and their solutions proved to be useful throughout the technology industry, such as data mining, industrial robotics, logistics, speech recognition, banking software, medical diagnosis, and search engines. The field of AI receives little to no credit for these successes. Many of AI's greatest innovations have been reduced to the status of just another item in the toolbox of computer science. This is now know as the "AI effect": "A lot of cutting edge AI has filtered into general applications, often without being called AI because once something becomes useful enough and common enough it's not labelled AI any more."

Deep learning (2000 – present)

In the first decades of the 21st century, access to large amounts of data (known as "big data"), faster computers and advanced machine learning techniques were successfully applied to many problems throughout the economy. By 2016, the market for AI related products, hardware and software reached more than 8 billion dollars. The applications of big data began to reach into other fields as well. Advances in deep learning drove progress and research in image and video processing, text analysis, and even speech recognition.

Deep learning is a branch of machine learning that models high level abstractions in data by using a deep graph with many processing layers. They are a new form of neural network (and fundamentally function as such), where additional layers (deep-ness) is used to help avoid problems like overfitting that are common to shallow networks. As such, deep neural networks are able to realistically generate more complex models as compared to their shallow counterparts.

Ethics & AI

As strong as our wish to create intelligent computers, people have also always been cautious about the possibilities of disaster(s) that could happen when we finally achieve computers that are intelligent. As early as the 1950s, by John von Neumann, stems a theory of the "emergence of superintelligence", which argues that with the invention of artificial intelligence, a superintelligence would abruptly trigger runaway technological growth, resuling in unfathomable changes to human civilization (typically ending in the enslavement or destruction of the human race). Von Neumann first used the term "*singularity*", which, in the context of technological progress causing accelerating change, means "the accelerating progress of technology and changes in the mode of human life, give the appearance of approaching some essential singularity in the history of the race beyond which human affairs, as we know them, could not continue" (**vonneumann**). According to this hypothesis, an upgradable intelligent machine (such as a computer running software-based artificial general intelligence) would enter a "runaway reaction" of self-improvement cycles, with each new and more intelligent generation appearing more and more rapidly, causing an intelligence explosion and resulting in a powerful superintelligence that would, qualitatively, far surpass all human intelligence.

The law of Moore, describing the exponential growth in computing technology is often used to support the singularity hypothesis. Ray Kurtzweil, a prominent singularity theorist, postulated a law of accelerated returns in which the speed of technological change is generalised using Moore's Law, also including material technology (especially nanotechnology), medical technology and others. Kurtzweil, however, reserves the term "singularity" for a rapid increase in artificial intelligence, and expects that "There will be no distinction, post-Singularity, between human and machine" (**kurtzweil**) by his prediction will occur in 2045.

Singularity is still just a hypothesis (that is, it is unproven until it happens), and there are many critics that debate that it will ever happen. Most common criticism attacks the assumption that computers can ever achieve intelligence, and postulate that, while computers are indeed rather successful in some intelligent tasks it does not make them intelligent in all different tasks, like us humans.

For instance, Steven Pinker stated (**pinker**):

> "There is not the slightest reason to believe in a coming singularity. The fact that you can visualize a future in your imagination is not evidence that it is likely or even possible. Look at domed cities, jet-pack commuting, underwater cities, mile-high buildings, and nuclear-powered automobiles – all staples of futuristic fantasies when I was a child that have never arrived. Sheer processing power is not a pixie dust that magically solves all your problems."

Whether Singularity will ever happen, it remains a fact that machines are becoming increasingly intelligent (or at least, are *appearing* increasingly intelligent), which has an impact on our day-to-day lives. It is becoming more and more a debate of the position of AI in our daily lives; what to think about autonomous cars, what will happen when many (repetitive) jobs will be replaced by robots, how much of our jobs can be automated by means of AI technology, which jobs should never be done (completely) by an AI? Nowadays, you cannot click on a news site nor open a newspaper without finding an article about the role of ethics in Artificial Intelligence. Technology is advancing, and many more AI applications that could only be dreamt of in the 1950-1970s can now be achieved (largely due to the increase in computational power available), but how much of that technology is actually wanted? And should we, if we could, stop the advancement in certain areas (like, for instance, autonomous weapon systems)? These are all valid questions that need to be asked.

Aside 1.6 — Asimov's Laws of Robotics.
Isaac Asimov first considered the issue of ethics of AI in the 1950s in his I, Robot series of science fiction stories. At the insistence of his editor John W. Campbell Jr., he proposed the Three Laws of Robotics to govern artificially intelligent systems. The Three Laws are the following:

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.

3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.

This could be seen as a first attempt to introduce ethics into artificial intelligent systems (even though, such systems were not yet possible in Asimov's time).
Much of Asimov's work was then spent testing the boundaries of his three laws to see where they would break down, or where they would create paradoxical or unanticipated behaviour. His work suggests that no set of fixed laws can sufficiently anticipate all possible circumstances (**irobot**).

1.2 Applied Artificial Intelligence

In this course, and in most companies nowadays, AI is synonym for Applied Artificial Intelligence, which is a subfield of generic AI research focussed on those aspects of AI that can be applied in real world situations. Related to what Searle called "weak AI"[1], the most common field of applied AI is by far the field of Machine Learning.

By some considered to be part of mathematics (statistical reasoning / statistical learning) or mainstream computer science (big data, data mining, clever algorithmics)[2], machine learning has had an enormous interest over the last few years, with prominent achievements like IBM's Watson, the speech recognition and assistant engines Siri, Google Now (Google Assistant) and Amazon Alexa (Amazon Dot), and Google AlphaMind and Deepmind (winning from the world champion Go). Many companies now see the advantages that AI can bring them, and many are eager to adopt and deploy AI techniques in their businesses.

Machine learning is the subfield of AI (or computer science) that gives computers the ability to learn without being explicitly programmed. Machine learning is a set of techniques that allows programmers to tell the computer what to do (given an input, and perhaps a desired output), without explicitly writing an algorithm to achieve that. That is, in traditional computer science, programmers write algorithms to direct the computer into what actions to take given a particular input to achieve a desired output. That is, the input is know, and the program (algorithms) are clear, in order to have the computer calculate the output (see Figure 1.1 below).

The result of a machine learning algorithm is typically a program that can be run by the computer to get the output from the specified input. Machine learning algorithms are typically categorised into three categories:

- **Supervised** – algorithms that are presented both the input and the desired ouput related with that input. The goal of such algorithms is to determine a general rule that maps the input to the output.
- **Unsupervised** – algorithms that are presented with only the input, and have to figure out themselves particular structure within it; the goal can be to detect

[1] Sometimes also called "narrow AI" or "applied AI", as the focus of the AI is much more narrow/applied than that of the general intelligence sought after by "strong AI".

[2] This is largely due to the "AI effect", discussed above.

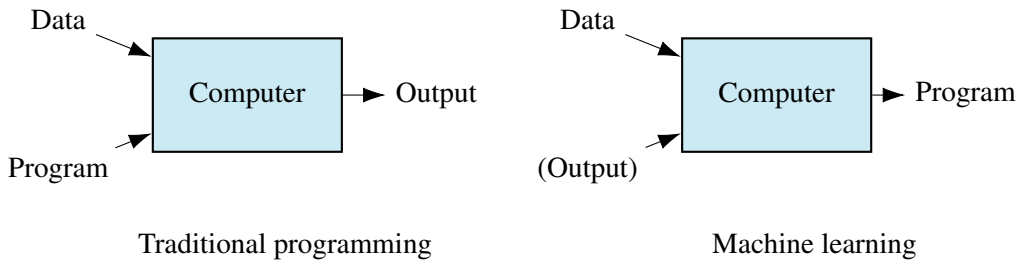


Figure 1.1 Traditional programming vs. machine learning

hidden patterns.

- **Reinforcement** – algorithms that interact with the environment to maximise long-term rewards. These algorithms are typically not rewarded for each step (as in supervised learning) but receive a single payment at the end.

Machine learning can be employed for various problems, the following is an overview of such problems (the list is not meant to be exhaustive but rather to give an idea of the kind of problems that ML is suitable for).

Classification

Items, samples, individuals are to be put in the right class. This involves the problem of trying to identify to which set of categories a new observation belongs. The characteristics of the categories is typically learned from a given set of labelled examples. Typical examples of classification include Spam filtering, Optical Character Recognition (OCR), Search Engines, Computer Vision.

Typical methods used for classification include: *neural networks* (see chapter 2), *support vector machines*, *decision trees*, and *k-nearest neighbours*.

Clustering

The objective of clustering is to find underlying structure in the data that is presented. This is typically an unsupervised task, since the algorithm is not predicting anything specific. Common questions that arise when clustering are: How many clusters are there? What are their sizes? Do elements in a sub-population have any common properties? Are sub-populations cohesive? Can they be further split up?

Typical methods used for clustering include: *k-means*, and *Gaussian mixture models*.

Regression

Regression analysis is the (statistical) process of trying to estimate the relationship among variables. Regression analysis helps one understand how the typical value of the dependent variable (or 'criterion') changes when any one of the independent variables is varied, while the other independent variables are kept fixed. For instance, trying to understand the relationship between the amount of hours spend on playing computer games vs. the average grade obtained in class (independent of, e.g., age, gender, etc.).

Typical methods used for regression include: *k-nearest neighbours* and *support vector machines*.

Dimensionality reduction

The data sets used with Machine Learning often have multiple variables to describe individuals/samples. These attributes of individuals can be thought of different dimensions, like the dimensions of a point in space (i.e., any particular point is described by 3 attributes, an x-, y, and z-position[3]). For data points, one can have many dimensions (think of, e.g., all the different attributes one can have to describe a student; next to physical attributes, like length, weight, gender, age, other attributes could be gathered as well, for instance, scores to individual tests/courses). Having many dimensions can make estimations of closeness cumbersome (lots of mathematics involved) and hard to explain. Often, a number of these dimensions correlate with each other, or are not that interesting given the problem at hand. Dimensionality reduction algorithms attempt to reduce the number of dimensions to a more manageable number (like, e.g., 3 to be able to make a meaningful plot).

An example for dimensionality reduction is *Principal Component Analysis*.

Density estimation

Density estimation is the construction of an estimate, based on observed data, of an unobservable underlying probability density function. The unobservable density function is thought of as the density according to which a large population is distributed; the data are usually thought of as a random sample from that population.

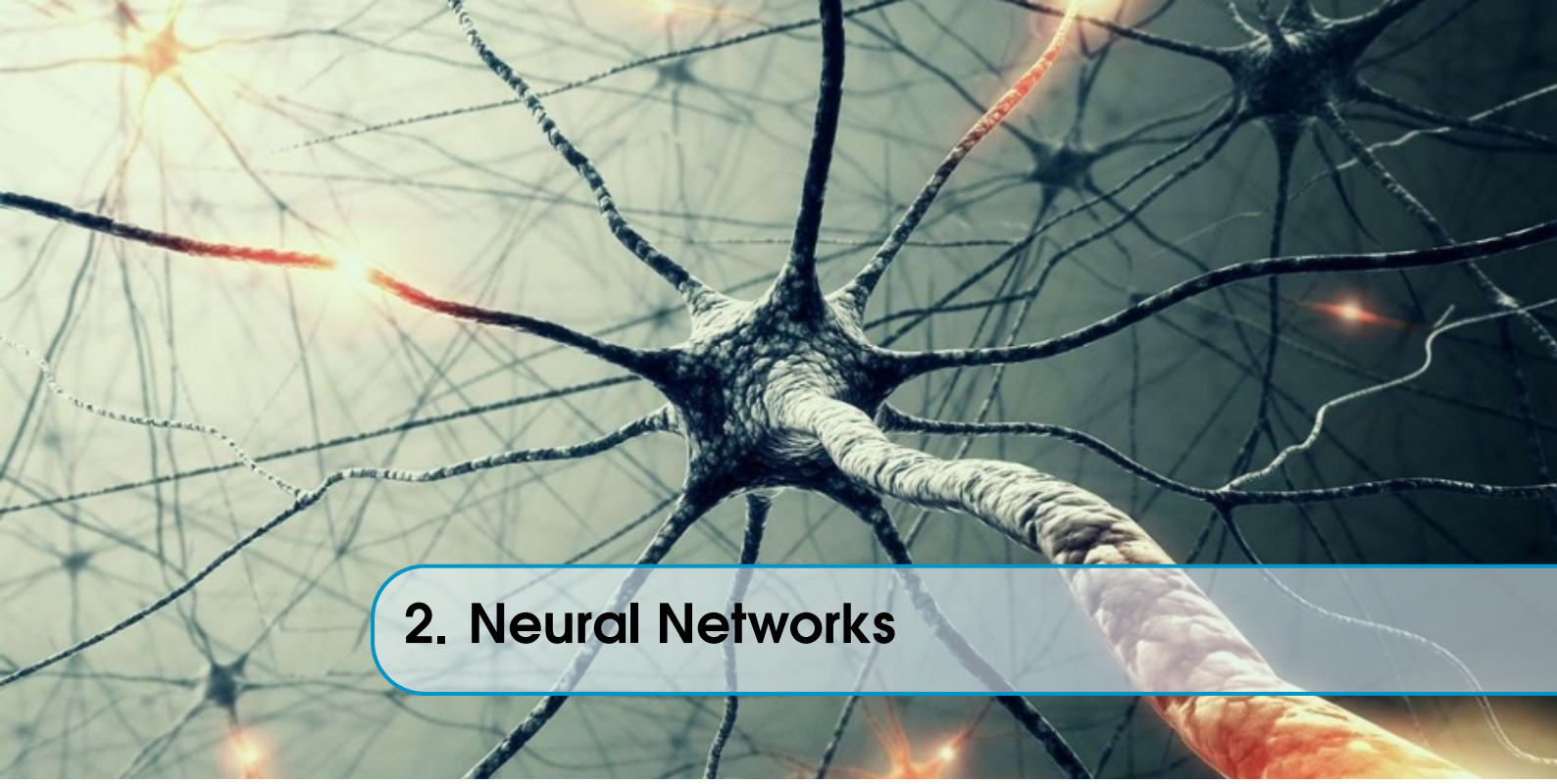
Many machine learning techniques require information about the probabilities of various events involving the data. A Bayesian approach (using probabilities), for instance, presupposes knowledge of the prior probabilities and the class-conditional probability densities of the attributes. This information is rarely available and must usually be estimated from available data.

Example methods include: *kernel density estimation* and *kernel function estimation*.

Prediction

By some seen as a specialisation of classification, machine learning algorithms can also be employed to perform diagnosis or prediction analysis. The algorithm is used to create stable inferences and make predictions given a (subset) of inputs. This can be done deterministically by means of *decision trees*, or probabilistically by means of *Bayesian networks*.

[3] One might argue that every point has a 4-th dimension, namely time.



2. Neural Networks

Our brains are capable of doing many great things. For instance, language processing and object recognition are tasks most people can do effortless. When you look at figure 2.1 it is almost impossible not to see the numbers '0034523', but when we want to write a computer program that reads these numbers we realise how hard this task really is. How do we tell a computer intuitive facts like that a '5' consists out of 2 straight lines and an incomplete circle? Or how do you explain to the computer that although the first two digits are drawn differently they both are a '0'?

Figure 2.1 A captcha of the number 0034523

Not only are we humans great in pattern recognition tasks, we are also able to learn new tasks without programming and apply previous learned skills in unknown fields. Computers should be jealous of us, were it not that they are great in many other tasks[1]. Besides, we humans are the one to blame of their lack of intelligence.

Artificial neural networks are an attempt to put the (human) brain approach to problem solving and learning into computers[2]. Although we know relative little about the brain as a whole, we known the functional working of neurons and how they work

[1] For example printing "hello world" a 1000 times on the screen.

[2] As computer engineers we are mostly interested in making the computer smarter, but this research field is actually two-fold. If we have a computer that performs equally to humans in the same tasks, but also makes the same mistakes as humans, then we have a mathematical model of how the human brain could work, which could help us in revealing the mysteries of the brain.

together in a network. This allowed the creation of a mathematical model of neural networks called artificial neural networks. Artificial neural networks are able to learn from examples. Instead of programming what a chair is[3], the neural network gets a lot of examples of chairs and learns itself how to recognise a chair.

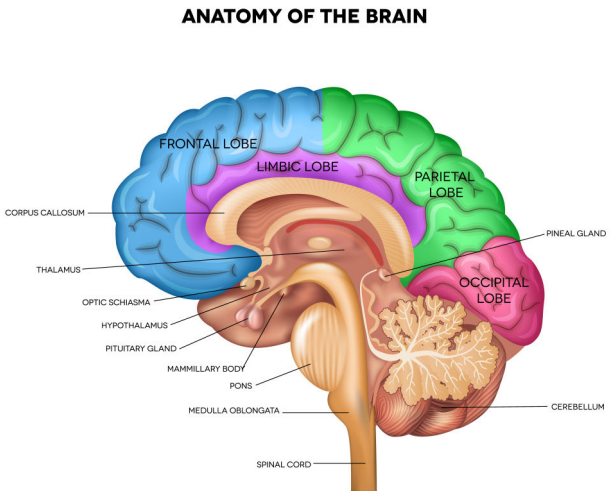
The recent successes and the media coverage of Google's AlphaGO (**alphamind**) and projects like DeepArt (**DeepArt**) make it seem like that every problem can now be tackled using neural networks. It certainly has proven its use, but before we start, we should realise that every technique has its limits. AlphaGo may defeat the world champion in Go, but is not able to bake a toasti (**NOS_Tosti**). Also in contrary to the impression given by the popular media, of course, neural networks are far from being the only AI systems capable of learning.

In this chapter we start by looking at artificial neurons and we work our way up to a complete learning artificial neural network.

2.1 (Artificial) neurons

As stated in the introduction, artificial neural networks are based on the brain, or more specific on the nerve-cells (within a brain), called *neurons*. In a human brain there are approximately 100 billion (100,000,000,000) neurons. We use these cells, among other things, to read, to control our muscles and to learn. In this section we look briefly at the biological neuron, after which we go into the mathematical model of a neuron and how it can be used in decision making.

Figure 2.2 The brain



2.1.1 Biological neurons

A neuron is a complex biological machine with as main task to process and send signals. Luckily, for our purpose we only need to know the functions of a few parts: the dendrites, the axon and the axon terminal (see figure 2.3). Every neuron consists of a cell body (soma). Branching out from the cell body are a number of fibres called

[3]Which is pretty hard. Try it: describe a chair to yourself and than think about if all chairs fit the description. How many legs does a chair have? Can a chair have wheels?

dendrites and a single long fibre called the axon. Dendrites branch into a bushy network around the cell, whereas the axon stretches out for a long distance, usually about a centimetre (100 times the diameter of the cell body) and as far as a meter in extreme cases. Eventually, the axon also branches into strands and substrands that connect to the dendrites and cell bodies of other neurons. The end point of a (sub)strand of an axon is called an axon terminal. The connecting junction of an axon terminal with a dendrite is called a synapse. The dendrites of a neuron are the place it receives its input and through the ((sub)strands) of the axon it sends its output.

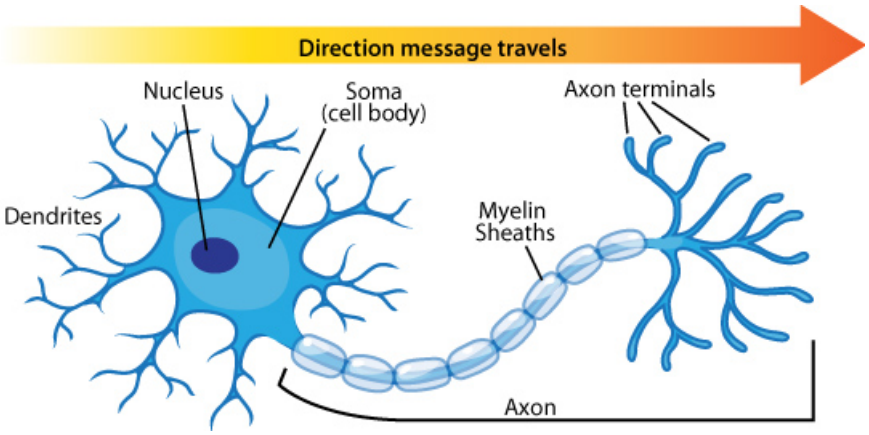


Figure 2.3 A neuron. In reality, an axon is about 100 times the diameter of the cell body.

A neuron can be in one of two states: it is firing (sending a signal along its axon) or it is in rest (not firing). Whether or not the neuron should fire depends on the signals it receives at its dendrites. These signals raise or lower the potential of the cell body. When the potential reaches a threshold, a signal, also called the action potential, is send along the axon. The signal spreads out to the axon terminals and the signal is transmitted to the next neuron(s).

A neuron can "learn", i.e. change when it fires, by modifying how strong the signal is transmitted at the synapses. This will effect the potential of the cell and thereby how soon it reaches the threshold. This can be experienced using after image illusions[4]. If you look at one colour for a long time your cones (special nerve cells in your eyes) will get less sensitive for that colour. In other words, their action potential will raise less when they “see” the colour. When you change your view to a white surface it seems like the surface has a different colour as the cones are still insensitive to the colour you were watching previously.

2.1.2 Artificial neuron

As computer engineers we could look at the neuron as a special type of logic gate: the dendrites represent the different inputs, the axon represents the output and the combination of the synapses and the threshold specify which type of gate it is. Figure 2.4 shows a representation of an artificial neuron. It has three inputs. In general it can have more or less inputs $x_1, \ldots, x_n$.

[4]For an example of the after image effect see: `https://youtu.be/GbHMLV4CZfI`

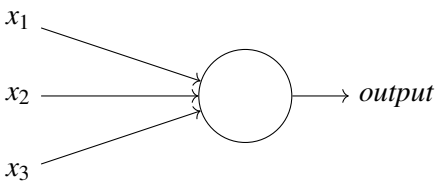


Figure 2.4 An artificial neuron with input x_1, x_2, x_3 and one output.

To compute the output we assign to each input a weight, $w_1, \ldots, w_n$. The weights represent the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_{i=1}^{n} w_i x_i$ is less than or greater than threshold value t. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

$$output = g = \begin{cases} 0 & \text{if } \sum_{i=1}^{n} w_i x_i < t \\ 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq t \end{cases} \tag{2.1}$$

This function is also called an *activation function*, in further reading denoted with a g.

And that is all we need to create an artificial neuron. As we see in section 2.3, the artificial neuron described here is a specific type of artificial neurons called a perceptron. When we understand the perceptron and networks of perceptrons it is easier to understand how other types of neurons work and why they exist.

We can now make an artificial neuron that can act as a logic gate (see figure 2.5, 2.6 and 2.7). By assigning the weights and the threshold we decide on the behaviour.

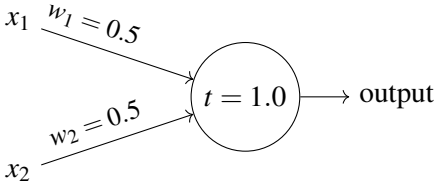


Figure 2.5 Appropriate weights and threshold to act as an AND-gate.

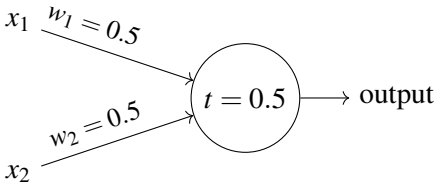


Figure 2.6 Appropriate weights and threshold to act as an OR-gate.

But we are not limited to logic gates. For example, we could make an artificial neuron that decides if we should go to a party. This decision can, of course, be made by weighing the following three factors:

- Are your friends going?
- Are there cats at the party?
- Is the party free?

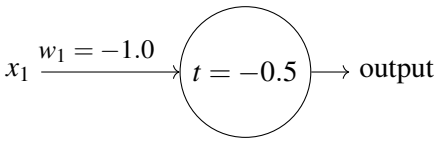


Figure 2.7 Appropriate weights and threshold to act as an INVERT-gate.

We can represent these three factors by corresponding binary variables x_1, x_2, and x_3. For instance, we'd have $x1 = 1$ if your friends are going, and $x1 = 0$ if your friends are not going. Similarly, $x_2 = 1$ if there are cats at the party, and $x_2 = 0$ if not. And similarly again for x_3 and whether there are entree costs to the party.

Suppose, that your friends make every party the best party possible, so much so that it doesn't matter if you have to pay or that there are no cats; you just have to be there. But if your friends are not there, the party needs to be free and have cats for you to be going. For this decision process we could use an artificial neuron. One way to do this is to choose a weight $w_1 = 0.6$ for the friends, a $w_2 = 0.3$ and $w_3 = 0.2$ for the other conditions and set the threshold t to 0.4. A larger value for the weight means that the factor matters more in the decision process. This unit can be seen in figure 2.8. By varying the weights and the threshold, we can get different models of decision-making. For instance, someone who thinks that cats are better than people (who doesn't?) might have a higher weight for w_2 and a lower weight for w_1.

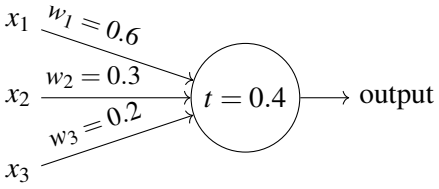


Figure 2.8 A possible perceptron to decide if we should go to a party.

2.1.3 The limits of perceptrons

We saw in the figures 2.5, 2.6 and 2.7 that perceptrons can represent the simple Boolean functions AND, OR and NOT, but what are the limits to the Boolean functions that can be represented with an artificial neuron? When we look at the activation function in equation 2.1 we see that it is a linear function. If we would represent our decision making in a two-dimensional plot based on the values of two inputs, then we can only draw one straight line to separate the two decisions. This is shown in figure 2.9 for an AND-gate. The white and black dots are linear separable. The line between the white and black dots represents the threshold. An artificial neuron can represent an AND-gate. A XOR-gate cannot be represented by an artificial neuron, as there is no single straight line that separates the white and the black dots (see figure 2.10).

2.2 A simple artificial neural network

A perceptron has it limits in decision making and is in no way close to a complete model for (human) decision making, but, just like logic-gates (see figure 2.11), we

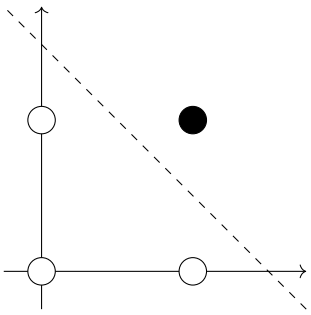


Figure 2.9 A two-dimensional plot of the decision making for an AND-gate.

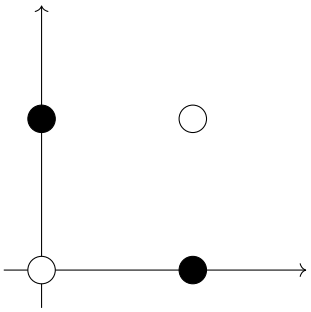


Figure 2.10 Plot of decision making for a XOR-gate. The white and black dots are not linear separable.

can combine them to make more subtle decisions. To get ourselves familiar with the terms used in neural networks we look in this section at the small network depicted in figure 2.12.

2.2.1 Units and layers

Artificial neurons in networks are often called *units*. In a network we can distinguish different type of units. The first layer of units with the input (x_1 and x_2 in figure 2.12) are called input units. These units do not get input from other units. Instead their activation depends on external variables (e.g. the grayscale of a pixel). The last layer of units in a network is called the output layer. The activation of the output units in this layer correspondents to the possible answers that we want to get out of the network (e.g., it is a cat). The layers between the input layer and the output layer are called hidden layers. A network without hidden layers is called a single-layer neural network (we do not count the input layer). Neural networks with one or more hidden layers are called multi-layer neural networks.

Layers are connected with each other by links, the connections between two units. As we already discussed, every input has a weight corresponding to it. We could

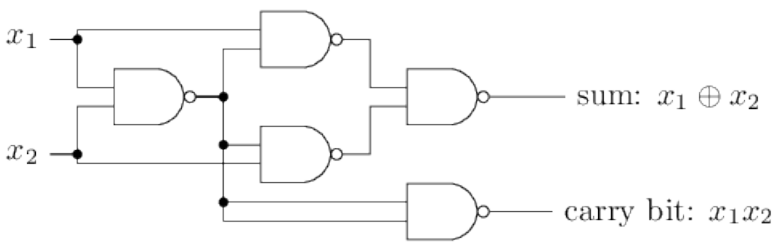


Figure 2.11 An adder built out of NAND-gates.

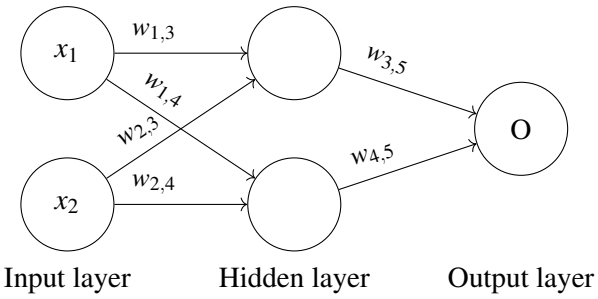


Figure 2.12 A small neural network. Weight $w_{i,j}$ is the weight of the link between unit i and unit j.

rephrase that to say that every link has a corresponding weight. Where weight $w_{i,j}$ is the weight of the link between unit i and unit j.

Note that, although a unit can have multiple outputs arrows, in reality a unit has only one output. Multiple arrows only depict that the output is transferred to multiple units.

2.2.2 Feed-forward network

The network seen in figure 2.12 is a *feed-forward network*. In a feed-forward network the links are unidirectional, and there are no cycles. Technically speaking, a feed-forward network is a directed acyclic graph (DAG). In more layman's terms, in a feed-forward network the activation of the units can only go to the next layer and not back to a previous layer. The significance of the lack of cycles is that computation can proceed uniformly from input units to output units. The activation from the previous time step (the last time the activation of the whole network was calculated) plays no part in the computation, because it is not fed back to an earlier unit. Hence, a feed-forward network simply computes a function of the input values that depends on the weight settings, it has no internal state other than the weights themselves. For instance, the function for the output of the network in figure 2.12 would be:

$$\begin{aligned} O &= g(w_{3,5}a_3 + w_{4,5}a_4) \\ &= g(w_{3,5}g(w_{1,3}a_1 + w_{2,3}a_2) + w_{4,5}g(w_{1,4}a_1 + w_{2,4}a_2)) \end{aligned} \tag{2.2}$$

where g is the activation function (see section 2.1.2), $w_{i,j}$ is the weight of the link between unit i to unit j and a_i is the activation of unit i.

A more specific type of a feed-forward network is the *layered feed-forward network*. In a layered feed-forward network each unit is linked only to units in the next layer. There are no links between units in the same layer, no links backward to a previous layer, and no links that skip a layer. The network in figure 2.12 is also a layered feed-forward network.

Neural networks that have cycles and/or bidirectional links are called recurrent networks. Note that our brain is a recurrent network as we otherwise had no short-term memory. In this course, we focus on feed-forward networks because they are relatively well-understood.

2.3 Artificial neuron revisited

The artificial neuron we described before is called a perceptron. There is another type of artificial neuron called a *sigmoid neuron*. In this section we describe the sigmoid neuron, but before we look at the sigmoid neuron let us simplify how we describe perceptrons.

2.3.1 Introducing Bias

In section 2.1.2 we described the working of a perceptron in equation 2.1 using inputs $x_1 \dots x_n$, corresponding weights $w_1 \dots w_n$ and threshold t. The condition $\sum_{i=1}^{n} w_i x_i > t$ is cumbersome, and we can make two notational changes to simplify it. The first change is to write it as a dot product, $\vec{w} \cdot \vec{x} = \sum_{i=1}^{n} w_i x_i$, where $\vec{w}$ and $\vec{x}$ are *vectors* whose components are the weights and inputs, respectively.

The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's bias: $b = -threshold$. Using the bias instead of the threshold, the perceptron activation function, g, can be rewritten:

$$output = g = \begin{cases} 0 & \text{if } \vec{w} \cdot \vec{x} + b < 0 \\ 1 & \text{if } \vec{w} \cdot \vec{x} + b \geq 0 \end{cases} \tag{2.3}$$

You can think of the bias as a measure of how easy it is to get the perceptron to output a 1. Or to put it in more biological terms, the bias is a measure of how easy it is to get the perceptron to fire. For a perceptron with a really big bias, it's extremely easy for the perceptron to output a 1. But if the bias is very negative, then it's difficult for the perceptron to output a 1.

We can simplify equation 2.3 and our calculations a bit more by making the bias a weight of the special input x_0 which will always be 1. This will give us the perceptron activation function, g,:

$$output = g = \begin{cases} 0 & \text{if } \vec{w} \cdot \vec{x} < 0 \\ 1 & \text{if } \vec{w} \cdot \vec{x} \geq 0 \end{cases} \tag{2.4}$$

where $\vec{w}$ is the vector whose components are the weights $w_0 \dots w_n$, $\vec{x}$ is the vector whose components are the inputs $x_0 \dots x_n$, w_0 is called the bias weight, with $x_0 = 1$.

This change from threshold to bias simplifies the calculations needed in neural networks, as now only vector multiplications are required. Later, in Section A, this means that we can more easily offload the neural network calculations to the GPU for training.

2.3.2 Sigmoid neuron

To understand why we have sigmoid neurons and why they are what they are we have to look at what we expect from a learning algorithm for neural networks. Suppose we have a network of perceptrons that we like to learn to solve some problem. For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. We would like the network to learn weights such that the

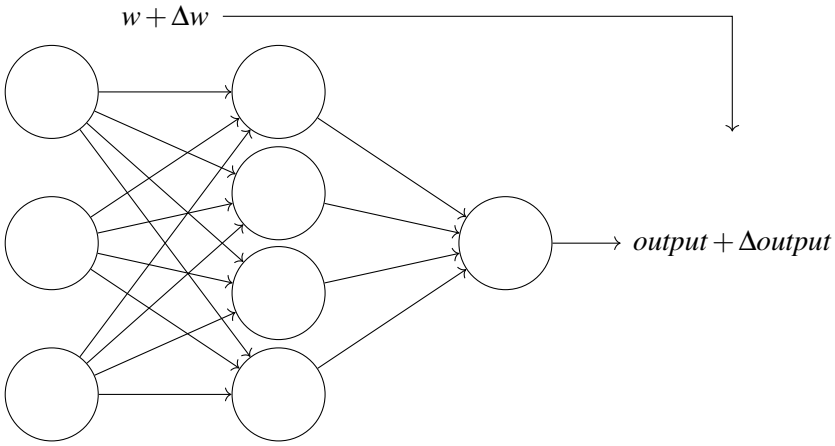


Figure 2.13 A small neural network. To be able to have an algorithm that can train a neural network, we want a small change in the weights, w, results in a small change in the output.

output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight in the network. What we would like is for this small change in weight to cause only a small corresponding change in the output from the network. This property will make learning easier. In figure 2.13 is schematically depicted what we want.

If it were true that a small change in a weight causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want. For example, suppose the network was mistakenly classifying an image as an "8" when it should be a "9". We could figure out how to make a small change in the weights and biases so the network gets a little closer to classifying the image as a "9". And then we can repeat this, changing the weights over and over to produce better and better output. The network would be learning.

The problem is that this is not what happens when our network contains perceptrons. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way. So while a "9" might now be classified correctly, the behaviour of the network on all the other images is likely to have completely changed in some hard-to-control way. That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behaviour.

We can overcome this problem by introducing a new type of artificial neurons called sigmoid neurons. Sigmoid neurons are similar to perceptrons, but modified such that small changes in their weights and bias cause only a small change in their output. That is the crucial fact which allows a network of sigmoid neurons to learn.

Just like a perceptron, the sigmoid neuron has an input vector $\vec{x}$ and a vector, $\vec{w}$, with weights for each connection. The big difference is that the possible inputs and output of a sigmoid neuron are not only 0's and 1's. The inputs and the output can take any value between 0 and 1. For instance, the value 0.563 is a valid input. To make this

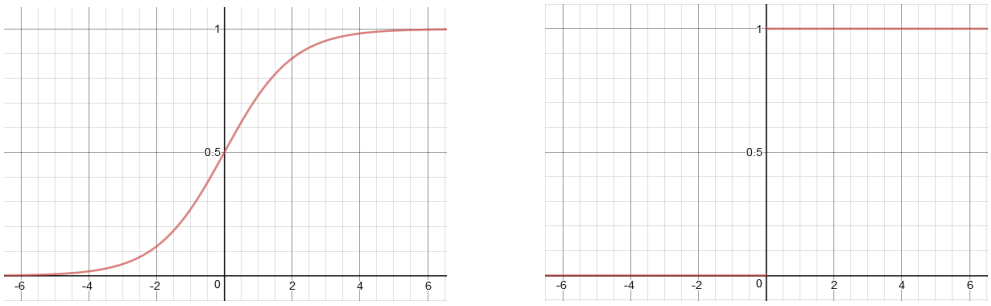


Figure 2.14 A plot of the sigmoid function (left) and step-function (right).

also possible for the output the sigmoid neuron uses a different activation function, g,:

$$output = g = \sigma(\vec{w} \cdot \vec{x}) \tag{2.5}$$

where σ is called the *sigmoid function*[5], and is defined by:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.6}$$

Figure 2.14 (left) shows a plot of the sigma function. We can see that it is a smoothed version of the activation function of the perceptron (also called *step-function*, see Figure 2.14 (right)). This means that we can represent the same functions as with the perceptrons (such as AND-gates, etc.). Furthermore, we have the desired feature for a learning algorithm: a small change in weights results in a small change in output.

2.4 Neural networks and learning

In the previous sections we collected all the parts to create a neural network. To make a learning neural network we only miss two things: examples to learn the task and a learning algorithm. Examples to learn from are called the *training set*. This consists of a set of inputs and the desired outcomes. For instance, in the case of number recognition, a set with images of numbers (input) each with a label of which number it is (desired output). This section explains how a neural network can learn when it has a training set.

2.4.1 Cost function

To learn humans need feedback. This is also true for neural networks. The function that tells the neural network how good it is doing is called the *cost function*. There are multiple types of functions than can be used as a cost function. Here, we use the mean squared error (MSE), also known as the quadratic cost function:

$$MSE = \frac{1}{n}\sum_{i=0}^{n} |\vec{y}_i - \vec{a}(\vec{x}_i)|^2 \tag{2.7}$$

[5]Incidentally, σ is sometimes called the logistic function, and this new class of neurons called logistic neurons. It is useful to remember this terminology, since these terms are used by many people working with neural networks. However, we stick with the sigmoid terminology.

Figure 2.15 A selection of the MNIST training set of handwritten digits.

The formula for this cost function $C(\vec{w})$ calculates the error (cost C) given a vector of weights ($\vec{w}$):[6]

$$C(\vec{w}) = \frac{1}{2}MSE = \frac{1}{2n}\sum_{i=0}^{n}|\vec{y}_i - \vec{a}(\vec{x}_i)|^2 \tag{2.8}$$

where n is the number of training inputs, x, $\vec{y}_i$ is the desired outcome of the network when $\vec{x}_i$ is the input and $\vec{a}(x_i)$ is the output of the neural network when $\vec{x}_i$ is the input. The notation $|\vec{v}|$ denotes the usual length function for a vector $\vec{v}$. [7]

Inspecting the form of the quadratic cost function, we see that $C(\vec{w})$ is non-negative, since every term in the sum is non-negative. Furthermore, the cost $C(\vec{w})$ becomes small, i.e., $C(\vec{w}) \approx 0$, precisely when $\vec{y}_i$ is approximately equal to the output, $\vec{a}$, for all training inputs, $\vec{x}_i$. So our training algorithm has done a good job if it can find weights and biases so that $C(\vec{w}) \approx 0$. By contrast, it's not doing so well when $C(\vec{w})$ is large - that would mean that $\vec{y}_i$ is not close to the output $\vec{a}$ for a large number of inputs. So the aim of our training algorithm will be to minimise the cost $C(\vec{w})$ as a function of the weights and biases. In other words, we want to find a set of weights and biases which make the cost as small as possible[8].

[6]The math for this cost function might seem complicated, but with a closer look we see it is literally the mean squared error. $|\vec{y}_i - \vec{a}(x_i)|$ is the difference (or distance) between the output that is given and the output that we want when given input $\vec{x}_i$. In other words $|\vec{y}_i - \vec{a}(x_i)|$ tells us how wrong the neural network is (i.e. how big the error is) when given input $\vec{x}_i$. As this value can be negative or positive and we are only interested in how wrong the neural network is, we square this value to make all values positive. This is the squared error part of the mean squared error. To get to MSE, we need to calculated the mean. This is done by summing ($\sum$) the squared error over all training inputs and dividing it by the number of training inputs, $i = 1, \ldots, n$, ($\frac{1}{2n}$). Actually we also divide by two (hence the 2 in $\frac{1}{2n}$). This has no effect on the functionality when minimising the function, but it makes the function derivative easier. As we use the derivative of this function this saves us some clutter. In short the MSE is nothing more than the average squared distance between the output of the neural network and the desired output.

[7]If we have a vector $\vec{v}$ with values v_1, v_2 and v_3, then $|v|$ is equals to $\sqrt{{v_1}^2 + {v_2}^2 + {v_3}^2}$.

[8]For some readers the questions might rise why we try to minimise the quadratic cost function, as we just as well could maximise the number of correct classified training samples. The problem with the number of correct classified training samples is that it is not a smooth function. Making a small change

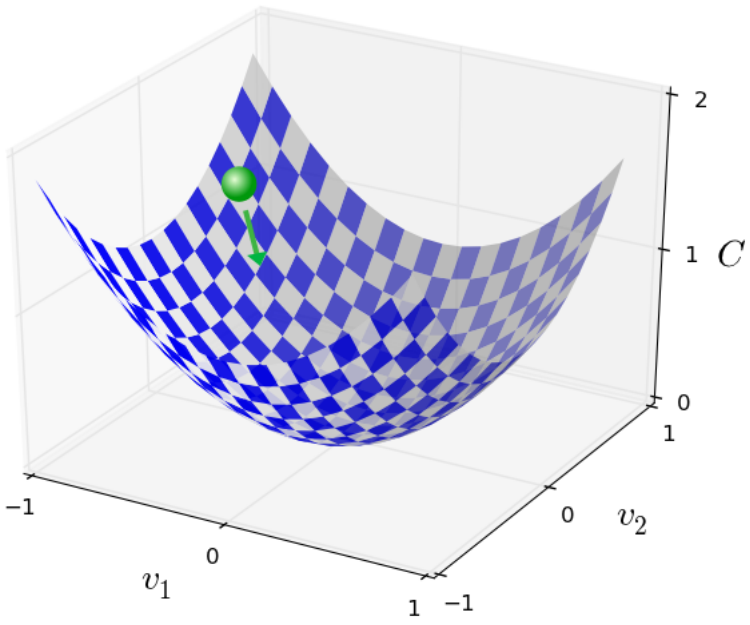


Figure 2.16 Gradient descent can be seen as taking small steps in the direction a ball would roll, until you reach a minimum.

2.4.2 Gradient descent

In section 2.4.1, we saw that the goal of a learning algorithm is to minimise the cost function $C(\vec{w})$. As $\vec{w}$ consists out of a lot of weights it takes too much time to calculated for each possible $\vec{w}$ the value of $C(\vec{w})$ to see where $C(\vec{w})$ has the lowest value. A solution for this problem is to use gradient descent. Gradient descent is an optimisation algorithm that approaches a (local) minimum of a function by taking steps in the direction of the negative of the gradient of the function at the current point.

For a first understanding of this algorithm, imagine a blind woman on a mountain that has as goal to go the valley (a minimum). The blind woman doesn't know where she is and can't look around to see which way she has to go. As a solution she feels around to find the slope of the mountain. Now she knows where the mountain goes down and she takes a step in that direction. By repeating this for each step, until she cannot go further down, she knows she ends in a (local) minimum.

Another way to visualise this technique can be seen in figure 2.16. The plane is the cost function with weights v_1 and v_2. A ball is placed at a random place on the plane. The ball is moved a small step into the direction it would roll. This is repeated until we reach the minimum of the plane. When the minimum is reached we have the desired values for v_1 and v_2.

in the weights of the units will most of the time result in no change in the number of correct classified training samples. This makes it hard to find which change we should make to the weights of the network. The quadratic cost function is smooth and, therefore, does not have this problem.

Back to our problem. We cannot calculate the minimum of the function, but we can start at a random position (random weights) and then use gradient descent to move down to a minimum. More formal, the algorithm starts with a random $\vec{w}$ and calculates which small change, $\Delta\vec{w}$, reduces $C(\vec{w})$. In other words, we want to find a $\Delta\vec{w}$, that has a negative $\Delta C(\vec{w})$. When $\Delta\vec{w}$ is applied to the weights $C(\vec{w})$ comes closer to a minimum. When we keep repeating this step until we cannot find a $\Delta\vec{w}$, such that $\Delta C(\vec{w})$ is negative, than we are at a (local) minimum and the network has learned its weights.

The $\Delta\vec{w}$ can be found by calculating the gradient on our current position, $\vec{w}$. The gradient can be found by using the derivatives[9] over each weight. The collection of the gradients is called the gradient vector, ∇C:

$$\nabla C = \left(\frac{\partial C}{\partial w_1}, \ldots, \frac{\partial C}{\partial w_n}\right)^T \tag{2.9}$$

where $\frac{\partial C}{\partial w_i}$ is the gradient over weight w_i and note that T here is the transpose operation, turning a row vector into an ordinary (column) vector.

Gradient descent tells us that if we move the weights in the negative direction of ∇C, then we move in the direction of a (local) minimum. With this we can create an update rule that tells us that in each step our new weights, $\vec{w}'$, become:

$$\vec{w}' = \vec{w} - \Delta\vec{w} = \vec{w} - \eta\nabla C \tag{2.10}$$

where η is called the learning rate and denotes the size of the step we take in the direction of $-\nabla C$. When η is too large we might overshoot a minimum, but a too small η requires a lot of steps making the algorithm very slow.

Without vectors, which might make it easier to understand, the update rule for weight w_i is:

$$w_i' = w_i - \Delta w_i = w_i - \eta\frac{\partial C}{\partial w_i} \tag{2.11}$$

2.4.3 Single-layer network

From the last section we know that we can train our neural networks using gradient descent and that we can do that using the derivatives of the cost function. But what are these derivatives? Here we work towards the update rule for our weights for single layer networks. In the next section we generalise this rule to work for multi-layer networks.

First take a look again at the cost function (see equation 2.8). It requires to look at all training examples for each step. This is a big calculation just for one step. To make our training algorithm faster (but a bit less accurate), we take a step after each training example. The cost function for just one training example is:

$$C = \frac{1}{2}|\vec{y} - \vec{a}(\vec{x})|^2 \tag{2.12}$$

[9] If you are not familiar (anymore) with what a derivative is we recommend watching the first 3 chapters of "essence of calculus" by 3Blue1Brown that can be found on youtube: `https://youtu.be/WUvTyaaNkzM`.

Now we can use gradient descent to reduce the squared error[10] by calculating the partial derivative of C with respect to each weight and updating the weight:

$$\begin{aligned} w_i' &= w_i - \eta \frac{\partial C}{\partial w_i} \\ &= w_i - \eta \frac{\partial (\frac{1}{2} |\vec{y} - \vec{a}(\vec{x})|^2)}{\partial w_i} \end{aligned} \tag{2.13}$$

In a single layer network (see figure 2.17) each weight has a direct influence on one of the output nodes. The update rule can therefore be rewritten for the weight from input node j to output node k:

$$w_{j,k} = w_{j,k} - \Delta w_{j,k} = w_{j,k} + \eta a_j g'(in_k)(y_k - a_k) \tag{2.14}$$

where a_j is the activation of input node j, g' is the derivative of the activation function [11], in_k is the summed input of output node k, y_k is the desired activation of output node k and a_k is the actual activation of output node k[12]. This updating rule is called the *delta rule*.

The learning algorithm starts by assigning random values to the weights. Then it keeps updating the weights using the delta rule. Note that the update of the weights happens simultaneously. So, first are all $\Delta w_{j,k}$ calculated and then are all weights updated using the $\Delta w_{j,k}$'s. When a minimum is reached the algorithm stops.

2.4.4 Multi-layer network

In multi-layer networks we have the problem that the weights to hidden layers do not directly influence the output. To be able to also train these weights we take the idea that hidden node j is "responsible" for some fraction of the error in the layer after it. To get to our update rule for the hidden layers we rewrite the delta rule to emphasise what we see as the error:

$$\Delta_k = g'(in_k)(y_k - a_k) \tag{2.15}$$

$$w_{j,k} = w'_{j,k} + \eta a_j \Delta_k \tag{2.16}$$

where Δ_k is the error of output node k.

If we would know Δ_j, the fraction of error that hidden node j is responsible of in all output nodes, then we can rewrite the update rule to get the update rule for the weights from node i to hidden node j:

$$w_{i,j} = w'_{i,j} + \eta a_i \Delta_j \tag{2.17}$$

[10] Note that we no longer have to calculate the mean of the squared error, as we are only looking at a single example.

[11] For the sigmoid neuron the derivative is given by $g' = g(1-g)$

[12] Note the change of sign in the equation: the (partial) derivative $\eta \frac{\partial (\frac{1}{2} |\vec{y} - \vec{a}(\vec{x})|^2)}{\partial w_i}$ is negative (as we are looking for the down slope), which means that the sign changes from a minus to a plus.

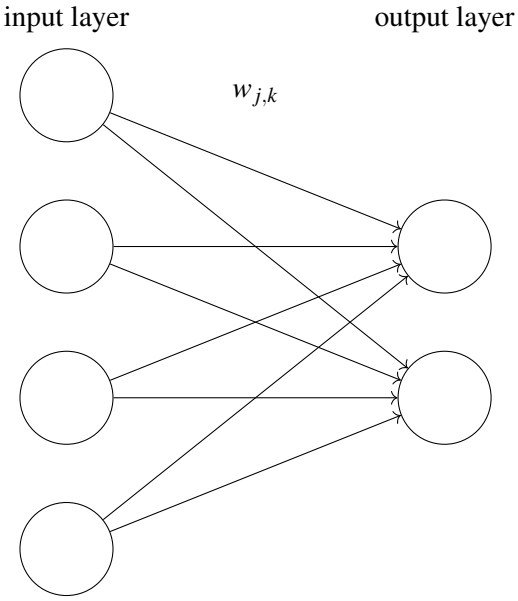


Figure 2.17 A single layer neural network

.

where a_i is the activation of node i. This looks quite similar as the rule for the weights to the output layer. We can calculate Δ_j with:

$$\Delta_j = g'(in_j) \sum_p w_{j,p} \Delta_p \tag{2.18}$$

where $w_{j,p}$ is the weight from node j to node p and Δ_p is the error of node p. If node p is an output node then Δ_p is calculated as Δ_k in equation 2.15 otherwise it is calculated the same way as Δ_j. In the formula for Δ_j we see that node j is held "responsible" for the error in the nodes it is connected to in the next layer dependent on its activation and its weights.

The algorithm is similar to that of the single-layer networks. The algorithm starts by assigning random values to the weights. Then for each example in the training set it calculates the activation for each node. As the activation of a node depends on the activation of the nodes of the previous layer, this is called forward propagation (the activation moves from the input layer to the output layer). Then the error deltas (Δ) for each node are calculated. As these are dependent on the next layer the delta calculation moves from the output layer to the input layer. The error propagates back through the network. When the deltas are known the weights are updated. One loop over all training samples is called an *epoch*. This is repeated for multiple epochs until some stopping criterion (such as a minimum in the cost function) is reached. This algorithm is called *backpropagation* named after the back propagation of the errors.

2.5 Exercises

Exercise 2.1 — NOR-Gate.
Determine weights and a threshold for a perceptron that would act as a NOR-gate with three inputs.

Exercise 2.2 — Neural ADDER.
Make an adder out of perceptrons. Draw the network and give the corresponding weights and biases.

Exercise 2.3 — Programming NNs.
In this exercise you are going to build a (naive) implementation of a neural networks[a] and use it to classify a number of data sets.

A) Neuron
Write a class to represent a neuron and its functions. Learning capabilities are not necessary yet. Use this class to implement the neuron of exercise 2.1 and the network of exercise 2.2.

B) Delta Rule
Add to the neuron class an update function. This function uses the delta rule to updates its weights. The function has as input the desired activation of the node. Test your new function to train the neuron of exercise 2.1.

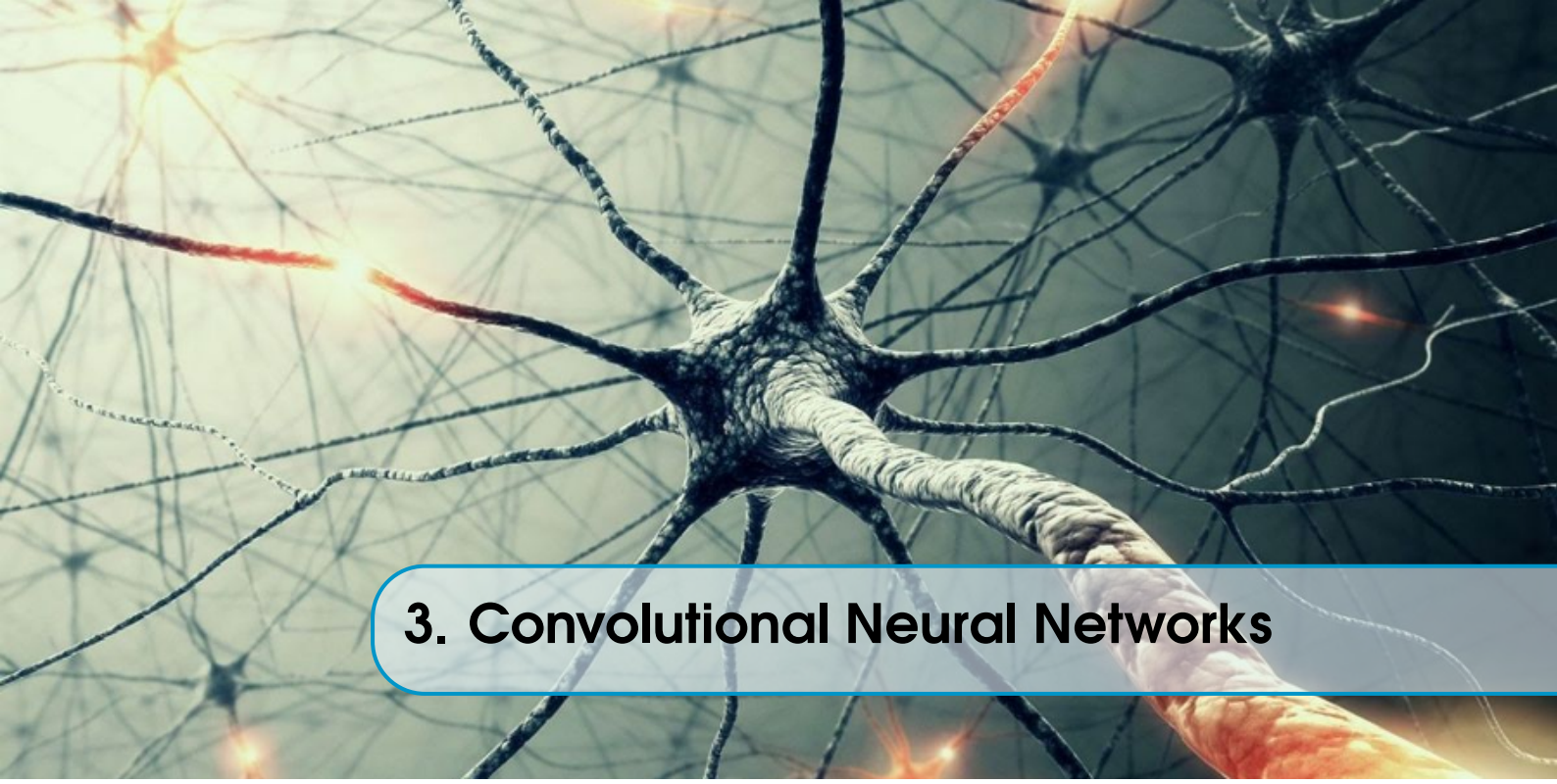
C) Backpropagation
Add backpropagation to your program. Create the XOR network from the sheets. Initialize the weights with random values. Train the network.

D) Iris dataset
Create a neural network, using your own code, that is able to classify correctly a high percentage of flowers from the iris dataset[b]. Make sure you use a train set and a test set.
Report the shape of your network and its score on the test set.

E) Weather data (optional)
Adjust your implementation to work on the weather data from the previous chapter. Can your NN perform as well as (or better than) your k-NN implementation?

[a] In the next section we discuss a "smarter", more mathematical, implementation of neural networks.

[b] `http://archive.ics.uci.edu/ml/datasets/Iris`



3. Convolutional Neural Networks

3.1 Introduction

Convolutional neural networks (CNNs or ConvNets) are very similar to ordinary neural networks as presented in chapter 2: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs and performs a dot product (see chapter A). The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. All tips and tricks for learning regular neural networks also still apply.

What makes CNNs different is that they make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function (the classification) more efficient to implement and vastly reduces the amount of parameters in the network (which, in turn, increases the efficiency in learning).

Regular neural networks do not scale well with full images. Small image datasets, like CIFAR-10, which contains images of only $32 \times 32 \times 3$ (32 pixels wide, 32 pixels high, 3 colour channels), can still be handled with a regular neural network, but requires a lot of weights. A single fully-connected neuron in a first hidden layer would have $32 * 32 * 3 = 3072$ weights (the weights are often also called *parameters*). While this still seems manageable, consider such a network for a slightly larger image; e.g., $200 \times 200 \times 3$, which leads to neurons with $200 * 200 * 3 = 120,000$ weights. Not to mention that we would probably want to have multiple of such neurons, which means the number of parameters of the network sky-rockets rather fast.

ConvNets, however, take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike regular neural networks, the layers of a ConvNet have neurons arranged in three dimension: **width**, **height**, **depth**[1]. For example, the input images in CIFAR-10 are an input

[1]Please note that *depth* is typically reserved to denote the number of (hidden) layers of a neural

volume of activations, and the volume has dimensions $32 \times 32 \times 3$ (width, height, depth, respectively, see right-hand side of Figure 3.1).

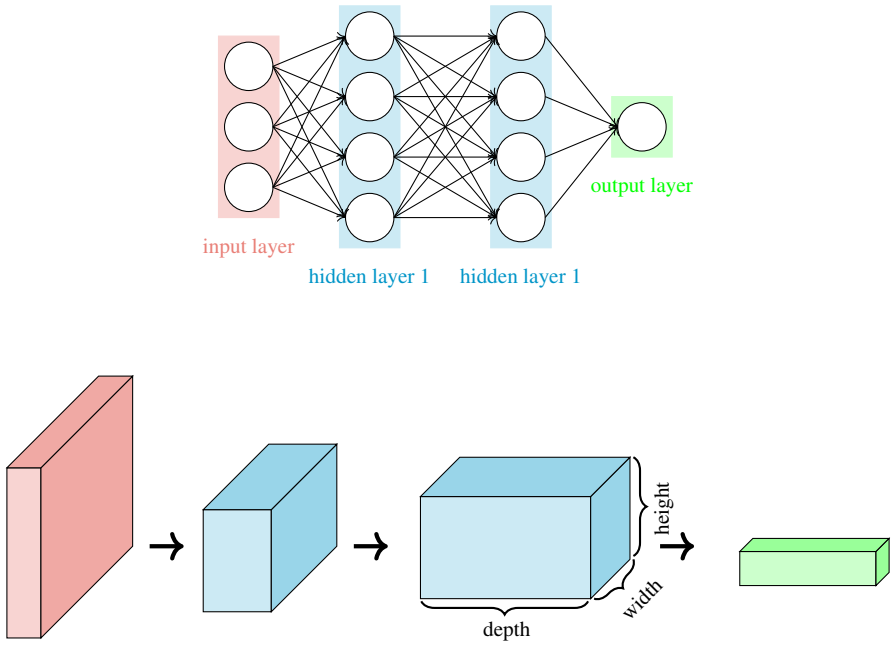


Figure 3.1 Top: a regular 3-layer neural network. Bottom: a convnet arranges its neurons in three dimensions (width, height, depth). Each layer in a convnet transforms the 3D input volume to a 3D output volume of neuron activations.

3.2 General

As described above, a simple ConvNet is a sequence of layers, where every layer transforms one volume of activations into another through a differentiable function. We distinguish three main types of layers in a convnet architecture: *convolutional layer*, *pooling layer*, and *fully-connected layer* (this latter functions exactly the same as in regular neural networks). A stack of such layers forms a full convent architecture.

A simple example for the CIFAR-10 classification could have the architecture [INPUT – CONV – RELU – POOL – FC]. In more detail:

- INPUT $[32 \times 32 \times 3]$ holds the raw pixel values of the image, in this case images with width 32, height 32, and three colour channels (R, G, B);
- CONV layer computes the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in a volume such as $[32 \times 32 \times 12]$ if we decide to use 12 filters;
- RELU layer applies an elementwise activation function, such as the $max(0,x)$ threshold at zero. This leaves the size of the volume unchanged;
- POOL layer performs a downsampling operation along the spatial dimensions (width, height), resulting in a volume such as $[16 \times 16 \times 12]$;

network. Here *depth* denotes the third dimension of the activation volume of a particular layer in the network.

- FC (i.e., fully-connected) layer computes the class scores, resulting in a volume of size $[1 \times 1 \times 10]$, where each of the 10 numbers corresponds to a class score (the CIFAR-10 dataset had 10 different classes, including cars, airplanes, etc.). As with ordinary neural networks and as the name implies, each neuron on this layer is connected to all the neurons on the previous volume.

In this way, convnets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters (CONV and FC) and others do not (INPUT, RELU and POOL). The CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers implement a fixed function. The parameters in the CONV/FC layers are trained with gradient descent (see Section 2.4.2) such that the class scores that the convnet computes is consistent with the labels in the training set.

3.2.1 Convolutional layer

CONV layers consist of a set of learnable *filters* (sometimes also called *kernels*). Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical layer on a first layer of a convnet might have the size $5 \times 5 \times 3$ (that is, 5 pixels wide and high, and 3 because input images have depth 3). During the forward pass, each filter is slid across the width and height of the input volume and is used to compute the dot products between the entries of the filter and the input at any position. As the filter slides over the width and height of the input volume, a 2-dimensional activation map is produced that gives the responses of that filter at every spatial position. Intuitively, the network learns filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some colour on the first layer, or eventually combinations of such patterns, like entire honeycomb or wheel-like patterns, on later layers of the network. Typically, a network has a set of filters per conv layer, and each produces a separate 2-dimensional activation map. The output volume of the layer is created by stacking these activation maps together along the depth dimension.

Local connectivity

When dealing with high-dimensional inputs such as images, it is impractical to connect all neurons to all neurons on the previous volume. Instead, neurons are only connected to a local region of the input volume. The spatial extend of this connectivity is a hyperparameter[2] called *receptive field* of the neuron (which kind of expresses the size of the filter related to the neuron). The extend of the connectivity along the depth axis is always equal to the depth of the input volume. That is, every filter always considers all depth slices of the input volume, but not all the width and height (at once).

Spatial arrangement

Neurons in the conv layer are thus locally connected to the input volume. Their arrangement in the output volume is determined by three hyperparameters: the *depth*, *stride*, and *zero-padding*:

[2]The term *hyperparameter* is often used to distinguish between the *parameters* of a layer (i.e., its weights and bias) and the characteristics of a layer. The latter are referred to as hyperparameters.

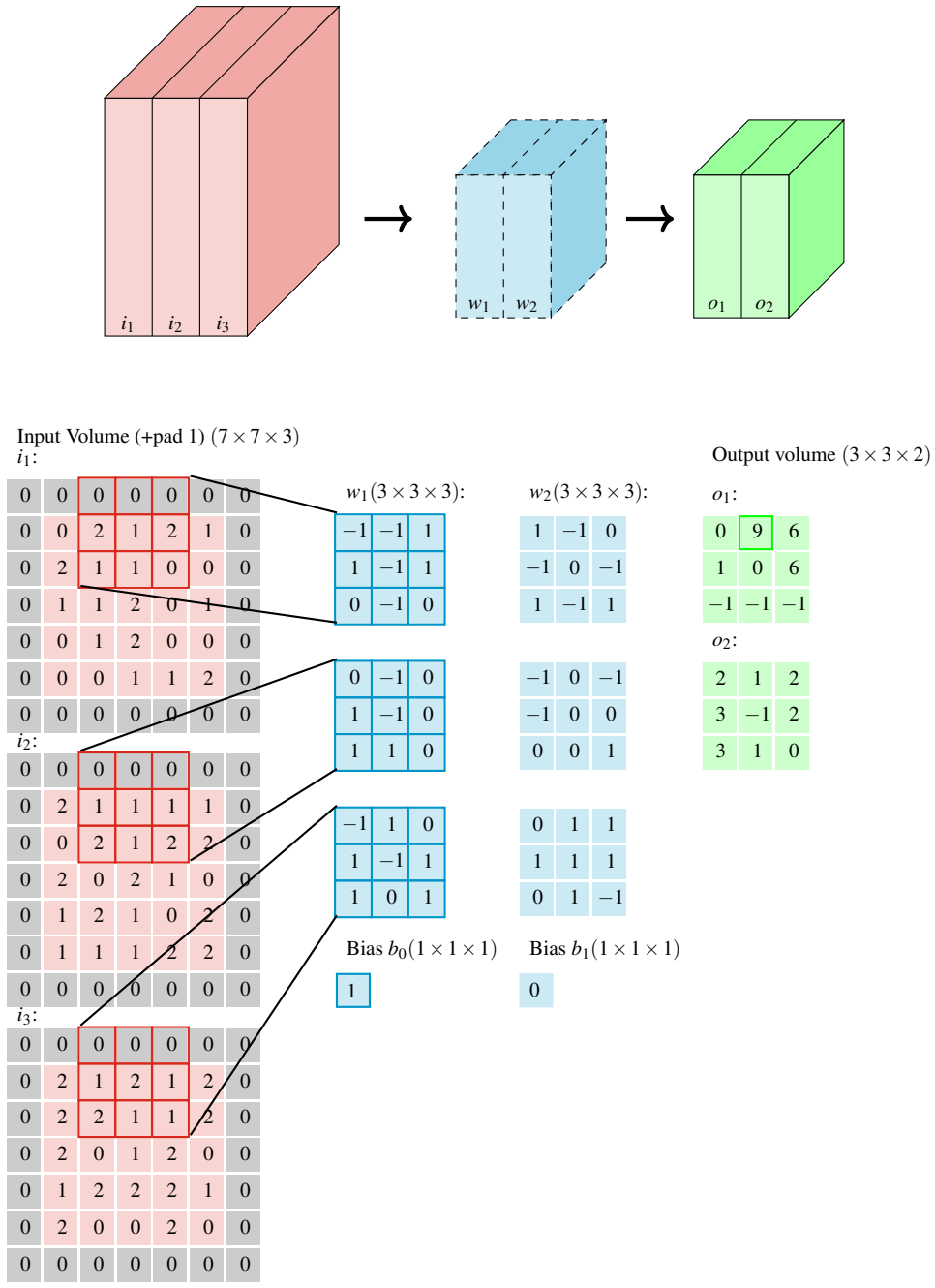


Figure 3.2 Example of hyperparameters and calculations in convolutional layers. Using two filters of 3×3 (with depth 3, as the input has depth 3),padding 1, and stride 2. Note that w_1 and w_2 are virtual: the neurons in the conv layer (in this case o_1 and o_2) are wired through the filters to the activation volume. That is, the neuron highlighted in output is actually wired (using the filters and weights specified in w_1) to the highlighted regions of the activation volume.

- First, *depth* of the output volume corresponds to the number of filters that are used in the convolutional layer (each looking for something different in the input). For example, if the first convolutional layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of colour. We refer to a set of neurons that are all looking at the same region of the input as a *depth column* (or *fibre*).

- Second, the *stride* determines the amount with which the filter slides across the input volume. When the stride is 1, the filter moves 1 pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filter jumps 2 pixels at a time. This produces smaller output volumes spatially.
- Sometimes it is convenient to pad the input volume with zeros around the border. The size of this *zero-padding* is a hyperparameter. The nice feature of zero padding is that it allows us to control the spatial size of the output volumes (commonly, padding is used to preserve the spatial size of the input volume so the input and output width and height are the same)[3].

The spatial size of the output volume can be computed as a function of the input volume size (W)[4], the receptive field size of the conv layer neurons (F)[5], the stride with which they are applied (S), and the amount of zero padding used (P) on the border. The correct formula for calculating how many neurons may "fit" is given by $(W-F+2P)/S+1$. For example, for a 7×7 input and a 3×3 filter with stride 1 and pad 0 we would get a 5×5 output (there are only 5 unique ways to fit a width of 3 on a width of 7, with stride 1). With stride 2 we would get a 3×3 output. Note that the answer to the function above needs to be an integer (a whole number), otherwise we cannot fit the filter in the input volume.

Parameter sharing

Parameter sharing scheme is used in convolutional layers to control the number of parameters (weights). As shown earlier, fully connecting all neurons and giving separate weights to each connection, quickly becomes unmanageable.

It turns out that the number of parameters can be drastically reduced by making a reasonable assumption: if one feature is useful to compute at some spatial position (x, y), then it should also be useful to compute at a different position (x_2, y_2). In other words, each of the neurons in a depth slice uses the same weights and bias.

Notice that if all neurons in a single depth slice are using the same weight vector, then the forward pass of the CONV layer can in each depth slice be computed as the **convolution** of the neuron's weights with the input volume (hence the name: convolution layer). This is also why it is common to refer to the set of weights as a *filter* or *kernel*, that is convolved with the input.

Summary

A convolutional layer:

- accepts a volume of size $W_1\times H_1\times D_1$;
- requires four hyperparameters:
  - number of filters K;
  - their spatial extend F;
  - the stride S;

[3] To determine the amount of padding required to make the output volume spatially equal to the input volume, use the following formula (for stride 1): $P=(F-1)/2$.

[4] As width and height are typically the same, we only need to use either to compute the output size.

[5] The receptive size of a conv layer is the size ($W_f\times H_f$) of the filters applied. All filters in a single layer typically have the same size, and are normally considered to be square (so F is equal to either W_f or H_f).

  - the amount of zero padding P.
- produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- with parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases;
- in the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

There are common conventions and rules of thumb that motivate these hyperparameters; for example, often $F = 3$, $S = 1$, and $P = 1$ are used.

3.2.2 Rectified linear unit (RELU) layer

In chapter 2 discussed as being a part of a neuron, in some architectures the activation function of the neurons is represented as a separate layer. This is nowadays more common, as it more clearly represents the hyperparameters of the network, and is easier to perform mathematically[6].

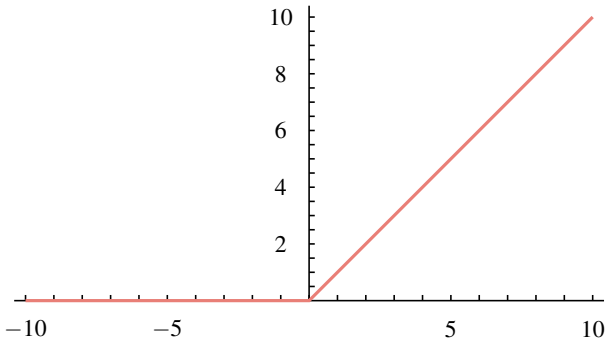


Figure 3.3 Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$.

The Rectified Linear Unit as activation has become very popular in the last few years. It computes the function $f(x) = (max)(0,x)$. In other words, the activation is simply thresholded at zero (see image below). There are several pros and cons to using the ReLUs:

(+) It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions.

(+) Compared to tanh/sigmoid neurons that involve extensive operations, the ReLU can be implemented by simply thresholding a matrix of activations at zero.

(-) Unfortunately, ReLU units can be fragile during training and can “die”. For example, a large gradient flowing through a ReLU neuron could cause the

[6] Instead of performing the activation function ‘within’ the neuron, the neuron now only calculates the appropriate dot products, after which its output (the result of the dot products) is passed through a simple filtering function.

weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing trhough the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. This may happen as much as 40% of your network's neurons if the learning rate is set too high. With proper setting of the learning rate this is less frequently an issue.

3.2.3 Pooling layer

Pooling layers are commonly inserted periodically in-between successive conv layers in a convnet architecture. The function of a pooling layer is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to control overfitting. The pooling layer operates independently on every depth slice of the input and resizes it spatially, for instance using the MAX operation. The most common form used is a pooling layer with filters of size 2×2 applied with a stride of 2. This downsamples every depth slice in the input by 2 along both the width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers. The depth dimension remains unchanged. More generally, the pooling layer:

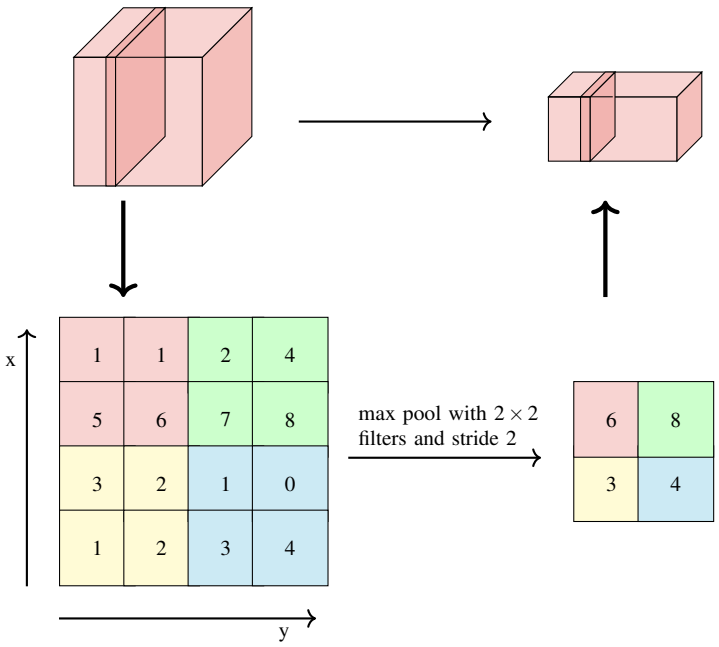


Figure 3.4 Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume

- accepts a volume of size $W_1 \times H_1 \times D_1$;
- requires two hyperparameters:
  - their spatial extend F;
  - the stride S.
- produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H1 - F)/S + 1$

  - $D_2 = D1$
- introduces zero parameters since it computes a fixed function of the input;
- Note that it is not common to use zero-padding for Pooling layers (they do not add anything useful).

It is worth noting that there are only two commonly seen variations of the max pooling layer found in practice: a pooling layer with $F = 3$, $S = 2$ (also called overlapping pooling), and more commonly $F = 2$,$S = 2$. Pooling sizes with larger receptive fields are too destructive.

In addition to max pooling, the pooling units can also perform other functions, such as *average pooling* or even *L2-norm pooling*. Average pooling was often used historically but has recently fallen out of favour compared to the max pooling, which has been shown to work better in practice.

3.2.4 Fully-connected layer

Neurons in a fully-connected layer have full connections to all activations in the previous layer, as seen in regular neural networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. See Chapter A for more information.

3.3 Architectures

Convolutional networks are commonly made up of only three types of layers: CONV, POOL (we assume max pool unless stated otherwise) and FC. We also explicitly write the RELU activation function as a layer, which applies elementwise non-linearity.

3.3.1 Layer patterns

The most common form of a convnet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size. At some point, it is common to transistion to fully-connected layers. The last fully-connected layer holds the output, such as the class scores. In other words, the most common convnet architecture follows the pattern:

$$\texttt{INPUT} \rightarrow [[\texttt{CONV} \rightarrow \texttt{RELU}] * \texttt{N} \rightarrow \texttt{POOL?}] * \texttt{M} \rightarrow [\texttt{FC} \rightarrow \texttt{RELU}] * \texttt{K} \rightarrow \texttt{FC}$$

where the $*$ denotes repetition, and the POOL? indicates an optional pooling layer. Moreover, $\texttt{N} \geq 0$ (and usually $\texttt{N} \leq 3$), $\texttt{M} \geq 0$, $\texttt{K} \geq 0$ (and usually $\texttt{K} < 3$).

It should be noted that it is preferred to use a small stack of filter CONV layers to one large receptive field CONV layer. For example, it can be shown that three 3×3 CONV layers on top of each other (with respective RELUs, of course) perform as well as a single layer with a 7×7 filter, while nearly halving the number of parameters required. Intuitively, stacking CONV layers with tiny filters as opposed to having one CONV layer with big filters allows us to express more powerfull features of the input, and with fewer parameters. As a practical disadvantage, though, multiple CONV layers require more memory to hold all the intermediate results when doing backpropagation.

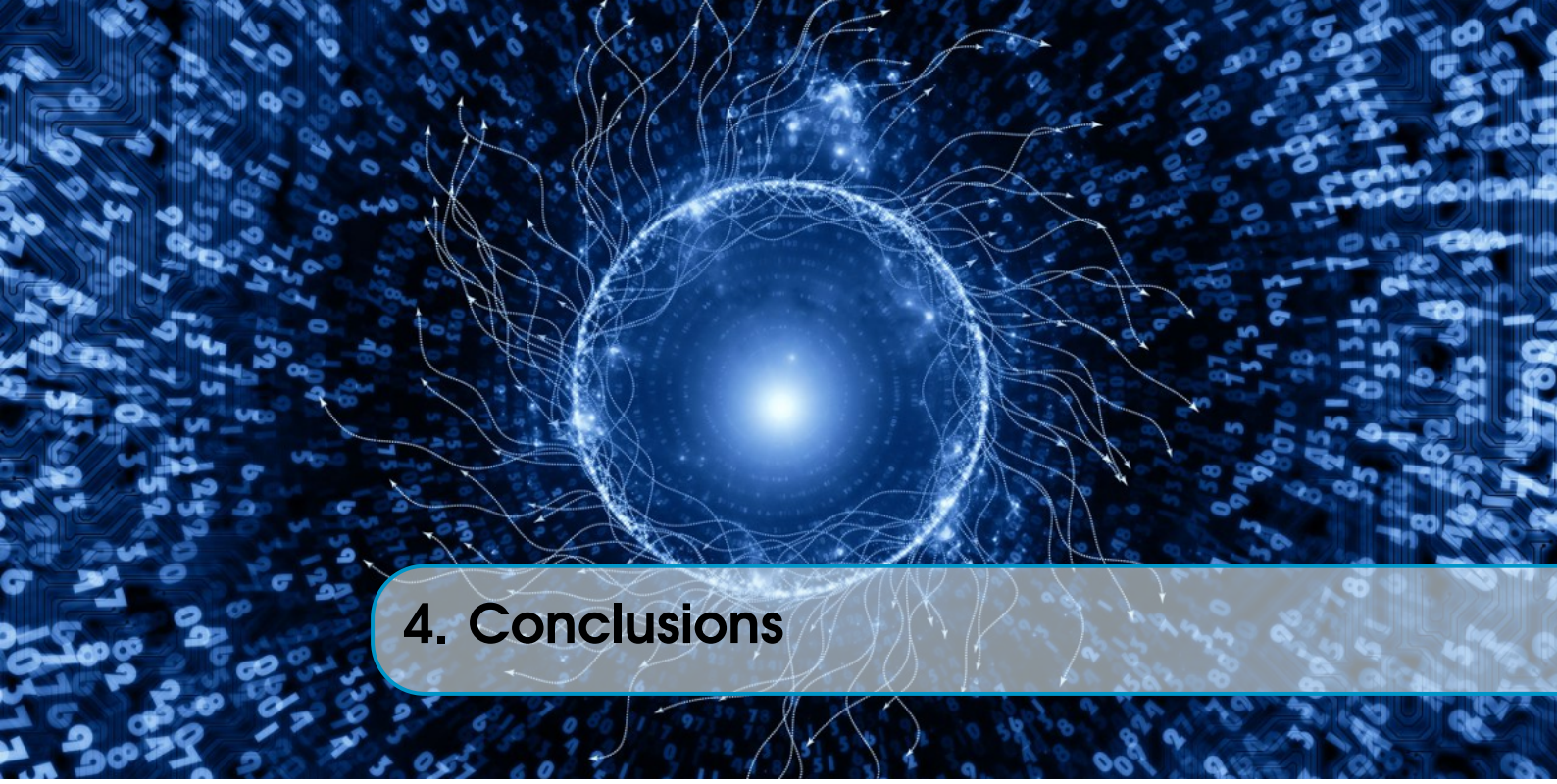
Layer sizing heuristics

Until now we have omitted mentions of common hyperparamters used in each of the layers in a convnet.

The *input layer* (that contains the image) should be divisible by 2 many times. Common numbers include 32, 64, 96, 224, 384, and 512.

The *conv layer* should be using small filters (preferable 3×3 or 5×5, using a stride of $S = 1$, and crucially, padding the input volume with zeros in such a way that the conv layer does not alter the spatial dimensions of the input. That is, when $F = 3$, then use $P = 1$ to retain the original size of the input. When $F = 5$, $P = 2$. Only use larger filter sizes (like 7×7 or so) on the very first conv layer that is looking at the input image (remember, calculate the required padding with $P = (F-1)/2$).

The *pool layers* are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max pooling with 2×2 receptive fields, and a stride of 2. Note that this discards exactly 75% if the activations in the input volume. Another slightly less common setting is to use 3×3 receptive fields with a stride of 2. It is very uncommon to see receptive field sizes for max pooling that are larger than 3 because the pooling is then too lossy and aggressive. This usually leads to worse performance.

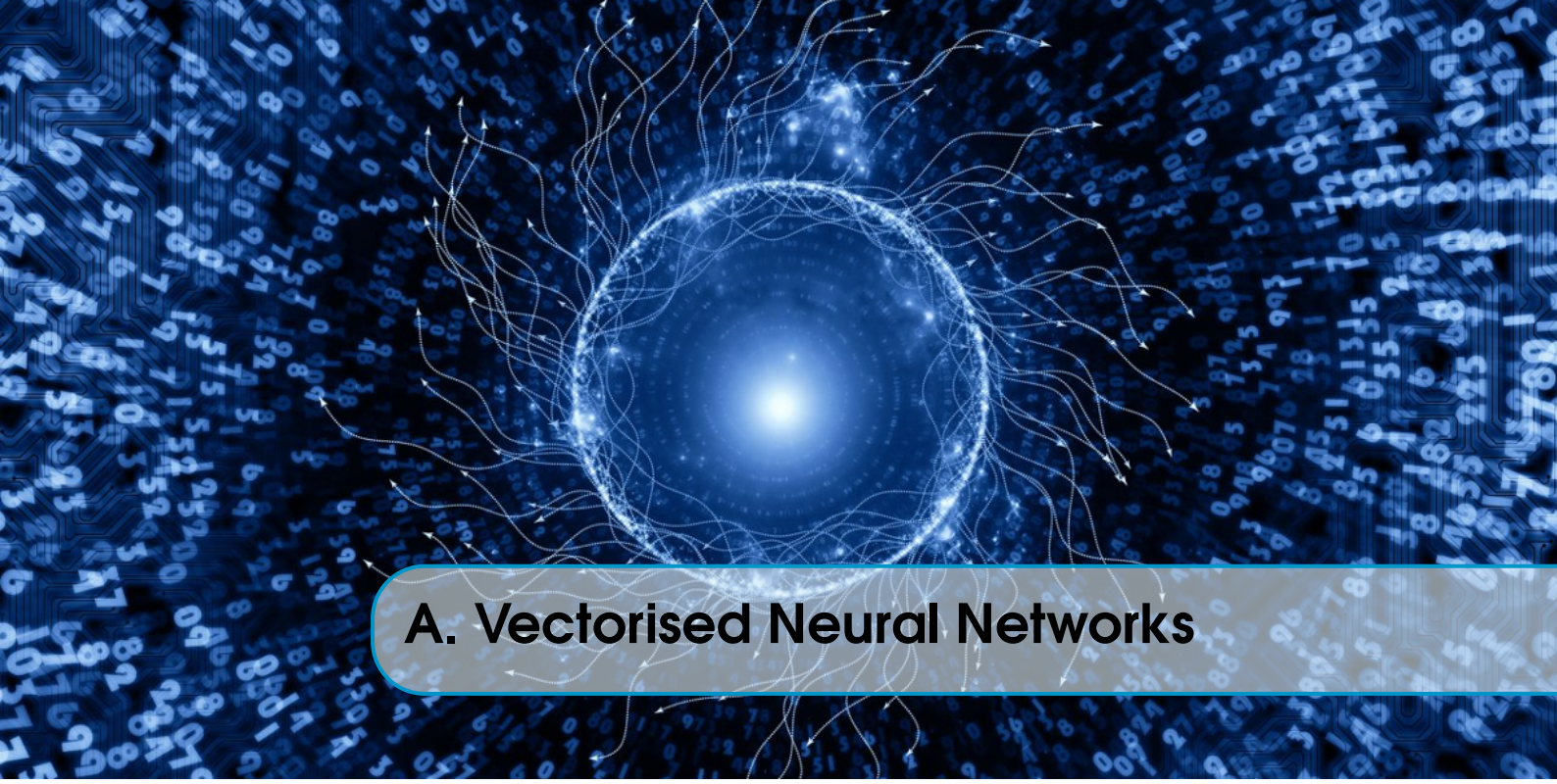


4. Conclusions

While the ideas and techniques used in neural networks has been around for nearly 70 years now, it was the developments of the recent years that made neural networks to what they are. With the 're-invention' of neural networks, and the introduction of convolutional neural networks, their strength has been increased significantly. You cannot open a newspaper nowadays and find yet another article about some AI application that has recently been introduced. 99% of those applications use (some form of) neural networks.

In this course we have tried to give a broad foundation to the technology of neural networks and deep learning, but also tried to paint the bigger picture of what AI represents and what can be achieved with it. There are many methods that incorporate some form of AI, and their application is wide-spread. Many companies nowadays agree that they should use some form of AI (most likely, some form of data processing through machine learning), yet only a few companies know exactly where to implement AI and how to use it optimally. There are still many challenges ahead for AI, and the correct application of it is one of its biggest.

The basic principles that we tried to teach in this course should remain true for most of the newer applications and developments within the field of AI. Only recently, Google researchers announced a new form of neural networks, called *Capsule networks* (**capsules**), which is meant to change the way image processing is handled by neural networks. While it makes significant changes to the architecture of 'traditional' neural networks, the underlying principles (layers, activations, weights, etc.) remain the same, and can easily be comprehended with the materials presented in this course.



A. Vectorised Neural Networks

Though the idea of Neural Networks is older than you might think (refer Chapter 1), it is only recently that the techniques started to yield the impressive results that have quickly become synonymous with its name. The reasons for this are myriad, but a large factor was the initial one: the march of technology is catching up with the requirements needed by complex neural networks. To utilise this increase in computing power, various optimisations have been developed for representing neural networks, and an important concept in this is parallelisation. By its very design as a great number of very simple entities, a neural network is a good fit for exploiting modern graphics cards, which consist of hundreds or thousands of cores able to perform a small number of relatively simple tasks. The modern GPU is optimised for matrix and vector calculations, because this is how 3D-graphics are represented. In this chapter we see how neural networks can be represented using basic linear algebra (the area of math concerned with vectors and matrices) to enable us to utilise this vast amount of parallel computing power.

A.1 Linear Algebra[1]

Before we take a look at neural networks, we first do a recap of the necessary linear algebra. This section is divided into subsections describing vectors and matrices.

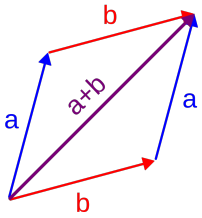
A.1.1 Vectors and Vector Spaces

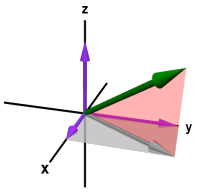
Vectors are a mathematical object for which three common intuitions exist. We look at each in turn, and then turn our attention to how these intuitions are helpful.

[1] Note that while we used $\vec{x}$ in the previous to denote a vector, in this chapter we use the more convenient $\mathbf{x}$ notation. See Section A.6 for an overview of the notations used in this chapter.

Arrows (Physics)

Vectors are represented by arrows in a 2-dimensional plane or a 3-dimensional space. Vectors can be combined to form new vectors, or scaled to become larger or smaller. Vectors are usually used to describe forces acting on objects, and are said to have magnitude (their length) and direction.





Arrays of Numbers (Data Science)

Vectors in data science are not limited to two or three dimensions. A vector is considered to be an ordered list of numbers of a fixed length. A vector of two dimensions contains two numbers (usually real numbers). Such a vector is said to be in $\mathbb{R}^2$. Vectors of two and three dimensions can be represented like their counterparts in physics, higher-dimensional vectors lack such a visualisation. Fortunately, most intuitions in lower dimensions scale well to higher dimensions.

$$\begin{bmatrix} 0.1359 \\ 0.4671 \\ -0.3379 \\ -0.2229 \\ -0.1364 \\ 0.3009 \end{bmatrix} + \begin{bmatrix} 0.0381 \\ -0.4876 \\ 0.9101 \\ 0.0011 \\ 0.4284 \\ -0.0139 \end{bmatrix} = \begin{bmatrix} 0.174 \\ -0.0205 \\ 0.5722 \\ -0.2288 \\ 0.292 \\ 0.287 \end{bmatrix}$$

Abstract (Mathematics)

Mathematically, a vector is defined more abstact, as an object within a vector space, for which a number of operations are defined. We can combine two vectors by adding them, putting the arrows end-to-end, or by adding every coefficient one at a time. Furthermore, we can also scale a vector by a *scalar* (usually a real number), by scaling the arrow or multiplying each coefficient by this number. In every vector space, there exists exactly one zero-vector, called $\mathbf{0}$, which is an arrow without length and consists of only 0 coefficients. Furthermore, for every vector a unique inverse (negative) can be found, for example $(-v_1, -v_2, \ldots, -v_n)$ for $(v_1, v_2, \ldots, v_n)$[2]. The formal definition is given in Definition A.1. This definition is a lot to swallow at first, and is mainly given for reference. The rules described here ensure that vectors behave as we expect them to. Furthermore, vectors can be written as a linear combination of basis-vectors, which means we can break them apart into a sum of products of coefficients and

[2]We will usually represent vectors as column vectors, i.e. vertically and between square brackets. Within sentences, a horizontal representation using parentheses is used instead.

basis-vectors. For example, using the standard basis for $\mathbb{R}^3$:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = a\mathbf{e_1} + b\mathbf{e_2} + c\mathbf{e_3} = a\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + b\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + c\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Definition A.1 — Definition of a Vector Space.
A vector field is a quadruple $(V, F, +, \cdot)$, where V is a set of vectors in our space, F is a field of scalars (coefficients of our vectors), $+ : V \times V \to V$ (vector addition) is an operation to add two vectors, and $\cdot : F \times V \to V$ (scalar multiplication) is an operation to scale a vector by a scalar. Furthermore, the following must hold:

- Vector addition is associative ($\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$) and commutative ($\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$).
- There must exist an additive identity $\mathbf{0}$, such that $\mathbf{v} + \mathbf{0} = \mathbf{v}$.
- Each vector must have an additive inverse: for every vector $\mathbf{v}$ we can find another vector $-\mathbf{v}$ such that $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$.
- Scalar multiplication is compatible with field multiplication, so $a(b\mathbf{v}) = (ab)\mathbf{v}$.
- The multiplicative identity of the scalar field F (generally 1 is also the identity element for scalar multiplication, so $1\mathbf{v} = \mathbf{v}$
- Scalar multiplication distributes over vector addition, so $a(\mathbf{u} + \mathbf{v}) = a\mathbf{u} + a\mathbf{v}$.
- Scalar multiplication distributes over field addition, so $(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$.

Note that while all 3-dimensional vectors ($\mathbb{R}^3$) form a vector space, and all 2-dimensional vectors ($\mathbb{R}^2$) form a vector space as well, that these are not the same spaces. We cannot add or subtract vectors in different spaces, so the following is invalid and does not have an answer:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = ?$$

Vector Operations

The three intuitions presented above each serve their own purpose: the physics intuition allows for easy visualisation, whereas the data science intuition gives us access to the data contained within the vectors allows us to carry out vector operations. The mathemathical intuition is more abstract, and mainly serves to establish a set of rules for managing vectors.

Exercise A.1 — Vector Addition and Scalar Multiplication.

Given the following vectors,

$$\mathbf{u} = \begin{bmatrix} -4 \\ 1 \\ 2 \end{bmatrix} \mathbf{v} = \begin{bmatrix} 0 \\ 9 \\ -6 \end{bmatrix} \mathbf{w} = \begin{bmatrix} 3 \\ -2 \\ -1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \mathbf{y} = \begin{bmatrix} 0 \\ -8 \end{bmatrix}$$

evaluate the following sums and multiplications (or explain why no answer exists):

a. $\mathbf{u}+\mathbf{v}$
b. $\mathbf{u}-\mathbf{w}$
c. $2\mathbf{v}$
d. $3\mathbf{u}-2\mathbf{v}+\mathbf{w}$
e. $\mathbf{x}+\mathbf{y}-\mathbf{y}$
f. $2\mathbf{x}+\mathbf{u}$

Inner Products

As seen above, we can add and subtract vectors (subtraction being defined as adding the inverse of a vector), and multiply any vector by a scalar. One further operation is necessary for the application of vectors in neural networks: The dot product or inner product. Any vector space equipped with an inner product is called an *inner product space*[3]. The inner product is an operation $\langle\ ,\ \rangle : V \times V \to F$, so that $\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u}_1\mathbf{v}_1 + \mathbf{u}_2\mathbf{v}_2 + \cdots + \mathbf{u}_n\mathbf{v}_n$. The inner product of $\mathbf{u}$ and $\mathbf{v}$ is also written as $\mathbf{u} \cdot \mathbf{v}$ and $\langle \mathbf{u}|\mathbf{v} \rangle$ (Dirac notation), the latter of which we will use for the rest of this chapter.

$$\mathbf{u} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$\langle \mathbf{u}|\mathbf{v} \rangle = 1 \times 1 + 2 \times 0 + 3 \times 1 = 4$$

Just as with vector spaces, elements from different inner product spaces cannot be combined using the inner product, meaning one cannot take the inner product between a 2 and 3-dimensional vector. Note, however, that the value of an inner product is a *scalar*, which can be multiplied by any vector using the same type of scalars.

Exercise A.2 — Inner Products.
Given the same vectors as above, compute the following inner products (or explain why no answer exists):

a. $\langle \mathbf{u}|\mathbf{v} \rangle$
b. $\langle \mathbf{v}|\mathbf{u} \rangle$
c. $\langle \mathbf{w}|\mathbf{x} \rangle$
d. $\langle \mathbf{u}|\mathbf{v} \rangle \mathbf{w}$
e. $\langle \langle \mathbf{u}|\mathbf{v} \rangle \mathbf{w}|\mathbf{w} \rangle$
f. $\langle \langle \mathbf{x}|\mathbf{y} \rangle \mathbf{w}|\mathbf{w} \rangle$
g. $\langle \langle \mathbf{x}|\mathbf{y} \rangle \mathbf{x}|\mathbf{w} \rangle$

Anything else?

There are many more operations defined on vectors with applications in areas such as computer graphics, which we ignore in this course.

[3] Not every vector space is an inner product space, although most vector spaces that we will concern ourselves with are. Soon, we encounter an example of a vector space without an inner product.

A.1.2 Matrices

Like vectors, matrices are collections of numbers[4]. Unlike vectors, matrices are two-dimensional: they have rows and columns. The vectors we have seen so far can be seen as special cases of matrices, with either a single row or a single column. Vectors are usually interpreted as $m \times 1$ matrices, or column-vectors. We can also write vectors as a single row, which is called a row-vector. We can switch between the two representations by *transposing* the matrix, which means we flip it around the diagonal running top-left to bottom right:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, M^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \mathbf{v}^T = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$M \in \mathbb{R}^{2\times 3}, M^T \in \mathbb{R}^{3\times 2}, \mathbf{v} \in \mathbb{R}^3, \mathbf{v}^T \in \mathbb{R}^3$$

Note that we use capital letters to denote matrices, and bold letters for vectors. Transposition is indicated by a superscript capital T. For $\mathbf{v}$ (and its transposed counterpart), we can choose whether to interpret it as a matrix or a vector, and match the notation convention for our choice. Here, we went with $\mathbf{v}$ being a vector.

Mathematically, matrices are also vectors, in that the set of m by n matrices ($\mathbb{R}^{m\times n}$) is also a vector space. This means that matrices can be added and scaled just like vectors can. There is, however, no inner product defined, so unless $m = 1$ or $n = 1$, $\mathbb{R}^{m\times n}$ is not an inner product space.

Matrix Products

Two matrices can be multiplied, provided they are of the correct dimensions: the number of columns in the first matrix *must* match the number of rows in the second. The resulting matrix will have the same number of rows as the first, and the same number of columns as the second. This implies matrix multiplication is not symmetrical (or *commutative*, as its formally called): $MN \neq NM$:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix} \neq \begin{bmatrix} ae+cf & be+df \\ ag+ch & bg+dh \end{bmatrix} = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

In this example, both matrices are in $\mathbb{R}^{2\times 2}$, so both MN and NM are defined (though the results are unlikely to be equal). This only happens when the first matrix has the same number of columns as the second, and vice versa. Take a look at some more examples:

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix} = \begin{bmatrix} ag+bj & ah+bk & ai+bl \\ cg+dj & ch+dk & ci+dl \\ eg+fj & eh+fk & ei+fl \end{bmatrix}$$

[4]Here, we are using the data science intuition from before; visually, matrices correspond to transformations of vectors, changing the arrows from our physics-based intuition.

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} g & h \\ i & j \\ k & l \end{bmatrix} = \begin{bmatrix} ag+bi+ck & ah+bj+cl \\ dg+ei+fk & dh+ej+fl \end{bmatrix}$$

So we can multiply a 2×3 matrix by a 3×2 matrix and vice versa. The results are not only different, but even have a different structure. In general, if $M \in \mathbb{R}^{a \times b}$ and $N \in \mathbb{R}^{b \times c}$, then $MN \in \mathbb{R}^{a \times c}$.

As a last example, consider the following matrix multiplication:

$$\begin{bmatrix} a & b \\ c & d \\ e & f \\ g & h \end{bmatrix} \begin{bmatrix} i & j & k \\ l & m & n \end{bmatrix} = \begin{bmatrix} ai+bl & aj+bm & ak+bn \\ ci+dl & cj+dm & ck+dn \\ ei+fl & ej+fm & ek+fn \\ gi+hl & gj+hm & gk \qquad hn \end{bmatrix}$$

Here, there is only one way we can multiply; the other way around would be invalid, just as adding two matrices of different dimensions would.

It might seem random which numbers get multiplied and added together to form each element, but there is a method to this madness: Every element at coordinate (x, y) of the resulting matrix has a value determined by row x of the first matrix and column y of the second. In fact, if $AB = C$, then $C_{x,y}$ is formed by the inner product of row x of A and column y of B:

$$AB = C \quad \Rightarrow \quad C_{x,y} = \langle A_{x,*} | B_{*,y} \rangle$$

Furthermore, the inner product of two vectors is equal to their matrix product, interpreting the first vector as a row and the second as a column.

$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \mathbf{v}^T = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$\langle \mathbf{v} | \mathbf{v} \rangle = \mathbf{v}^T \mathbf{v} = 1 \times 1 + 2 \times 2 + 3 \times 3 = 14$$

Aside A.1 — Outer Product.
You might wonder if multiplying the other way around is also defined. This operation is called the outer product, which instead of resulting in a scalar[a] is a square matrix of the vectors' dimensions. The notation common for this is $\mathbf{u} \otimes \mathbf{v}$ or $|\mathbf{u}\rangle \langle \mathbf{v}|$ in Dirac notation:

$$|\mathbf{v}\rangle \langle \mathbf{v}| = \mathbf{v}\mathbf{v}^T = \begin{bmatrix} 1 \times 1 & 1 \times 2 & 1 \times 3 \\ 2 \times 1 & 2 \times 2 & 2 \times 3 \\ 3 \times 1 & 3 \times 2 & 3 \times 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

[a] Which is also, if you feel pendantic, a 1×1 matrix.

Exercise A.3 — Matrix Multiplication.
Given the following vectors and matrices,

$$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, B = \begin{bmatrix} -3 & 0 \\ 0 & 2 \end{bmatrix}, \mathbf{u} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 8 \\ 2 \end{bmatrix}$$

evaluate the following multiplications:

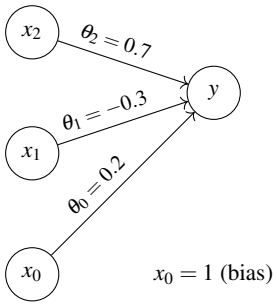
a. $A\mathbf{u}$
b. $B(A\mathbf{u})$
c. $(BA)\mathbf{u}$
d. $\mathbf{u}^T\mathbf{v}$
e. $\mathbf{v}^T\mathbf{u}$
f. $(A\mathbf{v}+B\mathbf{u})\mathbf{v}^T$

A.2 Neural Networks as Matrix-products

We can use the notation and intuitions described above to represent neural networks in a way that can be parallelised. The activation of each layer, whether it is input, output or hidden, is represented by a vector. Weights are represented in matrices, which transform the activation of a layer aßL to the activation of the next $a^{(L+1)}$ via matrix multiplication. Bias can be represented in multiple ways, but either requires an additional vector or an extra column in each weight-matrix. Gradient descent is used to train the network, like in the object-oriented representation described in Section 2.4.2.

Perceptron

We start by examining a simple example, a single perceptron:



Instead of considering each input x_n and each weight θ_n as a separate value, we store them in two vectors $\mathbf{x}$ and θ. Normally, we use the matrix Θ to store the weights between two layers, where each column matches a node in the domain (aßL, the input of the transformation) and each row matches a node in the co-domain ($a^{(L+1)}$, the output of the transformation). In this case, we have a single node in the co-domain, so the weight-matrix consists of a single row: a vector.

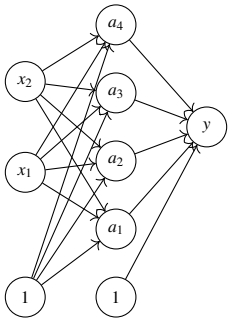
By taking the inner product of the input vector $\mathbf{x}$ and the weight vector θ, we can use a single operation to multiply each input value with its associated weight, and sum the result. So $y = \sigma(\theta_1 x_1 + \theta_2 x_2)$ becomes $y = \sigma\langle\theta|\mathbf{x}\rangle$[5].

[5]Remember, σ represents the *sigmoid activation function*, as introduced in Section 2.3.

Note that the bias is, in this case, added as a column to the weight matrix (or rather, in this simple example, a single value added to the weight matrix). This value is typically added as column 0, which in this vector example means θ_0. Furthermore, each input and activation layer is prepended[6] with a 0-index element, which will alway be 1 (as $1 \times \theta_0 = \theta_0$, the bias). Thus, the expansion of our inner product including the bias is $y = \sigma(\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2)$. In a later example, we will see another way to represent the bias using a separate vector.

Hidden Layer

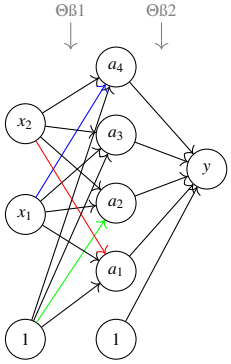
We now consider a more complicated network, by adding a single hidden layer consisting of 4 neurons.



We can calculate a_1 as $\langle \mathbf{x} | \theta ß \mathbf{1} \rangle$, a_2 as $\langle \mathbf{x} | \theta ß \mathbf{2} \rangle$, et cetera. But, as we have seen, we can do an entire series of inner products as a single operation using the matrix product. Multiplying $A\mathbf{b} = c$, each element $\mathbf{c}_y$ in $\mathbf{c}$ is calculated by $\langle \mathbf{A}_{x,*} | b_y \rangle$. We can use this to our advantage, and stack every weight vector $\theta ß \mathbf{n}$ for every neuron a_n into a weight matrix Θ. Multiplying this with our input vector $\mathbf{x}$ results in a new vector $\mathbf{a}$, consisting of all activation values in the succeeding layer.

$$\Theta ß 1 = \begin{bmatrix} - & \boldsymbol{\theta^1} & - \\ - & \boldsymbol{\theta^2} & - \\ - & \boldsymbol{\theta^3} & - \\ - & \boldsymbol{\theta^4} & - \end{bmatrix}, \quad \mathbf{a} = \sigma(\Theta ß 1 \mathbf{x})$$

The entire picture now looks like this:



$$\Theta ß 1 = \begin{bmatrix} 0.1 & -0.2 & -0.3 \\ 0.4 & -0.5 & 0.6 \\ -0.7 & 0.8 & 0.9 \\ -0.1 & 0.2 & 0.3 \end{bmatrix}$$

$$\Theta ß 2 = \begin{bmatrix} 0.1 & 0.1 & 0.2 & 0.3 & 0.5 \end{bmatrix}$$

[6] So, $\begin{bmatrix} x_1 & x_2 \end{bmatrix}$ becomes $\begin{bmatrix} 1 & x_1 & x_2 \end{bmatrix}$. We can write this as $\mathbf{a} \leftarrow \begin{bmatrix} 1 & \mathbf{a} \end{bmatrix}$.

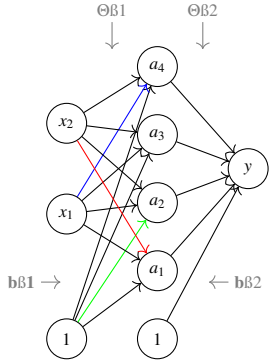
Each value in $\Theta ß1$ matches a weight between the input and hidden layer; $\Theta ß1_{ij}$ represents the weight from x_j to a_i. In this case, rows count from 1, columns count from 0 with column 0 representing the bias. Each value in $\Theta ß2$ matches a weight between the hidden layer and the output. In this case, it could have been represented by a vector because the output value y is a scalar, but to avoid mixing and matching lowercase and uppercase Greek letters we keep every weight a matrix for now. Calculating y from $\mathbf{x}$, $\Theta ß1$, and $\Theta ß2$ looks like this:

$$\mathbf{a} = \sigma(\Theta ß1\mathbf{x}), \quad \mathbf{a}' = \begin{bmatrix} 1 & \mathbf{a} \end{bmatrix}, \quad y = \sigma(\Theta ß2\mathbf{a}')$$

$$y = \sigma(\Theta ß2\begin{bmatrix} 1 & \sigma(\Theta ß1\mathbf{x}) \end{bmatrix})$$

Bias

So far, we have been representing the bias for each layer using an extra weight, which gets multiplied by 1 for each neuron in the layer. An alternative way of representing the bias is as a separate vector, to be added after the matrix multiplication. Using the same sample as above, we can also represent the network as shown below. This time, both rows and columns count from 1 as there is no bias column 0.



$$\Theta ß1 = \begin{bmatrix} -0.2 & -0.3 \\ -0.5 & 0.6 \\ 0.8 & 0.9 \\ 0.2 & 0.3 \end{bmatrix}, \mathbf{bß1} = \begin{bmatrix} 0.1 \\ 0.4 \\ -0.7 \\ -0.1 \end{bmatrix}$$

$$\Theta ß2 = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.5 \end{bmatrix}, \quad \mathbf{b}^{(2)} = \begin{bmatrix} 0.1 \end{bmatrix}$$

$$\mathbf{a} = \sigma(\Theta ß1\mathbf{x} + \mathbf{bß1}), \quad y = \sigma(\Theta ß2\mathbf{a} + \mathbf{bß2})$$

$$y = \sigma(\Theta ß2\sigma(\Theta ß1\mathbf{x} + \mathbf{bß1}) + \mathbf{bß2})$$

Notice that we have written $\mathbf{bß2}$ as a 1-element vector instead of a scalar, just as we write the single row matrix $\Theta ß2$ as a matrix instead of a vector.

Exercise A.4 — Bias Representation.
Confirm the equivalence of both bias representations by calculating the results in both instances.
Bonus: Prove both bias representations are the same.

A.2.1 Going Deeper

Now, consider a more complex example:

- 2 hidden layers
- 1024 input neurons
- 42 in hidden layer 1
- 28 in hidden layer 2
- 12 output neurons

To represent this, we need the following vectors:

- input vector $\mathbf{x} \in \mathbb{R}^{1024}$
- activation vector HL1 $\mathbf{aß1} \in \mathbb{R}^{42}$
- activation vector HL2 $\mathbf{aß2} \in \mathbb{R}^{28}$
- output vector $\mathbf{y} \in \mathbb{R}^{12}$

Which gives us the following weights and equalities:

$$\Theta ß1 \in \mathbb{R}^{42\times 1024+1}, \Theta ß2 \in \mathbb{R}^{28\times 42+1}, \Theta ß3 \in \mathbb{R}^{12\times 28+1}$$

$$\mathbf{aß1} = \sigma(\Theta ß1 \begin{bmatrix} 1 & \mathbf{x} \end{bmatrix})$$

$$\mathbf{aß2} = \sigma(\Theta ß2 \begin{bmatrix} 1 & \mathbf{aß1} \end{bmatrix}) = \sigma(\Theta ß2 \begin{bmatrix} 1 & \sigma(\Theta ß1 \begin{bmatrix} 1 & \mathbf{x} \end{bmatrix}) \end{bmatrix})$$

$$\mathbf{y} = \sigma(\Theta ß3 \begin{bmatrix} 1 & \mathbf{aß2} \end{bmatrix}) = \sigma(\Theta ß3 \begin{bmatrix} 1 & \sigma(\Theta ß2 \begin{bmatrix} 1 & \sigma(\Theta ß1 \begin{bmatrix} 1 & \mathbf{x} \end{bmatrix}) \end{bmatrix}) \end{bmatrix})$$

Or, with bias as a separate vector:

$$\Theta ß1 \in \mathbb{R}^{42\times 1024}, \Theta ß2 \in \mathbb{R}^{28\times 42}, \Theta ß3 \in \mathbb{R}^{12\times 28}$$

$$\mathbf{bß1} \in \mathbb{R}^{42}, \mathbf{bß2} \in \mathbb{R}^{28}, \mathbf{bß3} \in \mathbb{R}^{12}$$

$$\mathbf{aß1} = \sigma(\Theta ß1\mathbf{x} + \mathbf{bß1})$$

$$\mathbf{aß2} = \sigma(\Theta ß2\mathbf{aß1} + \mathbf{bß2}) = \sigma(\Theta ß2\sigma(\Theta ß1\mathbf{x} + \mathbf{bß1}) + \mathbf{bß2})$$

$$\mathbf{y} = \sigma(\Theta ß3\mathbf{aß2} + \mathbf{bß3}) = \sigma(\Theta ß3\sigma(\Theta ß2\sigma(\Theta ß1\mathbf{x} + \mathbf{bß1}) + \mathbf{bß2}) + \mathbf{bß3})$$

Exercise A.5 — Another Deep Neural Network.
Consider a network consisting of the following:

- 3 hidden layers
- 576 input neurons
- 64 neurons in hidden layer 1
- 16 neurons in hidden layer 2
- 16 neurons in hidden layer 3
- 4 output neurons

Which matrices do we need to implement this, and what are their dimensions? How is the output-vector $\mathbf{y}$ defined in terms of $\mathbf{x}$? Write the math out as we have seen above, both for bias-as-column and bias-as-vector representations.

A.2.2 Vectorised Cost Function

Like most forms of supervised machine learning, the cost-function is based on the difference between the output of the network and the expected answer. For each training example, we need to compare two vectors: the predicted answer $\mathbf{y}$ and the correct answer $\mathbf{y}'$. Here, we use the sum of squared errors, $J = (\mathbf{y}_1 - \mathbf{y}'_1)^2 + (\mathbf{y}_2 - \mathbf{y}'_2)^2 + \cdots + (\mathbf{y}_n - \mathbf{y}'_n)^2$. If, for increased clarity, we define an error vector $\mathbf{e} = \mathbf{y} - \mathbf{y}'$, we can write this as $J = \mathbf{e}_1\mathbf{e}_1 + \mathbf{e}_2\mathbf{e}_2 + \cdots + \mathbf{e}_n\mathbf{e}_n$. We have seen this pattern before: what we have here is the inner product of $\mathbf{e}$ and itself. Vectorised, we can thus write our cost function as $J = \langle \mathbf{e}|\mathbf{e}\rangle = \mathbf{e}^2$. Substituting our original difference between $\mathbf{y}$ and $\mathbf{y}'$ back for $\mathbf{e}$ yields $J = \langle \mathbf{y} - \mathbf{y}'|\mathbf{y} - \mathbf{y}'\rangle \quad \text{or} \quad J = (\mathbf{y} - \mathbf{y}')^2$.

A.2.3 Further Parallelisation

Up until now, we have represented each layer in our neural network as a vector, and used matrices to map between each. Each vector $\mathbf{x}$ represents a single input vector, associated with a single output vector $\mathbf{y}$. We can evaluate multiple examples in parallel by replacing our vectors by matrices. As an intermediate step, consider our deep example from Section A.2.1, but this time, interpret each n-vector as if it where a $1 \times n$ matrix (for the sake of saving trees, we will only consider the bias-as-vector representation, but the same holds for bias-as-column).

$$X \in \mathbb{R}^{1024\times 1}, A\text{ß}1 \in \mathbb{R}^{42\times 1}, A\text{ß}2 \in \mathbb{R}^{28\times 1}, Y \in \mathbb{R}^{12\times 1}$$

$$\Theta\text{ß}1 \in \mathbb{R}^{42\times 1024}, \Theta\text{ß}2 \in \mathbb{R}^{28\times 42}, \Theta\text{ß}3 \in \mathbb{R}^{12\times 28}$$
$$B\text{ß}1 \in \mathbb{R}^{42\times 1}, B\text{ß}2 \in \mathbb{R}^{28\times 1}, B\text{ß}3 \in \mathbb{R}^{12\times 1}$$

$$A\text{ß}1 = \sigma(\Theta\text{ß}1X + B\text{ß}1)$$
$$A\text{ß}2 = \sigma(\Theta\text{ß}2A\text{ß}1 + B\text{ß}2)$$
$$Y = \sigma(\Theta\text{ß}3A\text{ß}2 + B\text{ß}3)$$

Exercise A.6 — Vectors to Matrices.
Verify that the example given above is equivalent to the vector-based example from Section A.2.1.

We can add as many columns as we need to accomodate the number of training examples that we want to process in parallel. In general, to evaluate n examples at a time, we need the following dimensions for our Θ and B matrices:

$$X \in \mathbb{R}^{1024\times n}, A\text{ß}1 \in \mathbb{R}^{42\times n}, A\text{ß}2 \in \mathbb{R}^{28\times n}, Y \in \mathbb{R}^{12\times n}$$
$$B\text{ß}1 \in \mathbb{R}^{42\times n}, B\text{ß}2 \in \mathbb{R}^{28\times n}, B\text{ß}3 \in \mathbb{R}^{12\times n}$$

A.3 Learning

Now that we know how to represent our network, we can look towards learning. Like in the object-oriented representation from Section 2.4.2, we can use Gradient Descent. We can use multivariable calculus to find the gradient ∇J of our cost function, and move our weights and biasses accordingly. Tweaking this for a lot of variables quickly gets complicated, which is why we use the functions available to us in NumPy to avoid having to do this ourselves, which is generally error-prone. To get a feel of what is happening, take a look at the videos made by 3Blue1Brown[7] animating this process.

Exercise A.7 — Gradient Descent.
Watch the following videos on YouTube, and explain in your own words what the algorithm does.

- But what *is* a Neural network?
- Gradient descent, how neural networks learn (optional)
- What is backpropagation and what is it actually doing?
- Backpropagation calculus (optional)

A.4 Implementation using NumPy

We now turn our attention towards implementing what we have learned using NumPy. NumPy is a package which provides a large array of fundamental mathemathical functions and structures to Python. Even more bare-metal approaches exist for maximising GPU perfomance, such as OpenCL[8] and CUDA[9]. In this course, we focus on NumPy, which provides a nice middle ground between readability and performance.

A.4.1 Vectors

Vectors are represented as numpy arrays. A numpy array can be generated from a regular array using `numpy.array(regular_array)`, e.g. `numpy.array([1,2,3])`. The `numpy.array` function is idempotent: calling it on an existing numpy array results an unchanged array. This is useful when writing functions to ensure a parameter is a numpy array, as it allows one to convert the paramater regardless of whether it was already a Numpy array.

Numpy arrays support vector addition and scalar multiplication using overloaded versions of the regular $+$ and $*$ operators. For the inner product, the `dot` function is used: `numpy.dot(u,v)` can be used to calculate $\langle \mathbf{u}|\mathbf{v}\rangle$.

Caveats

Note that $\langle \mathbf{u}|\mathbf{v}\rangle$ is an entirely different operation from Python's `u*v`, which calculates the Hadamard product (see Aside A.2), which is different from the inner product. Furthermore, it is possible to add a scalar to a vector, which results in the scalar being

[7] A playlist with the neural network series by 3Blue1Brown is available at `https://youtu.be/aircAruvnKk`.

[8] `https://www.khronos.org/opencl`

[9] `https://developer.nvidia.com/cuda-zone`

broadcasted into a vector:

$$\begin{bmatrix}1\\2\\3\end{bmatrix} + 1 = \begin{bmatrix}1\\2\\3\end{bmatrix} + \begin{bmatrix}1\\1\\1\end{bmatrix} = \begin{bmatrix}2\\3\\4\end{bmatrix}$$

Broadcasting will be explained in more detail in Section A.4.3, after we have familiarised ourselves with numpy's notation for matrices.

Aside A.2 — The Hadamard Product.
The Hadamard product $\odot : V \times V \to V$ of two vectors, also referred to as pointwise multiplication, is defined as follows:

$$\mathbf{u} \odot \mathbf{v} = \begin{bmatrix}\mathbf{u}_1\mathbf{v}_1\\\mathbf{u}_2\mathbf{v}_2\\\vdots\\\mathbf{u}_n\mathbf{v}_n\end{bmatrix}$$

Notice that this means that the Hadamard product of two vectors in space $\mathbb{R}^n$ will always be another vector in $\mathbb{R}^n$. This matches the type Python expects of the $*$ function, which is likely why this product was chosen despite being a less common operation than the inner product. This behaviour is also consistent with the broadcasting described below.

A.4.2 Matrices

Matrices can be generated in numpy using the same `numpy.array` function as vectors, only using nested arrays: `numpy.array([[1,2,3],[2,3,4]])` represents the 2×3 matrix below:

$$\begin{bmatrix}1 & 2 & 3\\2 & 3 & 4\end{bmatrix}$$

Matrices addition and scalar multiplication work the same way as for vectors, as does the Hadamard product, broadcasting if necessary. The same `dot` function used for the inner product is also used for matrix multiplication.

A.4.3 Broadcasting

Broadcasting can occur whenever Python expects two operands to be of equal dimension. If this expectation is met, everything will proceed as normal. Otherwise, Python will attempt to correct the situation by expanding the smaller operand. This can only occur when one dimension of the operand is equal to 1, in which case Python repeats what it knows in order to correct the size of the operand. Thus, it is possible to add a scalar to a vector, or add a vector to a matrix:

$$\begin{bmatrix}1\\2\\3\end{bmatrix} + \begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix} = \begin{bmatrix}2 & 2 & 2\\3 & 3 & 3\\4 & 4 & 4\end{bmatrix}$$

When making use of broadcasting, keep in mind that a numpy array by default is interpreted as a row vector. It is possible to explicitly create row and column vectors, as shown in the following examples. This is useful for controlling broadcasting, or when for example an outer product is desired. Compare:

```
   numpy.array([1,2,3]) # Vector
 + numpy.array([[1,1,1],[1,1,1],[1,1,1]])
#= numpy.array([[2,3,4],[2,3,4],[2,3,4]])

   numpy.array([[1,2,3]]) # Explicit Row Vector
 + numpy.array([[1,1,1],[1,1,1],[1,1,1]])
#= numpy.array([[2,3,4],[2,3,4],[2,3,4]])

   numpy.array([[[1],[2],[3]]]) # Explicit Column Vector
 + numpy.array([[1,1,1],[1,1,1],[1,1,1]])
#= numpy.array([[2,2,2],[3,3,3],[4,4,4]])
```

Notice that only the third example corresponds to the addition using a column vector shown above.

A.4.4 Example: Perceptron

We can now return to our first example (Section A.2): the simple perceptron. We encode $\mathbf{x}$ and θ as numpy `arrays`, and take their inner product to determine y. Using 0.2 and 0.3 for the input values of $\mathbf{x}$, we get the following:

```
def sigmoid(x): # Also available from SciKit
    return 1 / (1 + math.e ** (-x))

x = numpy.array([1,0.2,0.3]) # Bias and two input values
theta = numpy.array([0.2,-0.3,0.7])
y = sigmoid(numpy.dot(theta, x)) # 0.5866175789173301
# alternative:
# y = sigmoid(theta.dot(x))
```

A.4.5 Hidden Layers

Moving on to our more complex example featuring a hidden layer, we can expand upon this. Remember that we have seen two ways to encode the bias: bias-as-column and bias-as-vector. This difference must be taken into account when translating our example into code.

Thanks to broadcasting, our sigmoid function will work on numpy arrays without change. For more complex functions (e.g. the *tanh* or tan^{-1} activation function), it is possible to create a vectorised version of said function by using Numpy's `vectorise` function. This creates a new function that will map its input function over an array.

```
from math import tanh

vectorised_activation = numpy.vectorise(tanh)
```

Bias-as-column

For this representation, we need to introduce one final bit of notation: prepending the bias to an activation layer: $\mathbf{a}' = \begin{bmatrix} 1 & \mathbf{a} \end{bmatrix}$. In Numpy, we can use the append function for this: `numpy.append(1, a)`. The resulting code looks as follows:

```
x = numpy.array([1,0.2,0.3]) # Bias and two input values
Theta1 = numpy.array([[0.1,-0.2,-0.3]
                      ,[0.4,-0.5,0.6]
                      ,[-0.7,0.8,0.9]
                      ,[-0.1,0.2,0.3]])
a = vectorised_sigmoid(numpy.dot(Theta1, x))
# a = [ 0.49250056,  0.61774787,  0.4329071 ,  0.50749944]
a_prime = numpy.append(1, a) # Add bias
Theta2 = numpy.array([[0.1,0.1,0.2,0.3,0.5]])
y = vectorised_sigmoid(numpy.dot(Theta2, a_prime))
# y = 0.65845607
```

Bias-as-vector

In this representation, we do not need to append anything; rather, we need some additional values:

```
x = numpy.array([0.2,0.3]) # Just two input values
Theta1 = numpy.array([[-0.2,-0.3]
                      ,[-0.5,0.6]
                      ,[0.8,0.9]
                      ,[0.2,0.3]])
b1 = numpy.array([[0.1]
                  ,[0.4]
                  ,[-0.7]
                  ,[-0.1]])
a = vectorised_sigmoid(numpy.dot(Theta1, x) + b1)
# a = [ 0.49250056,  0.61774787,  0.4329071 ,  0.50749944]
Theta2 = numpy.array([[0.1,0.2,0.3,0.5]])
b2 = 0.1
y = vectorised_sigmoid(numpy.dot(Theta2, a) + b2)
# y = 0.65845607
```

A.5 Exercises

Exercise A.8 — NOR-Gate.
As a warm-up exercise, we implement the NOR-gate from Chapter 2 using vector-representation.

A) Structure

Implement the NOR-Gate using NumPy's matrices and vectors. Use NumPy to generate a random initial vector and create a truth table of the network output.

B) Feed Forward Function

Generalise your NOR-Gate to a function `predict(x, Theta)` that, given an input vector x and a list of weight matrices $[\Theta_1, \Theta_2, \ldots, \Theta_n]$, predicts the associated y value.

C) Training

Using the code from github[a], train the network to correctly emulate the NOR-Gate. In order to use the backpropagation algorithm provided, you need to adapt your `predict` function from above to a step-wise `forward`-function (see the description and function profile in the `backprop.py`-file on the provided git repository).

[a] `https://github.com/aldewereld/nl.hu.ict.a2i.cnn`; only use backprop.py, the other files are used in Appendix B!

Exercise A.9 — MNIST.

The MNIST dataset is the "Hello World" of classification problems. The dataset consists of a large number (42000) labeled examples of handwritten digits. Each digit is represented as a 784-dimensional row, corresponding to the brightness values of 28×28 pixels.

A) Getting the Data

Download the MNIST dataset using the following code. Acquaint yourself with the dataset by using the function `view_image(int)` provided.

```
import pickle , gzip , os
from urllib import request
from pylab import imshow , show , cm


url = "http://deeplearning.net/data/mnist/mnist.pkl.gz"
if not os.path.isfile("mnist.pkl.gz"):
    request.urlretrieve(url, "mnist.pkl.gz")

f = gzip.open('mnist.pkl.gz', 'rb')
train_set, valid_set, test_set =
    pickle.load(f, encoding='latin1')
f.close()

def get_image(number):
    (X, y) = [img[number] for img in train_set]
    return (np.array(X), y)

def view_image(number):
    (X, y) = get_image(number)
    print("Label: %s" % y)
```

```
imshow(X.reshape(28,28), cmap=cm.gray)
show()
```

B) Training on MNIST

Design a neural network to classify digits from the MNIST dataset. Start by reasoning about the number and size of hidden layers, and document your considerations. Implement the network using Keras[a] and TensorFlow[b]. Train the network on the training portion of the dataset.

C) Evaluation

Evaluate the performance of your network using the test portion of the dataset. Explain your findings and offer some ideas on how performance could be improved.

D) Lather, Rinse, Repeat (optional)

Design a new neural network using the suggestions described in C and compare the performance. Were your ideas able to improve your network? Why (not)?

[a] See `https://keras.io` for details on Keras. Keras is a high-level front-end to TensorFlow.

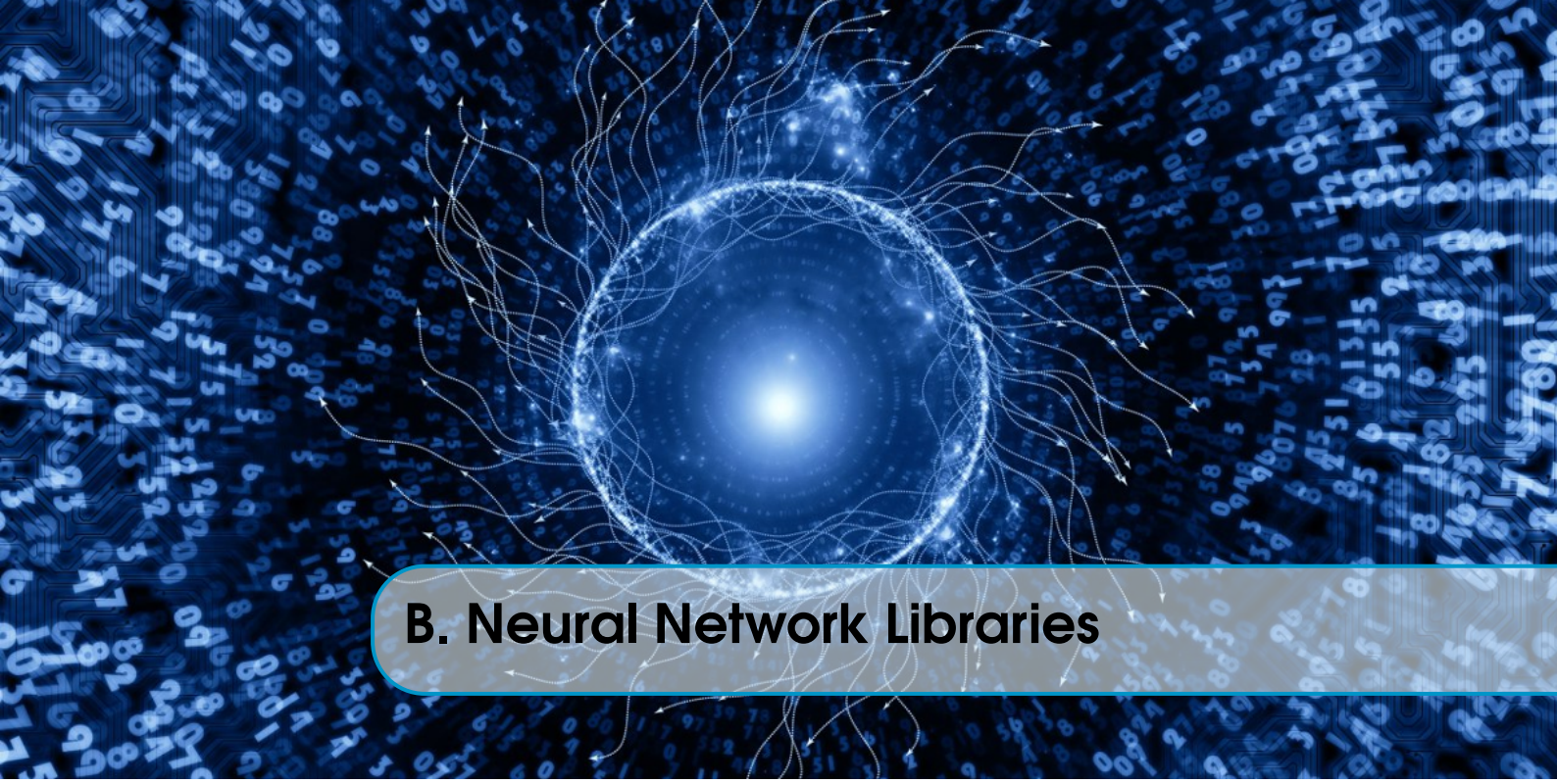
[b] See `https://www.tensorflow.org` for details on TensorFlow. You are allowed to use the Convolutional implementation of appendix B as a reference, but are required to implement a "normal" neural network instead.

A.6 Appendix: Notation Overview

- y is a number (scalar)
- $\mathbf{b}$ is a vector
- $\mathbf{b}_1$ is a number in b (a scalar)
- $\mathbf{bß2}$ is a different vector than $\mathbf{bß1}$ and $\mathbf{bß3}$; this notation will be used to mark different layers. The same goes for matrices.
- A is a matrix
- $A_{x,y}$ or A_{xy} is a number in A (row x, column y)
- $A_{x,*}$ is the vector created from row x of A
- θ and Θ are used for weights in our neural network. Pronounced as “theta”.
- $\langle \mathbf{a}|\mathbf{b} \rangle$ is the inner product of $\mathbf{a}$ and $\mathbf{b}$
- $c\mathbf{v}$ and cA are scalar products
- $\mathbf{v}A$, $A\mathbf{v}$ and AB are matrix multiplications

A.6.1 Asides (for completeness):

- $|\mathbf{a}\rangle \langle \mathbf{b}|$ is the outer product of $\mathbf{a}$ and $\mathbf{b}$
- $\mathbf{a} \odot \mathbf{b}$ is the Hadamard product of vectors $\mathbf{a}$ and $\mathbf{b}$
- $A \odot B$ is the Hadamard product of matrices A and B



B. Neural Network Libraries

In this appendix we introduce some of the frequently used libraries for using neural networks. There are, however, many different (good) libraries around, each of them with their own perks and tricks. It would be too much for this appendix to give coverage of all libraries. Neither are the following sections meant to be a complete tutorial for a given library; we introduce some common concepts and show how to build a 'simple' network. For complete coverage of a given library, we refer to the many tutorials available on the web. Given the basic knowledge about neural networks that is presented in this reader, most concepts used in libraries should be easy to understand.

B.1 TensorFlow[1]

TensorFlow is an open-source software library developed by Google Brain for machine learning. It is a symbolic math library, stressing the mathematical properties of neural networks as presented in Chapter A.

A tensor is a mathematical geometrical object that describes linear relations between geometric vectors, scalars and other tensors. Elementary examples of such relations include dot product, cross product and linear maps. As seen in Chapter A, these lie at the basis of neural networks, and combinations of these (combined into a flow) creates the same properties as expressed by neural networks as described earlier.

So much for explaining the reason why Google decided to call its open-source neural network library *TensorFlow*. In the following we show how to use TensorFlow to classify digits in the MNIST dataset, thus explaining several features of TensorFlow, including building Convolutional Neural Networks (see Appendix 3).

[1] For a complete overview of TensorFlow tutorials, start at `https://www.tensorflow.org/tutorials/`.

For our example MNIST classifier, we build a convolutional neural network with the following architecure:

1. **Convolutional layer #1:** applies 32 5×5 filters (extracting 5×5-pixel subregions), with ReLU activation function;
2. **Pooling layer #1:** performs max pooling with a 2×2 filter and stride of 2;
3. **Convolutional layer #2:** applies 64 5×5 filters, with ReLU activation function;
4. **Pooling layer #2:** again, performs max pooling with a 2×2 filter and stride of 2;
5. **Dense layer #1:** 1,024 neurons, with dropout regularisation rate of 0.4[2] (probability of 0.4 that any given element will be dropped during training);
6. **Dense layer #2 (Logits layer):** 10 neurons, one for each digit target class (0-9)[3].

The `tf.layers` module contains methods to create each of the three layer types above:

- `conv2d()` constructs a two-dimensional convolution layer. Takes number of filters, filter kernel size, padding, and activation function as arguments.
- `max_pooling2d()` constructs a two-dimensional pooling layer using the max-pool algorithm. Takes pooling filter size and stride as arguments.
- `dense()` constructs a dense layer. Takes number of neurons and activation function as arguments.

Each of these methods accepts a tensor as input and returns a transformed tensor as output. This makes it easy to connect one layer to another: just take the output from one layer-creation method and supply it as input to another[4].

Input layer

The methods in the `layers` module for creating convolutional and pooling layers for two-dimensional image data expect input tensors to have a shape of `[batch_size, image_width, image_height, channels]`, defined as follows:

- `batch_size` defines the size of the subset of examples to use when performing gradient descent during training.
- `image_width` defines the width of the example images.
- `image_height` defines the height of the example images.
- `channels` defines the number of colour channels in the example images (the depth). For colour image the number of channels is typically 3 (red, green, blue). For monochrome images, there is just 1 channel (black).

The MNIST dataset is composed of monochrome 28×28 pixel images, so the desired shape for our input layer is given by the following method that converts our input feature map (`features`) to this shape:

[2] *Dropout* is a technique to keep neural networks from overfitting. In essence, the network will occasionally (determined by the probability set for the dropout) disregard the outcome of particular neurons (including their weights to and from the neuron) in the calculation of the forward and backward pass. Simply set, it disables particular neurons randomly while training to boost training of the others.

[3] A *logit* is a sigmoid function (see Chapter 2); in TensorFlow, the logits layer indicates that this Tensor is the quantity that is being mapped to (so to say, the output Tensor).

[4] Full code of the example is available on `https://github.com/aldewereld/nl.hu.ict.a2i.cnn/blob/master/tf_mnist.py`.

```
input_layer = tf.reshape(features["x"], [-1, 28, 28,1])
```

Here, we have indicated a `batch_size` of -1, which specifies that this dimension should be dynamically computed based on the number of input values (in `features["x"]`, holding he size of all other dimensions constant. This allows us to vary the `batch_size` when feeding in the examples into the model. For example, if we feed the model batches of 5, `features["x"]` will contain 3,920 ($5 \times 28 \times 28 \times 1$) values, and `input_layer` will have a shape of `[5, 28, 28, 1]`.

Convolutional layer #1

In the first convolutional layer, we apply 32 5×5 filters to the input layer, with a ReLU activation function. We can use the `conv2d()` method in the `layers` module to create this layer as follows:

```
conv1 = tf.layers.conv2d(
  inputs = input_layer,
  filters=32,
  kernel_size=5,
  padding="same",
  activation=tf.nn.relu
)
```

The **`input`** argument specifies our input tensor, which must have the shape `[batch_size, image_width, image_height, channels]`. Here we connected our first convolutional layer to the `input_layer`, as defined above.

The **`filters`** argument specifies the number of filters to apply (here, 32), and **`kernel_size`** specifies the dimensions (using a single integer, here 5, which assumes the filter is square, otherwise **`kernel_size`**`=[5, 5]` can be used as well).

The **`padding`** argument specifies one of two enumerated values: `valid` (default value)[5] or `same`. To specify that the output tensor should have the same width and height as the input tensor, we set **`padding`**`=same` here, which instructs TensorFlow to add 0 values to the edges of the input tensor to preserve width and height of 28. (Without padding, a 5×5 convolution over a 28×28 tensor will produce a 24×24 tensor, as there are 24 valid positions (horizontally and vertically) to uniquely place a 5×5 filter with stride 1).

The **`activation`** argument specifies the activation function to apply to the output of the convolution. Here, we specify ReLU activation with `tf.nn.relu`.

The output tensor produced by `conv2d` with these settings has a shape of `[batch_size, 28, 28, 32]`: the same width and height as the input, but now with 32 channels holding the output of each of the filters.

Pooling layer #1

Next, we connect our first pooling layer to the convolutional layer we just created. We can use the `max_pooling2d()` method in `layers` to construct a layer that performs max pooling with a 2×2 filter and stride of 2:

[5] While not often used, 'valid' padding in TensorFlow only considers the valid positions of a filter on an input volume, and might drop columns when it does not fit. Using 'valid' padding will result in a smaller spatial volume (width×height) than the input volume.

```
pool1 = tf.layers.max_pooling2d(
  inputs=conv1,
  pool_size=2,
  strides=2
)
```

Again, **inputs** specifies the input tensor, with a shape of `[batch_size, image_width, image_height, channels]`. Here, our input tensor is `conv1`, the output of the first convolutional layer.

The **pool_size** argument specifies the size of the max pooling filter, again either as an integer in the case of a square filter or as an array `[2, 2]`.

The **strides** argument specifies the size of the stride. Here, we set a stride of 2, which means that the subregions extracted by the filter do not overlap. If you want to set different stride values for width and height, you can instead specify a tuple or a list (e.g., **strides**=`[3,6]`).

The output tensor produced by `max_pool2d()` has the size of `[batch_size, 14, 14, 32]`: the pooling filter halves the width and height, but leaves the depth unchanged.

Conv #2 and Pool #2

We can connect a second convolutional and pooling layer to our cnn using the methods described above. For the second conv layer we configure 64 5×5 filters with ReLU activation, and for our second pool layer, we use the same specs as for our first one:

```
conv2 = tf.layers.conv2d(
  inputs=pool1,
  filters=64,
  kernel_size=5,
  padding="same",
  activation=tf.nn.relu
)

pool2 = tf.layers.max_pooling2d(
  inputs=conv2,
  pool_size=2,
  strides=2
)
```

Dense layer

Next we add a dense layer (with 1,024 neurons and ReLU activation) to our cnn to perform classification on the features we extracted by the convolutional/pooling layers. Before we can connect the layer, however, we have to flatten the feature map (`pool2`) to shape `[batch_size, features]`, so that our tensor has only two dimensions:

```
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
```

In the `reshape()` operation above, the -1 again signifies that the `batch_size` dimensions will be dynamically calculated based on the number of examples in our input

data. Each example has $7 * 7 * 64$ features, so we want the `features` dimension to have a value of 3136 in total.

We can now use the `dense()` method in `layers` to connect our dense layer as follows:

```
dense = tf.layers.dense(
  inputs=pool2_flat,
  units=1024,
  activation=tf.nn.relu
)
```

The **`inputs`** argument specifies the input tensor. The `units` argument specifies the number of neurons in the dense layer (1,024). The **`activation`** argument takes the activation function; again, we use `tf.nn.relu` to add ReLU activation.

To help improve the results of our model, we also apply dropout regularisation to our dense layer, using the `dropout` method in `layers`:

```
dropout = tf.layers.dropout(
  inputs=dense,
  rate=0.4,
  training=mode == tf.estimator.ModeKeys.TRAIN
)
```

Again, **`inputs`** specifies the input tensor. The `rate` argument specifies the dropout rate; here we use a 40% random dropout during training. The `training` argument takes a boolean specifying whether or not the model is currently being run in training mode; dropout is only performed when `training` is `True`. Here, we check if the `mode` passed to our model function is `TRAIN` mode.

Logits layer

The final layer in our neural network is the logits layer, which returns the raw values for our predictions. We create a dense layer with 10 neurons (one for each target class 0-9), with linear activation (the default):

```
logits = tf.layers.dense(inputs=dropout, units=10)
```

Our final output tensor of the cnn, `logits`, has the shape `[batch_size, 10]`.

Generating predictions

The logits layer of the model returns predictions as raw values in a 2-dimensional tensor. Let's convert these raw values into two different formats that our model can return:

- the **predicted class** for each example: a digit from 0-9;
- the **probabilities** for each possible target class for each example.

For a given example, our predicted class is the element in the corresponding row of the logits tensor with the highest raw value. We can find the index of this element using the `tf.argmax` function:

```
tf.argmax(input=logits, axis=1)
```

The `input` argument specifies the tensor from which to extract maximum values – here `logits`. The `axis` argument specifies the axis of the `input` tensor along which to find the greatest value. Here, we want to find the largest value along the dimension with index of 1, which corresponds to our predictions (recall that our logits tensor has the shape `[batch_size, 10]`).

We can derive probabilities from our logits layer by applying softmax activation using `tf.nn.softmax`:

```
tf.nn.softmax(logits, name="softmax_tensor")
```

We compile our predictions in a dict, and return an `EstimatorSpec` objecT:

```
predictions = {
  "classes": tf.argmax(input=logits, axis=1),
  "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
}
if mode == tf.estimator.ModeKeys.PREDICT:
  return tf.estimator.EstimatorSpec(mode=mode,
          predictions=predictions)
```

Calculating loss

For both training and evaluating, we need to define a loss function that measures how closely the model's predictions match the target classes. For multiclass problems like MNIST, cross entropy is typically used as the loss metric. The following code calculates cross entropy when the model runs in either `TRAIN` or `EVAL` mode:

```
onehot_labels =
     tf.one_hot(indices=tf.cast(labels, tf.int32), depth=10)
loss = tf.losses.softmax_cross_entropy(
  onehot_labels=onehot_labels, logits=logits)
```

Our `labels` tensor contains a list of predictions for our examples, e.g. `[1, 9, ...]`. In order to calculate cross-entropy, we first need to convert `labels` to the corresponding one-hot encoding[6]:

```
[[0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
 ...]
```

Training

Now we have defined loss for our cnn, let's configure our model to optimise this loss value during training. We use a learning rate of 0.001 and stochastic gradient descent (see Section 2.4.2) as the optimisation algorithm:

```
if mode == tf.estimator.ModeKeys.TRAIN:
  optimizer =
```

[6]One hot encoding transforms categorical features to a format that works better with classification and regression algorithms. The label (categorical) is transformed to probabilities for each different classes. Only one of the columns takes a value of 1 for each sample, hence the name *one hot encoding*.

```
    tf.train.GradientDescentOptimizer(learning_rate=0.001)
  train_op = optimizer.minimize(
    loss=loss,
    global_step=tf.train.get_global_step())
  return
    tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)
```

Add evaluation metrics

To add an accuracy metric in our model, we define `eval_metric_ops` dict in EVAL mode:

```
eval_metric_ops = {
  "accuracy": tf.metrics.accuracy(
    labels=labels, predictions=predictions["classes"])}
return tf.estimator.EstimatorSpec(
  mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

Training and evaluating the CNN MNIST classifier

The creation of the network, as described above, is done in a user-defined function (let's name it `cnn_model_fn`), which is used to create an `Estimator`[7]. Before we do that, we have to load the training and evaluation examples. To run the model later, TensorFlow requires you to specify a `main()` function:

```
def main(unused_argv):
  # Load training and eval data
  mnist = tf.contrib.learn.datasets.load_dataset("mnist")
  train_data = mnist.train.images # Returns np.array
  train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
  eval_data = mnist.test.images # Returns np.array
  eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)
```

Next we can create the `Estimator`:

```
# Create the Estimator
mnist_classifier = tf.estimator.Estimator(
    model_fn=cnn_model_fn,
    model_dir="/tmp/mnist_convnet_model")
```

The `model_dir` argument specifies where the checkpoints (model data) will be stored. Change this to whatever suits you. The `model_fn` argument specifies the function that is used for training, evaluation, and prediction (which is what we built above).

We are now ready to train the model[8], which we can do by creating `train_input_fn` and calling `train` on `mnist_classifier`:

[7] `Estimator` is a TensorFlow class for performing high-level model training, evaluation, and inference.

[8] The full code on `https://github.com/aldewereld/nl.hu.ict.a2i.cnn/blob/master/tf_mnist.py` also includes logging options to show what has happened after a number of iterations. Please inspect the full code for this additional functionality.

```
# Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_data},
    y=train_labels,
    batch_size=100,
    num_epochs=None,
    shuffle=True)
mnist_classifier.train(
    input_fn=train_input_fn,
    steps=20000)
```

The `numpy_input_fn` transforms our training data into a training input function that is used during the training of the network. Note that we specify a `batch_size` of 100 here (which means that the model is trained on minibatches of 100 examples at each step). `num_epochs=None` indicates that the networks trains until the specified number of steps is reached. `shuffle=True` indicates that we shuffle the training data (to decrease the chance of overfitting).

Once training is complete, we want to evaluate the model to determine its accuracy on the MNIST test set. We call the `evaluate` method, which evaluates the metrics we specified in `eval_metric_ops` argument in the `model_fn`.

```
# Evaluate the model and print results
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": eval_data},
    y=eval_labels,
    num_epochs=1,
    shuffle=False)
eval_results =
  mnist_classifier.evaluate(input_fn=eval_input_fn)
print(eval_results)
```

Note the similarity with the code for training. However, we now set the `num_epochs=1`, so that the model evaluates the metrics over one epoch of data and returns the result. There is now also no need to shuffle the data.

Performance and GPU usage

The typical installation of TensorFlow uses the CPU for training the neural networks. In our example above, run on a Intel i5 3.2GHz processor with 12GB of RAM, it takes around 3 hours to train the network (batch_size 100, 20,000 iterations), which resulted in an accuracy of 96.9%.

TensorFlow can also be sped up by using GPU(s) for training. For now it only supports NVidia CUDA cores, but the speed increase can be rather significant. See `https://www.tensorflow.org/tutorials/using_gpu` for more instructions on how to install TensorFlow with GPU capabilities.

B.2 Lasagne[9]

Lasagne is a lightweight library to build and train neural networks in Theano. Theano, on the other hand, is a Python library that allows you to define, optimise, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. One of the advantages of Theano is the transparent use of the GPU. Theano is often seen as the precursor to TensorFlow, and was developed by the University of Montreal.

The first thing you will note when creating a Lasagne application is that while Lasagne is built on top of Theano, it is meant as a supplement helping with some tasks, not as a replacement. Therefore, you will always mix Lasagne with some vanilla Theano code.

Here we built again a MNIST digit classifier, using the same architecture as described above. First we start with the necessary imports.

```
import numpy as np
import theano
import theano.tensor as T

import lasagne
```

Building the model

Lasagne allows you to define an arbitrarily structured neural network by creating and stacking or merging layers. Since every layer knows its immediate incoming layers, the output layer (or output layers) of a network double as a handle to the network as a whole, so usually this is the only thing we pass to the rest of the code.

The function for building the a Convolutional Neural Network (cnn) is named `build_cnn()`, which creates two conv layers and pooling stages, a fully-connected hidden layer and a fully-connected output layer (same as we did above). The function starts by creating the input layer:

```
def build_cnn(input_var=None):
  network = lasagne.layers.InputLayer(shape=(None, 1, 28, 28),
                                      input_var=input_var)
```

The four numbers in the shape tuple represent, in order: `(batchsize, channels, rows, columns)`. Here, we have set the batchsize to `None`, which means the network will accept input data of arbitrary batchsize after compilation. If you know the batchsize beforehand and do not need this flexibility, you should set the batchsize here, as it allows Theano to apply some optimisations.

Next we add the `Conv2DLayer` on top with 32 filters of size 5×5:

```
network = lasagne.layers.Conv2DLayer(
    network, num_filters=32, filter_size=5,
    nonlinearity=lasagne.nonlinearities.rectify,
    stride=1, pad="same",
```

[9] Tutorials for Lasagne and Theano are available, respectively, here: `http://lasagne.readthedocs.io/en/latest/user/tutorial.html` and here `http://deeplearning.net/software/theano/tutorial/`.

```
    W=lasagne.init.GlorotUniform())
```

`nonlinearity` takes a nonlinearity function, several of which are defined in `lasagne.nonlinearities`. Here we have chosen the linear rectifier, so we obtain ReLUs. Finally, `lasagne.init.GlorotUniform()` gives the initialiser for the weight matrix `W`. This particular initialiser samples weights from a uniform distribution of a carefully chosen range. Other initialisers are available in `lasagne.init`, and alternatively, `W` could also have been initialised from a Theano shared variable or numpy array of the correct shape (as `GlorotUniform()` is the default, we omit it from now on). `stride` defines the stride, either as an integer or as a 2-element tuple (when you want a different stride horizontally and vertically). `pad` denotes the padding performed to the input volume. It accepts either an integer (for custom defined padding), a tuple (allows different padding per dimension), `"full"` (pads with one less than the filter size on both sides), `"same"` (ensures output volume has equal width and height as input volume for stride 1), or `"valid"` (an alias for 0; no padding).

Next we apply max-pooling of factor 2 in both dimensions, using a `MaxPool2DLayer` instance:

```
network = lasagne.layers.MaxPool2DLayer(
    network, pool_size=2, stride=2)
```

Again, **`pool_size`** can be specified as integer, like here, or as an tuple (e.g. `(2, 2)`). We specify a stride of 2 to ensure the regions of the max-pool do not overlap.

We add another convolution and pooling layer like the ones above:

```
network = lasagne.layers.Conv2DLayer(
    network, num_filters=64, filter_size=5,
    nonlinearity=lasagne.nonlinearity.rectify,
    stride=1, pad="same")
network = lasagne.layers.MaxPool2DLayer(
    network, pool_size=2, stride=2)
```

Then a fully-connected layer of 1,024 units with 40% dropout on its inputs (using `lasagne.layers.dropout` shortcut inline):

```
network = lasagne.layers.DenseLayer(
    lasagne.layers.dropout(network, p=.4),
    num_units=1024,
    nonlinearity=lasagne.nonlinearity.rectify)
```

And finally, a 10-unit softmax output layer:

```
network = lasagne.layers.DenseLayer(
    network,
    num_units=10,
    nonlinearity=lasagne.nonlinearities.softmax)

return network
```

Loading the data

Last we define some code to load the MNIST dataset and return it in the form of regular numpy arrays. There is no Lasagne involved at all, so for the purpose of this tutorial, we regard it as[10]:

```
def load_dataset():
  ...
  return X_train, y_train, X_val, y_val, X_test, y_test
```

`X_train.shape` is `(50000, 1, 28, 28)`, to be interpreted as: 50,000 images of 1 channel, 28 rows and 28 columns each. `y_train.shape` is simply `(50000,)`, that is, it is a vector the same length of `X_train` giving an integer class label for each image – namely, the digit between 0 and 9 depicted in the image.

Training the model

First we define a short helper function for synchronously iterating over two numpy arrays of input data and targets, respectively, in minibatches of a given number of items. For the purpose of the tutorial we short it to:

```
def iterate_minibatches(inputs, targets,
                        batchsize, shuffle=False):
  if shuffle:
    ...
  for ...:
     yield inputs[...], targets[...]
```

All that is relevant is that it is a generator function that serves one batch of inputs and targets at a time until the given dataset (in `inputs` and `targets`) is exhausted, either in sequence or in a random order.

The actual training is started in the `main()` function.

```
# Load the dataset
X_train, y_train, X_val, y_val, X_test, y_test = load_dataset()
# Prepare Theano variables for input and targets
input_var = T.tensor4('inputs')
target_var = T.ivector('targets')
# Create neural network model
network = build_cnn(input_Var)
```

The first line loads the inputs and targets of the MNIST dataset as numpy arrays, split into training, validation and test data. The next two statements define symbolic Theano variables that represent a mini-batch of inputs and targets in all the Theano expressions we generate for network training and inference. They are not tied to any data yet, but their dimensionality and data type is fixed already and matches the actual inputs and targets we will process later.

[10] See full code example on `https://github.com/aldewereld/nl.hu.ict.a2i.cnn/blob/master/lasagne_mnist.py`.

Loss and update expressions

Continuing, we create a loss expression to be minimised in training:

```
prediction = lasagne.layers.get_output(network)
loss = lasagne.objectives.categorical_crossentropy(
    prediction, target_var)
loss = loss.mean()
```

The first step generates a Theano expression for the network output given the input variable linked to the network's input layer(s). The second step defines a Theano expression for the categorical cross-entropy loss between said network output and the targets. Finally, as we need a scalar loss, we simply take the mean over the mini-batch. Depending on the problem you are solving, you might need different loss functions (there are others in `lasagne.objectives`).

Having the model and the loss function defined, we create update expressions for training the network. An update expression describes how to change the trainable parameters of the network at each presented mini-batch. Again, we use Stochastic Gradient Descent (SGD) here, but there are others available in `lasagne.updates`.

```
params =
  lasagne.layers.get_all_params(network, trainable=True
updates =
  lasagne.updates.sgd(loss, params, learning_rate=0.001)
```

The first step collects all Theano `SharedVariable` instances making up the trainable parameters of the layer, and the second step generates an update expression for each parameter.

Compilation

Equipped with all the necessary Theano expressions, we are now ready to compile a function performing a training step:

```
train_fn = theano.function(
    [input_var, target_var],
    loss, updates=updates)
```

This tells Theano to generate and compile a function taking two inputs – a mini-batch of images and a vector of corresponding targets – and returning a single output: the training loss. Additionally, each time it is invoked, it applies all parameter updates in the `updates` dictionary, thus performing a gradient descent step.

For validation, we compile a second function:

```
val_fn = theano.function(
     [input_var, target_var],
     [test_loss, test_acc])
```

This one also takes a mini-batch of images and targets, then returns the (deterministic) loss and classification accuracy, not performing any updates.

Finally, we write the training loop. In essence, we need to do the following:

```
for epoch in range(num_epochs):
  for batch in iterate_minibatches(X_train, y_train,
                                   100, shuffle=True):
    inputs, targets = batch
    train_fn(inputs, targets)
```

This uses our dataset iteration helper function to iterate over the training data in random order, in mini-batches of 100 items each, for `num_epochs` epochs, and calls the training function we compiled to perform an update step of the network parameters.

The complete code (available on `https://github.com/aldewereld/nl.hu.ict.a2i.cnn/blob/master/lasagne_mnist.py`) shows how to monitor training and performance of the network.

Performance and GPU use

At first glance, the Lasagne implementation appears to be much slower than the TensorFlow implementation mentioned above, but the comparison is not completely fair. The Lasagne implementation runs *per epoch*, while the TensorFlow runs *per step*. An epoch is a run over the complete dataset, while a step is simply a run over a single batchsize. So, as MNIST contains 50,000 images (of which 10,000 are reserved for validation), a single epoch equals $40,000/100 = 400$ steps (that is, dataset/batchsize). To get a fair comparison, we need to run $20,000/400 = 50$ epochs of training to get similar results.

On an Intel i5 3.2GHz processor with 12GB RAM, the training of the network required around 2.5 hours, and achieved a performance of 99.14%.

Like TensorFlow, Theano can be sped up significantly by using the GPU (and again, unfortunately, only NVidia CUDA GPUs are supported). Details about adding CUDA support to Lasagne and Theano can be found at `http://lasagne.readthedocs.io/en/latest/user/installation.html#gpu-support`.