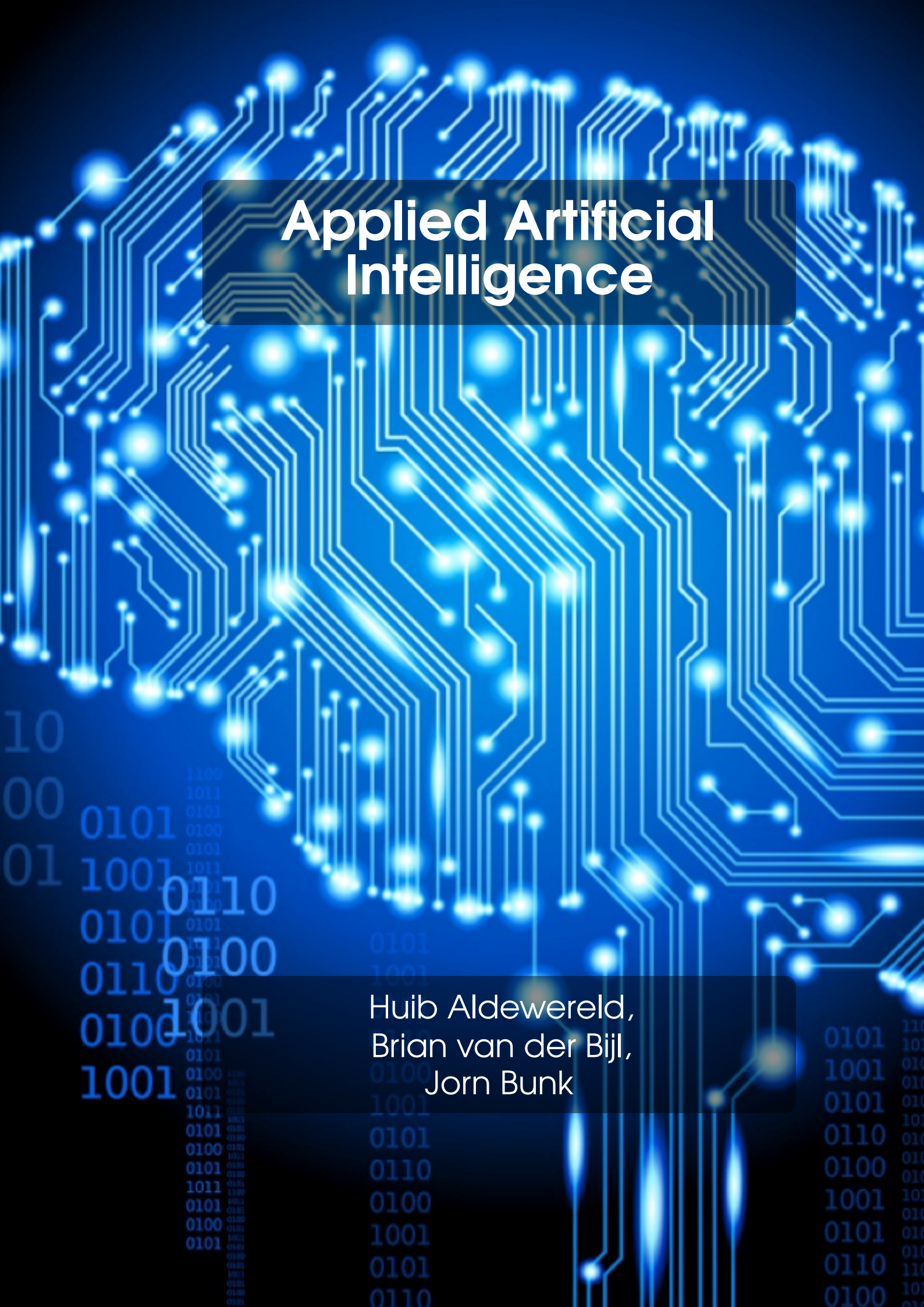
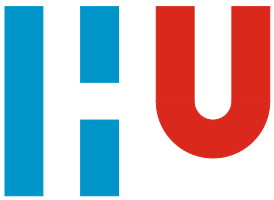


Applied Artificial Intelligence

Huib Aldewereld,
Brian van der Bijl,
Jorn Bunk

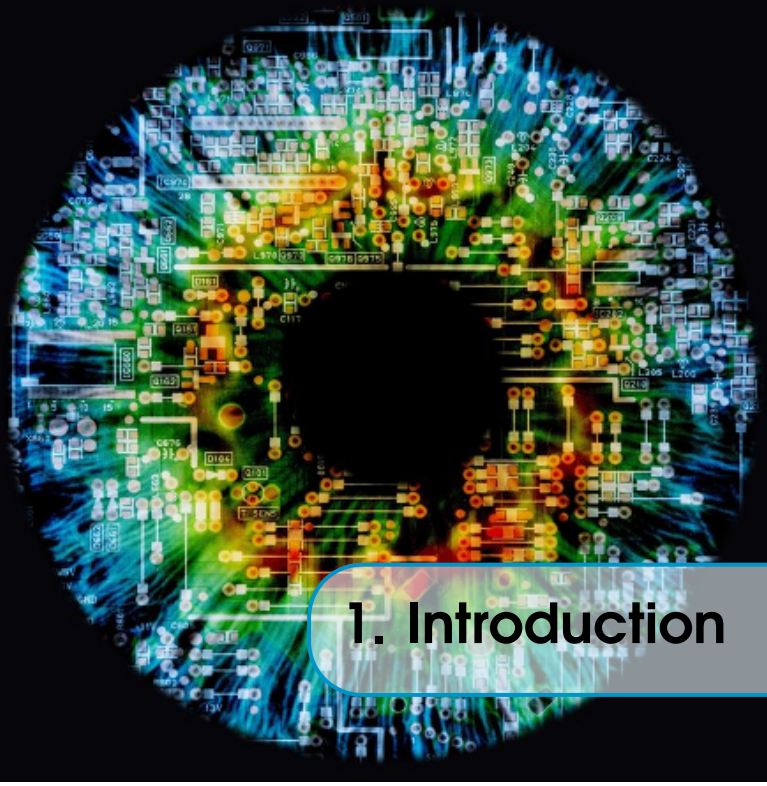


This reader was verified by:

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

First release, October 2017

Contents



1. Introduction

Artificial Intelligence (AI) is omnipresent nowadays; not just in science fiction novels, films, and TV series, but also in our day-to-day lives. Think about techniques that we use (daily) like navigation, language processing (both in speech recognition and speech generation, but also in, for instance, the grammar and spelling check in word processors), and data analytics and usage.

It can therefore not be denied that AI is making the transfer from a pure scientific nature to an applied reality. It is thus important that Computer Engineers are aware of the current possibilities of AI and how to apply those techniques in their jobs.

This reader is the accompanying document for the course *Applied Artificial Intelligence* (TCTI-VKAAI-17) from the University of Applied Sciences Utrecht. In this course, we take a short step back, and discuss what AI is, and how it originated. This is meant to be a background on the practical issues of AI and getting to know which AI can and which it cannot solve. This introduction into AI is presented in chapter 2.

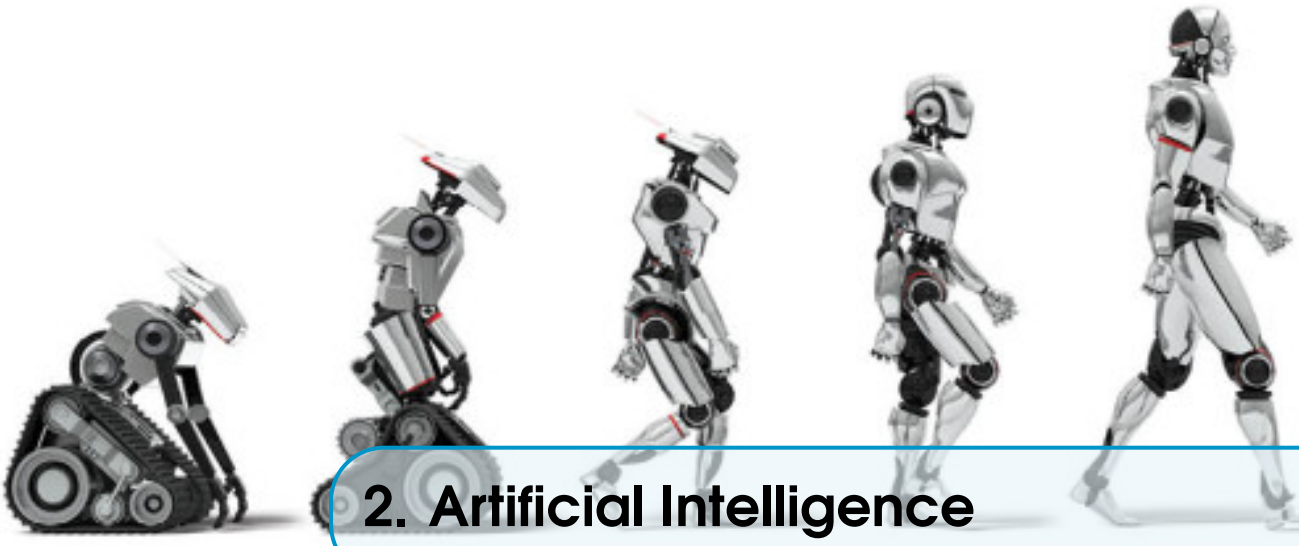
The main focus of the course is then shifted to the topic of Neural Networks. Neural networks originated from the minds of Cognitive Psychologists as a model of how the human brain functions, but has been slowly but surely evolved over the years into one of the most promising areas of (applied) artificial intelligence. Many of the recent developments in AI (that made the news) have been build using techniques from neural networks; IBM Watson wins *Jeopardy!* (Markoff, 2011, February 16), Google AlphaMind defeats world champion Go ("Artificial Intelligence: Google's AlphaGo beats Go master Lee Se-dol", 2016, March 12).

But what exactly is a neural network? How does it function? What problems can you solve with it? How does it do that? In chapter 4 we discuss the basics of neural networks, and their fundamental workings.

Neural networks, as many of the advances of AI in the recent years, benefit greatly from the increase in computing power that we have gained since their inception (about 60 years ago). CPUs nowadays can perform many more computations than CPUs

back then, and can process much larger problems due to the vast increases in memory available. Recent developments into GPU processing have further increased this computational capacity. Though it has to be noted that the main benefit from GPU computation is on parallel computation, not for linear computations. In chapter 5 we show that the computations needed for training and using neural networks, largely, adhere to these parallel constraints for benefiting from GPU optimalisations. Moreover, we show how (simple) neural networks can be programmed using the GPU to gain that computational benefit.

Finally, training a neural network is a large factor in the (potential) realisation of applications using such techniques. There are many factors to be considered (topology, type, weights, ...). As these factors influence each other, it is hard to make decisions on the right combination(s), making it difficult to optimise your solution. Due to the vast number of possible combinations, trying and testing each combination is infeasible. However, using another AI technique, namely Evolutionary Computing, likely candidates of combinations that would perform well can be found rather easily (and, maybe more importantly, fast). Evolutionary computing is an algorithmic approach to optimalisation, largely based on ideas taken from Evolution Theory (Darwin, 1859). In chapter 6 we explain the basics of evolutionary algorithms and how to apply these to optimise the training of a neural network.



2. 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 2.1 — 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, ca. 8, Book X) and (Wikipedia, 2005, May 11). Picture by (Wikipedia, 2005, May 11).

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 2.2) has captivated scientists for many ages.

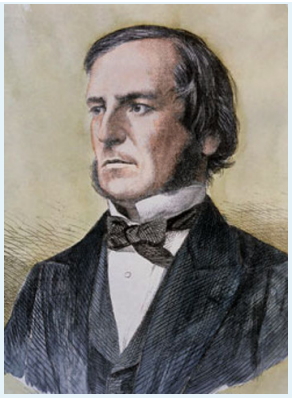
Aside 2.2 — *Calculemus!* – A history of formal reasoning.
The search for a formal, mathematical, system for reasoning started with the Syllogisms of Aristotle. The famous example: *"If all men are moral, and Socrates is a man, one can derive that Socrates is mortal"*.

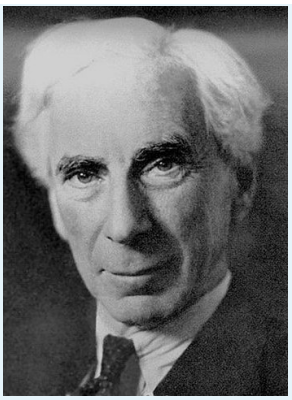
This idea of formal reasoning was further enriched over the years by philosophers like Descartes and Hobbes, until the German philosopher Leibniz dreamt of a formal apparatus that would allow one to precisely determine the truth of things by means of calculation – *calculus ratiocinator*.

> "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 (Doxiadis & Papadimitriou, 2009). 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.

2.1 A brief history of Artificial Intelligence

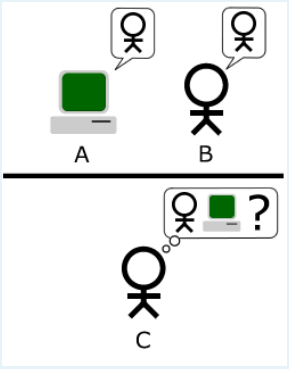
Birth (1952 – 1956)

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.

McCulloch and Pitts (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 2.3). A simplified version of this test allowed Turing to argue convincingly that a "thinking machine" was at least *plausible*.

Aside 2.3 — 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 (Wikipedia, 2001, August 26).

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, Shaw, and 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, 1967)

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.

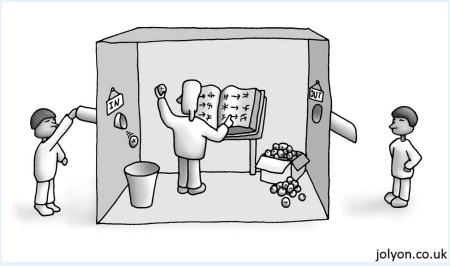
Due to the lack of progress in AI, funding agencies (such as British Government, DARPA and the National Research Council) became frustrated and cut off almost all funding for undirected research in AI. To make matters worse, AI received many critiques even from researchers within the field, that AI might 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 2.4), and that symbolic reasoning (i.e., using logic) was never going to create intelligence. A book by Minsky and Papert (1969) 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 2.4 — Chinese room.
The Chinese Room argument, a thought-experiment by John Searle in (Searle, 1980), 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 2.5 — *Elephants don't play chess* (Embodied AI).
In a 1990 paper titled *"Elephants Don't Play Chess"* (Brooks, 1990) 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."

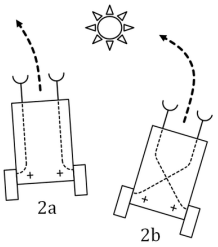
Brooks was wrong.
Picture by Ethiriel Photography.

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" (Ulam, 1958, May). 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, 2005) 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 (Spectrum, 2008, June 1):

> "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 2.6 — 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 (Asimov, 2008).

2.2 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 2.1 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.

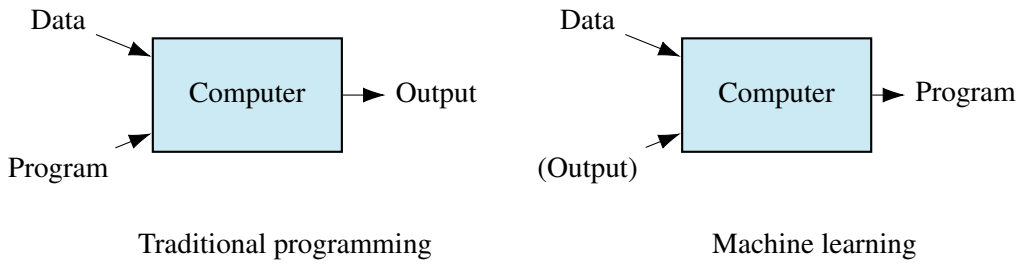


Figure 2.1 Traditional programming vs. machine learning

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 4), *k-nearest neighbours* (see section 3), *decision trees*, and *support vector machines*.

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* (see section 3 below), 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.

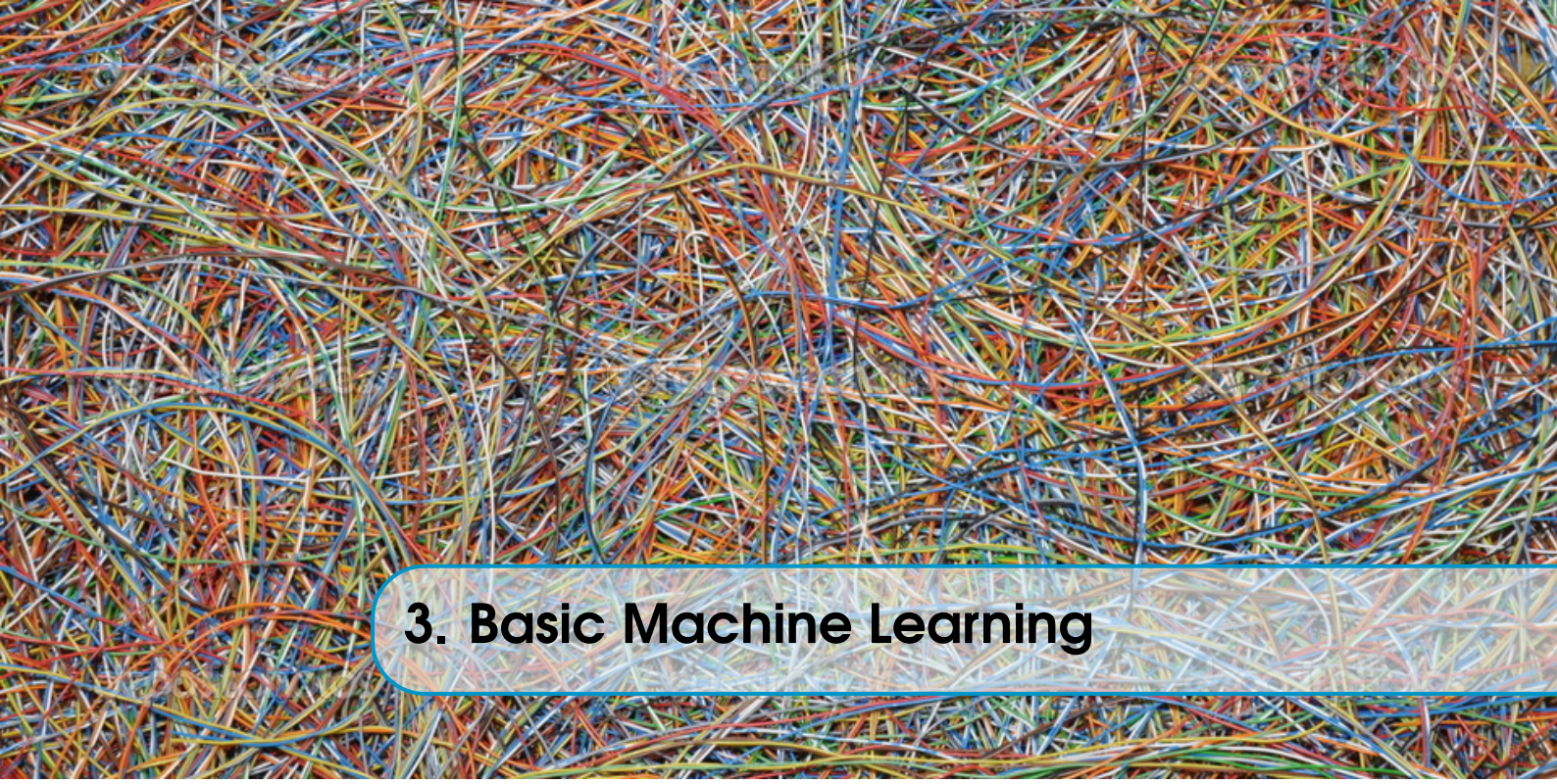
Many machine learning techniques require information about the probabilities of various events involving the data. A Bayesian approach (using probabilities), for instance, presupposes knowledge of the prior probabilities and the class-conditional probability densities of the attributes. This information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



3. Basic Machine Learning

As an example of Machine Learning algorithms, in this chapter we discuss the Classification algorithm of k-Nearest Neighbours and the Clustering algorithm of K-Means. As shown in section 2.2 there are many more machine learning methods, the two presented here provide a good and generic introduction into the field of machine learning.

3.1 Classification: k-nearest neighbours

Classification is the generic process of trying to categorise something. It encompasses the process in which ideas or objects are recognized, differentiated, and understood (put in some class). In general, classification is the problem of trying to identify to which set of categories (also called sub-populations or *classes*) a new observation belongs. Observations can relate to physical objects (e.g., people, animals, books, etc.) or more abstract concepts (like events, ideas, etc.). The categorisation is done based on a given training set of data containing observations (instances) whose category membership is known. An example is "spam" filtering, where incoming (before unseen) emails are categorised as either "spam" or "ham" (not-spam)[1] given a body of emails of which it is known (or has been indicated) whether they are "spam" or "ham".

The availability of a training set of which the class membership is known, makes classification a **supervised** machine learning technique. An algorithm that implements classification is known as a **classifier**. It can be seen as a function that maps input data to a category.

[1] SPAM© was a type of canned cooked meat popularized after the second world war, which was not quite ham. The use of the word for unsolicited (electronic) advertisement was derived from the Monty Python sketch, in which the word was increasingly inserted into the conversation (Wikipedia, 2001, July 31).

A classification algorithm examines the individual observations (*instances)* and analyses of each of them a set of quantifiable properties, known as *explanatory variables* or *features*. These can be, for example, *age*, *length*, *weight*, etc. of a human being (to be classified as 'adult' or 'child'), or the relative frequency of the occurrences of "spammy words" in an email. A combination of such quantifiable properties is called a *feature vector*.

A feature vector is a n-dimensional vector (it contains n attributes) of numerical features that represents some object[2]. For instance, our representation of a human being, mentioned above, by *age*, *length*, and *weight* has 3-dimensional feature vector. More concretely, the feature vector $(56, 185, 81)$ is then the representation of 'Barack Obama'. When representing images, the feature values might correspond to the pixels of an image, while representing texts the features might be the frequencies of occurrence of textual terms.

The *vector space* can then be understood as space that can contain all the possible values in the feature vectors, and represents a complete picture of the domain. Each feature of the feature vectors relate to a single dimension of the vector space, in much the same way as the x, y and z position (feature) of a point relates to a position on the x-, y- and z-axis of a graph. A feature vector with 20 attributes thus relates to a 20-dimension vector space (which can be difficult to grasp, or graph).

Classes or *categories* can then be understood as a particular partition (section) of the vector space, where all instances that fall into that partition are considered to be of the same 'type'. Note that the differentiation between classes can be made on a single feature (e.g., everyone below the age of 18 is considered 'child', everyone above is an 'adult'), or a combination of features (e.g., specific combinations of length and weight are considered healthy, others are considered unhealthy).

3.1.1 Intuitions

There are multiple ways to perform classification. One could use Bayesian (probability) based calculations or statistical (Gaussian) models to predict the probabilities that a particular observation belongs to a particular class. These models can be extremely powerful, but are rather cumbersome with (complex) mathematics. Instead, we introduce the *k-Nearest Neighbours algorithm*, which performs in a very similar way as that we humans would, which makes it much easier to understand.

Consider the classification problem presented below in Figure 3.1. This is a typical classification problem; we have a training set of labelled data (the red circles and the blue triangles), and have been given the task to determine whether a new observation (the white box) is either a red circle or a blue triangle.

When given the task of trying to determine the class of the box in Figure 3.1, humans do not calculate the priors of it belonging to 'blue' or 'red', nor do humans use statistics to predict the probability of it being 'red' (or 'blue'). Instead, humans will eyeball the image, and (correctly) determine that the box will *probably* be a 'red circle', merely on the fact that it is *much closer* to the red examples, than to the

[2]When a variable x is in fact a vector, we will indicate that with an overset arrow: $\vec{x}$ for clarity. The different attributes of $\vec{x}$, we typically number from 1 to n: $\vec{x} = (x_1, \ldots, x_n)$. A random attribute of $\vec{x}$ is indicated as x_i.

blue examples. This intuition of classification based on closeness is what drives the k-Nearest Neighbour algorithm.

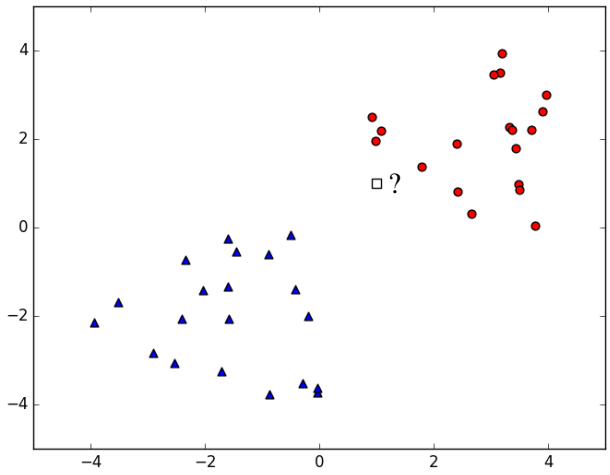


Figure 3.1 Classification problem: to which class does the box belong?

While with a more complex vector space (our example uses only two features, since it is easier to plot) a human could not eyeball it any more, as long as can come up with some measure of 'closeness', we could still apply the same principles to determine classes of observations in, say, a 20-dimensional feature space.

In essence, by looking at the closest neighbour (of which we know the class), we are partitioning the vector space into parts where the boundaries between two points is determined by the fact that every point on the boundary is an equal distance to either point. This type of partitioning is called *Voronoi tesselation*. Figure 3.2 below shows an example of such a Voronoi tesselation (right) of the vector space shown (left). By looking at the position of the box, it can now be determined that in the problem presented in Figure 3.2, the box is closest to a blue triangle (it is within the region of a triangle) and is therefore probably a triangle.

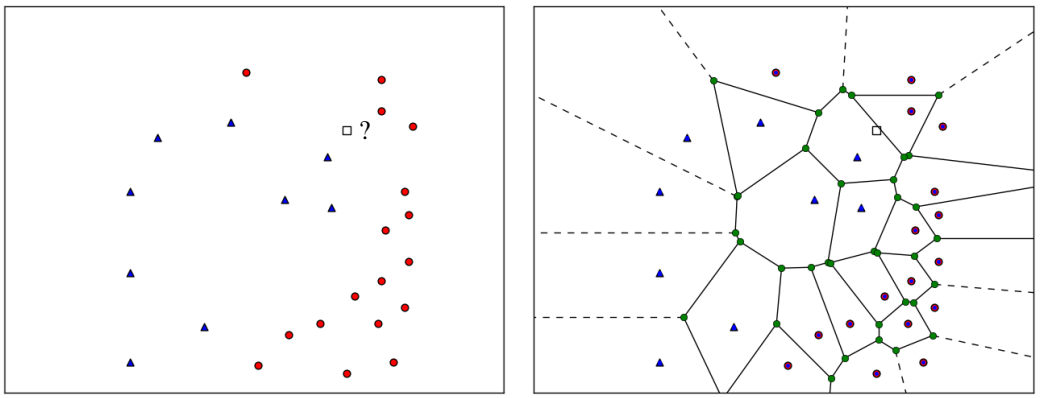


Figure 3.2 Feature space (left) and Voronoi tesselation (right).

The combinations of all regions of the triangles and all the regions of the circles defines the transition of when a new observation is classified as either a triangle or as a circle. This is the boundary between the set of regions of either example, and is called the *classification boundary*. Figure 3.3 below shows the classification boundary for the given example. Observations to the left of the classification boundary are classified as triangle, those to the right of it are classified as circles.

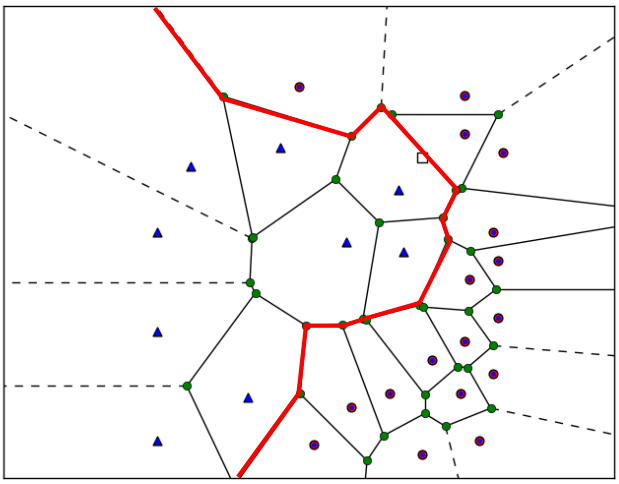


Figure 3.3 Classification boundary for k-NN with $k = 1$.

Note that the classification boundary is not linear, and reflects well the different classes, which is rather impressive for such a simple algorithm. However, this method is not without its problems, as we see in the following Section.

3.1.2 Outliers

The classification problem presented above was rather straight-cut. While the classification boundary has a rather complex shape, the training set is still very neat (all triangles are on one side, the circles on the other). In reality, it often happens that training samples are much more scattered throughout each other. In some extreme cases, we speak of outliers (see Figure 3.4 below). The single red circle on the left side of the feature space is an odd duck, and could be the result of some error (perhaps noise during measurement or a typo in the data), but it could also be that our classification examples are not as clear as we have seen above.

Suddenly, the classification boundary is even more complex, and non-continuous. There now is some region in which instances are classified as triangles (between the orange and red lines) and everything else is classified as circles. Again, this is a simplified view to highlight a problem, more outliers will increase the complexity of the classification boundary even further.

If the single circle on the left of the training set is indeed an error, this shows that errors can have strong effects on the outcome of the classifier. A single error radically

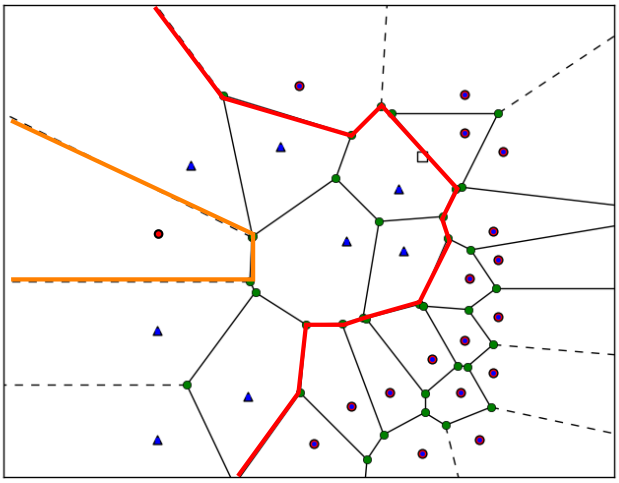


Figure 3.4 An outlier and the effect on the classification boundary (orange).

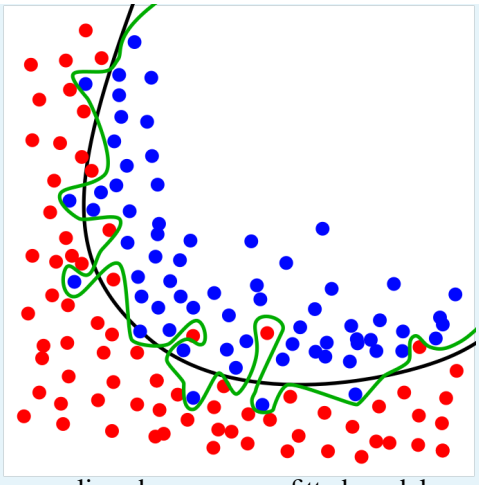
changes the shape of the classification boundary, and this is unwanted; the boundary exactly follows the training data.

Until now, we have only looked at a single neighbour of an instance. If we take more neighbours into consideration (e.g., the closest 3 or the closest 5) and use those to determine the class of a new observation, the classifier becomes much more stable. In this sense, we take the closest k[3] training samples and derive a class from them (e.g., by taking a mean, or a majority vote). In our example above (see Figure 3.4), the number of triangles on the left side of the image will outweigh the single circle, thus removing its destructive effect on the classification boundary.

Aside 3.1 — Overfitting vs. underfitting.
The problem described above, when a learned boundary, or other property that is derived from the data, exactly models the training data is called *overfitting*. In essence, you have made a single perfect classifier for one set of data: the classifier will work flawlessly on the training data, but can perform significantly worse on other (new) data.

When using machine learning, it has to be taken into account that one is typically working with noisy data. That is, the samples that you have to train your algorithm are *not* an exact representation of the problem/solution. There could be parts missing, or the data available could be distorted by noise, etc. Whatever the problem might be, it is never a wise decision to exactly model the training data available, since your algorithm looses all possibilities of generalising to other (similar, but slightly different) instances that have not yet been seen.

[3] Hence the name of the algorithm.



The green line shows an overfitted model, exactly following the training data. The black line shows a regularised model.

The opposite of overfitting is *underfitting*, when you design your algorithm too loosely. In that case, the classifier will not be precise enough with respect to the instances presented. Training machine learning algorithm is often trying to find a balance between the two.

Image by Wikipedia.

3.1.3 The k-NN algorithm

Now we understand the intuition behind distance based classification, we can introduce the k-Nearest Neighbours (k-NN) algorithm. The k-NN algorithm is given below:

Algorithm 3.1 — k-Nearest Neighbours.
Given:

- Training set X of examples $(\vec{x}_i, y_i)$ where
  - $\vec{x}_i$ is feature vector of example i; and
  - y_i is class label of example i.
- Feature vector $\vec{x}$ of test point that we want to classify.

Do:

1. Compute distance $D(\vec{x}, \vec{x}_i)$;
2. Select k closest instances $\vec{x_{j_1}}, \ldots, \vec{x_{j_k}}$ with class labels $y_{j_1}, \ldots, y_{j_k}$;
3. Output class y^*, which is most frequent in $y_{j_1}, \ldots, y_{j_k}$.

■

The k-NN algorithm is simple to understand. We take our new observation $\vec{x}$ and compare that to all available training examples, by computing the distance $D(\vec{x}, \vec{x}_i)$ between the two. Next, we select the k closest training samples to $\vec{x}$ and collect their labels $y_{j_1}, \ldots, y_{j_k}$. We can then determine the answer by selecting the label y^* by looking which label occurs most often.

There are multiple ways to calculate distances between vectors, but for now we

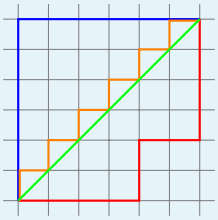
use the Euclidean distance (straight-line distance between two points[4]).

Aside 3.2 — Distance measures.
So far we have only considered the most straightforward distance measure to determine the distance between points (in both K-Means and in k-Nearest Neighbours). This measure, Euclidean distance, is the straight-line distance between two points in space, like "as the crow flies" in geometric distances ("vogelvluchtafstand"). It is an intuitive and much used distance measure, but it has some drawbacks.

A drawback of Euclidean distance is that it evaluates all the dimensions of the points equally important. That is, given a 2-dimensional problem (where points have an x-value and an y-value), Euclidean distances considers a step on the x-axis as important as a step on the y-axis. That is, $(2,1) \rightsquigarrow (2,2)$ is as far apart as $(2,1) \rightsquigarrow (3,1)$. This can produce big differences when the scale of each of the dimensions is different. For instance, when x represents *age* and y represents *grossincome*, the influence of y on the distance between two points is much larger than that of x (as y, for typical people, is measured in thousands (of euros), while age is measured in single years.

Other, numerical, distance measures available are:

- **Manhattan Distance**: the distance between two points calculated by strict horizontal and/or vertical paths (that is, along the grid lines), as opposed to the diagonal ("as the crow flies") Euclidean distance.



Blue, red, orange: Manhattan distance (note they are all the same: 12);
Green: Euclidean distance (8.5).

- **Minkowski Distance**: a generic measure of distance, generalised from the Euclidean and Manhattan distance described above. Minkowski distance is calculated by: $\sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}$
When p is 1, this equates to the Manhattan distance, when p is 2 one gets the Euclidean distance. With p reaching ∞, one obtains the Chebyshev distance.

5	4	3	2	2	2	2	2
5	4	3	2	1	1	1	2
5	4	3	2	1	**k**	1	2
5	4	3	2	1	1	1	2
5	4	3	2	2	2	2	2
5	4	3	3	3	3	3	3
5	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5

Chebyshev (or chessboard) distance between each field and the location of the king at (6, 6); it indicates that the king can move on both axes at the same speed.

[4]The calculation of the Euclidean distance is a straight-forward application of the law of Pythagoras: the distance d between points $(a_1,\ldots,a_n)$ and $(b_1,\ldots,b_n)$ is $d^2 = (a_1-b_1)^2+\ldots+(a_n-b_n)^2$.

- **Chebyshev Distance**: this measure defines the distance between two vectors as the greatest of their differences along any coordinate dimension. It is best visualised by the moves of a king on a chessboard (see above).

Another disadvantage is that Euclidean distance only works for numeric attributes on an interval or ratio scale. For categorical scales (e.g., *gender* or *hair colour*) it is impossible to calculate a Euclidean distance. Instead, one can use the **Hamming Distance**, which counts the number of substitutions (changes) that need to be made to get from one sample to another. That is to say, it counts the number of attributes that are different between two given points.

The selection of the resulting class label from the set of labels $y_{j_1}, \ldots, y_{j_k}$ can be understood as a majority vote. Each y_j in the set counts as a ballot ('stembiljet') in the vote, and the class that is selected the most is considered the winner.

3.1.4 Problems with k-NN

While k-NN is intuitively simple it performs quite well for such a simple algorithm. However, there are some inherent problems with k-NN, that we address here.

How to choose K?

The biggest question when applying the k-NN algorithm, is how to determine the best k; that is, how many neighbours should you look at to get the best (stable) result in your classification.

The choice of the value of k has a strong effect on the performance of k-NN. If the k is chosen too large, everything will be classified as the class that is most common in the data (that is, the one that has the most ballots in the vote). When k approaches the size of your training set, the closeness to samples of a particular class have no longer an influence on the classification, but the amount of samples that are available (of each class) in the data set start to overrule the vote.

If the value of k is chosen too low, as we have seen before, the classifier becomes less stable, and is much more receptive to the influence of outliers. This leads to an unstable classification boundary. The value of k has a large impact on the smoothness of the classification boundary.

See Figure 3.5:

- $k = 1$ will classify the green circle as a red triangle;
- $k = 3$ will still classify it as a red triangle;
- $k = 4$ leads to a tie (see next Section);
- $k = 5$ changes classification to a blue square;
- $k = n$ keeps classification as blue square (there are more square examples than triangle examples in the training data).

To select the proper k, we have to use a technique often applied in machine learning. We set aside a portion of the training data (i.e., those randomly selected samples of which we know the class are removed from the set of samples given to the algorithm); this set is called the *validation set*. Next we run the algorithm with different sizes of k and test on this validation set which number of k performs the best on classifying the validation samples. Since we know the class of each of the samples in the validation set, we can calculate a classification error for each k based

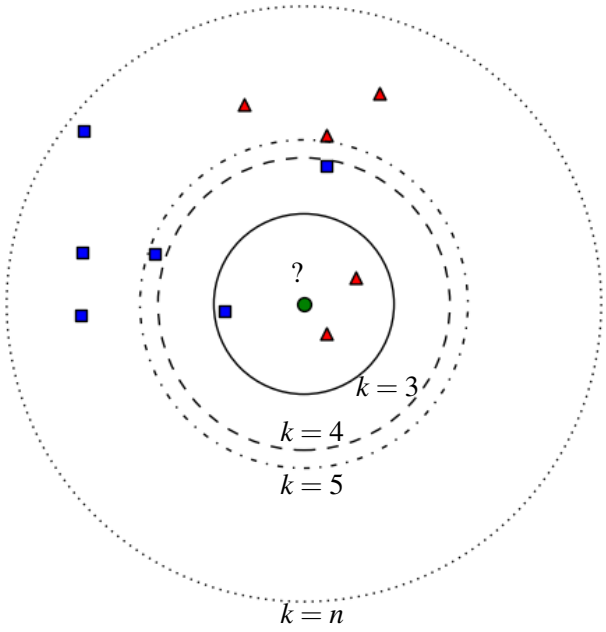


Figure 3.5 The influence of the value of k.

on the number of correct/incorrect classifications made by the algorithm. We can now determine the best k, we select the k that gives the best generic performance.

Resolving ties

It can occur that the classification vote ends in a tie (see the example in Figure 3.5 above). In those rare cases when there are an equal amount of positive and negative votes there are a number things to do:

- **Use an odd** k: when dealing with binary classification, using an odd number of neighbours in the vote makes it impossible to have as much positive as negative votes. However, when dealing with a multi-class classification problem (where there are more than two classes), this solves nothing, however, as ties can still occur.
- **Breaking ties**: other possibilities of solving ties are:
  - At random: choose a class at random (by flipping a coin). The rational is that it does not matter, as the observation is obviously on the border between the two classes.
  - Use the *prior*; that is, choose the class that is most often occurring in the training set. The rational is that, since the prior is more common (more occurring), the probability that you guess right by selecting that class is higher.
  - **Nearest**: use a lower k (e.g. $k = 1$, since that will never result in a tie) to select the class.

Why is k-NN slow?

While k-NN performs well for its complexity, it not a very forgiving algorithm. With larger training sets, the performance (time/memory it takes to calculate an answer)

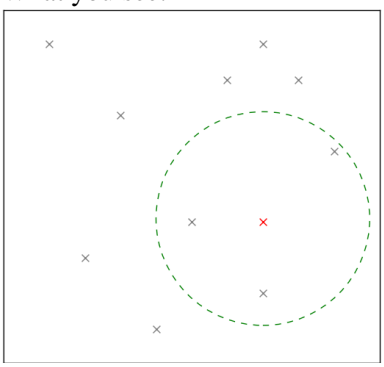
becomes increasingly worse.

The main problem of k-NN is the fact that, since the computer has no representation of distance, dimensions or closeness. In order to assess a new observation given a training set of data, it has to compute the distance of the observation to each of the available training samples. And with large amounts of training data available (which tends to increase the robustness of your classifier), the classifier will perform slower and slower. In the same way, each additional dimension in the features of samples increases the amount of calculations needed to compute the nearest k elements (which is why it often helps to reduce dimensions, see Section 2.2, *before* using k-NN).

Consider the Figure 3.6 below.

Figure 3.6 Comparison: find the nearest neighbour to the red cross.

What you see:



What the computer sees:

```
data = np.array([
 (1, 9), (2, 3), (4, 1), (3,7),
 (5, 4), (6, 8), (7, 2), (8, 8),
 (7, 9), (9, 6)
])
testPoint = (7, 4)
```

To the computer, the training data is just an array of feature vectors, which each have to be compared to `testPoint` to calculate the distance. For a training set of n samples, with d features, this takes $n * d$ comparisons for every point you try to classify.

The easiest fix for this problem, as mentioned above, is the removal of (a number of) dimensions. This can be done in a simplistic way, by throwing away attributes that are not important to the classification process; e.g. in our earlier example, where we represented humans as a vector of $(age, length, weight)$, and try to classify them as 'child' or 'adult', we can remove the *length* and *weight* attributes. For more complex dimensional reduction, one could use more complex (mathematical) techniques such as Principle Component Analysis, which tries to find the most discriminative dimensions (which could be combinations of some of the dimensions in the feature space).

Another fix would be by reducing the number of training examples. The idea here is identify a small number of individuals $m \ll n$ of potential neighbours, and use only those to classify new observations. In essence, you are trying to find *prototypical examples* of the classes, and use those to do the classification. This idea is exploited in the K-D Trees algorithm (which can be thought of as a combination of k-NN and decision trees; exploiting the best of both techniques).

3.2 Clustering: K-Means

Clustering is the problem of trying to determine whether there is any underlying structure in the data that is available. For instance, if you have data about a number of

individuals, can those individuals be grouped in some way(s) such that members of each group seemingly belong together, yet also not appear to be belong to some other group.

That is, clustering tries to come up with groups (called *clusters*) such that the intra-cluster distance is minimised, yet the inter-cluster distance is maximised. The intra-cluster distance is a measure of the similarity of objects within a group (how much are they similar to each other); the inter-cluster distance is the difference (from the group as a whole) to other groups that have been identified.

Clustering is typically an **unsupervised** task as it does not try to predict anything specific. It tries to find sub-populations in the data, but it can be difficult to figure out how many of those there are and what their size is? Moreover, are the clusters based on common properties and are the clusters cohesive or can they be split further?

This latter aspect of clustering, whether elements in a cluster have common properties, differentiates between two forms of clustering: monothetic and polythetic clustering. *Monothetic clustering* groups members on a single common property, for instance, all elements have an age between 20 and 35 years, or each has a particular answer to a particular question of an inquiry. *Polythetic clustering*, on the other hand, groups elements based on similarity, where the similarity is not driven by a single common property. Based on all attributes of the elements, the distance between the elements alone defines their membership to the different groups (that is, all attributes are considered to determine group membership).

Another important difference between clustering algorithms is whether the clusters can have an overlap. *Hard clustering* are those algorithms that assign a unique group to each element, meaning that elements belong to a cluster or it does not belong to that cluster. *Soft clustering*, however, is less strict, and allows elements to be part of multiple clusters. Not only can elements be part of multiple clusters, the degree by which they belong to that cluster is a scale (real number).

Lastly, clusters can either be *flat* or *hierarchical*. Flat clustering makes single groups, and groups are not dividable into smaller groups. In hierarchical clustering, however, one can allocate multiple levels of groups, where groups on a higher level can be divided into smaller groups on the next level (think of the taxonomy of animals/plants in biology).

The K-Means algorithm that we discuss here (see Section 3.2.1 below) is only one example of a clustering algorithm; there are others. K-Means is a polythetic, hard boundary, flat clustering algorithm. An example of a monothetic (hard boundary, hierarchical) clustering algorithm is k-D trees (the combination of k-Nearest Neighbours and Decision Trees). An example of a soft boundary clustering method (polythetic, flat) is Gaussian Mixture Models, which is a statistical method that tries to plot (multi-dimensional) Gaussians to determine the degree of membership of the elements[5].

[5]Neither k-D Trees nor Gaussian Mixture models are part of the content of this course, and you are not required to understand these. The mentioning of them here is merely to illustrate different methods for the same problem.

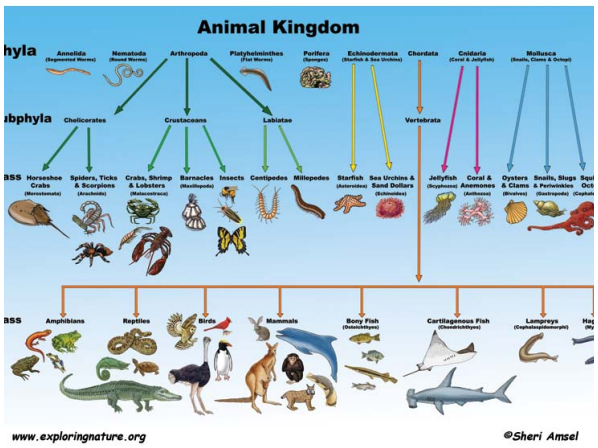


Figure 3.7 Taxonomy of the animal kingdom.

3.2.1 The K-means algorithm

In contrast to the k-Nearest Neighbours algorithm, K-Means is not derived from the intuitions that can be derived from humans performing a clustering task. Intuitively, humans would cluster based on closeness, which is much more similar to what is done in Gaussian Mixture Models, instead of what is done in K-Means.

However, the K-Means algorithm is much easier to explain and understand (since it does not require any knowledge of Gaussians and statistics). K-Means clustering aims to partition the observations into k clusters in which each observation belongs to the cluster with the nearest mean, which serves as a prototype of the cluster. That is to say, K-Means tries to find k prime examples to represent the entire domain, and matches all individuals/observations to their closest examplar[6]. The partitioning of the feature space by means of these prototypes again results in a set of Voronoi cells, as shown earlier in Figure 3.2.

Formally, the algorithm thus looks as follows:

Algorithm 3.2 — K-Means Clustering.

Given:

- Training set X of examples $\{\vec{x}_1, \ldots, \vec{x}_n\}$ where[a]
  - $\vec{x}_i$ is the feature vector of example i
- A set K of centroids $\{\vec{c}_1, \ldots, \vec{c}_k\}$

Do:

1. For each point $\vec{x}_i$:
   (a) Find the nearest centroid $\vec{c}_j$;
   (b) Assign point $\vec{x}_i$ to cluster j;
2. For each cluster $j = 1, \ldots, k$:
   (a) Calculate new centroid $\vec{c}_j$ as the mean of all points $\vec{x}_i$ that are assigned to cluster j.

■

[a] Note that we do not have a class label; we are trying to determine classes here.

[6] What a *prototype* or *examplar* exactly is will become clear in a little while.

Intuitively, the K-Means algorithm does the following. One starts by (randomly) placing k points, which will function as the prototypes (they are called *centroids*, as they represent the centre of the cluster). Then, repeatedly, the following two steps are performed: 1) each observation is compared to all of these centroids, and put into the same class as their closest centroid. 2) When all observations have been (re)clustered, the mean of the cluster (it's centre) is calculated, and the centroid is shifted to that position.

Those two steps are repeated as long as changes happen; that is, the algorithm stops when no observation is put in a different cluster than it was in before the iteration began.

Lets try to illustrate the algorithm by means of pictures. Given is the following set of points (the grey dots in the top left graph of Figure 3.8) and two centroids placed at random (the yellow and red triangles). The orange line represents the (imaginary) line that indicates the boundary between the red cluster and the yellow cluster, being that it connects all points that are an equal distance away from the red centroid as from the yellow centroid. This line allows us to quickly determine in which cluster a point should fall, the computer, however, has to calculate the distance between a point and each centroid to determine the closest centroid (and thus determine the class of that point). In the top right of Figure 3.8, it is shown that each point is given a label (a

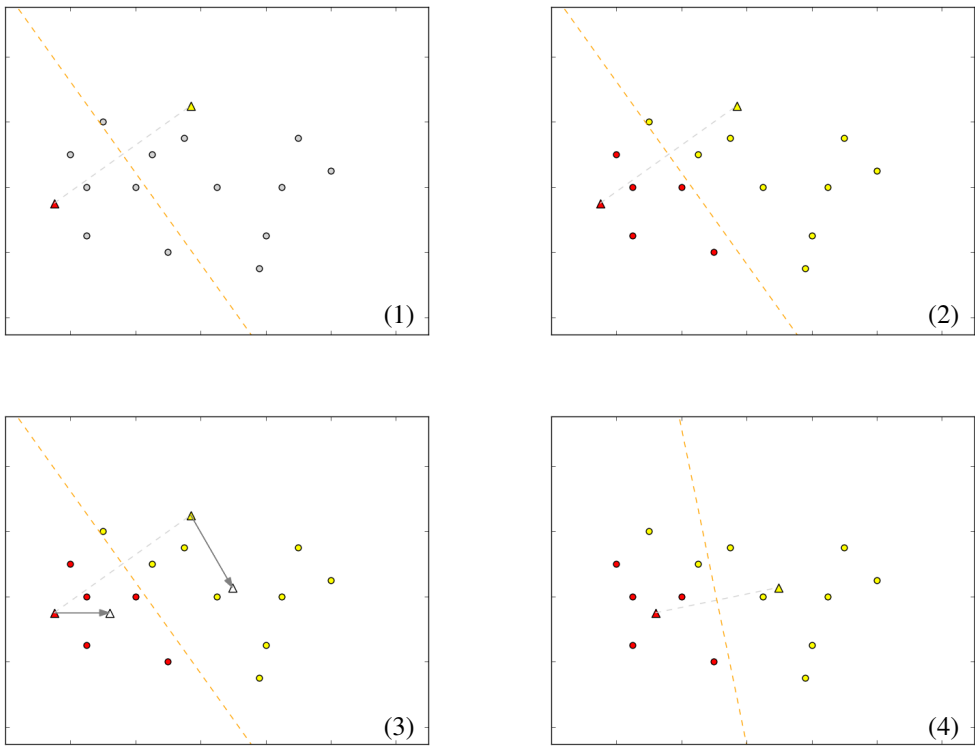


Figure 3.8 The first iteration of K-Means on a dataset. (1) place centroid at random; (2) colour points same as closest centroid; (3) recalculate position centroid as middle of cluster; (4) boundary shifts as result of moving centroids.

colour in this case) to match it's closest centroid. Given this new clustering of the points, we can now determine the new (corrected) centre of the cluster, by calculating the mean of all points within each cluster (except the centroid, because it is not a

true observation). The centroids are moved to this new position (bottom-left), which naturally also moves the boundary between the classes (bottom-right).

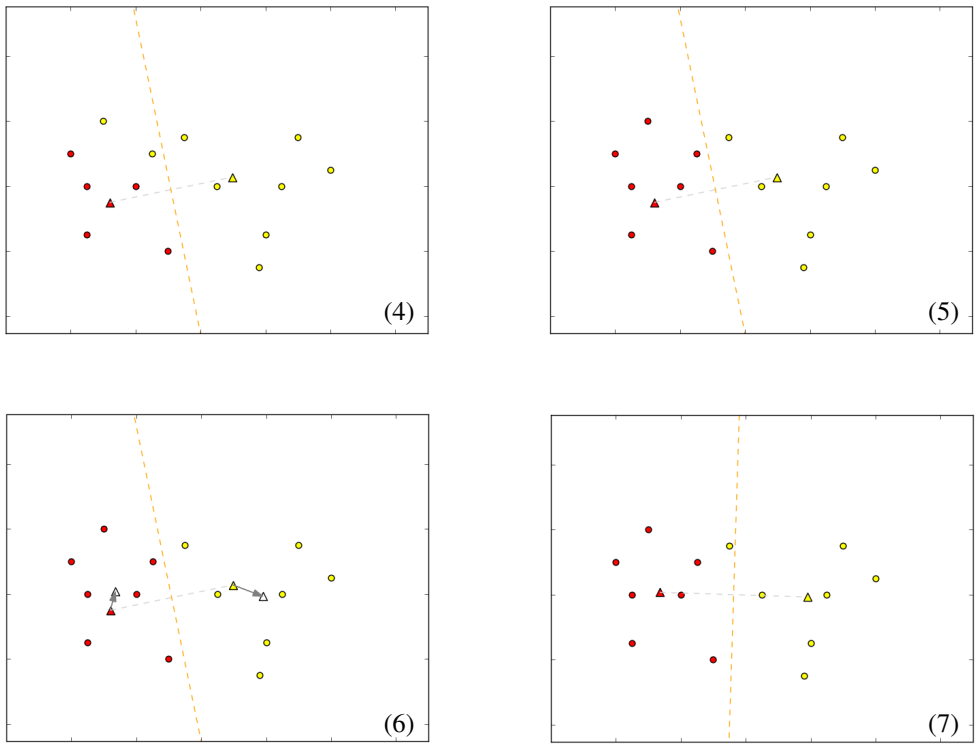


Figure 3.9 The second iteration of K-Means on dataset.

In the next iteration (see Figure 3.9), we start again from this situation, by recalculating the distance of each point to the centroids, colouring the same as their closest centroid, resulting in a new composition of the clusters. After that we re-calculate the centres of the clusters, and again move the centroids to their new position.

This process is continued until no point changes cluster in the relabelling step of the algorithm. The final iteration is shown in Figure 3.10.

3.2.2 Properties of K-means

As shown above, the K-Means algorithm is a rather simple algorithm to create clusters. Despite its simplicity, K-Means is quite effective in finding good divisions, and, more importantly, it is **fast**. As in the example above, it typically takes only a few iterations to find a stable clustering.

K-Means is good in what it does. It minimizes the aggregate intra-cluster distance, which is like we described in the generic clustering problem description above. It means that it tries to minimize the distance between members within a group, yet maximise the distance between the groups (centres). This leads to the most stable division of the feature space. The aggregate intra-cluster distance can be calculated as the mean distance from each point in a cluster to the centroid of that cluster. If Euclidean distances is used (like we did before in k-NN), this measure is the same as the standard deviation as used in statistics.

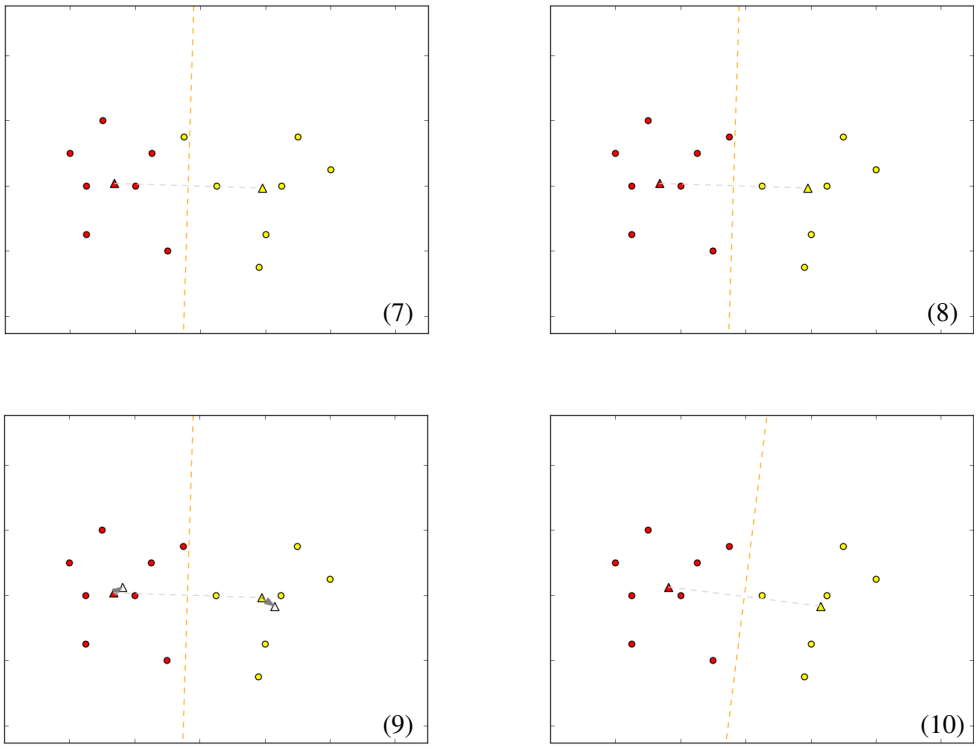


Figure 3.10 The final iteration of K-Means on dataset.

A major problem of K-Means is that it can converge to 'stable' solutions, that are actually not the best solution available. For instance, when we look at the problem given in the left-hand side of Figure 3.11, we would group the left two points in one cluster, and the right two points in another. However, an unlucky choice of start positions for our centroids could mean that K-Means converges to the distribution on the right-hand side of that figure instead.

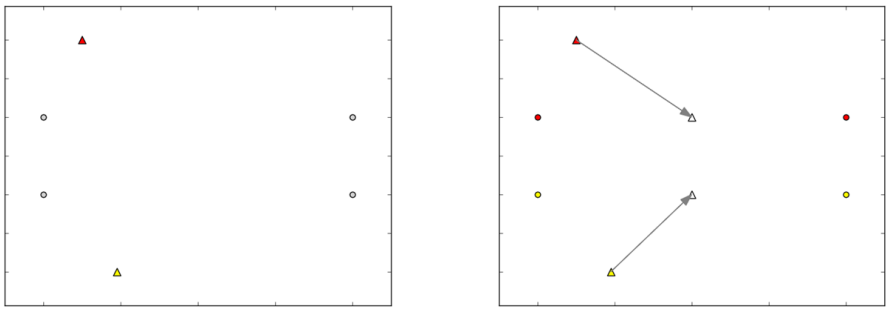


Figure 3.11 Local convergence of K-Means.

Also in different examples (see Figure 3.12), the start position of the centroids can determine the clusters that K-Means will find. Not all of these solutions are equally good, so running K-Means once might leave you with an under-performing clustering. To solve this issue, it is advisable to run K-Means a number of times (each time with another, random start for the centroids) and selecting the best result. To determine the

best clustering result, calculate the aggregate intra-cluster distance (for each cluster), and pick the one that yields the smallest distance in total. This is the result that has the most cohesive clusters.

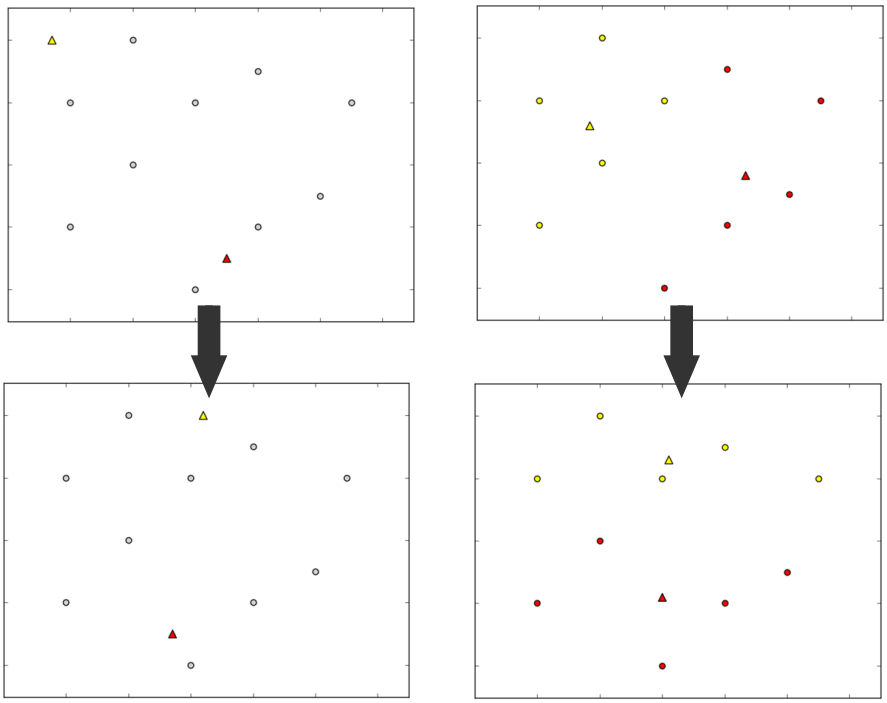


Figure 3.12 Converging to different clusters, depending on start position of centroids.

3.2.3 Choosing K

A final issue to solve, before using K-Means to determine clusters in your data, is how big K needs to be; that is, how many clusters are there actually in the data? Running K-Means with different numbers of centroids changes the performance of the clustering algorithm. In its extreme, running with a small K (e.g., say 2), leads to a very different aggregate intra-cluster distance for the clusters then when running with $K = n$ (K as big as the number of points in your data set). In the latter example, the intra-cluster distance would become 0, as every centroid will be placed on exactly one data point, and every cluster will only contain that single data point.

In some problems, determining the right K is rather easy, as one can derive the number of clusters from the problem itself. For instance, when trying to cluster digit recognition data (trying to use features to determine whether a hand-written digit is a 0, or a 1, ...); as there are 10 different digits, you expect 10 different clusters. In this case, the expected class labels determine the size of K.

In cases where we do not know a priori how many classes there might be, we have to find a good way to determine the best size of K. As mentioned, we can determine the performance of the K-Means clustering by calculating the aggregate intra-cluster distance. A good measure here is using aggregated squared distances (as it removes the need to take care of negative values). The aggregate intra-cluster

distance is calculated by the following formula: $V = \sum_j^k \sum_{x_i \to c_j}^n (c_j - x_i)^2$. This means as much as: for every cluster j, we calculate the squared distance of each point in that cluster x_i to the centroid of that cluster c_j.

Now, we could run K-Means several times with different values for K, and calculate the aggregate intra-cluster distance for each run, but this does not provide us with enough information about which K to choose. As mentioned above, when every point in your data has its own centroid/cluster, the value of V will be minimal (i.e., 0), however, this is a rather severe overfitting of the algorithm (see Aside 3.1). Basically, we have made every point in our training set its own cluster and it does not say anything about shared properties/attributes among the data that could be used to determine in which class a new observation should be placed; that is, there is no generalisation possible in this case.

So, how do we determine the correct size of K. It should be bigger than 1 (why would you never ever run K-Means with $K = 1$?), but smaller than n. We could try to split our data into two sets (training and validation), as we did with k-NN, and train on one set, and calculate the aggregate intra-cluster distance on the other; but this still has the problem that larger K perform better. This has to do with the fact that K-Means makes a Voronoi partition of the vector space, and a larger K means more (smaller) cells (more partitions) in the space. Every point of the validation set will always be in such a cell, and the smaller the cells, the smaller the distance to the centroid (of that cell). Hence, it leaves us with the same problem that we already had, $K = n$ performs best (but is not the best solution!).

The only way to correctly determine the right size of K is by doing a few runs with different values for K, calculating the aggregate intra-cluster distance of each run, and plotting these in a graph. Such a graph is shown in Figure 3.13. This kind of plot is called a *scree plot* due to its resemblance of a scree (see right side of Figure 3.13). It takes a deep dive, at the left-hand side of the plot, and suddenly eases to near horizontal at some point.

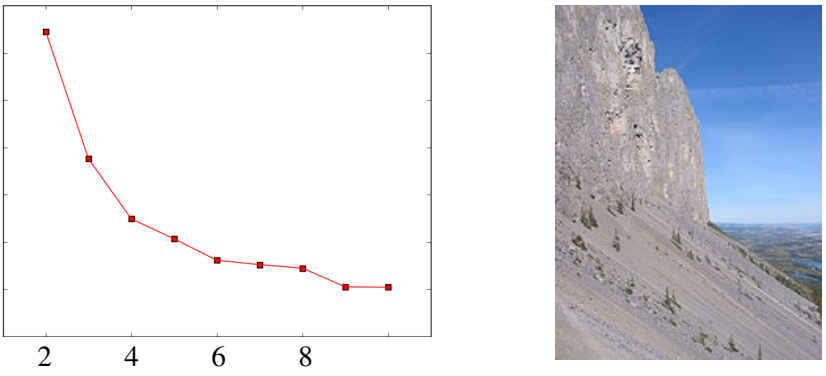


Figure 3.13 A Scree Plot for our earlier example (left) and a *scree* (cliff) that gave the plot its name (right).

The scree plot tells us something rather important. When increasing the size of K for the smaller values of K (e.g., switching from $K = 2$ to $K = 3$, etc.) a rather large decrease is achieved in the aggregate intra-cluster distance (on the vertical axis of the plot in Figure 3.13). On the other hand, for the larger values of K (e.g., moving

from $K = 9$ to $K = 10$), almost no decrease is achieved. The scale on the y-axis is in hundreds or thousands (does not really matter), so the increase from $K = 2$ to $K = 3$ leads to a decrease of almost $2,500$, whereas the increase from $K = 9$ to $K = 10$ is almost 0. This shows that a increase to $K = 3$ leads to a much better result (compared to $K = 2$), yet an increase of $K = 10$ has not such an effect when compared to $K = 9$.

It now comes down to a balance between *good enough* and *generality*; increasing K while the decrease in the aggregate intra-cluster distance is significant makes sense, increasing K when the improvement of the aggregate intra-cluster distance is only minimal hurts the generality of the solution (i.e., you are overfitting to the training data). This means that the 'best' K is the point where the steep drop ends and the 'flatness' begins (or, so to say, where the 'mountain' ends and the 'rubble' begins). This point can be visually identified (in the Figure above, this is at $K = 5$). If you prefer, you could do this mathematically, by maximising the second derivative of the scree plot[7].

3.3 Exercises

In the following exercises we are going to program our own implementation of the k-NN and K-Means algorithms. There is a dataset available on the electronic learning environment (ELE): `dataset1.csv`. This dataset includes meteorological data (weather data) from the weather station at De Bilt for the year 2000. The dataset is part of the freely available weather data gathered by the KNMI since 1901, for several weather stations in The Netherlands[8].

The data that we are using has 11 attributes:

- `YYYYMMDD`: date in year, months, days;
- `FG`: day average windspeed (in 0.1 m/s);
- `TG`: day average temperature (in 0.1 degrees Celsius);
- `TN`: minimum temperature of day (in 0.1 degrees Celsius);
- `TX`: maximum temperature of day (in 0.1 degrees Celsius);
- `SQ`: amount of sunshine that day (in 0.1 hours); -1 for less than 0.05 hours;
- `DR`: total time of precipitation (in 0.1 hours);
- `RH`: total sum of precipitation that day (in 0.1 mm); -1 for less than 0.05mm.

Data is entered in a comma-seperated format, which can be easily loaded into Python by using the `numpy`-module (the use of `numpy` is recommended as it supplies you with valuable additions to work with vectors and arrays). Use the following code-excerpt to import the data into Python:

```
import numpy as np

data = np.genfromtxt('dataset1.csv',
```

[7] While this is beyond the scope of this course, for the interested reader: the second derivative of a graph signifies the rate of change of that figure (the first derivative gives you the change, the second gives you the rate of change); the point where the second derivative is maximal is the point where the rate of changes is largest, which is when the steep decline stops, and the slower flatness begins.

[8] For interested readers, the complete dataset (all years, all weather stations) is available on `https://www.knmi.nl/nederland-nu/klimatologie/daggegevens`.

```
            delimiter=';',
            usecols=[1,2,3,4,5,6,7],
            converters=
               {5: lambda s: 0 if s == b"-1" else float(s),
                7: lambda s: 0 if s == b"-1" else float(s)})
```

Note that the `converters` are needed to transform the -1's in the columns `SQ` (hours sunshine) and `RH` (amount of rain) to a more useful `0`. This is a minor correction to make the data better represent what it means, improving the distance calculations between different data points.

We deliberately skip column `0` since that contains our date, which we will use to create the labels for the data. Instead of using the year, month, day as label, we will generalise to seasons, which we do as follows[9]:

```
dates = np.genfromtxt('dataset1.csv',
                      delimiter=';',
                      usecols=[0])
labels = []
for label in dates:
    if label < 20000301:
        labels.append('winter')
    elif 20000301 <= label < 20000601:
        labels.append('lente')
    elif 20000601 <= label < 20000901:
        labels.append('zomer')
    elif 20000901 <= label < 20001201:
        labels.append('herfst')
    else: # from 01-12 to end of year
        labels.append('winter')
```

The ELE also contains a validation set (`validation1.csv`), which contains validation data taken from 2001. Import the validation data in the same way as described above, and, like above, create the labels for the data[10].

Exercise 3.1 — k-Nearest Neighbours.
In this first exercise we use the `data`-array with weather data from 2000 to train a season-classifier by means of k-NN. Look at algorithm 3.1 to base your (bare metal) implementation of the classifier. The classifier will be able to determine the season based on the amount of rain, temperature, etc. of a given day.

Use the `validation`-array to determine how well your classifier functions for various k. What is the best value for k? You can determine the error of your classifier by letting it determine the season of the points in the validation set, and comparing that prediction to the label of that point (which you know). Express the

[9] Note that we use the meteorological seasons, not the astronomical ones, which makes sense as we are working with meteorological data.

[10] Note, the labeling scheme above needs to be changed since the dates are now starting with `2001` instead of `2000`.

error as a percentage of the validation-set that was classified incorrectly.

Next use your classifier to determine in the season of these following days (also available in `days.csv` on ELE):

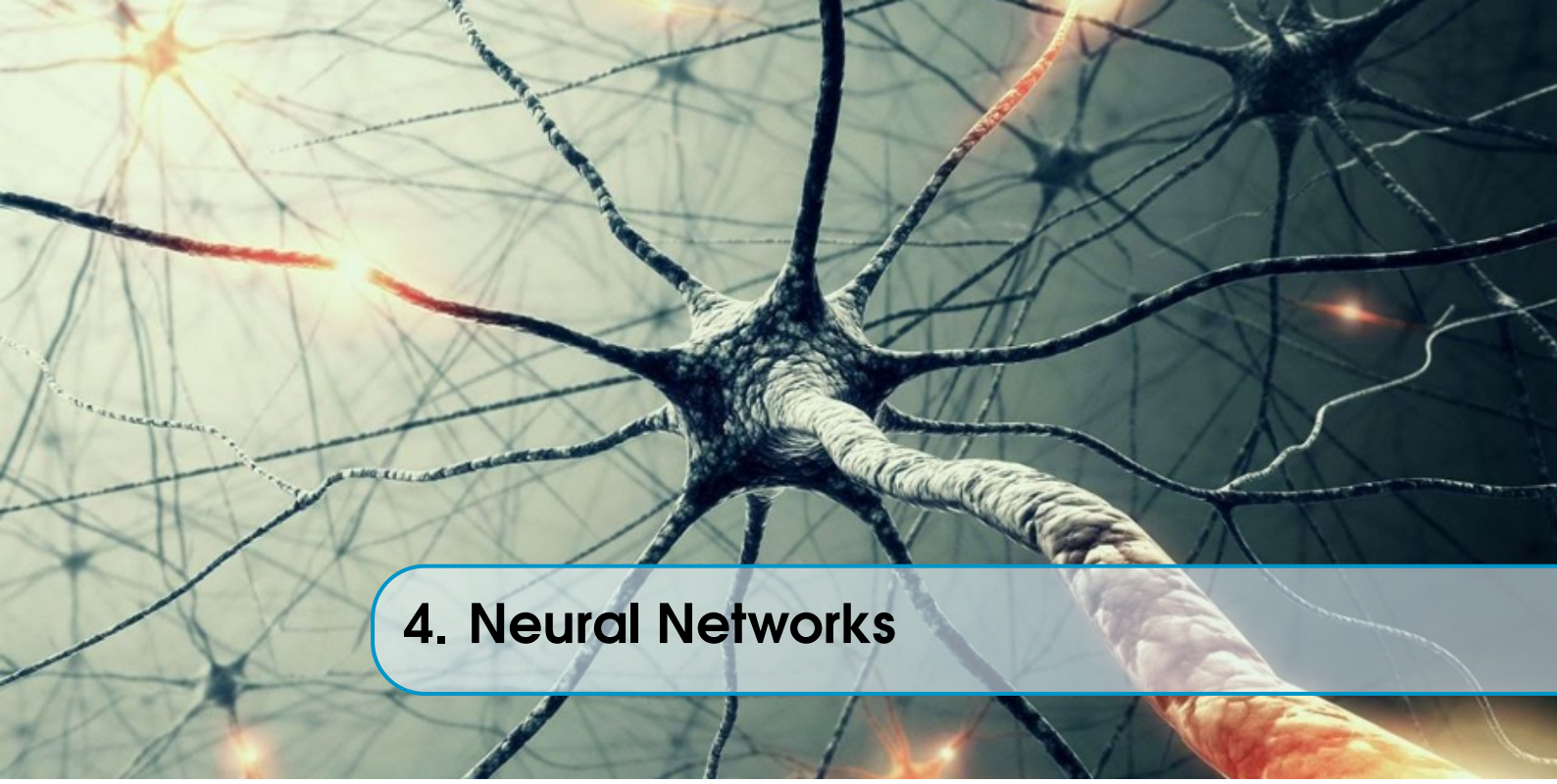
	FG	TG	TN	TX	SQ	DR	RH
40;52;2;102;103;0;0	4	5.2	0.2	10.2	10.3	0	0
25;48;-18;105;72;6;1	2.5	4.8	-1.8	10.5	7.2	0.6	1
23;121;56;150;25;18;18	2.3	12.1	5.6	15.0	2.5	1.8	1.8
27;229;146;308;130;0;0	2.7	22.9	14.6	30.8	13.0	0	0
41;65;27;123;95;0;0	4.1	6.5	2.7	12.3	9.5	0	0
46;162;100;225;127;0;0	4.6	16.2	10.0	22.5	12.7	0	0
23;-27;-41;-16;0;0;-1	2.3	-2.7	-4.1	-1.6	0	0	0.05
28;-78;-106;-39;67;0;0	2.8	-7.8	-10.6	-3.9	6.7	0	0
38;166;131;219;58;16;41	3.8	16.6	13.1	21.9	5.8	1.6	4.1

Next we are going to re-cluster the available data. This would show us whether we can indeed find 4 different groups in the data (one for each season). We again need to import the data and labels as we did above (we will need the labels later to verify whether the clusters made by K-Means indeed reflect the true season of the data points).

Exercise 3.2 — K-Means.
Write a bare metal K-Means clustering implementation, using the algorithm in 3.2, to cluster the data in `dataset1.csv`.

Use the maximum vote principle to cluster the data into the 4 different seasons. Use the true label of the data points to have them vote (within each cluster) as to what that clusters 'meaning' is (that is, a cluster with mostly 'winter' data points, is probably the 'winter'-cluster, etc.).

How many clusters can you (reliably) detect? Use the method described above (using a scree plot) to determine the optimal size of K for this dataset. What value for K is that? Did you expect that answer?



4. 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 4.1 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 4.1 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 math ematical 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.

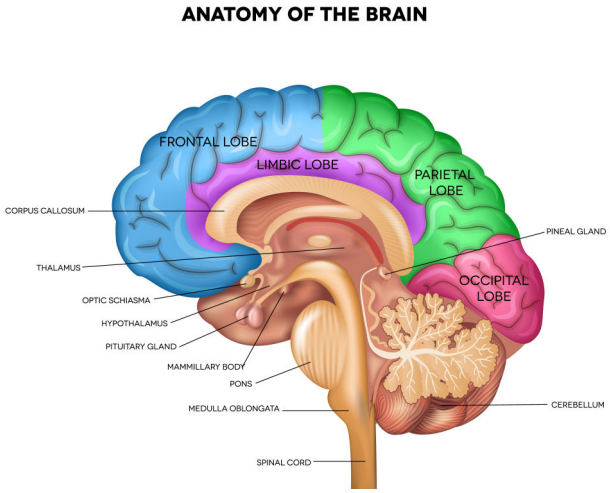
The recent successes and the media coverage of Google's AlphaGO ("Artificial Intelligence: Google's AlphaGo beats Go master Lee Se-dol", 2016, March 12) and projects like DeepArt (Wikipedia, 2016, May 15) make it seem like that every problem can now be tackled using neural networks. It certainly has 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 (Kasteleijn, 2017). 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.

4.1 (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 4.2
The brain



4.1.1 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 4.3). Every neuron consists

[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?

of a cell body (soma). Branching out from the cell body are a number of fibres called 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.

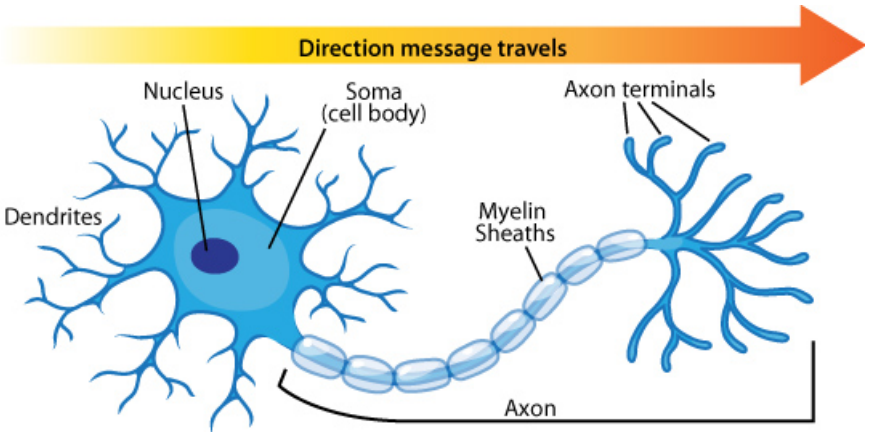


Figure 4.3 A neuron. In reality, an axon is about 100 times the diameter of the cell body.

.

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.

4.1.2 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 4.4 shows a representation of an artificial neuron. It has three inputs. In general it can

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

have more or less inputs $x_1, \ldots, x_n$.

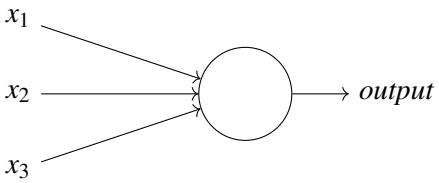


Figure 4.4 An artificial neuron with input x_1, x_2, x_3 and one output.

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:

$$output = g = \begin{cases} 0 & \text{if } \sum_{i=1}^{n} w_i x_i < t \\ 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq t \end{cases} \tag{4.1}$$

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 4.3, 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 4.5, 4.6 and 4.7). By assigning the weights and the threshold we decide on the behaviour.

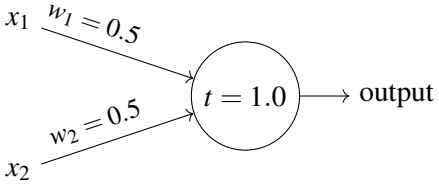


Figure 4.5 Appropriate weights and threshold to act as an AND-gate.

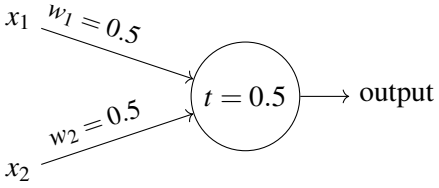


Figure 4.6 Appropriate weights and threshold to act as an OR-gate.

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?

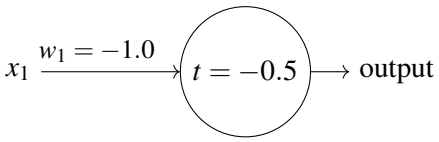


Figure 4.7 Appropriate weights and threshold to act as an INVERT-gate.

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 4.8. 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.

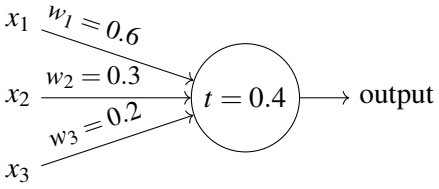


Figure 4.8 A possible perceptron to decide if we should go to a party.

4.1.3 The limits of perceptrons

We saw in the figures 4.5, 4.6 and 4.7 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 4.1 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 4.9 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 4.10).

4.2 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 4.11), we

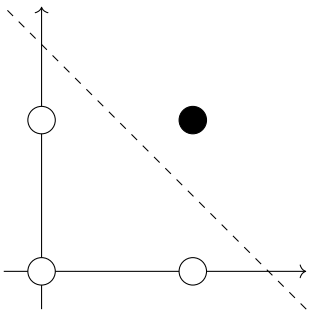


Figure 4.9 A two-dimensional plot of the decision making for an AND-gate.

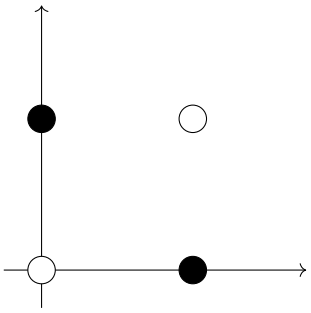


Figure 4.10 Plot of decision making for a XOR-gate. The white and black dots are not linear separable.

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 4.12.

4.2.1 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 4.12) 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

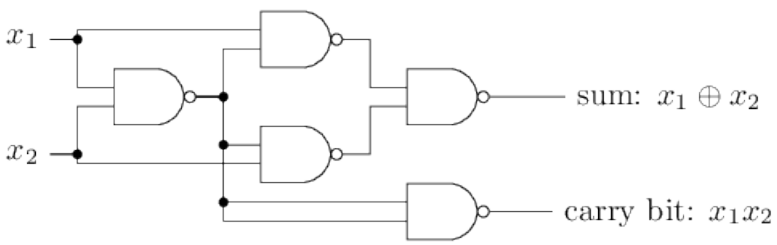


Figure 4.11 An adder built out of NAND-gates.

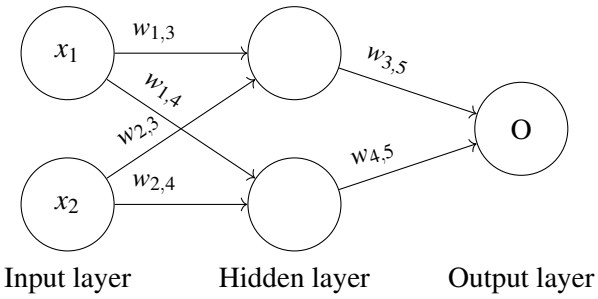


Figure 4.12 A small neural network. Weight $w_{i,j}$ is the weight of the link between unit i and unit j.

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.

4.2.2 Feed-forward network

The network seen in figure 4.12 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 4.12 would be:

$$\begin{aligned} O &= g(w_{3,5}a_3 + w_{4,5}a_4) \\ &= g(w_{3,5}g(w_{1,3}a_1 + w_{2,3}a_2) + w_{4,5}g(w_{1,4}a_1 + w_{2,4}a_2)) \end{aligned} \tag{4.2}$$

where g is the activation function (see section 4.1.2), $w_{i,j}$ 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 4.12 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.

4.3 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.

4.3.1 Introducing Bias

In section 4.1.2 we described the working of a perceptron in equation 4.1 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, $\vec{w} \cdot \vec{x} \equiv \sum_{i=1}^{n} w_i x_i$, where $\vec{w}$ and $\vec{x}$ are *vectors* whose components are the weights and inputs, respectively[5]. 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: $b \equiv -threshold$. Using the bias instead of the threshold, the perceptron activation function, g, can be rewritten:

$$output = g = \begin{cases} 0 & \text{if } \vec{w} \cdot \vec{x} + b < 0 \\ 1 & \text{if } \vec{w} \cdot \vec{x} + b \geq 0 \end{cases} \tag{4.3}$$

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 4.3 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,:

$$output = g = \begin{cases} 0 & \text{if } \vec{w} \cdot \vec{x} < 0 \\ 1 & \text{if } \vec{w} \cdot \vec{x} \geq 0 \end{cases} \tag{4.4}$$

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, and x_0 is always -1.

This change from threshold to bias simplifies the calculations needed in neural networks, as now only vector multiplications are required. Later, in Section 5, this means that we can more easily offload the neural network calculations to the GPU for training.

4.3.2 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

[5] The $\equiv$ means that what is on the left side is equivalent to what is on the right side. In other words, $\vec{w} \cdot \vec{x}$ is the same as $\sum_{i=1}^{n} w_i x_i$.

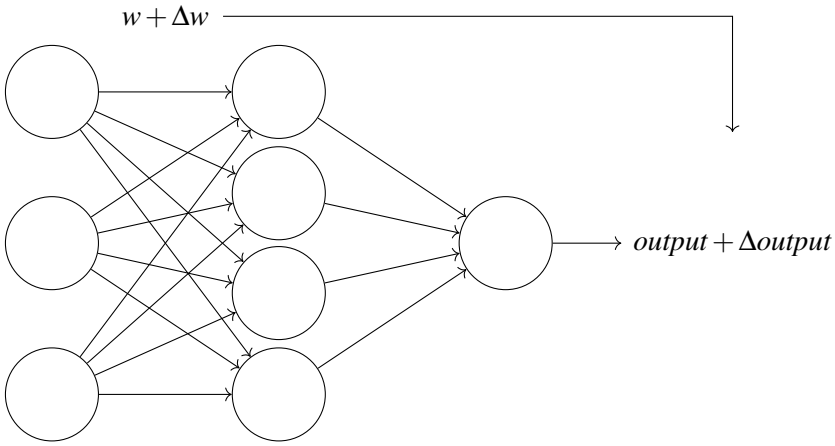


Figure 4.13 A small neural network. To be able to have an algorithm that can train a neural network, we want a small change in the weights, w, results in a small change in the output.

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 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 4.13 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, $\vec{w}$,

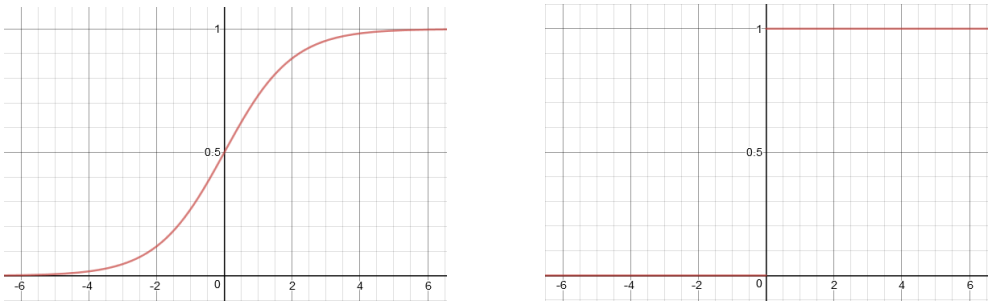


Figure 4.14 A plot of the sigmoid function (left) and step-function (right).

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

$$output = g = \sigma(\vec{w} \cdot \vec{x}) \tag{4.5}$$

where σ is called the *sigmoid function*[6], and is defined by:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{4.6}$$

Figure 4.14 (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 4.14 (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.

4.4 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.

4.4.1 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

[6] 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 4.15 A selection of the MNIST training set of handwritten digits.

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

$$C(\vec{w}) = MSE = \frac{1}{2n}\sum_{i=0}^{n} |\vec{y}_i - \vec{a}(\vec{x}_i)|^2 \tag{4.7}$$

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] [8]

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[9].

[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] 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, see Aside 3.2 on page 29) 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.

[9] For some readers the questions might rise why we try to minimise the quadratic cost function, as we

4.4.2 Gradient descent

In section 4.4.1, 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 4.16. 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.

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[10] over each weight. The collection of the gradients is called the gradient vector, ∇C:

$$\nabla C = \left(\frac{\partial C}{\partial w_1}, \ldots, \frac{\partial C}{\partial w_n}\right)^T \tag{4.8}$$

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:

$$\vec{w}' = \vec{w} - \Delta\vec{w} = \vec{w} - \eta\nabla C \tag{4.9}$$

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 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.

[10] 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`.

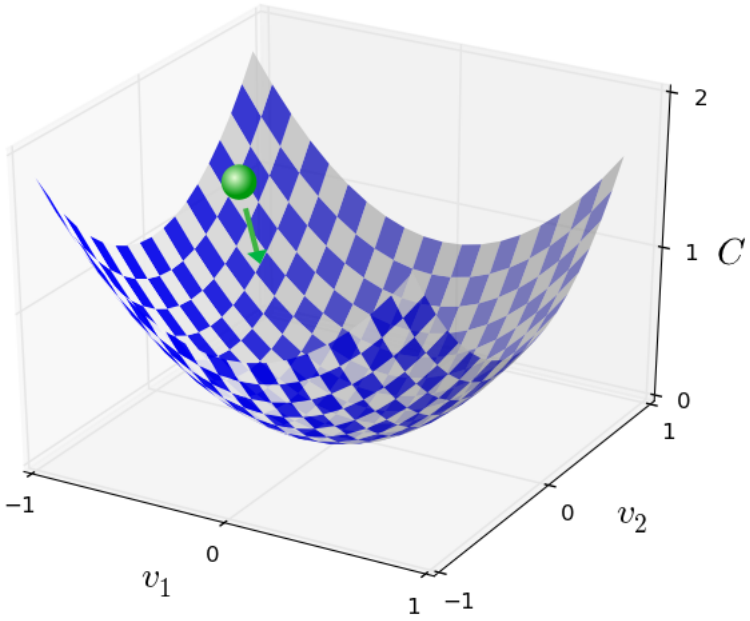


Figure 4.16 Gradient descent can be seen as taking small steps in the direction a ball would roll, until you reach a minimum.

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:

$$w_i' = w_i - \Delta w_i = w_i - \eta \frac{\partial C}{\partial w_i} \tag{4.10}$$

4.4.3 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 4.7). 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:

$$C = \frac{1}{2} |\vec{y} - \vec{a}(\vec{x})|^2 \tag{4.11}$$

Now we can use gradient descent to reduce the squared error[11] by calculating the partial derivative of C with respect to each weight and updating the weight:

$$\begin{aligned} w_i' &= w_i - \eta \frac{\partial C}{\partial w_i} \\ &= w_i - \eta \frac{\partial (\frac{1}{2} |\vec{y} - \vec{a}(\vec{x})|^2)}{\partial w_i} \end{aligned} \tag{4.12}$$

In a single layer network (see figure 4.17) 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:

$$w_{j,k} = w_{j,k} - \Delta w_{j,k} = w_{j,k} + \eta a_j g'(in_k)(y_k - a_k) \tag{4.13}$$

where a_j is the activation of input node j, g' is the derivative of the activation function [12], 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[13]. 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.

4.4.4 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:

$$\Delta_k = g'(in_k)(y_k - a_k) \tag{4.14}$$

$$w_{j,k} = w'_{j,k} + \eta a_j \Delta_k \tag{4.15}$$

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:

$$w_{i,j} = w'_{i,j} + \eta a_i \Delta_j \tag{4.16}$$

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

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

[13] 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.

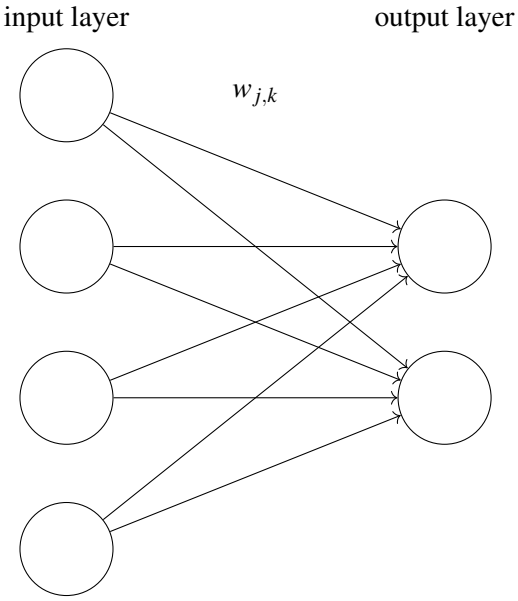


Figure 4.17 A single layer neural network

.

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:

$$\Delta_j = g'(in_j) \sum_p w_{j,p} \Delta_p \tag{4.17}$$

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 4.14 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.

4.5 Exercises

Exercise 4.1 — NOR-Gate.
Determine weights and a threshold for a perceptron that would act as a NOR-gate with three inputs.

Exercise 4.2 — Neural ADDER.
Make an adder out of perceptrons. Draw the network and give the corresponding weights and biases.

Exercise 4.3 — 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 4.1 and the network of exercise 4.2.

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 4.1.

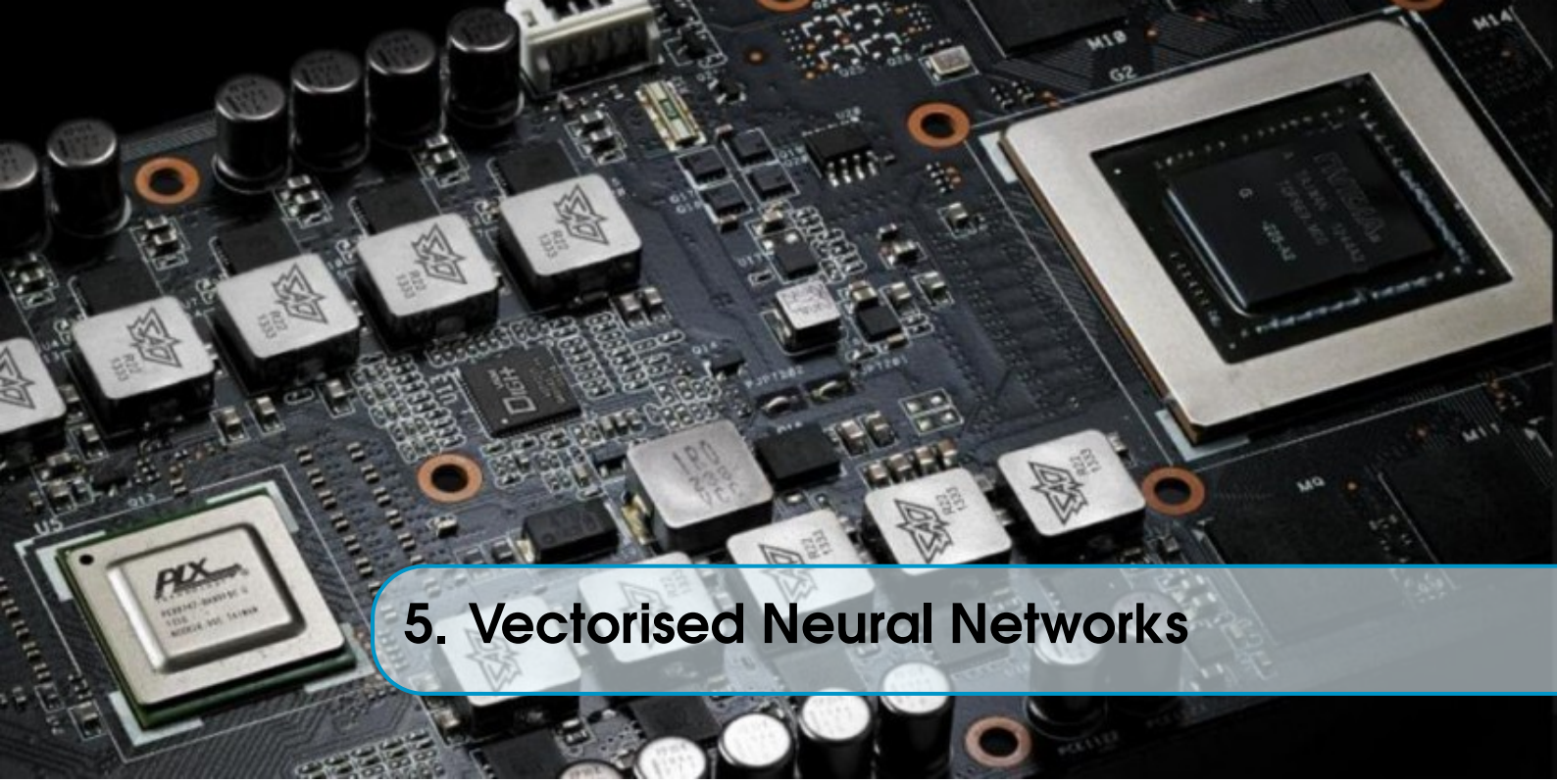
C) Backpropagation
Add backpropagation to your program. Create the XOR network from the sheets. Initialize the weights with random values. Train the network.

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`



5. Vectorised Neural Networks

Though the idea of Neural Networks is older than you might think (refer Chapter 2), 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.

5.1 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.

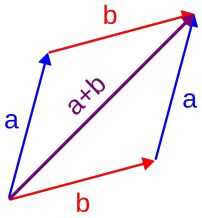
5.1.1 Vectors and Vector Spaces

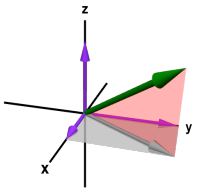
Vectors are a mathematical object for which three common intuitions exist. We look at each in turn, and then turn our attention to how these intuitions are helpful.

[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 5.6 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 5.1. 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 5.1 — 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 mathmetical intuition is more abstract, and mainly serves to establish a set of rules for managing vectors.

Exercise 5.1 — 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 5.2 — 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.

5.1.2 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.

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

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:

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

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 5.1 — 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 5.3 — 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:

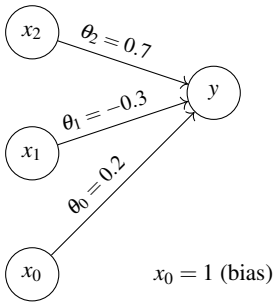
a. $A\mathbf{u}$
b. $B(A\mathbf{u})$
c. $(BA)\mathbf{u}$
d. $\mathbf{u}^T\mathbf{v}$
e. $\mathbf{v}^T\mathbf{u}$
f. $(A\mathbf{v}+B\mathbf{u})\mathbf{v}^T$

5.2 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 4.4.2.

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.

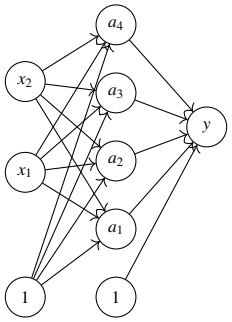
By taking the inner product of the input vector $\mathbf{x}$ and the weight vector θ, we can use a single operation to multiply each input value with its associated weight, and sum the result. So $y = \sigma(\theta_1 x_1 + \theta_2 x_2)$ becomes $y = \sigma\langle\theta|\mathbf{x}\rangle$[5].

[5]Remember, σ represents the *sigmoid activation function*, as introduced in Section 4.3.

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

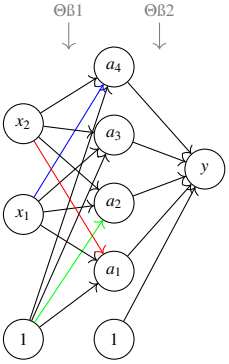
We now consider a more complicated network, by adding a single hidden layer consisting of 4 neurons.



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 $\mathbf{a} \leftarrow \begin{bmatrix} 1 & \mathbf{a} \end{bmatrix}$.

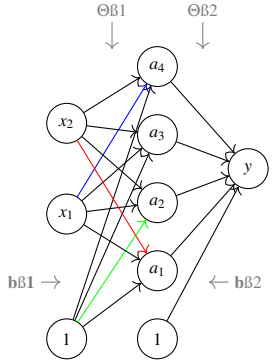
Each value in $\Theta ß1$ matches a weight between the input and hidden layer; $\Theta ß1_{ij}$ represents the weight from x_j to a_i. In this case, rows count from 1, columns count from 0 with column 0 representing the bias. Each value in $\Theta ß2$ matches a weight between the hidden layer and the 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 5.4 — Bias Representation.
Confirm the equivalence of both bias representations by calculating the results in both instances.
Bonus: Prove both bias representations are the same.

5.2.1 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 5.5 — 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.

5.2.2 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 $J = \langle \mathbf{y} - \mathbf{y}'|\mathbf{y} - \mathbf{y}'\rangle \quad \text{or} \quad J = (\mathbf{y} - \mathbf{y}')^2$.

5.2.3 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 5.2.1, 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).

$$X \in \mathbb{R}^{1024\times 1}, Aß1 \in \mathbb{R}^{42\times 1}, Aß2 \in \mathbb{R}^{28\times 1}, Y \in \mathbb{R}^{12\times 1}$$

$$\Theta ß1 \in \mathbb{R}^{42\times 1024}, \Theta ß2 \in \mathbb{R}^{28\times 42}, \Theta ß3 \in \mathbb{R}^{12\times 28}$$
$$Bß1 \in \mathbb{R}^{42\times 1}, Bß2 \in \mathbb{R}^{28\times 1}, Bß3 \in \mathbb{R}^{12\times 1}$$

$$Aß1 = \sigma(\Theta ß1X + Bß1)$$
$$Aß2 = \sigma(\Theta ß2Aß1 + Bß2)$$
$$Y = \sigma(\Theta ß3Aß2 + Bß3)$$

Exercise 5.6 — Vectors to Matrices.
Verify that the example given above is equivalent to the vector-based example from Section 5.2.1.

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:

$$X \in \mathbb{R}^{1024\times n}, Aß1 \in \mathbb{R}^{42\times n}, Aß2 \in \mathbb{R}^{28\times n}, Y \in \mathbb{R}^{12\times n}$$
$$Bß1 \in \mathbb{R}^{42\times n}, Bß2 \in \mathbb{R}^{28\times n}, Bß3 \in \mathbb{R}^{12\times n}$$

5.3 Learning

Now that we know how to represent our network, we can look towards learning. Like in the object-oriented representation from Section 4.4.2, 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 5.7 — 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)

5.4 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.

5.4.1 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 5.2), 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 5.4.3, after we have familiarised ourselves with numpy's notation for matrices.

Aside 5.2 — 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.

5.4.2 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.

5.4.3 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.

5.4.4 Example: Perceptron

We can now return to our first example (Section 5.2): 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))
```

5.4.5 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
```

5.5 Exercises

Exercise 5.8 — NOR-Gate.
As a warm-up exercise, we implement the NOR-gate from Chapter 4 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 5.9 — 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.

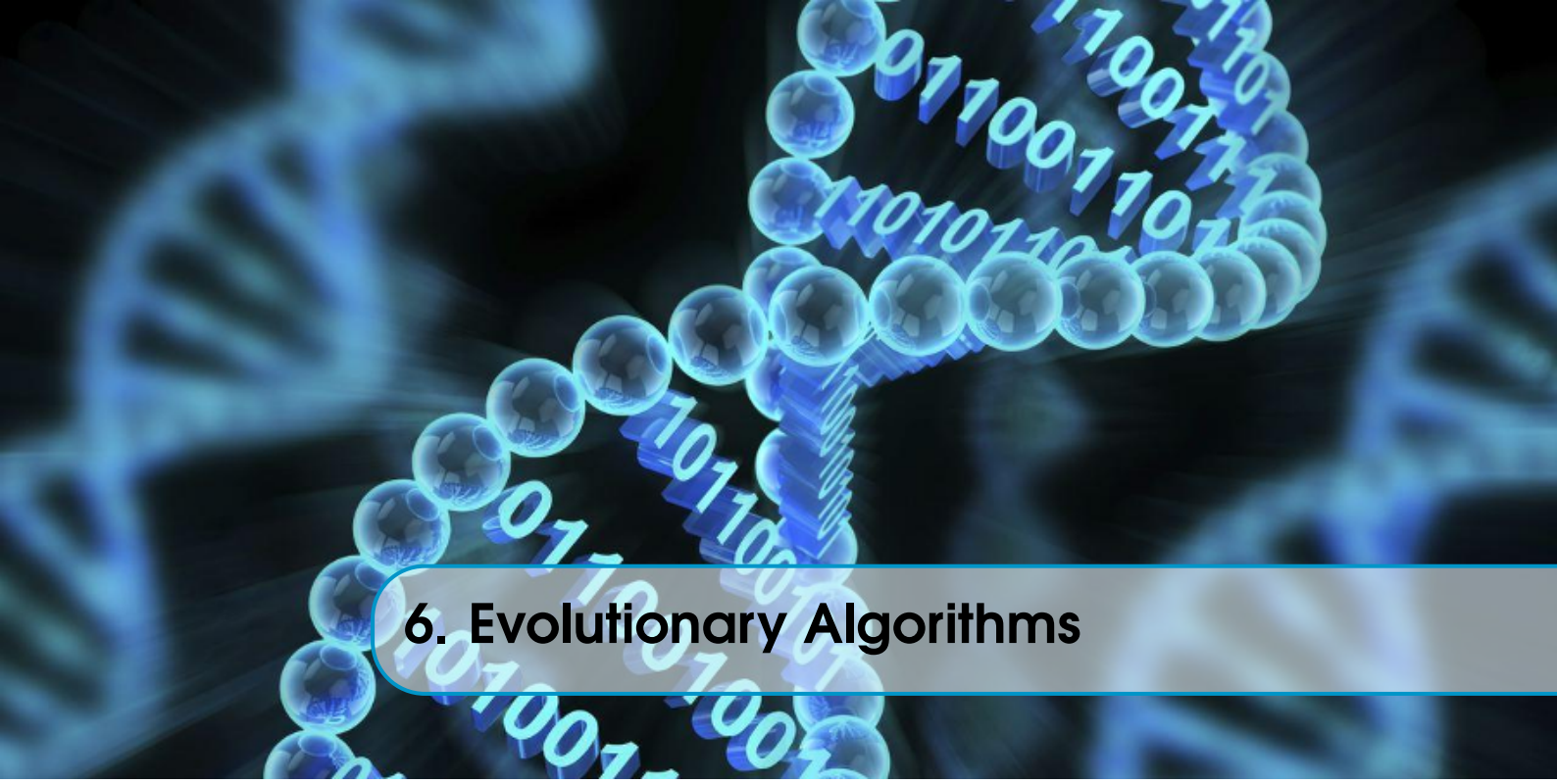
[b] See `https://www.tensorflow.org` for details on TensorFlow. You are allowed to use the Convolutional implementation of appendix B as a reference, but are required to implement a "normal" neural network instead.

5.6 Appendix: Notation Overview

- y is a number (scalar)
- $\mathbf{b}$ is a vector
- $\mathbf{b}_1$ is a number in b (a scalar)
- $\mathbf{b\beta 2}$ is a different vector than $\mathbf{b\beta 1}$ and $\mathbf{b\beta 3}$; 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

5.6.1 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



6. Evolutionary Algorithms

Finally, we turn our eyes to some of the coolest sounding AI techniques: *Evolutionary* and *Genetic Algorithms*. In the previous (e.g., see Chapter 4), we have already seen that AI sometimes tries to mimic elements of real life (like our brains) to solve issues in Computer Science. Evolutionary algorithms is similar in that sense as the ideas behind it are inspired by biological evolution (see Aside 6.1).

Evolutionary algorithms, and even wider, evolutionary computing, are a collection of optimisation algorithms, including[1]:

- **Genetic algorithm** (GA): the most popular type of EA, which seeks the solution of a problem in the form of strings of numbers, by applying operators such as recombination and mutations. Often used for optimisation problems.
- **Genetic programming** (GP): a specific implementation of GA from the field of AI that creates optimised forms of computer programs. Given a number of (basic) program elements, the algorithm tries to combine these such that a suitable (well-performing) program is created for a known problem.
- **Evolutionary programming**: similar to GP, but the structure of the program is fixed and its numerical parameters are allowed to evolve.
- **Gene expression programming**: like GP, GEP evolves computer programs but it explores a genotype-phenotype system, where computer programs of different sizes are encoded in linear chromosomes of fixed length.
- **Evolution strategy**: algorithm that works with vectors of real numbers as representations of solutions and typically uses self-adaptive mutation rates.
- **Neuro-evolution**: similar to GP but the genomes represent artificial neural networks by describing structure and connection weights.

In relation to the previous material of this course, it should not surprise you that we are trying to achieve neuro-evolution; that is, create an optimisation algorithm that

[1] These following techniques largely differ in gene encoding and implementation details, but are considered a part of the same set of evolution inspired optimisation algorithms.

can determine the best structure and weights for our neural networks. Before we get to the evolution of neural networks (see Section 6.2), we first have to explain what genetic algorithms are (see Section 6.1) and how they function.

6.1 Genetic algorithms

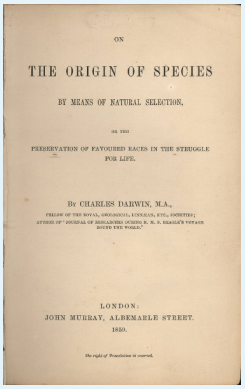
Genetic algorithms are a metaheuristic from the field of computer science (more specifically, operations research). Being inspired by the concepts of biological evolution, they belong to the larger class of *evolutionary algorithms*. While genetic and evolutionary algorithms are considered a part of computer science, the study of the use of evolution inspired algorithms, broadly called *evolutionary computing*, is in fact a sub-field of Artificial Intelligence.

A *metaheuristic* is a high-level procedure designed to find, generate, or select a heuristic[2] (partial search algorithm) that may provide a sufficiently good solution to an optimisation problem, especially with incomplete or imperfect information or limited computation capacity. Metaheuristics sample a set of solutions which is too large to be completely sampled; that is, they exploratively search a solution space without fully searching through that space to find a good enough solution to the problem. Compared to optimisation algorithms and iterative methods (e.g., binary search), metaheuristics do not guarantee that a globally optimal solution can be found.

Evolutionary algorithms (and therefore also genetic algorithms) use mechanisms inspired by evolution, such as *reproduction*, *mutation*, *recombination*, and *selection* to solve global optimisation or search problems.

Aside 6.1 — Principles of evolution: *On the Origin of Species*.
In the mid-19th century, Charles Darwin formulated the scientific theory of evolution by natural selection, published in his book *On the Origin of Species* (1859). Evolution by natural selection is a process demonstrated by the observation that more offspring are produced than can possibly survive, along with three facts about populations:

1. traits vary among individuals with respect to morphology, physiology, and behaviour (phenotypic variation);
2. different traits confer different rates of survival and reproduction (differential fitness); and
3. traits can be passed from generation to generation (heritability of fitness).


ON
THE ORIGIN OF SPECIES
BY MEANS OF NATURAL SELECTION,
OR THE
PRESERVATION OF FAVOURED RACES IN THE STRUGGLE
FOR LIFE.
By CHARLES DARWIN, M.A.,
LONDON:
JOHN MURRAY, ALBEMARLE STREET.
1859.

Thus, in successive generations members of a population are replaced by progeny of parents better adapted to survive and reproduce in the biophysical environment in which natural selection takes place.

While not truly similar to natural selection as the goal of the selection is often

[2] A heuristic is a search method that uses knowledge of the problem to avoid having to search the entire solution space. While this provides efficiency (solutions can be found faster), the disadvantage is that it is not guaranteed that the (overall) best solution will be found.

predetermined in evolutionary algorithms, they adhere to the same principles of evolution: survival of traits is pertained by the implication of that trait on the overall fitness (that is, characteristics that make a (sub-)population perform better, survive longer), and important traits are passed from generation to generation (that is, children look like a combination of their parents, inheriting traits from both of them).

In principle, evolutionary algorithms function in this rudimentary manner:

1. Generation of an initial population of individuals (often at random);
2. Evaluation of the fitness of each individual in that population;
3. Repetition of the following re-generational steps until termination (or until a pre-defined maximal number of generations):
   (a) Select the best-fit individuals for reproduction;
   (b) Breed new individuals through crossover and mutation operations to give birth to offspring.
   (c) Evaluate the individual fitness of new individuals.
   (d) Replace least-fit population with new individuals.

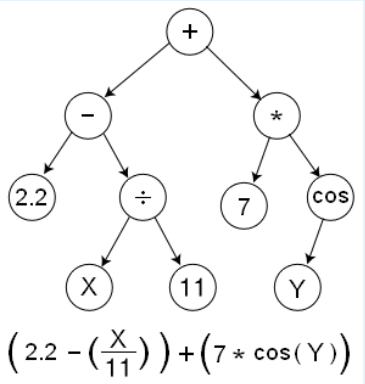
Aside 6.2 — What's in a name.
The use of Evolutionary principles for automated problem solving originated in the 1950s. It was not until the 1960s that three distinct interpretations of this idea started to be developed in three different places.

Evolutionary programming was introduced by Lawrence J. Fogel in the US, while John Henry Holland called his method a *genetic algorithm*. In Germany Ingo Rechenberg and Hans-Paul Schwefel introduced *evolution strategies*. These areas developed separately for about 15 years. From the early nineties on they are unified as different representatives ("dialects") of one technology, called *evolutionary computing*. Also in the early nineties, a fourth stream following the general ideas had emerged – *genetic programming*. Since the 1990s, nature-inspired algorithms are becoming an increasingly significant part of evolutionary computation.

Figure 6.1 2006 NASA ST5 spacecraft antenna, found by an evolutionary computer design program to create the radiation pattern.

In the following we look at each of these steps in turn (with a running example) to show how genetic algorithms are used.

Aside 6.3 — Genetic programming.
Genetic programming (GP) is an AI technique whereby computer programs are encoded as a set of genes that are then modified (evolved) using an evolutionary algorithm (often a genetic algorithm). The results are computer programs able to perform well in a predefined task.
The methods used to encode a computer program in an artificial chromosome and to evaluate its fitness with respect to the predefined task differ between techniques and is still a subject of research. As an example, a computer program could be represented as a tree structure (favouring functional programming languages), which can be evaluated in a recursive way, making mathematical expressions easy to evolve and evaluate.



The genetic algorithm is then employed to optimise the structure and position of elements within the tree, with the aim of creating the fittest program to solve the problem.

Picture by Wikipedia.

6.1.1 Optimisation

In mathematics and computer science, optimisation is the selection of a best element (with regard to some criterion) from some set of available alternatives. In the simplest case, an optimisation problem consists of maximising or minimising a real function by systematically choosing input values from within an allowed set and computing the value of the function. More generally, optimisation includes finding "best available" values of some objective function given a defined domain (or input), including a variety of different types of objective functions and different types of domains.

The purpose of an optimisation problem is to find a solution $s_{max} \in S$, that is find the maximal solution s_{max} in a given solution space S. Because the solution space is often too large to search systematically (that is, select each solution individually and determine whether it is the best (so far)), heuristic searches are often applied. Genetic algorithms is one of these, but gradient descent, presented earlier in Section 4.4.2, is another[3]. A third optimisation technique, that is close to how genetic algorithms work, is (local) *hill climbing*.

As solution spaces are often quite big, and as their shape is not uniform, it can be quite challenging to find a optimal value. Consider the solution space in Figure 6.2, if the fitness function (that is, the function that determines the score of a specific point) was known on forehand, it would be easy to determine where the maximum value would be (at least, it would be for us, as we can try to determine where the peak of the figure is). Unfortunately, the exact shape of the fitness function is not so

[3]Note that gradient descent is doing the exact opposite by trying to minimise the value of s; that is, find the lowest point s_{min} in the solution space S. The principle is the same, however.

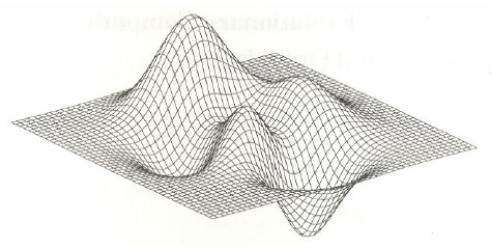


Figure 6.2 A fitness landscape.

obvious[4], and only local information is known (we only know whether the points next to us are higher or lower), it can be very difficult to find the absolute maximum. In principle, this is how local hill climbing works, however. Local hill climbing starts a some random point and tries (randomly) its neighbours to see if one of those has a higher fitness than itself.

Algorithm 6.1 — Local hill climbing.

- Generate (or select) a starting solution s_0 at random, initialise time $t = 0$;
- Repeat until convergence:
  1. create s_{new} by changing s_t (take a step left, forward, right, backward);
  2. if the fitness of s_new is bigger than that of s_t, then $s_{t+1} = s_{new}$ (we take s_{new} as are next 'starting point');
  3. else $s_{t+1} = s_t$ (that is, we go back to our previous best point);
  4. $t = t + 1$ (increase time).

■

The most important aspect of the local hill climbing algorithm is *changing* s_t. If the steps taken are too small, progress will be very slow; but if the steps are too large, optimal values might be overstepped and never found.

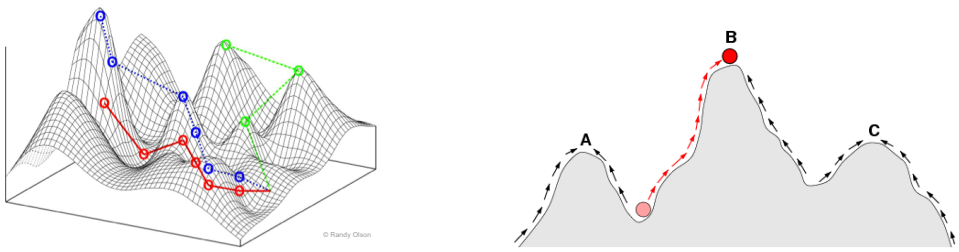


Figure 6.3 Hill climbing in 3D (left) and 2D (right).

An important disadvantage of hill climbing is that the algorithm is prone to getting stuck in local optima (that is, a peak that is smaller than the optimal peak, but high enough that a step in either direction does not affect an immediate improvement). Consider for instance the red and green paths in the left-hand side of Figure 6.3; or the paths *A* and *C* in the right-hand side of Figure 6.3. It is therefore important to run the algorithm multiple times, with different starting points.

On an abstract level, genetic algorithms function in similar matter. By selecting the best traits in individuals, genetic algorithms try to only take the steps that are

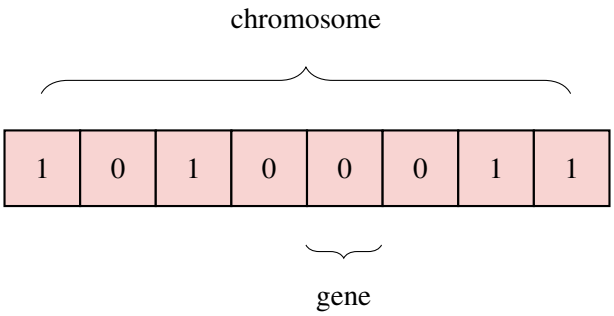
[4]Not to mention that most fitness functions are not 2-dimensional in shape; it is very difficult to imagine a 20-dimension shape and determine where the optimal value is.

'uphill'. The biggest distinction between genetic algorithms and the hill climbing algorithm is, however, that genetic algorithms use many different starting points (thus trying to circumvent the local optima problem of hill climbing) and can, given correct parameter choices, escape local optima.

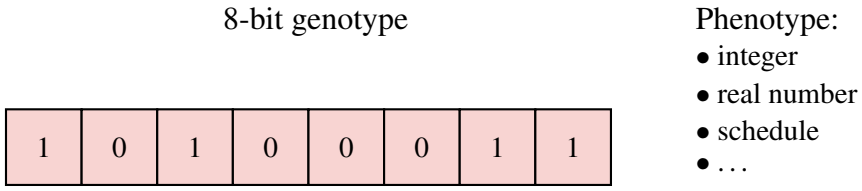
6.1.2 Problem definition

To use a genetic algorithm on a optimisation problem, we have to start with a representation of the problem and its solutions. Candidate solutions in a genetic algorithm are called *individuals* or *phenotypes*. The representation of an individual is called a *genotype* (or genome), and there are many options to encode the individuals into a genotype. The encoding must be done in such a way that we can solve the problem. We have to consider the evaluation method used to determine the fitness of the individuals.

Genome representations can range from binary strings to arrays (or lists) of integer values or characters (or even program elements, in the case of genetic programming). An example of a binary representation is shown below:



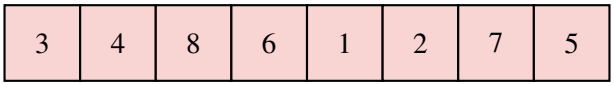
Given a representation of our genome, we have all the freedom to translate that into a phenotype. The genotype is the representation that is used in the search space (solution space); the phenotype is what is being evaluated.



An example of an integer phenotype (given the genotype in the figure above) would be using the typical binary representation; 163. If we would want to represent numbers between 2.5 and 20.5 (with maximal 256 values), the phenotype would be $2.5 + \frac{163}{256}(20.5 - 2.5) = 13.9609$.

In some cases, for instance in parameter optimisation, we would rather base our genotype directly on real numbers. In this case we use a tuple (or list) of numbers, rather than a binary representation. For instance, to encode the order in which cities are visited in a Travelling Salesman Problem (TSP) can be done as follows[5]:

[5]This encoding is ambiguous; it could either mean that the position in the genome determines the



Initialisation of the genetic algorithm is then done by generating a number of random individuals; this group of individuals is typically called a *population*. Initialisation can be done in two different manners:

- chosen randomly (uniform) from the entire search space:
  - binary strings: equal chance of generating a 0 or a 1 for every gene of every genome;
  - real numbers: uniform selected from a closed interval (does not work well with open intervals);
- by using results from previous populations or by using heuristics:
  - advantage: less iterations required to reach convergence/optimum;
  - disadvantage: possible loss of genetic diversity;
  - disadvantage: can introduce a initial bias, that is hard to escape.

Example

In our running example we use a rather simple problem: trying to create a list of N numbers that equal to X when summed together. If we set $N = 5$ and $X = 200$, then all of these would be appropriate solutions (and many more):

```
lst = [40,40,40,40,40]
lst = [50,50,50,25,25]
lst = [200,0,0,0,0]
```

First we have to initialise our genetic algorithm by creating a population of randomly generated individuals. In our current problem, each list of N numbers is an individual.

```
from random import randint

def individual(length, min, max):
    """
    Creates an individual for a population

    :param length: the number of values in the list
    :param min:    the minimum value in the list of values
    :param max:    the maximal value in the list of values
    :return:
    """
    return [ randint(min, max) for x in range(length) ]
```

And the population is then a collection of individuals:

```
def population(count, length, min, max):
```

position in the route (i.e., city 3 is visited first, then city 4, etc.), or the position in the genome determines the city, and the number indicates the position in the route (i.e., city 5 is visited first, then city 6, etc.). Either encoding would work, as long as the fitness function is defined correctly.

```
    """
    Create a number of individuals (i.e., a population).

    :param count:  the desired size of the population
    :param length: the number of values per individual
    :param min:    the minimum in the individual's values
    :param max:    the maximal in the individual's values
    """
    return [ individual(length, min, max) \
             for x in range(count) ]
```

6.1.3 Evaluation

The evolution is an iterative process, with the population in each iteration called a *generation*. In each generation, the *fitness* of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. This *fitness function* determines the fitness of each of the individuals.

Determining the fitness of the individuals can be a costly process, especially when concerning real-life problems. To reduce the evaluation time, you could skip evaluating individuals that have not changed since last generation.

Evaluation can be done with subroutines, black-box simulators and even external processes (for instance, robots). The evaluation should not take too long to execute, as this counteracts the effectiveness of the genetic algorithm. For instance, when trying to determine the structure of a neural network, fitness can be determined by training each solution to see if it has the potential to solve the problem; but training should not be too long, since your genetic algorithm would run forever if you would train each (generated) individual for a couple of hours before determining its fitness.

Sometimes you encounter a problem where the phenotype has to cope with some particular requirement (for instance, no number in the TSP problem described above should occur twice in a genome). This could be solved in the genome, by ensuring that the requirement is never violated, or by adding a penalty to the fitness function.

Example, continued

In our simple problem, the fitness is a function of the distance between the sum of an individuals numbers and the target number X. We can implement the fitness function as follows:

```
from functools import reduce
from operator import add

def fitness(individual, target):
    """
    Determine the fitness of an individual. Lower is better.

    :param individual: the individual to evaluate
    :param target: the sum that we are aiming for (X)
```

```
    """
    sum = reduce(add, individual, 0)
    return abs(target-sum)
```

In this case our objective is to minimise the distance between the sum of an individual's numbers and the target value X. A positive correlation between the fitness and the fitness score might be preferable; in that case, a higher fitness (a better match) has a higher score. In our case, the best score is 0 and every worse score is higher, which might be counter-intuitive.

It might be helpful to create a function to determine the population's average fitness (to show whether we make progress in our evolution):

```
def grade(population, target):
    """
    Find average fitness for a population

    :param population: population to evaluate
    :param target: the value that we are aiming for (X)
    """
    summed = reduce(add, (fitness(x, target) \
             for x in population), 0)
    return summed / len(population)
```

Next we need a way to evolve our population, that is, to advance the population from one generation to the next.

6.1.4 Iterative improvement

The next step in genetic algorithms is where the magic happens. Consider a population of moose which are ruthlessly hunted by a pack of wolves. With each generation the weakest are eaten by the wolves, and then only the strongest moose are left to reproduce and have children. The idea of evolution is that the traits that help with survival (the best stamina, the best hooves for running, the strongest muscles, etc.) survive in the population and are passed to the next generation. The same principle is applied in genetic algorithms.

The more fit individuals of the population are selected in some manner from the current population, and each individual's genome is modified (recombined and possibly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm,

Mutation

Mutation is a genetic operator used to maintain genetic diversity between generations. It is analogous to biological mutations, and alters one or more gene values in a chromosome. Mutation can severely change the solution from the previous solution, thus introducing a new 'starting point' for the hill climbing like behaviour of the genetic algorithm. Remember, to ensure that the optimisation does not get stuck in local optima, we have to start in many different positions in the solution space; mutation accomplishes this (in a rather destructive manner). The mutation occurs

during evolution according to a user-definable mutation probability. While mutation is a strong counter against local optima, the mutation probability should not be set too high, or the search will turn into a primitive random search.

There are a number of important aspects to remember when using mutation:

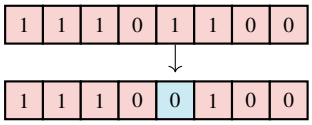
- at least 1 mutation operator should allow for searching the whole search space;
- the size of the mutation-step should be controllable (not too large);
- mutation should lead to valid chromosomes.

The classic mutation operator is a random flip of bits in the bit string representation. Every bit (gene) of every bit string (chromosome) has a small chance P_m to flip from a 0 to a 1, or vice versa. This works well for simple genotype-phenotype representations, but can be disruptive for more complex encodings (like floating point representations). In more restrictive gene encodings, like permutation problems (like the TSP problem mentioned earlier), mutations are rather implemented as swaps, inversions or scrambles.

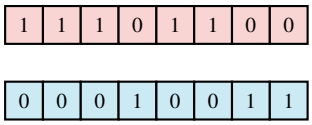
The main purpose of mutation in genetic algorithms is for preserving and introducing diversity. Mutation should allow the algorithm to avoid local optima by preventing the population of chromosomes from becoming too similar to each other, this slowing or even stopping evolution. This reasoning also explains why most genetic algorithms avoid only taking the fittest of the population in a generation but rather random (or semi-random) select the best to promote that genetic diversity.

Examples of mutation include the following.

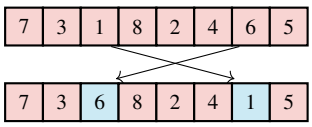
- Bit string mutation: flips one bit in a bit encoding.



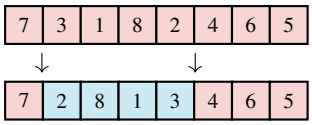
- Flip bit mutation: inverts all bits in a bit encoding.



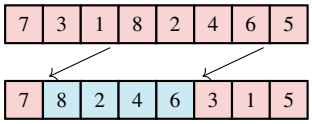
- Uniform mutation: used in integer representations. Mutated genes are changed with a uniform random value selected between the upper and lower bounds.
- Non-uniform mutation: the changes made to mutated genes decreases as generations progress. This keeps the population from stagnating in the early phases of evolution.
- Gaussian noise: more typical with integer representations. Mutated genes are changed with a Gaussian random value (that is, higher probability for small changes, low probability for large changes).
- Swap: used for ordering-specific representations; select two genes in a chromosome and swap their values.



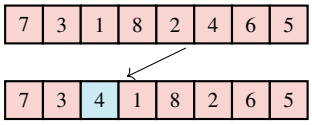
- Inversion: choose two genes and reverse order of those in between.



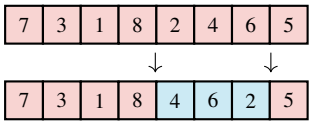
- Displacement: choose a portion of the chromosome (a section between two genes), and move the entire section to another part of the chromosome.



- Insertion: choose and remove a random gene from the chromosome and insert in some other location.



- Scramble: choose two genes in the chromosome and randomly shuffle all genes in between.



Crossover

Recombination in genetic algorithms is done by means of the crossover operator. Crossover combines chromosomes of the parents (typically two, but crossover with more than two parents is also possible) to create chromosomes for the next generations. It is analogous to reproduction and biological crossover, upon which the operator is based. There are different methods of performing crossover in genetic algorithms.

Crossover typically combines two parents into one or two children. Multiple crossovers might happen in a single reproduction. It is important to note that:

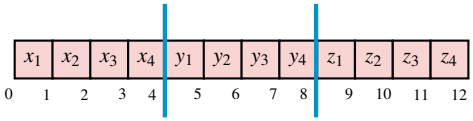
- children have to inherit something from each parent, otherwise you are performing mutation;
- crossover operations have to be designed with the representation (encoding) in mind, to ensure that crossover has the best effect;
- crossover has to generate new valid chromosomes.

Crossover is meant to promote the strong elements of each parent to create the best (evolutionary seen) offspring[6]. In that sense, it has a different function than mutation, which is meant to promote genetic diversity. If crossover is done wrong, and only retains traits from one parent, but none of the other, than you are, in fact, performing

[6]Note that whether an offspring is indeed the best (evolutionary) combination of both parents is determined in the next evaluation round, not at reproduction.

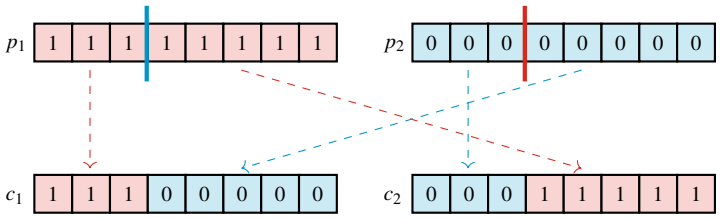
a mutation on the chromosome of the first parent (the traits of the second parent are lost).

Crossover has to be designed in combination with the encoding scheme. For example, if the genome represents a concatenation of three 4-bit numbers, like

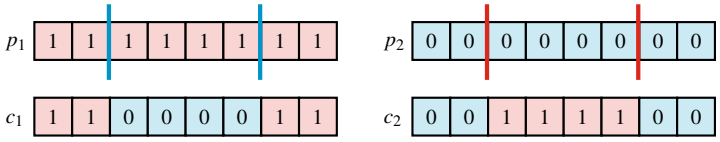


where each number (x, derived from $x_1x_2x_3x_4$, etc.) represents a parameter to be optimised, the crossover should be applied on points 4 or 8 (that is, after bit 4 or after bit 8, as shown in the image above) to retain important traits (that is, the value of x, y, or z) from their parents. Variations for the values of x, y, and z should be handled by mutation, finding the right combination of (optimal) x, y, and z is handled by crossover.

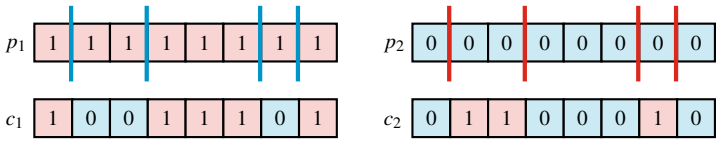
Single-point crossover combines two parents based on a single crossover point. A single gene position is chosen (based on the genotype encoding) to split the genomes of the parents, and recombine them in a crossed manner.



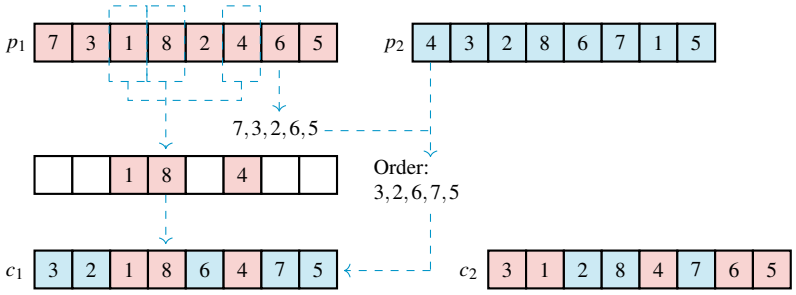
Crossover can also be performed on multiple crossover points. For instance, *two-point crossover* (the crossover points are predetermined depending on the genotype encoding):



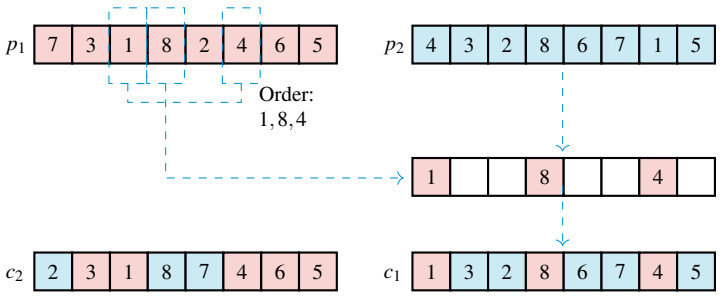
Uniform crossover uses a fixed mixing ratio between the parents. Unlike single- and two-point crossover, uniform crossover enables the parents chromosomes to contribute on the gene-level rather than on segment level. While this increases the amount of solution space being searched (it is more explorative than traditional crossover) it can be rather destructive on the sub-chromosome level (see our earlier point of choosing crossover points wisely). If the mixing rate is set to 0.5, offspring will have approximately half of the genes from either parent, but the crossover points are chosen at random.



Earlier we mentioned that crossover should produce new valid chromosomes. This demand is especially important in encodings where ordering is important (again, as in our TSP problem). When ordering in genomes is important, you cannot simply cut the genome in half and connect it to another half genome, since then cities might appear multiple times in the offspring (which is forbidden in our problem). Other crossover methods have to be applied, which keep the ordering information in place while creating the offspring. This can be done, for instance, by copying a random part of the first parent to the child, and then place the remaining genes in the child by using their ordering in the second parent. This is called *position-based crossover*.



An alternative for problems where ordering is important is *order-based crossover*, where a number of genes are selected from random positions in the first parent and reordered in the same way in the second to create the offspring. The second offspring is then created by doing the same with the parents swapped.



While mutation should always be performed in a genetic algorithm, one can choose to not apply any kind of recombination. Mutation ensures enough diversity and thereby enough coverage of the entire search space, which is needed to ensure that the algorithm does not get stuck in local optima. Mutation has an *explorative* function.

Crossover, on the other hand, is used to optimise based on common or well-performing traits. That is, crossover ensures that certain chromosome sections that function well, are retained from generation to generation, thus trying to move the solutions closer to the optimum. It has a *exploitive* nature. Changes to the genomes depend on the entire population (you cannot create children with particular traits if that trait is none existent in the population). As the population converges to the optimum, the power of the crossover operation diminishes. Since most individuals in the population now have the (most) important traits, all the children will have them as well.

Genetic algorithms that only use mutation and no recombination are called *evolutionary strategies*. The choice whether to use a genetic algorithm (with recombination) or an evolutionary strategy depends on a few key aspects:

- are there building blocks in your solution space?
- is there a semantically meaningful recombination operator?

Either of these indicate that exploitation of a particular (sub)structure in the solution space can be performed, and therefore, the use of a recombination method is advised.

Selection strategies

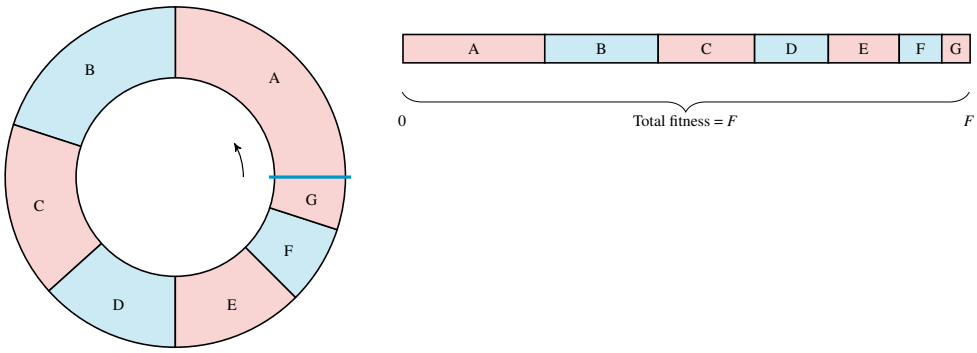
Now we know how to generate new offspring for the next generation, we still need to determine which individuals from the current generation to select as parents. We want a method of ensuring that the best individuals have a higher probability to become parents than the weaker individuals (remember the story about the moose at the beginning of this section). This creates a selection pressure that ensures that the population can improve itself.

For the benefit of genetic diversity, it is wise to not only select the best individuals, but also select a (small) portion of the weaker individuals. Remember that the best individuals are considered 'best' by our fitness function, which can have local optima. If all the best individuals are actually on or near local optima, and we would only use them to reproduce the next generation, we would surely converge to one of those local optima. However, if we mix in some of the weaker individuals (which, according to our fitness function perform worse than the others, but might be closer to the actual optimum) we retain some diversity in the search, allowing the genetic algorithm to escape the local optima. There are several different selection strategies, we discuss a few of them here.

In *fitness proportional selection*, or *roulette-wheel selection*, the fitness level of an individual is used to associate a probability of selecting that individual for reproduction. That is, every individual i, with fitness f_i is assigned a probability as follows:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

The probability p_i that individual i is selected is proportional to its fitness f_i compared to the fitness of the population, as denoted by the sum of all fitnesses $\sum_{j=1}^{N} f_j$. This could be imagined as a Roulette wheel in a casino. Giving each individual a piece of the wheel proportional to its relative fitness, and then turning the wheel for a number of times to determine which individuals are used as parents for the next generation.



In fitness proportional selection, there is small change that high fitness individuals are not selected for the next generation. In the much simpler truncation selection this is never the case. On the other hand, fitness proportionate selection has a chance to keep some of the weaker individuals, mean a higher genetic diversity in the selection strategy. On average, one can expect that every individual i with fitness f_i is chosen $\frac{f_i}{\bar{f}}$ times (that is, the proportion of its fitness f_i with respect to the average fitness $\bar{f}$ of the entire population).

There are some inherent disadvantages with fitness proportional selection.

- Danger of premature convergence; good scoring individuals (much better than the rest of the population, but not necessarily anywhere near the optimum) can take over the entire population rather quickly.
- When fitness scores start to converge, the selection pressure drops (being nearly as good as all the others will give you a fair chance to be selected for the next generation).

These can be circumvented by adapting the fitness function. When fitness ranges from 0 (worst) to 1 (best) and the fitness of a population always sums to 1, the fitness function adapts to the performance of the population and removes both of the above mentioned problems.

Another method of selection is *truncate selection*. Also called *rank-based selection*, truncate selection ranks the individuals based on their fitness score, and only the top percentage is allowed to reproduce. Truncate selection is less sophisticated than the other strategies mentioned here and is not often used in practice. Without a proper reinsertion of lower performing individuals to increase the genetic diversity, rank-based selection runs the risk of quickly converging to a local optimum.

The last strategy that we discuss is called *tournament selection*. Tournament selection involves running several "tournaments" among a few individuals chosen at random from the population. The winner of each tournament (the one with the best fitness) is selected for reproduction. Tournament selection has the benefit of increased genetic diversity, over the simpler truncate selection, since tournaments can be held with the lowest ranking individuals (and therefore, lower ranked individuals still have a chance to reproduce). On the other hand, tournament selection can adjust the selection pressure by adjusting the size of the tournaments. If the tournaments are held with more individuals, weaker individuals have a smaller chance of being selected.

Tournament selection happens as follows:

Algorithm 6.2 — Tournament selection.

1. choose k (the tournament size) individuals from the population at random (this selection is called a *pool*);
2. choose the best individual from pool with probability p;
3. choose the second best individual with probability $p*(1-p)$;
4. choose the third best individual with probability $p*(1-p)^2$);
5. and so on...

■

Deterministic tournament selection (when $p = 1$) selects only the best individual in each tournament. A one-way tournament (when $k = 1$) is the same as random selection. Chosen individuals can be removed from the population if desired, otherwise individuals can be selected more than once for the next generation.

After selecting the right individuals, and knowing how they can reproduce to form the next generation, there is one more element that can influence the working of a genetic algorithm: the *replacement strategy*. In true biology, parents (hopefully the fittest) have kids (one or more) but are themselves never part of the next generation. Every individual from the former generation is eliminated from the population by the creation of the next generation (not physically, for most species, but at least semantically).

In some cases, this is not wanted, since offspring created by mutation and recombination could (randomly) destroy the best individuals (the best genome sequences) in the population. The *elitist strategy* is a selection strategy where a limited number of individuals with the best fitness values are chosen to pass to the next generation (avoiding crossover and mutation). This ensures that the best individual is never lost until other individuals surpass it. The number of elite individuals should never be too high, however, otherwise the population tends to degenerate.

Example, continued

In our example, we perform a simple selection based on proportional fitness as determined by the fitness function presented earlier. We score each individual, sort them (best first) and select a portion from the front of this list (truncate selection). The portion that we select is predefined as the `retain` value of our evolution mechanism. These selected individuals are allowed to reproduce, for which we use a simple single-point crossover. For now, the reproduction method is rather simple, we take the first `N/2` digits from the father and the last `N/2` digits from the mother to create a new child[7].

```
>>> father = [1,2,3,4,5,6]
>>> mother = [10,20,30,40,50,60]
>>> child = father[:len(father)/2] + mother[len(mother)/2:]
>>> child
[1,2,3,40,50,60]
```

[7] Note that there is not really a father and a mother, but rather a *first* parent and a *second* parent.

It is okay for individuals to be a parent multiple times, but no parent should be both father and mother of a child (as, effectively, the child will be an exact copy of that parent).

To promote genetic diversity, we also select a portion of the lesser fit individuals, and allow them to participate in the mix of creating new offspring. Abandoning the metaphor, one of the dangers of optimisation algorithms is getting stuck in local optima and thus being unable to find the real optimum. By including individuals who are not performing well, we decrease the likelihood of getting stuck in a local optimum (remember Section 6.1.1).

We breed as many new children to repopulate the population; that is, if we select 20% of the population as the fittest, and 5% for genetic diversity, we need to breed 75 new individuals to fill up a population of 100. The new population is thus composed of the selected parents (elitists), and the newly created offspring.

We also apply mutation as another method of avoiding local optima. In our example we do this by, at random, selecting and changing a value from an individual (that is, change a single gene of an individual). We only do this for the new individuals (that is, the individuals that were added this generation).

Putting it all together results in the following function to evolve a generation:

```
from random import random

def evolve(population, target, retain=0.2,
           random_select=0.05, mutate=0.01):
    """
    Function for evolving a population, that is, creating
    offspring (next generation population) from combining
    (crossover) the fittest individuals of the current
    population

    :param population: the current population
    :param target: the value that we are aiming for
    :param retain: the portion of the population that we
        allow to spawn offspring
    :param random_select: the portion of individuals that
        are selected at random, not based on their score
    :param mutate: the amount of random change we apply to
        new offspring
    :return: next generation population
    """
    graded = [ (fitness(x, target), x) for x in population ]
    graded = [ x[1] for x in sorted(graded) ]
    retain_length = int(len(graded)*retain)
    parents = graded[:retain_length]

    # randomly add other individuals to promote genetic
    # diversity
```

```
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)

    # crossover parents to create offspring
    desired_length = len(population) - len(parents)
    children = []
    while len(children) < desired_length:
        male = randint(0, len(parents)-1)
        female = randint(0, len(parents)-1)
        if male != female:
            male = parents[male]
            female = parents[female]
            half = int(len(male) / 2)
            child = male[:half] + female[half:]
            children.append(child)

    # mutate some individuals
    for individual in children:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual)-1)
            # this mutation is not ideal, because it
            # restricts the range of possible values,
            # but the function is unaware of the min/max
            # values used to create the individuals
            individual[pos_to_mutate] = \
                randint(min(individual), max(individual))

    parents.extend(children)
    return parents
```

Now we have all the pieces of a genetic algorithm, and we can try it out to see if it works. Commonly, a genetic algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. Here, we use the former.

```
target = 444    # X
p_count = 100   # number of individuals in population
i_length = 7    # N
i_min = 0       # value range for generating individuals
i_max = 70
p = population(p_count, i_length, i_min, i_max)
fitness_history = [grade(p, target),]
for _ in range(100): # we stop after 100 generations
    p = evolve(p, target)
    score = grade(p, target)
    fitness_history.append(score)
```

```
    print(score, end=", ")
```

With our default values of 20% survival, 5% random selection, and 1% mutations, it takes about 5 generations to reach a close optimum (it takes a further 20 or so generations to reach near true optimum):

```
134.03, 83.72, 36.71, 16.81, 6.19, 4.22, 4.0, 4.0, 3.97, 3.82,
3.48, 2.2, 2.08, 2.2, 2.0, 2.0, 2.11, 1.99, 1.94, 1.8, 1.0,
1.0, 1.07, 1.08, 0.99, 0.98, 0.81, 0.28, 0.1, 0.02, 0.01,
0.06, 0.11, 0.07, 0.09, 0.09, 0.13, 0.01, 0.0, 0.09, 0.0, 0.0,
0.11, 0.0, 0.08, 0.04, 0.05, 0.0, 0.0, 0.0, 0.11, 0.0, 0.06,
0.0, 0.05, 0.05, 0.0, 0.05, 0.0, 0.07, 0.07, 0.0, 0.07, 0.06,
0.12, 0.0, 0.0, 0.18, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.1,
0.1, 0.0, 0.0, 0.08, 0.0, 0.06, 0.0, 0.09, 0.1, 0.0, 0.03,
0.0, 0.0, 0.19, 0.08, 0.0, 0.0, 0.0, 0.0, 0.0, 0.11, 0.0, 0.0,
```

Of course, every run is different, and sometimes it takes longer, and sometimes it takes less (to reach a perfect score). Note that the algorithm happily generates all requested 100 generations, even when it already achieved a perfect score, and due to mutations, it does not stay at the perfect score, but "wanders around it a bit".

6.1.5 Limitations

Literature on genetic algorithms (GA) describes a large number of successful applications, but there are also many cases in which GAs perform poorly. Given a particular potential application, how do we know if a GA is a good method? Unfortunately, there is no solid answer, though there is some agreement that GAs have a chance to perform well within the following intuitions:

- search space is large; and
- search space is known to not be perfectly smooth and unimodal (that is, consist of a single smooth "hill"); or
- search space is not well understood; or
- fitness function is noisy; and
- task does not require global optimum to be found (that is, finding a good enough solution quickly is more important).

If the search space is not large, then it makes more sense to search it exhaustively, because one can then be sure that the true optimum is found. If the search space is smooth or unimodal, a gradient-descent (or -ascend) algorithm will be more efficient (as such algorithms exploit the search space smoothness much better than GAs). If the search space is well-understood (like Travelling Salesman Problem), search by using heuristics and domain-knowledge outperform GAs. When the fitness function is noisy (like, for example, it involves taking error-prone measurements from the real-world), GAs work much better, since they accumulate fitness over many generations, whereas single-candidate-search methods (like hill climbing) are much more led astray by the noise. GAs are much better at overcoming the presence of small amounts of noise.

These intuitions, of course, do not rigorously predict when a GA will be an effective search procedure competitive with other procedures. A GA's performance

depends greatly on details such as the method for encoding candidate solutions, the operators, the parameter settings, and the particular criterion for success.

Aside 6.4 — Randomness and GAs: the infinite monkey theory.
The classic infinite monkey theorem states that a monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely type a given text, such as the complete works of William Shakespeare.

In fact the monkey would almost surely type every possible finite text an infinite number of times. However, the probability that all the monkeys that ever lived would type a complete work such as Shakespeare's Hamlet is so tiny that it would be extremely unlikely (but technically not impossible).

Now consider that we would apply evolutionary elements to stack the situation in our favour. Let assume that we would only allow monkeys that type 'parts of' the complete works of Shakespeare to reproduce the next generation of typing monkeys[a], would we in some time (after a finite number of generations) have a higher probability of them typing Shakespeare?

Unfortunately, this is not the case. Monkeys type one letter at a time, independently of previously typed letters[b]. This completely random generation of texts had too little intrinsic (internal) relation that it would be nearly impossible to keep those traits over future generations.

[a] Do consider that this kind of selective breeding to create an "über-monkey" is considered to be unethical and should be avoided in practice. So let us remind you that this is merely a thought-experiment, no monkeys were harmed in the process.

[b] In fact, an actual study has shown that monkey loose interests in typewriting devices rather quickly, and never produce much text at all. They rather smash the keyboard, or urinate and defecate on it, instead.

6.2 Evolving neural networks

Neuro-evolution is a form of Artificial Intelligence that combines evolutionary algorithms with neural networks. The optimisation problem that it tries to solve (by using an EA) can be either the neural network parameters, the topology, or the rules/weights, or any combination of these. A main advantage of this approach is that it can be applied more widely than supervised learning algorithms, as presented in Section 4.4.2, which requires a (large) set of correct input-output pairs. Neuro-evolution, on the other hand, only requires a measure of the network's performance at a particular task, which could also be measured without providing labelled examples of desired combinations (e.g., the outcome of a game, without predetermining the desired strategy, can still be measured by means of wins and losses).

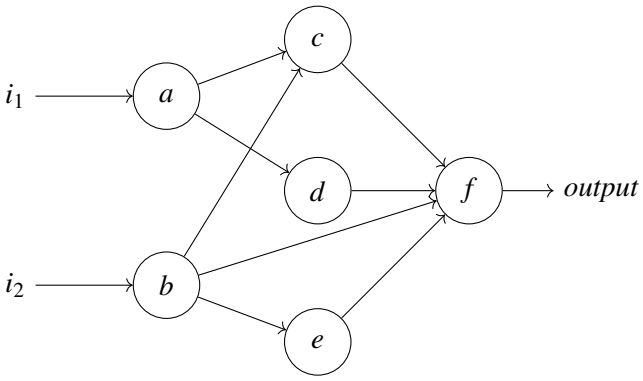
6.2.1 Evolving parameters / structure

The topology and parameters used to initialise a neural network have a large impact on the performance. The chosen topology and parameters can determine the success and failure of the application of the neural network. It is, however, rather difficult to completely determine the best topology and parameters on forehand, based on only your knowledge of the problem domain. Usually, the topology and parameters are decided on by trial and error, and there is a great need for a methodological approach to design neural networks for a particular application. Neuro-evolution could help here, by trying to find the optimal combination of topology and parameters for the network.

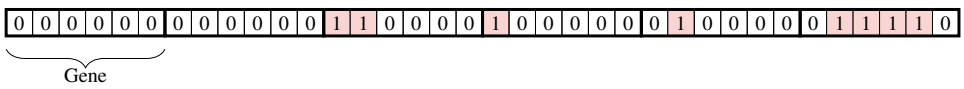
The basic idea is to evolve the topology and parameters of the network, thus searching for the best choices of numbers of neurons, number of layers, layer types, learning algorithm used, etc. To achieve this, we need to find a correct encoding for our problem.

The connection topology of a neural network can be represented by a square connectivity matrix. Each entry in the matrix defines the type of connection from one neuron (column) to another neuron (row), where 0 means no connection and 1 denotes a connection (for which the weight can be changed through learning). To change the matrix into a chromosome, we string the rows of the matrix together.

	a	b	c	d	e	f
a	0	0	0	0	0	0
b	0	0	0	0	0	0
c	1	1	0	0	0	0
d	1	0	0	0	0	0
e	0	1	0	0	0	0
f	0	1	1	1	1	0

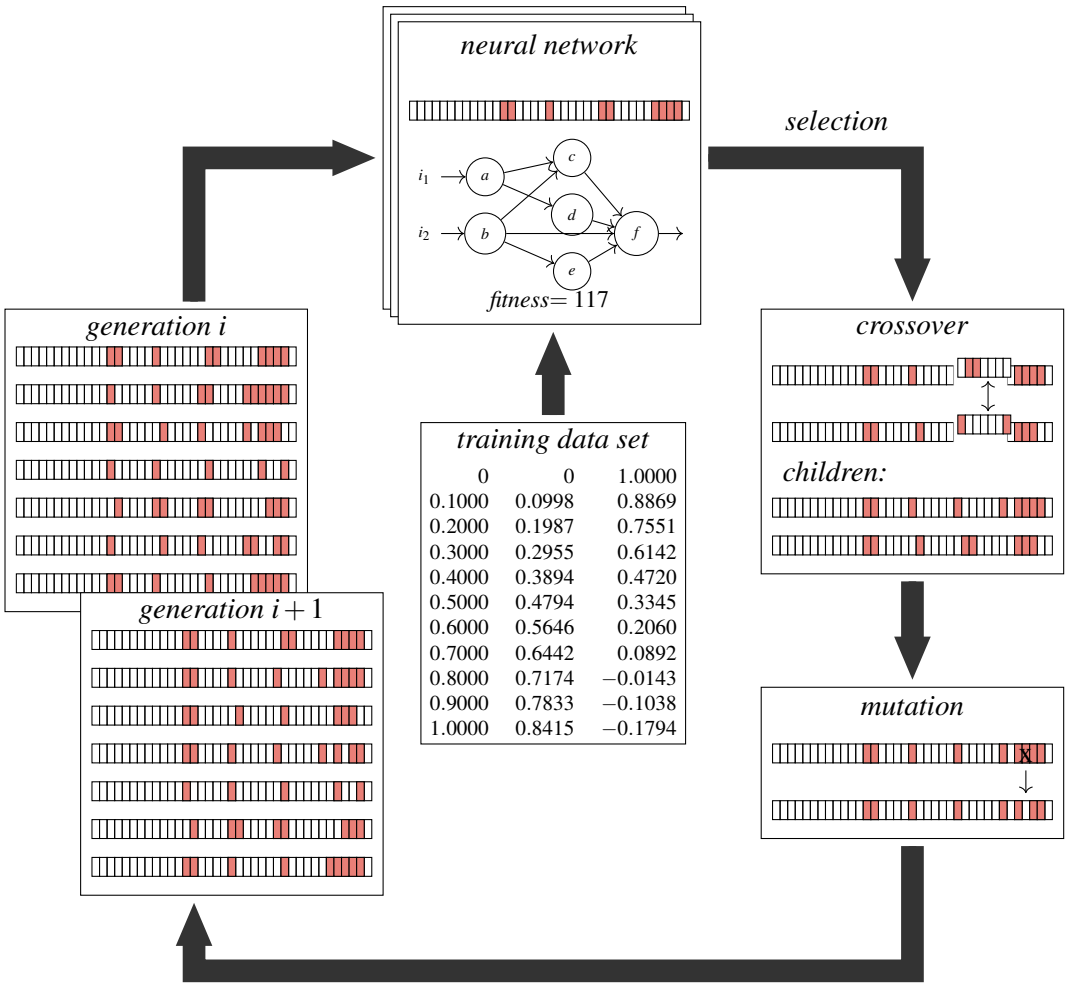


Chromosome:



Gene

We now apply our typical evolutionary components (fitness, selection, recombination, etc.) to obtain the following optimisation cycle:



This method of encoding (called *direct encoding*) and evolving the topology of neural networks becomes increasingly cumbersome when the complexity of the network rises (when more and more nodes are added). As the size of the network grows, the size of the required chromosome increases quickly, leading to problems in performance. Moreover, direct encoding cannot represent repeated or nested structures in the network, which are common for some problems.

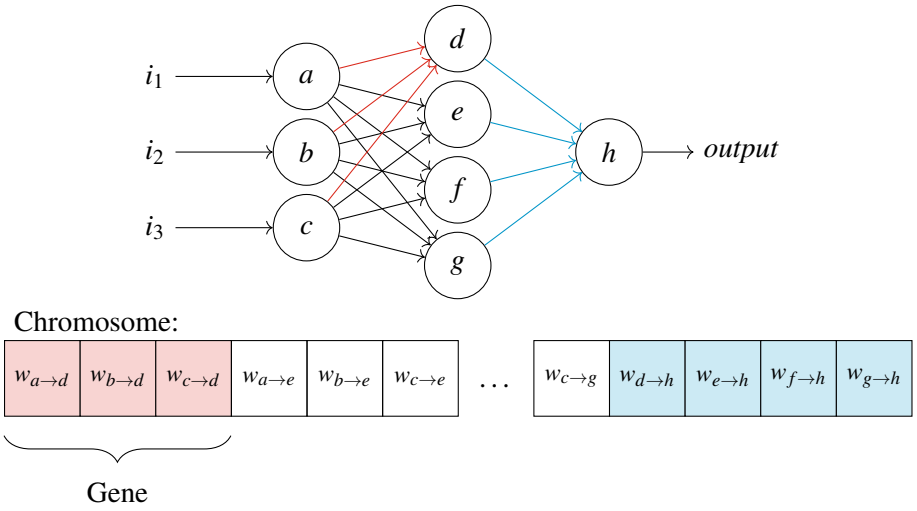
An alternative encoding, called *grammatical encoding* tries to solve this problem. This encoding represents the network encoding as grammars; the genetic algorithm evolves the grammars, but the fitness is tested only after a "development" step in which a network develops from the grammar. That is, the grammar is a "genotype", while the "phenotype" is a network derived from that grammar. The exact encoding scheme used in this method, and the execution of it are beyond the scope of this discussion.

6.2.2 Evolving weights

While this subject is beyond the scope of this course, as we have learned back-propagation in Section 4.4.2. Back-propagation is in most situation the better way of learning the weights in a neural network, and is therefore preferred. In some situations,

however, back-propagation can get stuck in one of the many local optima present, and has difficulties escaping these. In such cases, using a genetic algorithm to learn the weights instead could be helpful.

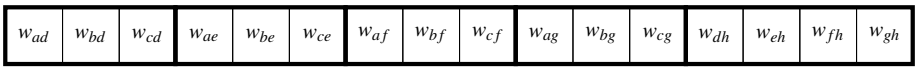
To apply genetic algorithms to the learning of weights, we use the vector representation as presented in the previous chapter. The chromosomes in our population are a sequence of (real) numbers, where each number represents a weight of a connection in the neural network. We group numbers related to a single neuron in a gene:



Next we need to define a fitness function for evaluating the performance of the chromosomes. The function must estimate the performance of a given network, which can easily be calculated by having the network classify a number of examples. The number of examples should not be too small, since this increases the risk of overfitting (the neural network performs well on the examples, but not on the real problem). However, too much examples means that the evaluation of a population takes more time. An alternative is to select a portion of a large data set for testing every selection round (but make sure that all networks are scored using the same examples).

The simplest function to score networks (based on an example set) is using the sum of squared errors (see Section 4.4.2 earlier for details). The smaller the sum, the fitter the network. This means that the genetic algorithm attempts to find a set of weights that minimises the sum of squared errors.

There are a number of choices for the genetic operators. In some cases, an evolutionary strategy is used which means that only mutation is allowed, but crossover can be applied as well, if designed correctly. A crossover operator (single-point crossover, two-point crossover, or even uniform crossover) needs to adhere to the weights that belong to a single neuron; that is, crossover can be performed, as long as the crossover points are chosen to match the gene encoding, that is, split only on the segments that represent the weights on the links related to a single neuron.



The mutation operator can be chosen, for instance, to add a small random value between -1 and 1 to each weight in a randomly chosen gene (again, remember that a single gene represents multiple weights).

As mentioned at the start of this section, while the optimisation of weights of a neural network by means of an evolutionary algorithm is possible, it is rarely used. This is because the genetic algorithm has many more parameters that need to be selected to find the optimal training strategy, and because back-propagation usually functions well enough. The latter is especially true in larger (thousands to millions of neurons) networks.

6.3 Exercises

Exercise 6.1 — Card problem.
You have 10 cards numbered from 1 to 10. You have to choose a way of dividing them into two piles, so that the cards in the first pile **sum** to a number as close as possible to 36, and the remaining cards in the other pile **multiply** to a number as close as possible to 360.

Genotype encoding

Each card can be put in `Pile_0` or `Pile_1`, there are 1024 possible ways of sorting them into two piles, and you have to find the best. Think of a sensible way of encoding any possible solution-attempt as a genotype.

Fitness

Some of these solution attempts will be closer to the target than others. Think of a sensible way of evaluating any solution-attempt and scoring it with a fitness-measure.

Assignment

Write a program to run a Genetic Algorithm with your genotype encoding and fitness function. Run it once for a suitable (probably very large – you choose) number of generations. Then repeat that same run a number of times (like, 100) and see if you get the same answer each time, see how much variance there is between each run.

Exercise 6.2 — Wing design.
You have 4 variables that represent possible parameter settings for the design of an aircraft wing. A, B, C, and D, each of which can be any whole number between 0 and 63 (use a bit encoding per parameter).
Your aerodynamics model tells you that the *Lift* of the wing is the following:

$$Lift = (A-B)^2 + (C+D)^2 - (A-30)^3 - (C-40)^3$$

Find values of A B C D, each within their allowed range 0-63, that maximises *Lift*.

Assignment

Write a program to run a Genetic Algorithm with your genotype encoding and fitness function. Run it once for a suitable (probably very large – you choose) number of generations. Then repeat that same run a number of times (like, 100) and see if you get the same answer each time, see how much variance there is between each run.

Exercise 6.3 — Face Recognition (Optional).

In this exercise we are going to combine evolutionary algorithms with neural networks to create a facial recognition system.

A data set consisting of pictures (96×96 pixels, greyscale) is available on the electronic learning environment. The images are pictures (centered) of lecturers of the Computer Engineering team. Your task is to use the available data set to create and train a neural network that can detect which person is represented on a given image.

Use an evolutionary algorithm to determine the most suitable configuration (topology, etc.) of your neural network. The neural networks should be created by using available library implementation[a]

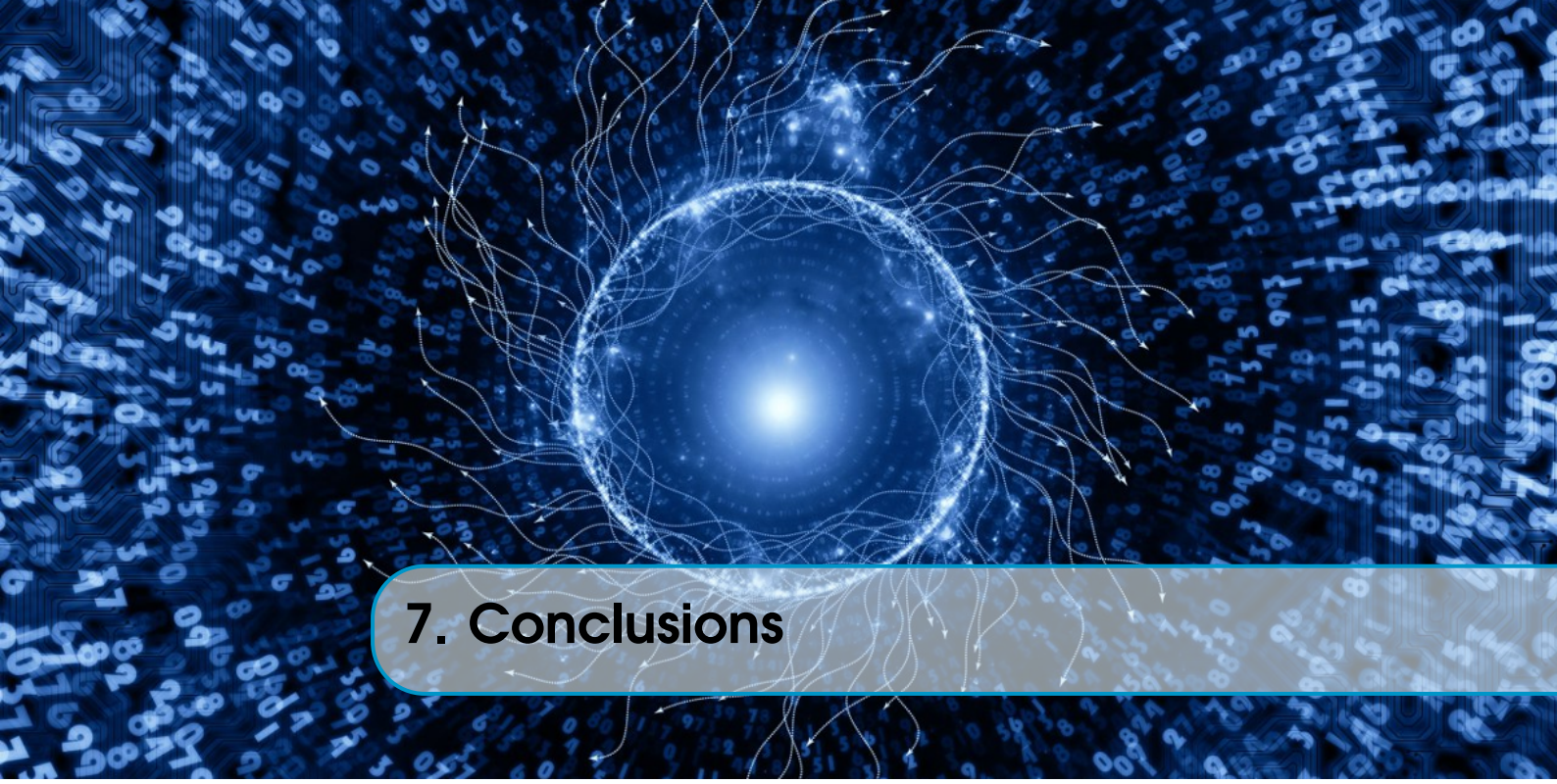
The optimisation of the network and the training of the neural network are done in *your* time; during grading we will assess how well your (best) neural network performs on a new set of images. That means that you need to be able to save (and later load) a neural network (layers, weights, etc.).

Hand-in

You need to hand-in the following elements:

- the implementation (code) of the evolutionary algorithm that was used to determine the NN architecture;
- a short description of the best architecture (found by your EA) (describe number of layers, layer types, etc.) and a brief explanation why you think this is the best architecture;
- an implementation (code) that can read your best network and perform a classification based on a given (similar, but different) dataset to assess it's performance.

[a] For example, see Appendix B for descriptions of the use of neural network libraries, including `TensorFlow` and `Theano`. You are not restricted to either of these, you may choose your own.

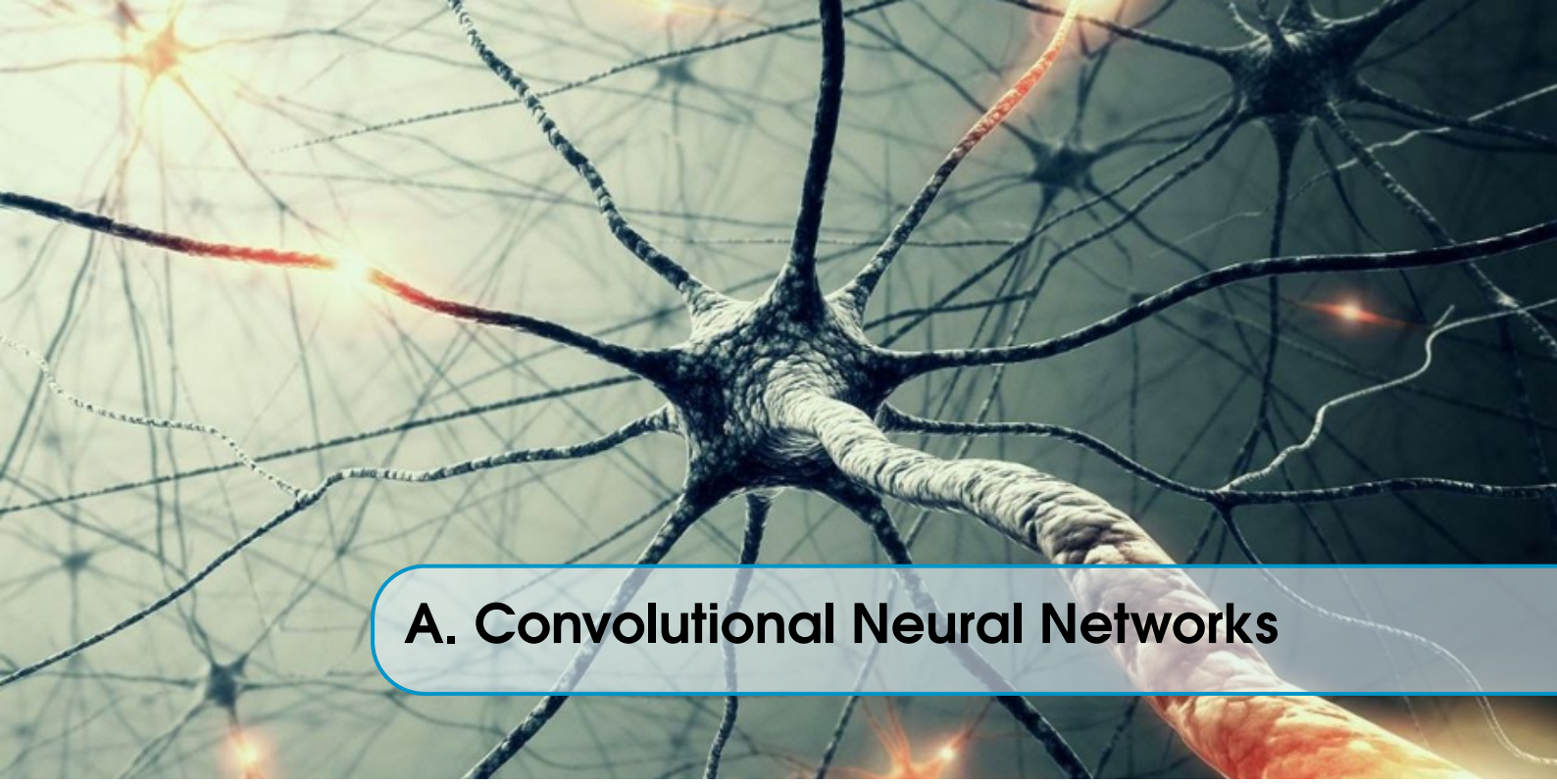


7. Conclusions

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* (Sabour, Frosst, & Hinton, 2017), 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. Convolutional Neural Networks

This section provides an overview of the latest in techniques for neural networks. In recent years it has become apparent that the use of convolution in neural networks is the preferred method of avoiding overfitting when dealing with image data. The goal of this chapter is to act as a reference for the terms and intuitions concerning convolutional neural networks, or to serve as a starting point for further studies. The material in this chapter is not part of the exam, you may consider this chapter (including the following appendices) a large aside in the reader[1].

A.1 Introduction

Convolutional neural networks (CNNs or ConvNets) are very similar to ordinary neural networks as presented in chapter 4: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs and performs a dot product (see chapter 5). 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

[1] For an even further explanation of Convolutional Neural Networks, please consider watching the excellent tutorial by Stanford University: `https://youtu.be/AQirPKrAyDg`.

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**[2]. For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions $32 \times 32 \times 3$ (width, height, depth, respectively, see right-hand side of Figure A.1).

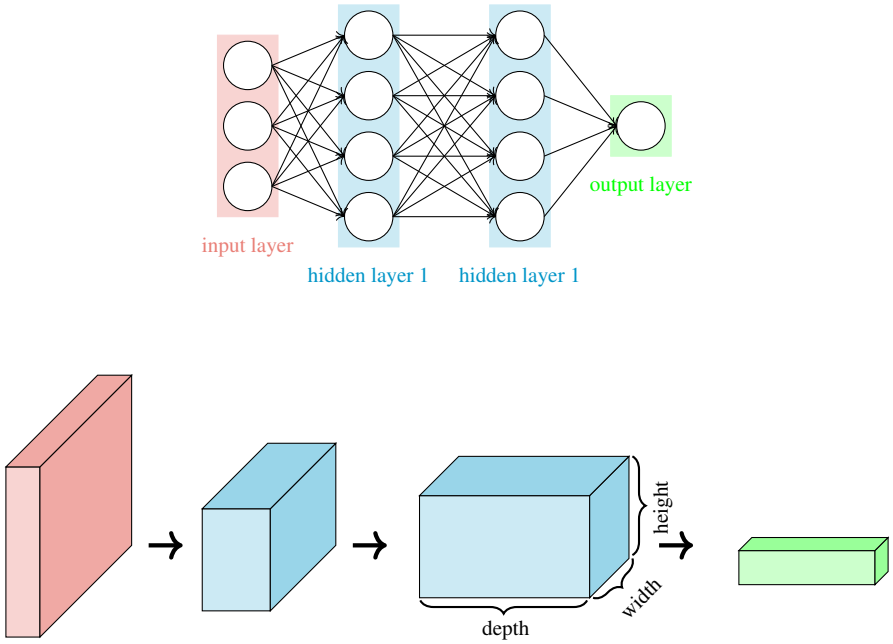


Figure A.1 Top: a regular 3-layer neural network. Bottom: a convnet arranges its neurons in three dimensions (width, height, depth). Each layer in a convnet transforms the 3D input volume to a 3D output volume of neuron activations.

A.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:

[2]Please note that *depth* is typically reserved to denote the number of (hidden) layers of a neural network. Here *depth* denotes the third dimension of the activation volume of a particular layer in the network.

- 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 $[16 \times 16 \times 12]$;
- 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 4.4.2) such that the class scores that the convnet computes is consistent with the labels in the training set.

A.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[3] 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).

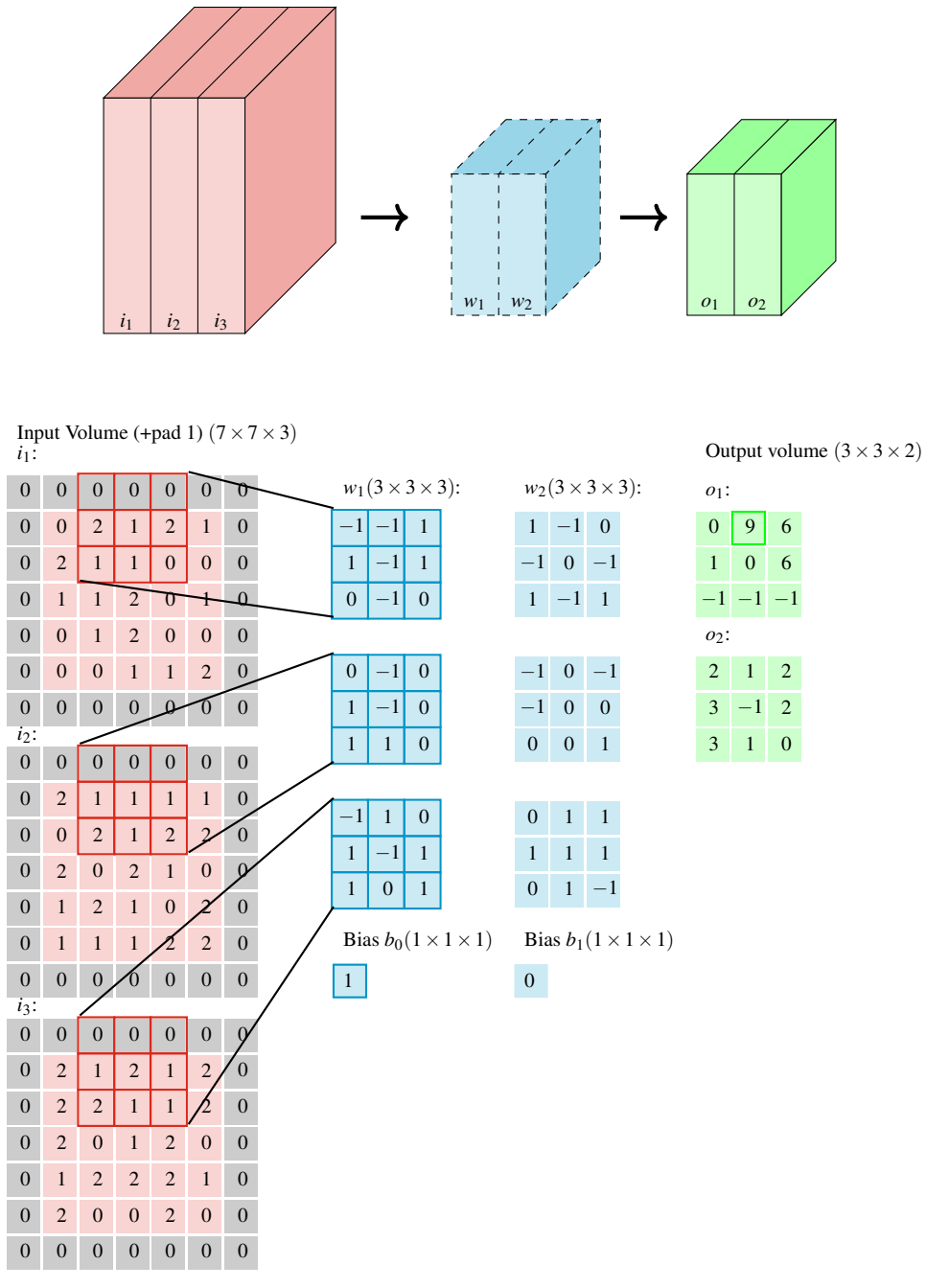


Figure A.2 Example of hyperparameters and calculations in convolutional layers. Using two filters of 3×3 (with depth 3, as the input has depth 3),padding 1, and stride 2. Note that w_1 and w_2 are virtual: the neurons in the conv layer (in this case o_1 and o_2) are wired through the filters to the activation 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.

[3]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.

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*:

- 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)[4].

The spatial size of the output volume can be computed as a function of the input volume size (W)[5], the receptive field size of the conv layer neurons (F)[6], 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:

[4]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): $P=(F-1)/2$.

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

[6]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).

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;
  - 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.

A.2.2 Rectified linear unit (RELU) layer

In chapter 4 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[7].

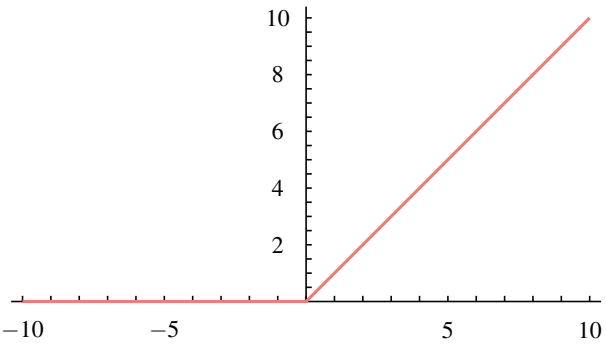


Figure A.3 Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$.

The Rectified Linear Unit as activation has become very popular in the last few

[7] 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.

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 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.

A.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:

- 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

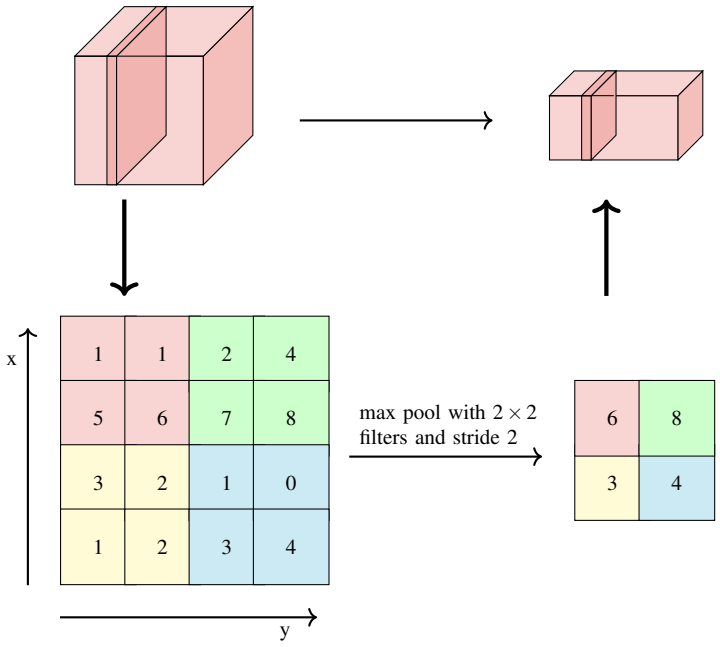


Figure A.4 Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume

has been shown to work better in practice.

A.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 5 for more information.

A.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.

A.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.

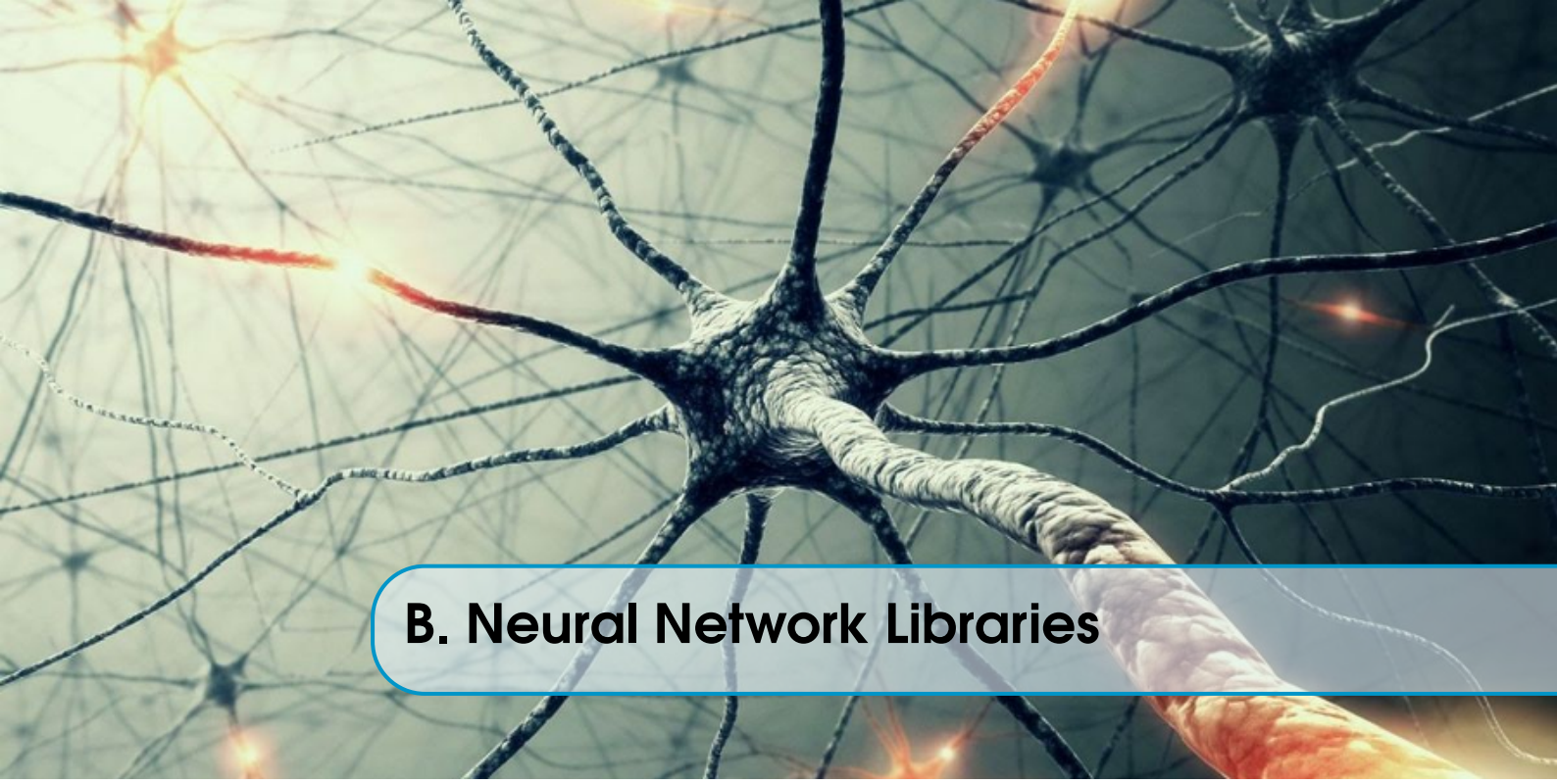
Layer sizing heuristics

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.



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 5.

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 5, 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 A).

[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 4); 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 4.4.2) 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[10] (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
```

[10] For instance, if you want to use this library with weights learned by NeuroEvolotion, see Section 6.2.

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[11]:

```
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.

[11] 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.

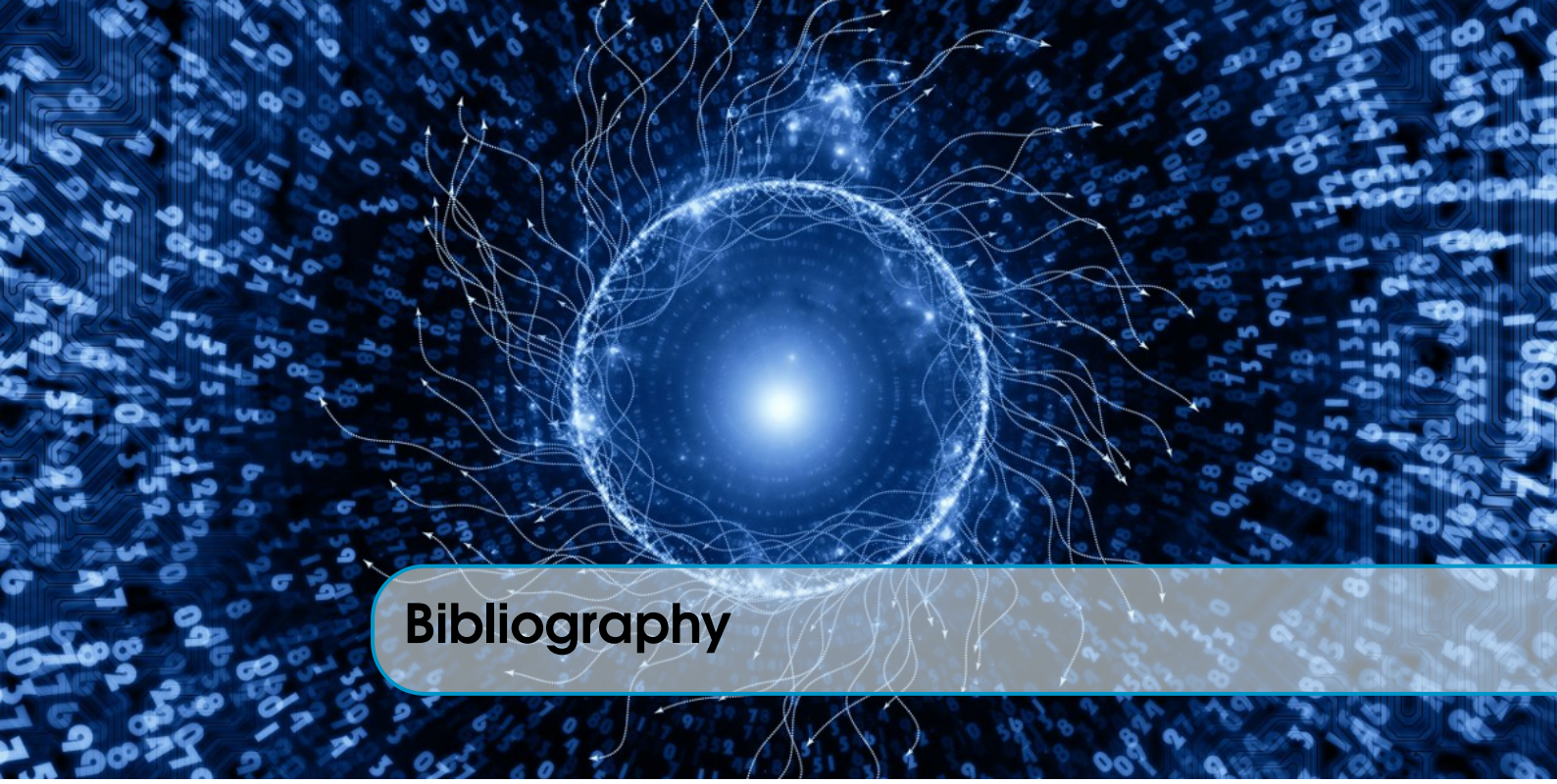
The complete code (available on `https://github.com/aldewereld/nl.hu.ict.a2i.cnn/blob/master/lasagne_mnist.py`) shows how to monitor training and performance of the network.

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`.



Bibliography

Artificial Intelligence: Google's AlphaGo beats Go master Lee Se-dol (2016, March 12). BBC News. URL: http://www.bbc.com/news/technology-35785875.

Asimov, Isaac (2008). *I, Robot*. New York: Bantam. ISBN: 0-553-38256-X.

Brooks, Rodney A (1990). "Elephants don't play chess". In: *Robotics and autonomous systems* 6.1-2, pp. 3–15.

Darwin, Charles (1859). "On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life". In: *Nature* 5.121, p. 502.

Doxiadis, Apostolos and Christos H. Papadimitriou (2009). *Logicomix – An Epic Search for Truth*. Bloomsbury Publishing, London.

Kasteleijn, Nando (2017). *Computer verslaat grootmeester bordspel, maar kan geen tosti maken*. URL: http://nos.nl/artikel/2175632-computer-verslaat-grootmeester-bordspel-maar-kan-geen-tosti-maken.html (visited on 08/28/2017).

Kurtzweil, Ray (2005). *The Singularity is Near*. Penguin Group.

Markoff, J. (2011, February 16). *Computer Wins on 'Jeopardy!': Trivial, It's Not*. New York Times. URL: http://www.nytimes.com/2011/02/17/science/17jeopardy-watson.html.

McCulloch, W.S. and W. Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biophysics*.

Minsky, Marvin (1967). *Computation: Finite and Infinite Machines*. Englewood Cliffs: Prentice-Hall.

Minsky, Marvin and Seymour Papert (1969). *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, Cambridge MA.

Newell, A., J.C. Shaw, and H.A. Simon (1959). "Report on general problem-solving program". In: *Proceedings of the International Conference on Information Processing*, pp. 256–264.

Ovid (ca. 8). *Metamorphoses*.

Sabour, Sara, Nicholas Frosst, and Geoffrey Hinton (2017). "Dynamic Routing between Capsules". In:

Searle, John (1980). "Minds, Brains and Progams". In: *Behavioral and Brain Sciences* 3.3.

Spectrum, IEEE (2008, June 1). *Tech Luminaries Address Singularity*. URL: https://spectrum.ieee.org/computing/hardware/tech-luminaries-address-singularity (visited on 09/08/2017).

Turing, Alan (1950). "Computing Machinery and Intelligence". In: *Mind* LIX.236, pp. 433–460.

Ulam, Stanislaw (1958, May). "Tribute to John von Neumann". In: *Bulletin of the American Mathematical Society* 64.3.

Weizenbaum, Joseph (1976). *Computer power and human reason: from judgment to calculation*. W.H. Freeman and Company.

Wikipedia (2001, August 26). *Turing test*. URL: https://en.wikipedia.org/wiki/Turing_test (visited on 06/16/2017).

— (2001, July 31). *Spam (Monty Python)*. URL: https://en.wikipedia.org/wiki/Spam_(Monty_Python) (visited on 08/01/2017).

— (2005, May 11). *Pygmalion (mythology)*. URL: https://en.wikipedia.org/wiki/Pygmalion_(mythology) (visited on 06/27/2017).

— (2016, May 15). *Deepart*. URL: https://en.wikipedia.org/wiki/DeepArt (visited on 08/28/2017).