A Brief Introduction to

# Neural Networks

David Kriesel
dkriesel.com

**Download location:**
http://www.dkriesel.com/en/science/neural_networks

**NEW – for the programmers:**
Scalable and efficient NN framework, written in JAVA
http://www.dkriesel.com/en/tech/snipe

# A small preface

"Originally, this work has been prepared in the framework of a seminar of the University of Bonn in Germany, but it has been and will be extended (after being presented and published online under `www.dkriesel.com` on 5/27/2005). First and foremost, to provide a comprehensive overview of the subject of neural networks and, second, just to acquire more and more knowledge about LaTeX . And who knows – maybe one day this summary will become a real preface!"

*Abstract of this work, end of 2005*

The above abstract has not yet become a preface but at least a *little preface*, ever since the extended text (then 40 pages long) has turned out to be a download hit.

## Ambition and intention of this manuscript

The entire text is written and laid out more effectively and with more illustrations than before. I did all the illustrations myself, most of them directly in LaTeX by using XYpic. They reflect what I would have liked to see when becoming acquainted with the subject: Text and illustrations should be memorable and easy to understand to offer as many people as possible access to the field of neural networks.

Nevertheless, the mathematically and formally skilled readers will be able to understand the definitions without reading the running text, while the opposite holds for readers only interested in the subject matter; everything is explained in both colloquial and formal language. Please let me know if you find out that I have violated this principle.

### The sections of this text are mostly independent from each other

The document itself is divided into different parts, which are again divided into chapters. Although the chapters contain cross-references, they are also individually acces-

sible to readers with little previous knowledge. There are larger and smaller chapters: While the larger chapters should provide profound insight into a paradigm of neural networks (e.g. the classic neural network structure: the *perceptron* and its learning procedures), the smaller chapters give a short overview – but this is also explained in the introduction of each chapter. In addition to all the definitions and explanations I have included some excursuses to provide interesting information not directly related to the subject.

Unfortunately, I was not able to find free German sources that are multi-faceted in respect of content (concerning the paradigms of neural networks) and, nevertheless, written in coherent style. The aim of this work is (even if it could not be fulfilled at first go) to close this gap bit by bit and to provide easy access to the subject.

## Want to learn not only by reading, but also by coding? Use SNIPE!

***SNIPE***[1] is a well-documented JAVA library that implements a framework for neural networks in a speedy, feature-rich and usable way. It is available at no cost for non-commercial purposes. It was originally designed for high performance simulations with lots and lots of neural networks (even large ones) being trained simultaneously. Recently, I decided to give it away as a professional reference implementation that covers network aspects handled within this work, while at the same time being faster and more efficient than lots of other implementations due to the original high-performance simulation design goal. Those of you who are up for learning by doing and/or have to use a fast and stable neural networks implementation for some reasons, should definetely have a look at Snipe.

However, the aspects covered by Snipe are not entirely congruent with those covered by this manuscript. Some of the kinds of neural networks are not supported by Snipe, while when it comes to other kinds of neural networks, Snipe may have lots and lots more capabilities than may ever be covered in the manuscript in the form of practical hints. Anyway, in my experience almost all of the implementation requirements of my readers are covered well. On the Snipe download page, look for the section "Getting started with Snipe" – you will find an easy step-by-step guide concerning Snipe and its documentation, as well as some examples.

1 Scalable and Generalized Neural Information Processing Engine, downloadable at `http://www.dkriesel.com/tech/snipe`, online JavaDoc at `http://snipe.dkriesel.com`

**SNIPE:** This manuscript frequently incorporates Snipe. Shaded Snipe-paragraphs like this one are scattered among large parts of the manuscript, providing information on how to implement their context in Snipe. **This also implies that those who do not want to use Snipe, just have to skip the shaded Snipe-paragraphs!** The Snipe-paragraphs assume the reader has had a close look at the "Getting started with Snipe" section. Often, class names are used. As Snipe consists of only a few different packages, I omitted the package names within the qualified class names for the sake of readability.

## It's easy to print this manuscript

This text is completely illustrated in color, but it can also be printed as is in monochrome: The colors of figures, tables and text are well-chosen so that in addition to an appealing design the colors are still easy to distinguish when printed in monochrome.

## There are many tools directly integrated into the text

Different aids are directly integrated in the document to make reading more flexible: However, anyone (like me) who prefers reading words on paper rather than on screen can also enjoy some features.

### In the table of contents, different types of chapters are marked

Different types of chapters are directly marked within the table of contents. Chapters, that are marked as "fundamental" are definitely ones to read because almost all subsequent chapters heavily depend on them. Other chapters additionally depend on information given in other (preceding) chapters, which then is marked in the table of contents, too.

### Speaking headlines throughout the text, short ones in the table of contents

The whole manuscript is now pervaded by such headlines. Speaking headlines are not just title-like ("Reinforcement Learning"), but centralize the information given in the associated section to a single sentence. In the named instance, an appropriate headline would be "Reinforcement learning methods provide feedback to the network, whether it

behaves good or bad". However, such long headlines would bloat the table of contents in an unacceptable way. So I used short titles like the first one in the table of contents, and speaking ones, like the latter, throughout the text.

### Marginal notes are a navigational aid

The entire document contains marginal notes in colloquial language (see the example in the margin), allowing you to "scan" the document quickly to find a certain passage in the text (including the titles).

New mathematical symbols are marked by specific marginal notes for easy finding (see the example for $x$ in the margin).

### There are several kinds of indexing

This document contains different types of indexing: If you have found a word in the index and opened the corresponding page, you can easily find it by searching for ***highlighted text*** – all indexed words are highlighted like this.

Mathematical symbols appearing in several chapters of this document (e.g. $\Omega$ for an output neuron; I tried to maintain a consistent nomenclature for regularly recurring elements) are separately indexed under "Mathematical Symbols", so they can easily be assigned to the corresponding term.

Names of persons written in SMALL CAPS are indexed in the category "Persons" and ordered by the last names.

## Terms of use and license

Beginning with the epsilon edition, the text is licensed under the *Creative Commons Attribution-No Derivative Works 3.0 Unported License*[2], except for some little portions of the work licensed under more liberal licenses as mentioned (mainly some figures from Wikimedia Commons). A quick license summary:

1. You are free to redistribute this document (even though it is a much better idea to just distribute the URL of my homepage, for it always contains the most recent version of the text).

2 `http://creativecommons.org/licenses/by-nd/3.0/`

2. You may not modify, transform, or build upon the document except for personal use.
3. You must maintain the author's attribution of the document at all times.
4. You may not use the attribution to imply that the author endorses you or your document use.

For I'm no lawyer, the above bullet-point summary is just informational: if there is any conflict in interpretation between the summary and the actual license, the actual license always takes precedence. Note that this license does not extend to the source files used to produce the document. Those are still mine.

### How to cite this manuscript

There's no official publisher, so you need to be careful with your citation. Please find more information in English and German language on my homepage, respectively the subpage concerning the manuscript[3].

## Acknowledgement

Now I would like to express my gratitude to all the people who contributed, in whatever manner, to the success of this work, since a work like this needs many helpers. First of all, I want to thank the proofreaders of this text, who helped me and my readers very much. In alphabetical order: Wolfgang Apolinarski, Kathrin Gräve, Paul Imhoff, Thomas Kühn, Christoph Kunze, Malte Lohmeyer, Joachim Nock, Daniel Plohmann, Daniel Rosenthal, Christian Schulz and Tobias Wilken.

Additionally, I want to thank the readers Dietmar Berger, Igor Buchmüller, Marie Christ, Julia Damaschek, Jochen Döll, Maximilian Ernestus, Hardy Falk, Anne Feldmeier, Sascha Fink, Andreas Friedmann, Jan Gassen, Markus Gerhards, Sebastian Hirsch, Andreas Hochrath, Nico Höft, Thomas Ihme, Boris Jentsch, Tim Hussein, Thilo Keller, Mario Krenn, Mirko Kunze, Maikel Linke, Adam Maciak, Benjamin Meier, David Möller, Andreas Müller, Rainer Penninger, Lena Reichel, Alexander Schier, Matthias Siegmund, Mathias Tirtasana, Oliver Tischler, Maximilian Voit, Igor Wall, Achim Weber, Frank Weinreis, Gideon Maillette de Buij Wenniger, Philipp Woock and many others for their feedback, suggestions and remarks.

3 http://www.dkriesel.com/en/science/neural_networks

Additionally, I'd like to thank Sebastian Merzbach, who examined this work in a very conscientious way finding inconsistencies and errors. In particular, he cleared lots and lots of language clumsiness from the English version.

Especially, I would like to thank Beate Kuhl for translating the entire text from German to English, and for her questions which made me think of changing the phrasing of some paragraphs.

I would particularly like to thank Prof. Rolf Eckmiller and Dr. Nils Goerke as well as the entire Division of Neuroinformatics, Department of Computer Science of the University of Bonn – they all made sure that I always learned (and also had to learn) something new about neural networks and related subjects. Especially Dr. Goerke has always been willing to respond to any questions I was not able to answer myself during the writing process. Conversations with Prof. Eckmiller made me step back from the whiteboard to get a better overall view on what I was doing and what I should do next.

Globally, and not only in the context of this work, I want to thank my parents who never get tired to buy me specialized and therefore expensive books and who have always supported me in my studies.

For many "remarks" and the very special and cordial atmosphere ;-) I want to thank Andreas Huber and Tobias Treutler. Since our first semester it has rarely been boring with you!

Now I would like to think back to my school days and cordially thank some teachers who (in my opinion) had imparted some scientific knowledge to me – although my class participation had not always been wholehearted: Mr. Wilfried Hartmann, Mr. Hubert Peters and Mr. Frank Nökel.

Furthermore I would like to thank the whole team at the notary's office of Dr. Kemp and Dr. Kolb in Bonn, where I have always felt to be in good hands and who have helped me to keep my printing costs low - in particular Christiane Flamme and Dr. Kemp!

Thanks go also to the Wikimedia Commons, where I took some (few) images and altered them to suit this text.

Last but not least I want to thank two people who made outstanding contributions to this work who occupy, so to speak, a place of honor: My girlfriend Verena Thomas, who found many mathematical and logical errors in my text and discussed them with me, although she has lots of other things to do, and Christiane Schultze, who carefully reviewed the text for spelling mistakes and inconsistencies.

*David Kriesel*

# Contents

# Part I

# From biology to formalization – motivation, philosophy, history and realization of neural models

# Chapter 1

# Introduction, motivation and history

How to teach a computer? You can either write a fixed program – or you can enable the computer to learn on its own. Living beings do not have any programmer writing a program for developing their skills, which then only has to be executed. They learn by themselves – without the previous knowledge from external impressions – and thus can solve problems better than any computer today. What qualities are needed to achieve such a behavior for devices like computers? Can such cognition be adapted from biology? History, development, decline and resurgence of a wide approach to solve problems.

## 1.1 Why neural networks?

There are problem categories that cannot be formulated as an algorithm. Problems that depend on many subtle factors, for example the purchase price of a real estate which our brain can (approximately) calculate. Without an algorithm a computer cannot do the same. Therefore the question to be asked is: *How do we learn to explore such problems?*

Exactly – we *learn*; a capability computers obviously do not have . Humans have a brain that can learn. Computers have some processing units and memory. They allow the computer to perform the most complex numerical calculations in a very short time, but they are not adaptive. If we compare computer and brain[1], we will note that, theoretically, the computer should be more powerful than our brain: It comprises $10^9$

1 Of course, this comparison is - for obvious reasons - controversially discussed by biologists and computer scientists, since response time and quantity do not tell anything about quality and performance of the processing units as well as neurons and transistors cannot be compared directly. Nevertheless, the comparison serves its purpose and indicates the advantage of parallelism by means of processing time.

| | Brain | Computer |
|---|---|---|
| No. of processing units | $\approx 10^{11}$ | $\approx 10^{9}$ |
| Type of processing units | Neurons | Transistors |
| Type of calculation | massively parallel | usually serial |
| Data storage | associative | address-based |
| Switching time | $\approx 10^{-3}$s | $\approx 10^{-9}$s |
| Possible switching operations | $\approx 10^{13}\frac{1}{s}$ | $\approx 10^{18}\frac{1}{s}$ |
| Actual switching operations | $\approx 10^{12}\frac{1}{s}$ | $\approx 10^{10}\frac{1}{s}$ |

**Table 1.1:** The (flawed) comparison between brain and computer at a glance. **Inspired by:** [Zel94]

transistors with a switching time of $10^{-9}$ seconds. The brain contains $10^{11}$ neurons, but these only have a switching time of about $10^{-3}$ seconds.

The largest part of the brain is working continuously, while the largest part of the computer is only passive data storage. Thus, the brain is parallel and therefore performing close to its theoretical maximum, from which the computer is orders of magnitude away (Table 1.1). Additionally, a computer is static - the brain as a biological neural network can reorganize itself during its "lifespan" and therefore is able to learn, to compensate errors and so forth.

Within this text I want to outline how we can use the said characteristics of our brain for a computer system.

So the study of artificial neural networks is motivated by their similarity to successfully working biological systems, which - in comparison to the overall system - consist of very simple but numerous nerve cells that work massively in parallel and (which is probably one of the most significant aspects) have the ***capability to learn***. There is no need to explicitly program a neural network. For instance, it can learn from training samples or by means of encouragement - with a carrot and a stick, so to speak (*reinforcement learning*).

One result from this learning procedure is the capability of neural networks to ***generalize and associate data***: After successful training a neural network can find reasonable solutions for similar problems of the same class that were not explicitly trained. This in turn results in a high degree of ***fault tolerance*** against noisy input data.

Fault tolerance is closely related to biological neural networks, in which this characteristic is very distinct: As previously mentioned, a human has about $10^{11}$ neurons that

continuously reorganize themselves or are reorganized by external influences (about $10^5$ neurons can be destroyed while in a drunken stupor, some types of food or environmental influences can also destroy brain cells). Nevertheless, our cognitive abilities are not significantly affected. Thus, the brain is tolerant against internal errors – and also against external errors, for we can often read a really "dreadful scrawl" although the individual letters are nearly impossible to read.

Our modern technology, however, is not automatically fault-tolerant. I have never heard that someone forgot to install the hard disk controller into a computer and therefore the graphics card automatically took over its tasks, i.e. removed conductors and developed communication, so that the system as a whole was affected by the missing component, but not completely destroyed.

A disadvantage of this distributed fault-tolerant storage is certainly the fact that we cannot realize at first sight what a neural neutwork knows and performs or where its faults lie. Usually, it is easier to perform such analyses for conventional algorithms. Most often we can only transfer knowledge into our neural network by means of a *learning procedure*, which can cause several errors and is not always easy to manage.

Fault tolerance of data, on the other hand, is already more sophisticated in state-of-the-art technology: Let us compare a record and a CD. If there is a scratch on a record, the audio information on this spot will be completely lost (you will hear a pop) and then the music goes on. On a CD the audio data are *distributedly* stored: A scratch causes a blurry sound in its vicinity, but the data stream remains largely unaffected. The listener won't notice anything.

So let us summarize the main characteristics we try to adapt from biology:

- Self-organization and learning capability,
- Generalization capability and
- Fault tolerance.

What types of neural networks particularly develop what kinds of abilities and can be used for what problem classes will be discussed in the course of this work.

In the introductory chapter I want to clarify the following: "*The* neural network" does not exist. There are different paradigms for neural networks, how they are trained and where they are used. My goal is to introduce some of these paradigms and supplement some remarks for practical application.

We have already mentioned that our brain works massively in parallel, in contrast to the functioning of a computer, i.e. every component is active at any time. If we want

to state an argument for massive parallel processing, then the ***100-step rule*** can be cited.

### 1.1.1 The 100-step rule

Experiments showed that a human can recognize the picture of a familiar object or person in $\approx 0.1$ seconds, which corresponds to a neuron switching time of $\approx 10^{-3}$ seconds in $\approx 100$ discrete time steps of ***parallel*** processing.

A computer following the von Neumann architecture, however, can do practically nothing in 100 time steps of *sequential* processing, which are 100 assembler steps or cycle steps.

Now we want to look at a simple application example for a neural network.

### 1.1.2 Simple application examples

Let us assume that we have a small robot as shown in fig. 1.1 on the next page. This robot has eight distance sensors from which it extracts input data: Three sensors are placed on the front right, three on the front left, and two on the back. Each sensor provides a real numeric value at any time, that means we are always receiving an input $I \in \mathbb{R}^8$.

Despite its two motors (which will be needed later) the robot in our simple example is not capable to do much: It shall only drive on but stop when it might collide with an obstacle. Thus, our output is binary: $H = 0$ for "Everything is okay, drive on" and $H = 1$ for "Stop" (The output is called $H$ for "halt signal"). Therefore we need a mapping

$$f : \mathbb{R}^8 \to \mathbb{B}^1,$$

that applies the input signals to a robot activity.

#### 1.1.2.1 The classical way

There are two ways of realizing this mapping. On the one hand, there is the *classical way*: We sit down and think for a while, and finally the result is a circuit or a small computer program which realizes the mapping (this is easily possible, since the example is very simple). After that we refer to the technical reference of the sensors, study their characteristic curve in order to learn the values for the different obstacle distances, and embed these values into the aforementioned set of rules. Such procedures are applied

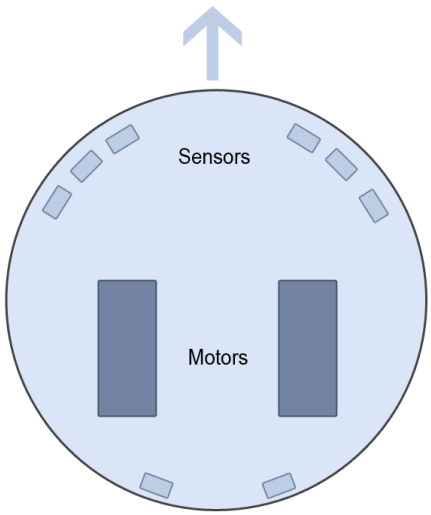



**Figure 1.1:** A small robot with eight sensors and two motors. The arrow indicates the driving direction.

in the classic artificial intelligence, and if you know the exact rules of a mapping algorithm, you are always well advised to follow this scheme.

#### 1.1.2.2 The way of learning

On the other hand, more interesting and more successful for many mappings and problems that are hard to comprehend straightaway is the *way of learning*: We show different possible situations to the robot (fig. 1.2 on the following page), – and the robot shall learn on its own what to do in the course of its robot life.

In this example the robot shall simply learn when to stop. We first treat the neural network as a kind of ***black box*** (fig. 1.3 on the next page). This means we do not know its structure but just regard its behavior in practice.

The situations in form of simply measured sensor values (e.g. placing the robot in front of an obstacle, see illustration), which we show to the robot and for which we specify whether to drive on or to stop, are called *training samples*. Thus, a training sample consists of an exemplary input and a corresponding desired output. Now the question is how to transfer this knowledge, the information, into the neural network.

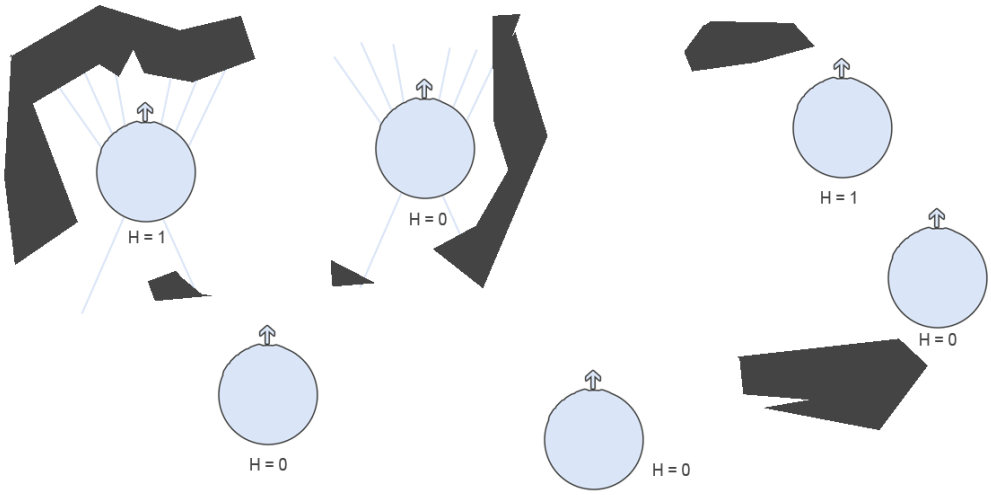



**Figure 1.2:** The robot is positioned in a landscape that provides sensor values for different situations. We add the desired output values $H$ and so receive our learning samples. The directions in which the sensors are oriented are exemplarily applied to two robots.

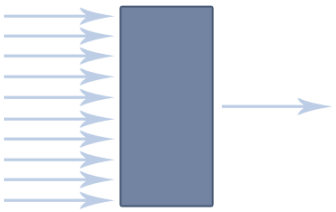


**Figure 1.3:** Initially, we regard the robot control as a black box whose inner life is unknown. The black box receives eight real sensor values and maps these values to a binary output value.

The samples can be taught to a neural network by using a simple *learning procedure* (a learning procedure is a simple algorithm or a mathematical formula. If we have done everything right and chosen good samples, the neural network will *generalize* from these samples and find a universal rule when it has to stop.

Our example can be optionally expanded. For the purpose of direction control it would be possible to control the motors of our robot separately[2], with the sensor layout being the same. In this case we are looking for a mapping

$$f : \mathbb{R}^8 \to \mathbb{R}^2,$$

which gradually controls the two motors by means of the sensor inputs and thus cannot only, for example, stop the robot but also lets it avoid obstacles. Here it is more difficult to analytically derive the rules, and de facto a neural network would be more appropriate.

Our goal is not to learn the samples by heart, but to realize the *principle* behind them: Ideally, the robot should apply the neural network in any situation and be able to avoid obstacles. In particular, the robot should query the network continuously and repeatedly *while* driving in order to continously avoid obstacles. The result is a constant cycle: The robot queries the network. As a consequence, it will drive in one direction, which changes the sensors values. Again the robot queries the network and changes its position, the sensor values are changed once again, and so on. It is obvious that this system can also be adapted to dynamic, i.e changing, environments (e.g. the moving obstacles in our example).

## 1.2 A brief history of neural networks

The field of neural networks has, like any other field of science, a long ***history of development*** with many ups and downs, as we will see soon. To continue the style of my work I will not represent this history in text form but more compact in form of a timeline. Citations and bibliographical references are added mainly for those topics that will not be further discussed in this text. Citations for keywords that will be explained later are mentioned in the corresponding chapters.

The history of neural networks begins in the early 1940's and thus nearly simultaneously with the history of programmable electronic computers. The youth of this field of

2 There is a robot called *Khepera* with more or less similar characteristics. It is round-shaped, approx. 7 cm in diameter, has two motors with wheels and various sensors. For more information I recommend to refer to the internet.

**Figure 1.4:** Some institutions of the field of neural networks. From left to right: John von Neumann, Donald O. Hebb, Marvin Minsky, Bernard Widrow, Seymour Papert, Teuvo Kohonen, John Hopfield, "in the order of appearance" as far as possible.

research, as with the field of computer science itself, can be easily recognized due to the fact that many of the cited persons are still with us.

### 1.2.1 The beginning

**As soon as 1943** Warren McCulloch and Walter Pitts introduced models of neurological networks, recreated threshold switches based on neurons and showed that even simple networks of this kind are able to calculate nearly any logic or arithmetic function [MP43]. Furthermore, the first computer precursors ("***electronic brains***")were developed, among others supported by Konrad Zuse, who was tired of calculating ballistic trajectories by hand.

**1947:** Walter Pitts and Warren McCulloch indicated a practical field of application (which was not mentioned in their work from 1943), namely the recognition of spacial patterns by neural networks [PM47].

**1949:** Donald O. Hebb formulated the classical *Hebbian rule* [Heb49] which represents in its more generalized form the basis of nearly all neural learning procedures. The rule implies that the connection between two neurons is strengthened when both neurons are *active at the same time*. This change in strength is proportional to the product of the two activities. Hebb could postulate this rule, but due to the absence of neurological research he was not able to verify it.

**1950:** The neuropsychologist Karl Lashley defended the thesis that brain information storage is realized as a *distributed system*. His thesis was based on experiments on rats, where only the extent but not the location of the destroyed nerve tissue influences the rats' performance to find their way out of a labyrinth.

### 1.2.2 Golden age

**1951:** For his dissertation MARVIN MINSKY developed the neurocomputer ***Snark***, which has already been capable to adjust its weights[3] automatically. But it has never been practically implemented, since it is capable to busily calculate, but nobody really knows what it calculates.

**1956:** Well-known scientists and ambitious students met at the ***Dartmouth Summer Research Project*** and discussed, to put it crudely, how to simulate a brain. Differences between top-down and bottom-up research developed. While the early supporters of ***artificial intelligence*** wanted to *simulate* capabilities by means of software, supporters of neural networks wanted to achieve system behavior by imitating the smallest parts of the system – the neurons.

**1957-1958:** At the MIT, FRANK ROSENBLATT, CHARLES WIGHTMAN and their coworkers developed the first successful neurocomputer, the ***Mark I perceptron*** , which was capable to recognize simple numerics by means of a $20 \times 20$ pixel image sensor and electromechanically worked with 512 motor driven potentiometers - each potentiometer representing one variable weight.

**1959:** FRANK ROSENBLATT described different versions of the perceptron, formulated and verified his *perceptron convergence theorem*. He described neuron layers mimicking the retina, threshold switches, and a learning rule adjusting the connecting weights.

**1960:** BERNARD WIDROW and MARCIAN E. HOFF introduced the ***ADALINE*** (***ADAptive LInear NEuron***) [WH60], a fast and precise adaptive learning system being the first widely commercially used neural network: It could be found in nearly every analog telephone for real-time adaptive echo filtering and was trained by menas of the *Widrow-Hoff rule* or *delta rule*. At that time Hoff, later co-founder of Intel Corporation, was a PhD student of Widrow, who himself is known as the inventor of modern microprocessors. One advantage the delta rule had over the original perceptron learning algorithm was its *adaptivity*: If the difference between the actual output and the correct solution was large, the connecting weights also changed in larger steps – the smaller the steps, the closer the target was. Disadvantage: missapplication led to infinitesimal small steps close to the target. In the following stagnation and out of fear of scientific unpopularity of the neural networks ADALINE was renamed in ***adaptive linear element*** – which was undone again later on.

3 We will learn soon what weights are.

**1961:** Karl Steinbuch introduced technical realizations of associative memory, which can be seen as predecessors of today's neural associative memories [Ste61]. Additionally, he described concepts for neural techniques and analyzed their possibilities and limits.

**1965:** In his book *Learning Machines*, Nils Nilsson gave an overview of the progress and works of this period of neural network research. It was assumed that the basic principles of self-learning and therefore, generally speaking, "intelligent" systems had already been discovered. Today this assumption seems to be an exorbitant overestimation, but at that time it provided for high popularity and sufficient research funds.

**1969:** Marvin Minsky and Seymour Papert published a precise mathematical analysis of the perceptron [MP69] to show that the perceptron model was not capable of representing many important problems (keywords: *XOR problem* and *linear separability*), and so put an end to overestimation, popularity and research funds. The implication that more powerful models would show exactly the same problems and the forecast that the entire field would be a *research dead end* resulted in a nearly complete decline in research funds for the next 15 years – no matter how incorrect these forecasts were from today's point of view.

### 1.2.3 Long silence and slow reconstruction

The research funds were, as previously-mentioned, extremely short. Everywhere research went on, but there were neither conferences nor other events and therefore only few publications. This isolation of individual researchers provided for many independently developed neural network paradigms: They researched, but there was no discourse among them.

In spite of the poor appreciation the field received, the basic theories for the still continuing renaissance were laid at that time:

**1972:** Teuvo Kohonen introduced a model of the ***linear associator***, a model of an associative memory [Koh72]. In the same year, such a model was presented independently and from a neurophysiologist's point of view by James A. Anderson [And72].

**1973:** Christoph von der Malsburg used a neuron model that was non-linear and biologically more motivated [vdM73].

**1974:** For his dissertation in Harvard Paul Werbos developed a learning procedure called *backpropagation of error* [Wer74], but it was not until one decade later that this procedure reached today's importance.

**1976-1980 and thereafter:** Stephen Grossberg presented many papers (for instance [Gro76]) in which numerous neural models are analyzed mathematically. Furthermore, he dedicated himself to the problem of keeping a neural network capable of learning without destroying already learned associations. Under cooperation of Gail Carpenter this led to models of ***adaptive resonance theory*** (ART).

**1982:** Teuvo Kohonen described the ***self-organizing feature maps*** (SOM) [Koh82, Koh98] – also known as Kohonen maps. He was looking for the mechanisms involving self-organization in the brain (He knew that the information about the creation of a being is stored in the genome, which has, however, not enough memory for a structure like the brain. As a consequence, the brain has to organize and create itself for the most part).

John Hopfield also invented the so-called Hopfield networks [Hop82] which are inspired by the laws of magnetism in physics. They were not widely used in technical applications, but the field of neural networks slowly regained importance.

**1983:** Fukushima, Miyake and Ito introduced the neural model of the ***Neocognitron*** which could recognize handwritten characters [FMI83] and was an extension of the Cognitron network already developed in 1975.

### 1.2.4 Renaissance

Through the influence of John Hopfield, who had personally convinced many researchers of the importance of the field, and the wide publication of backpropagation by Rumelhart, Hinton and Williams, the field of neural networks slowly showed signs of upswing.

**1985:** John Hopfield published an article describing a way of finding acceptable solutions for the Travelling Salesman problem by using *Hopfield nets*.

**1986:** The *backpropagation of error* learning procedure as a generalization of the delta rule was separately developed and widely published by the *Parallel Distributed Processing* Group [RHW86a]: Non-linearly-separable problems could be solved by multilayer perceptrons, and Marvin Minsky's negative evaluations were disproven at a single blow. At the same time a certain kind of fatigue spread in the field of artificial intelligence, caused by a series of failures and unfulfilled hopes.

**From this time on,** the development of the field of research has almost been explosive. It can no longer be itemized, but some of its results will be seen in the following.

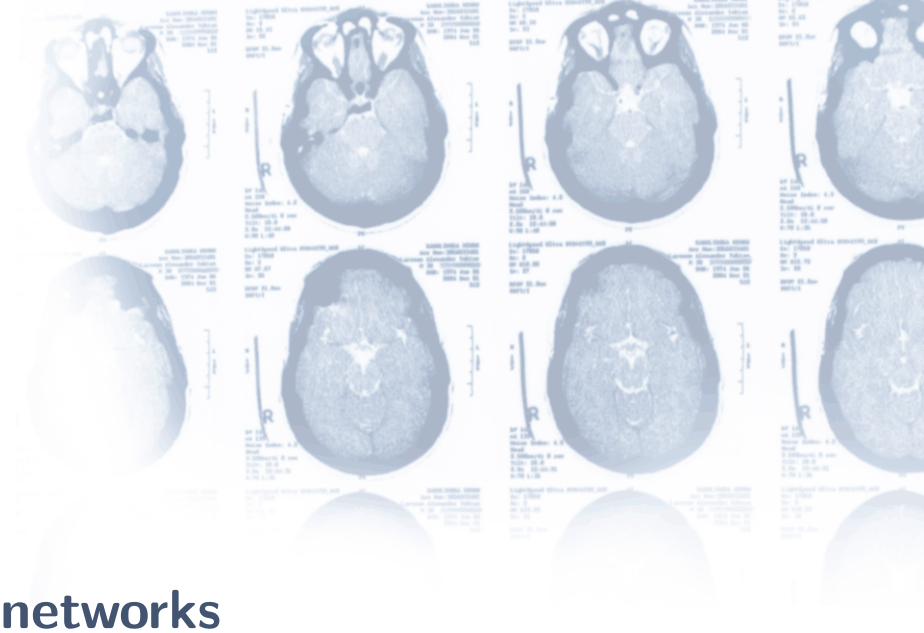
## Exercises

**Exercise 1.** Give one example for each of the following topics:

- A book on neural networks or neuroinformatics,
- A collaborative group of a university working with neural networks,
- A software tool realizing neural networks ("simulator"),
- A company using neural networks, and
- A product or service being realized by means of neural networks.

**Exercise 2.** Show at least four applications of technical neural networks: two from the field of pattern recognition and two from the field of function approximation.

**Exercise 3.** Briefly characterize the four development phases of neural networks and give expressive examples for each phase.



# Chapter 2

# Biological neural networks

How do biological systems solve problems? How does a system of neurons work? How can we understand its functionality? What are different quantities of neurons able to do? Where in the nervous system does information processing occur? A short biological overview of the complexity of simple elements of neural information processing followed by some thoughts about their simplification in order to technically adapt them.

Before we begin to describe the technical side of neural networks, it would be useful to briefly discuss the biology of neural networks and the cognition of living organisms – the reader may skip the following chapter without missing any technical information. On the other hand I recommend to read the said excursus if you want to learn something about the underlying neurophysiology and see that our small approaches, the technical neural networks, are only caricatures of nature – and how powerful their natural counterparts must be when our small approaches are already that effective. Now we want to take a brief look at the nervous system of vertebrates: We will start with a very rough granularity and then proceed with the brain and up to the neural level. For further reading I want to recommend the books [CR00,KSJ00], which helped me a lot during this chapter.

## 2.1 The vertebrate nervous system

The entire information processing system, i.e. the vertebrate ***nervous system***, consists of the central nervous system and the peripheral nervous system, which is only a first and simple subdivision. In reality, such a rigid subdivision does not make sense, but here it is helpful to outline the information processing in a body.

### 2.1.1 Peripheral and central nervous system

The ***peripheral nervous system*** (***PNS***) comprises the nerves that are situated outside of the brain or the spinal cord. These nerves form a branched and very dense network throughout the whole body. The peripheral nervous system includes, for example, the spinal nerves which pass out of the spinal cord (two within the level of each vertebra of the spine) and supply extremities, neck and trunk, but also the cranial nerves directly leading to the brain.

The ***central nervous system*** (***CNS***), however, is the "main-frame" within the vertebrate. It is the place where information received by the sense organs are stored and managed. Furthermore, it controls the inner processes in the body and, last but not least, coordinates the motor functions of the organism. The vertebrate central nervous system consists of the ***brain*** and the ***spinal cord*** (Fig. 2.1 on the facing page). However, we want to focus on the brain, which can - for the purpose of simplification - be divided into four areas (Fig. 2.2 on page 18) to be discussed here.

### 2.1.2 The cerebrum is responsible for abstract thinking processes.

The ***cerebrum*** (***telencephalon***) is one of the areas of the brain that changed most during evolution. Along an axis, running from the lateral face to the back of the head, this area is divided into two hemispheres, which are organized in a folded structure. These cerebral hemispheres are connected by one strong nerve cord ("***bar***") and several small ones. A large number of neurons are located in the ***cerebral cortex*** (***cortex***) which is approx. 2-4 cm thick and divided into different ***cortical fields***, each having a specific task to fulfill. ***Primary cortical fields*** are responsible for processing qualitative information, such as the management of different perceptions (e.g. the ***visual cortex*** is responsible for the management of vision). ***Association cortical fields***, however, perform more abstract association and thinking processes; they also contain our memory.

### 2.1.3 The cerebellum controls and coordinates motor functions

The ***cerebellum*** is located below the cerebrum, therefore it is closer to the spinal cord. Accordingly, it serves less abstract functions with higher priority: Here, large parts of motor coordination are performed, i.e., balance and movements are controlled and errors are continually corrected. For this purpose, the cerebellum has direct sensory

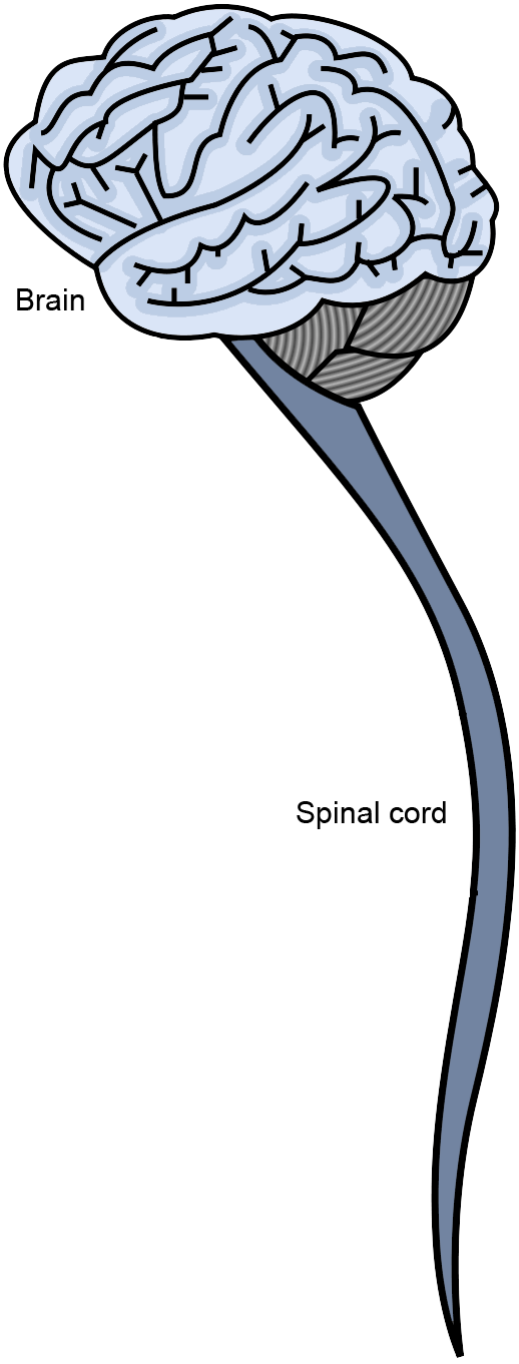



**Figure 2.1:** Illustration of the central nervous system with spinal cord and brain.

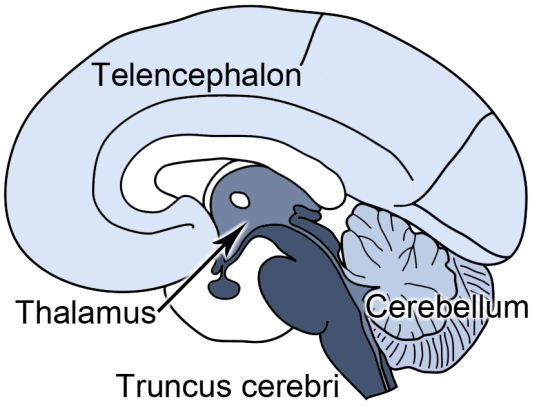



**Figure 2.2:** Illustration of the brain. The colored areas of the brain are discussed in the text. The more we turn from abstract information processing to direct reflexive processing, the darker the areas of the brain are colored.

information about muscle lengths as well as acoustic and visual information. Furthermore, it also receives messages about more abstract motor signals coming from the cerebrum.

In the human brain the cerebellum is considerably smaller than the cerebrum, but this is rather an exception. In many vertebrates this ratio is less pronounced. If we take a look at vertebrate evolution, we will notice that the cerebellum is not "too small" but the cerebum is "too large" (at least, it is the most highly developed structure in the vertebrate brain). The two remaining brain areas should also be briefly discussed: the diencephalon and the brainstem.

### 2.1.4 The diencephalon controls fundamental physiological processes

The ***interbrain*** (***diencephalon***) includes parts of which only the ***thalamus*** will be briefly discussed: This part of the diencephalon mediates between sensory and motor signals and the cerebrum. Particularly, the thalamus decides which part of the information is transferred to the cerebrum, so that especially less important sensory perceptions can be suppressed at short notice to avoid overloads. Another part of the diencephalon is the ***hypothalamus***, which controls a number of processes within the body. The diencephalon is also heavily involved in the human circadian rhythm ("internal clock") and the sensation of pain.

### 2.1.5 The brainstem connects the brain with the spinal cord and controls reflexes.

In comparison with the diencephalon the ***brainstem*** or the (***truncus cerebri***) respectively is phylogenetically much older. Roughly speaking, it is the "extended spinal cord" and thus the connection between brain and spinal cord. The brainstem can also be divided into different areas, some of which will be exemplarily introduced in this chapter. The functions will be discussed from abstract functions towards more fundamental ones. One important component is the ***pons*** (=bridge), a kind of transit station for many nerve signals from brain to body and vice versa.

If the pons is damaged (e.g. by a cerebral infarct), then the result could be the ***locked-in syndrome*** – a condition in which a patient is "walled-in" within his own body. He is conscious and aware with no loss of cognitive function, but cannot move or communicate by any means. Only his senses of sight, hearing, smell and taste are generally working perfectly normal. Locked-in patients may often be able to communicate with others by blinking or moving their eyes.

Furthermore, the brainstem is responsible for many fundamental reflexes, such as the blinking reflex or coughing.

All parts of the nervous system have one thing in common: information processing. This is accomplished by huge accumulations of billions of very similar cells, whose structure is very simple but which communicate continuously. Large groups of these cells send coordinated signals and thus reach the enormous information processing capacity we are familiar with from our brain. We will now leave the level of brain areas and continue with the cellular level of the body - the level of neurons.

## 2.2 Neurons are information processing cells

Before specifying the functions and processes within a neuron, we will give a rough description of neuron functions: A neuron is nothing more than a switch with information input and output. The switch will be activated if there are enough stimuli of other neurons hitting the information input. Then, at the information output, a pulse is sent to, for example, other neurons.

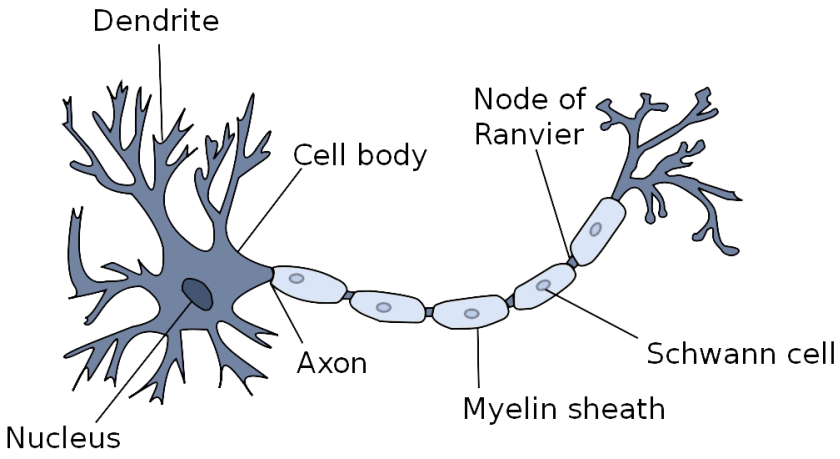



**Figure 2.3:** Illustration of a biological neuron with the components discussed in this text.

### 2.2.1 Components of a neuron

Now we want to take a look at the components of a neuron (Fig. 2.3). In doing so, we will follow the way the electrical information takes within the neuron. The dendrites of a neuron receive the information by special connections, the synapses.

#### 2.2.1.1 Synapses weight the individual parts of information

Incoming signals from other neurons or cells are transferred to a neuron by special connections, the ***synapses***. Such connections can usually be found at the dendrites of a neuron, sometimes also directly at the soma. We distinguish between electrical and chemical synapses.

The ***electrical synapse*** is the simpler variant. An electrical signal received by the synapse, i.e. coming from the *presynaptic* side, is directly transferred to the *postsynaptic* nucleus of the cell. Thus, there is a direct, strong, unadjustable connection between the signal transmitter and the signal receiver, which is, for example, relevant to shortening reactions that must be "hard coded" within a living organism.

The ***chemical synapse*** is the more distinctive variant. Here, the electrical coupling of source and target does not take place, the coupling is interrupted by the ***synaptic cleft***. This cleft electrically separates the presynaptic side from the postsynaptic one.

You might think that, nevertheless, the information has to flow, so we will discuss how this happens: It is not an electrical, but a *chemical* process. On the presynaptic side of the synaptic cleft the electrical signal is converted into a chemical signal, a process induced by chemical cues released there (the so-called ***neurotransmitters***). These neurotransmitters cross the synaptic cleft and transfer the information into the nucleus of the cell (this is a very simple explanation, but later on we will see how this exactly works), where it is reconverted into electrical information. The neurotransmitters are degraded very fast, so that it is possible to release very precise information pulses here, too.

In spite of the more complex functioning, the chemical synapse has - compared with the electrical synapse - utmost advantages:

**One-way connection:** A chemical synapse is a one-way connection. Due to the fact that there is no direct electrical connection between the pre- and postsynaptic area, electrical pulses in the postsynaptic area cannot flash over to the presynaptic area.

**Adjustability:** There is a large number of different neurotransmitters that can also be released in various quantities in a synaptic cleft. There are neurotransmitters that stimulate the postsynaptic cell nucleus, and others that slow down such stimulation. Some synapses transfer a strongly stimulating signal, some only weakly stimulating ones. The adjustability varies a lot, and one of the central points in the examination of the learning ability of the brain is, that here the synapses are variable, too. That is, over time they can form a stronger or weaker connection.

#### 2.2.1.2 Dendrites collect all parts of information

***Dendrites*** branch like trees from the cell nucleus of the neuron (which is called *soma*) and receive electrical signals from many different sources, which are then transferred into the nucleus of the cell. The amount of branching dendrites is also called ***dendrite tree***.

#### 2.2.1.3 In the soma the weighted information is accumulated

After the cell nucleus (***soma***) has received a plenty of activating (=stimulating) and inhibiting (=diminishing) signals by synapses or dendrites, the soma accumulates these signals. As soon as the accumulated signal exceeds a certain value (called threshold

value), the cell nucleus of the neuron activates an electrical pulse which then is transmitted to the neurons connected to the current one.

#### 2.2.1.4 The axon transfers outgoing pulses

The pulse is transferred to other neurons by means of the ***axon***. The axon is a long, slender extension of the soma. In an extreme case, an axon can stretch up to one meter (e.g. within the spinal cord). The axon is electrically isolated in order to achieve a better conduction of the electrical signal (we will return to this point later on) and it leads to dendrites, which transfer the information to, for example, other neurons. So now we are back at the beginning of our description of the neuron elements. An axon can, however, transfer information to other kinds of cells in order to control them.

### 2.2.2 Electrochemical processes in the neuron and its components

After having pursued the path of an electrical signal from the dendrites via the synapses to the nucleus of the cell and from there via the axon into other dendrites, we now want to take a small step from biology towards technology. In doing so, a simplified introduction of the electrochemical information processing should be provided.

#### 2.2.2.1 Neurons maintain electrical membrane potential

One fundamental aspect is the fact that compared to their environment the neurons show a difference in electrical charge, a *potential*. In the ***membrane*** (=envelope) of the neuron the charge is different from the charge on the outside. This difference in charge is a central concept that is important to understand the processes within the neuron. The difference is called ***membrane potential***. The membrane potential, i.e., the difference in charge, is created by several kinds of charged atoms (***ions***), whose concentration varies within and outside of the neuron. If we penetrate the membrane from the inside outwards, we will find certain kinds of ions more often or less often than on the inside. This descent or ascent of concentration is called a ***concentration gradient***.

Let us first take a look at the membrane potential in the resting state of the neuron, i.e., we assume that no electrical signals are received from the outside. In this case, the membrane potential is $-70$ mV. Since we have learned that this potential depends on the concentration gradients of various ions, there is of course the central question of how to maintain these concentration gradients: Normally, diffusion predominates

and therefore each ion is eager to decrease concentration gradients and to spread out evenly. If this happens, the membrane potential will move towards 0 mV, so finally there would be no membrane potential anymore. Thus, the neuron actively maintains its membrane potential to be able to process information. How does this work?

The secret is the membrane itself, which is permeable to some ions, but not for others. To maintain the potential, various mechanisms are in progress at the same time:

**Concentration gradient:** As described above the ions try to be as uniformly distributed as possible. If the concentration of an ion is higher on the inside of the neuron than on the outside, it will try to diffuse to the outside and vice versa. The positively charged ion $K^+$ (potassium) occurs very frequently within the neuron but less frequently outside of the neuron, and therefore it slowly diffuses out through the neuron's membrane. But another group of negative ions, collectively called $A^-$, remains within the neuron since the membrane is not permeable to them. Thus, the inside of the neuron becomes negatively charged. Negative A ions remain, positive K ions disappear, and so the inside of the cell becomes more negative. The result is another gradient.

**Electrical Gradient:** The electrical gradient acts contrary to the concentration gradient. The intracellular charge is now very strong, therefore it attracts positive ions: $K^+$ wants to get back into the cell.

If these two gradients were now left alone, they would eventually balance out, reach a steady state, and a membrane potential of $-85$ mV would develop. But we want to achieve a resting membrane potential of $-70$ mV, thus there seem to exist some disturbances which prevent this. Furthermore, there is another important ion, $Na^+$ (sodium), for which the membrane is not very permeable but which, however, slowly pours through the membrane into the cell. As a result, the sodium is driven into the cell all the more: On the one hand, there is less sodium within the neuron than outside the neuron. On the other hand, sodium is positively charged but the interior of the cell has negative charge, which is a second reason for the sodium wanting to get into the cell.

Due to the low diffusion of sodium into the cell the intracellular sodium concentration increases. But at the same time the inside of the cell becomes less negative, so that $K^+$ pours in more slowly (we can see that this is a complex mechanism where everything is influenced by everything). The sodium shifts the intracellular equilibrium from negative to less negative, compared with its environment. But even with these two ions a standstill with all gradients being balanced out could still be achieved. Now the last piece of the puzzle gets into the game: a "pump" (or rather, the protein ***ATP***) actively transports ions against the direction they actually want to take!

**Sodium** is actively pumped out of the cell, although it tries to get into the cell along the concentration gradient and the electrical gradient.

**Potassium,** however, diffuses strongly out of the cell, but is actively pumped back into it.

For this reason the pump is also called ***sodium-potassium pump***. The pump maintains the concentration gradient for the sodium as well as for the potassium, so that some sort of steady state equilibrium is created and finally the resting potential is $-70$ mV as observed. All in all the membrane potential is maintained by the fact that the membrane is impermeable to some ions and other ions are actively pumped against the concentration and electrical gradients. Now that we know that each neuron has a membrane potential we want to observe how a neuron receives and transmits signals.

#### 2.2.2.2 The neuron is activated by changes in the membrane potential

Above we have learned that sodium and potassium can diffuse through the membrane - sodium slowly, potassium faster. They move through channels within the membrane, the sodium and potassium channels. In addition to these permanently open channels responsible for diffusion and balanced by the sodium-potassium pump, there also exist channels that are not always open but which only response "if required". Since the opening of these channels changes the concentration of ions within and outside of the membrane, it also changes the membrane potential.

These controllable channels are opened as soon as the accumulated received stimulus exceeds a certain threshold. For example, stimuli can be received from other neurons or have other causes. There exist, for example, specialized forms of neurons, the sensory cells, for which a light incidence could be such a stimulus. If the incoming amount of light exceeds the threshold, controllable channels are opened.

The said threshold (the ***threshold potential***) lies at about $-55$ mV. As soon as the received stimuli reach this value, the neuron is activated and an electrical signal, an ***action potential***, is initiated. Then this signal is transmitted to the cells connected to the observed neuron, i.e. the cells "listen" to the neuron. Now we want to take a closer look at the different stages of the action potential (Fig. 2.4 on the next page):

**Resting state:** Only the permanently open sodium and potassium channels are permeable. The membrane potential is at $-70$ mV and actively kept there by the neuron.

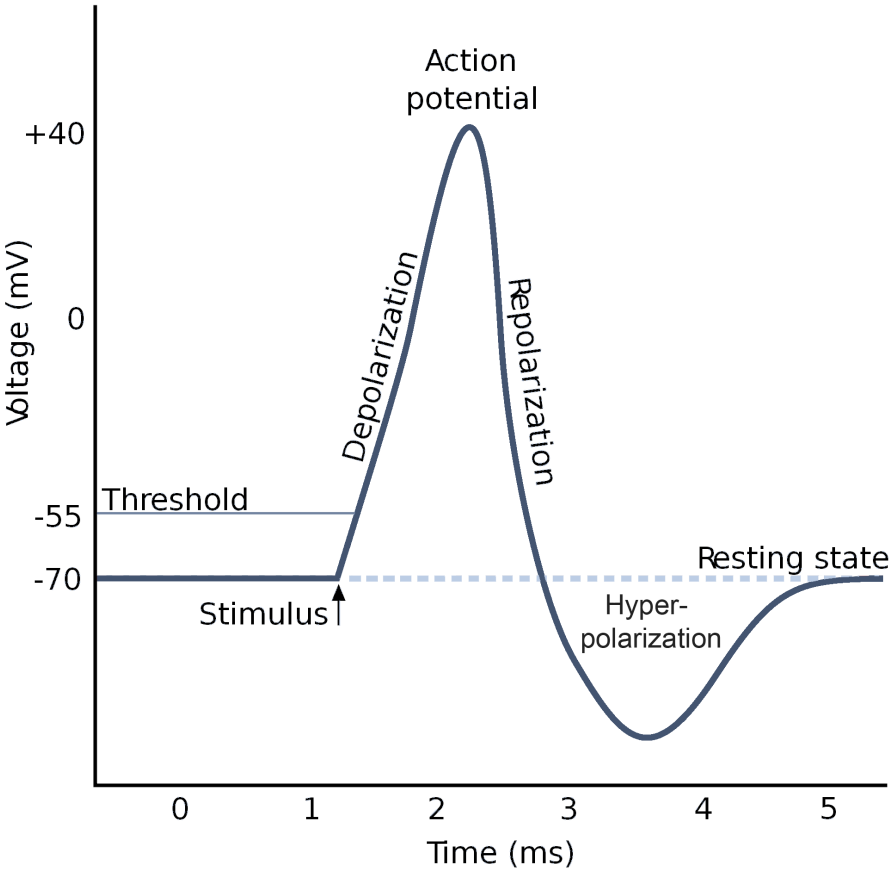



**Figure 2.4:** Initiation of action potential over time.

**Stimulus up to the threshold:** A ***stimulus*** opens channels so that sodium can pour in. The intracellular charge becomes more positive. As soon as the membrane potential exceeds the threshold of $-55$ mV, the action potential is initiated by the opening of many sodium channels.

**Depolarization:** Sodium is pouring in. Remember: Sodium wants to pour into the cell because there is a lower intracellular than extracellular concentration of sodium. Additionally, the cell is dominated by a negative environment which attracts the positive sodium ions. This massive influx of sodium drastically increases the membrane potential - up to approx. $+30$ mV - which is the electrical pulse, i.e., the action potential.

**Repolarization:** Now the sodium channels are closed and the potassium channels are opened. The positively charged ions want to leave the positive interior of the cell. Additionally, the intracellular concentration is much higher than the extracellular one, which increases the efflux of ions even more. The interior of the cell is once again more negatively charged than the exterior.

**Hyperpolarization:** Sodium as well as potassium channels are closed again. At first the membrane potential is slightly more negative than the resting potential. This is due to the fact that the potassium channels close more slowly. As a result, (positively charged) potassium effuses because of its lower extracellular concentration. After a ***refractory period*** of $1-2$ ms the resting state is re-established so that the neuron can react to newly applied stimuli with an action potential. In simple terms, the refractory period is a mandatory break a neuron has to take in order to regenerate. The shorter this break is, the more often a neuron can fire per time.

Then the resulting pulse is transmitted by the axon.

#### 2.2.2.3 In the axon a pulse is conducted in a saltatory way

We have already learned that the ***axon*** is used to transmit the action potential across long distances (remember: You will find an illustration of a neuron including an axon in Fig. 2.3 on page 20). The axon is a long, slender extension of the soma. In vertebrates it is normally coated by a ***myelin sheath*** that consists of ***Schwann cells*** (in the PNS) or ***oligodendrocytes*** (in the CNS) [1], which insulate the axon very well from electrical activity. At a distance of $0.1-2$mm there are gaps between these cells, the

1 Schwann cells as well as oligodendrocytes are varieties of the ***glial cells***. There are about 50 times more glial cells than neurons: They surround the neurons (glia = glue), insulate them from each other, provide energy, etc.

so-called ***nodes of Ranvier***. The said gaps appear where one insulate cell ends and the next one begins. It is obvious that at such a node the axon is less insulated.

Now you may assume that these less insulated nodes are a disadvantage of the axon - however, they are not. At the nodes, mass can be transferred between the intracellular and extracellular area, a transfer that is impossible at those parts of the axon which are situated between two nodes (***internodes***) and therefore insulated by the myelin sheath. This mass transfer permits the generation of signals similar to the generation of the action potential within the soma. The action potential is transferred as follows: It does not continuously travel along the axon but jumps from node to node. Thus, a series of depolarization travels along the nodes of Ranvier. One action potential initiates the next one, and mostly even several nodes are active at the same time here. The pulse "jumping" from node to node is responsible for the name of this pulse conductor: ***saltatory conductor***.

Obviously, the pulse will move faster if its jumps are larger. Axons with large internodes (2 mm) achieve a signal dispersion of approx. 180 meters per second. However, the internodes cannot grow indefinitely, since the action potential to be transferred would fade too much until it reaches the next node. So the nodes have a task, too: to constantly amplify the signal. The cells receiving the action potential are attached to the end of the axon – often connected by dendrites and synapses. As already indicated above, the action potentials are not only generated by information received by the dendrites from other neurons.

## 2.3 Receptor cells are modified neurons

Action potentials can also be generated by sensory information an organism receives from its environment through its sensory cells. Specialized ***receptor cells*** are able to perceive specific stimulus energies such as light, temperature and sound or the existence of certain molecules (like, for example, the sense of smell). This is working because of the fact that these sensory cells are actually modified neurons. They do not receive electrical signals via dendrites but the existence of the stimulus being specific for the receptor cell ensures that the ion channels open and an action potential is developed. This process of transforming stimulus energy into changes in the membrane potential is called ***sensory transduction***. Usually, the stimulus energy itself is too weak to directly cause nerve signals. Therefore, the signals are amplified either during transduction or by means of the ***stimulus-conducting apparatus***. The resulting action potential can be processed by other neurons and is then transmitted into the thalamus, which is, as we have already learned, a gateway to the cerebral cortex and

therefore can reject sensory impressions according to current relevance and thus prevent an abundance of information to be managed.

### 2.3.1 There are different receptor cells for various types of perceptions

***Primary receptors*** transmit their pulses directly to the nervous system. A good example for this is the sense of pain. Here, the stimulus intensity is proportional to the amplitude of the action potential. Technically, this is an amplitude modulation.

***Secondary receptors***, however, continuously transmit pulses. These pulses control the amount of the related neurotransmitter, which is responsible for transferring the stimulus. The stimulus in turn controls the frequency of the action potential of the receiving neuron. This process is a frequency modulation, an encoding of the stimulus, which allows to better perceive the increase and decrease of a stimulus.

There can be individual receptor cells or cells forming complex sensory organs (e.g. eyes or ears). They can receive stimuli within the body (by means of the ***interoceptors***) as well as stimuli outside of the body (by means of the ***exteroceptors***).

After having outlined how information is received from the environment, it will be interesting to look at how the information is *processed*.

### 2.3.2 Information is processed on every level of the nervous system

There is no reason to believe that all received information is transmitted to the brain and processed there, and that the brain ensures that it is "output" in the form of motor pulses (the only thing an organism can actually do within its environment is to move). The information processing is entirely decentralized. In order to illustrate this principle, we want to take a look at some examples, which leads us again from the abstract to the fundamental in our hierarchy of information processing.

- ▷ It is certain that information is processed in the cerebrum, which is the most developed natural information processing structure.
- ▷ The midbrain and the thalamus, which serves – as we have already learned – as a gateway to the cerebral cortex, are situated much lower in the hierarchy. The filtering of information with respect to the current relevance executed by the midbrain is a very important method of information processing, too. But even the thalamus does not receive any preprocessed stimuli from the outside. Now let us continue with the lowest level, the sensory cells.

- On the lowest level, i.e. at the receptor cells, the information is not only received and transferred but directly processed. One of the main aspects of this subject is to prevent the transmission of "continuous stimuli" to the central nervous system because of ***sensory adaptation***: Due to continuous stimulation many receptor cells automatically become insensitive to stimuli. Thus, receptor cells are not a direct mapping of specific stimulus energy onto action potentials but depend on the past. Other sensors change their sensitivity according to the situation: There are taste receptors which respond more or less to the same stimulus according to the nutritional condition of the organism.
- Even *before* a stimulus reaches the receptor cells, information processing can already be executed by a preceding signal carrying apparatus, for example in the form of amplification: The external and the internal ear have a specific shape to amplify the sound, which also allows – in association with the sensory cells of the sense of hearing – the sensory stimulus only to increase *logarithmically* with the intensity of the heard signal. On closer examination, this is necessary, since the sound pressure of the signals for which the ear is constructed can vary over a wide exponential range. Here, a logarithmic measurement is an advantage. Firstly, an overload is prevented and secondly, the fact that the intensity measurement of intensive signals will be less precise, doesn't matter as well. If a jet fighter is starting next to you, small changes in the noise level can be ignored.

Just to get a feeling for sensory organs and information processing in the organism, we will briefly describe "usual" light sensing organs, i.e. organs often found in nature. For the third light sensing organ described below, the single lens eye, we will discuss the information processing in the eye.

### 2.3.3 An outline of common light sensing organs

For many organisms it turned out to be extremely useful to be able to perceive electromagnetic radiation in certain regions of the spectrum. Consequently, sensory organs have been developed which can detect such electromagnetic radiation and the wavelength range of the radiation perceivable by the human eye is called *visible range* or simply *light*. The different wavelengths of this electromagnetic radiation are perceived by the human eye as different colors. The visible range of the electromagnetic radiation is different for each organism. Some organisms cannot see the colors (=wavelength ranges) we can see, others can even perceive additional wavelength ranges (e.g. in the UV range). Before we begin with the human being – in order to get a broader knowledge of the sense of sight– we briefly want to look at two organs of sight which, from an evolutionary point of view, exist much longer than the human.

**Figure 2.5:** Compound eye of a robber fly

#### 2.3.3.1 Compound eyes and pinhole eyes only provide high temporal or spatial resolution

Let us first take a look at the so-called ***compound eye*** (Fig. 2.5), which is, for example, common in insects and crustaceans. The compound eye consists of a great number of small, ***individual eyes***. If we look at the compound eye from the outside, the individual eyes are clearly visible and arranged in a hexagonal pattern. Each individual eye has its own nerve fiber which is connected to the insect brain. Since the individual eyes can be distinguished, it is obvious that the number of pixels, i.e. the spatial resolution, of compound eyes must be very low and the image is blurred. But compound eyes have advantages, too, especially for fast-flying insects. Certain compound eyes process more than 300 images per second (to the human eye, however, movies with 25 images per second appear as a fluent motion).

***Pinhole eyes*** are, for example, found in octopus species and work – as you can guess – similar to a pinhole camera. A pinhole eye has a very small opening for light entry, which projects a sharp image onto the sensory cells behind. Thus, the spatial resolution is much higher than in the compound eye. But due to the very small opening for light entry the resulting image is less bright.

#### 2.3.3.2 Single lens eyes combine the advantages of the other two eye types, but they are more complex

The light sensing organ common in vertebrates is the ***single lense eye***. The resulting image is a sharp, high-resolution image of the environment at high or variable light intensity. On the other hand it is more complex. Similar to the pinhole eye the light enters through an opening (***pupil***) and is projected onto a layer of sensory cells in the eye. (***retina***). But in contrast to the pinhole eye, the size of the pupil can be adapted to the lighting conditions (by means of the ***iris*** muscle, which expands or contracts the pupil). These differences in pupil dilation require to actively focus the image. Therefore, the single lens eye contains an additional adjustable ***lens***.

#### 2.3.3.3 The retina does not only receive information but is also responsible for information processing

The light signals falling on the eye are received by the retina and directly preprocessed by several layers of information-processing cells. We want to briefly discuss the different steps of this information processing and in doing so, we follow the way of the information carried by the light:

**Photoreceptors** receive the light signal und cause action potentials (there are different receptors for different color components and light intensities). These receptors are the real light-receiving part of the retina and they are sensitive to such an extent that only one single photon falling on the retina can cause an action potential. Then several photoreceptors transmit their signals to one single

**bipolar cell.** This means that here the information has already been summarized. Finally, the now transformed light signal travels from several bipolar cells [2] into

**ganglion cells.** Various bipolar cells can transmit their information to one ganglion cell. The higher the number of photoreceptors that affect the ganglion cell, the larger the field of perception, the ***receptive field***, which covers the ganglions – and the less sharp is the image in the area of this ganglion cell. So the information is already reduced directly in the retina and the overall image is, for example, blurred in the peripheral field of vision. So far, we have learned about the information processing in the retina only as a top-down structure. Now we want to take a look at the

2 There are different kinds of bipolar cells, as well, but to discuss all of them would go too far.

**horizontal and amacrine cells.** These cells are not connected from the front backwards but laterally. They allow the light signals to influence themselves *laterally* directly during the information processing in the retina – a much more powerful method of information processing than compressing and blurring. When the horizontal cells are excited by a photoreceptor, they are able to excite other nearby photoreceptors and at the same time inhibit more distant bipolar cells and receptors. This ensures the clear perception of outlines and bright points. Amacrine cells can further intensify certain stimuli by distributing information from bipolar cells to several ganglion cells or by inhibiting ganglions.

These first steps of transmitting visual information to the brain show that information is processed from the first moment the information is received and, on the other hand, is processed in parallel within millions of information-processing cells. The system's power and resistance to errors is based upon this massive division of work.

## 2.4 The amount of neurons in living organisms at different stages of development

An overview of different organisms and their neural capacity (in large part from [RD05]):

**302 neurons** are required by the nervous system of a *nematode worm*, which serves as a popular model organism in biology. Nematodes live in the soil and feed on bacteria.

**$10^4$ neurons** make an *ant* (To simplify matters we neglect the fact that some ant species also can have more or less efficient nervous systems). Due to the use of different attractants and odors, ants are able to engage in complex social behavior and form huge states with millions of individuals. If you regard such an ant state as an individual, it has a cognitive capacity similar to a chimpanzee or even a human.

**With $10^5$ neurons** the nervous system of a *fly* can be constructed. A fly can evade an object in real-time in three-dimensional space, it can land upon the ceiling upside down, has a considerable sensory system because of compound eyes, vibrissae, nerves at the end of its legs and much more. Thus, a fly has considerable differential and integral calculus in high dimensions implemented "in hardware". We all know that a fly is not easy to catch. Of course, the bodily functions are also controlled by neurons, but these should be ignored here.

**With $0.8 \cdot 10^6$ neurons** we have enough cerebral matter to create a *honeybee*. Honeybees build colonies and have amazing capabilities in the field of aerial reconnaissance and navigation.

**$4 \cdot 10^6$ neurons** result in a *mouse*, and here the world of vertebrates already begins.

**$1.5 \cdot 10^7$ neurons** are sufficient for a *rat*, an animal which is denounced as being extremely intelligent and are often used to participate in a variety of intelligence tests representative for the animal world. Rats have an extraordinary sense of smell and orientation, and they also show social behavior. The brain of a *frog* can be positioned within the same dimension. The frog has a complex build with many functions, it can swim and has evolved complex behavior. A frog can continuously target the said fly by means of his eyes while jumping in three-dimensional space and and catch it with its tongue with considerable probability.

**$5 \cdot 10^7$ neurons** make a *bat*. The bat can navigate in total darkness through a room, exact up to several centimeters, by only using their sense of hearing. It uses acoustic signals to localize self-camouflaging insects (e.g. some moths have a certain wing structure that reflects less sound waves and the echo will be small) and also eats its prey while flying.

**$1.6 \cdot 10^8$ neurons** are required by the brain of a *dog*, companion of man for ages. Now take a look at another popular companion of man:

**$3 \cdot 10^8$ neurons** can be found in a *cat*, which is about twice as much as in a dog. We know that cats are very elegant, patient carnivores that can show a variety of behaviors. By the way, an *octopus* can be positioned within the same magnitude. Only very few people know that, for example, in labyrinth orientation the octopus is vastly superior to the rat.

**For $6 \cdot 10^9$ neurons** you already get a chimpanzee, one of the animals being very similar to the human.

**$10^{11}$ neurons** make a *human*. Usually, the human has considerable cognitive capabilities, is able to speak, to abstract, to remember and to use tools as well as the knowledge of other humans to develop advanced technologies and manifold social structures.

**With $2 \cdot 10^{11}$ neurons** there are nervous systems having more neurons than the human nervous system. Here we should mention elephants and certain whale species.

Our state-of-the-art computers are not able to keep up with the aforementioned processing power of a fly. Recent research results suggest that the processes in nervous systems might be vastly more powerful than people thought until not long ago: Michaeva et al. describe a separate, synapse-integrated information way of information processing [MBW+10]. Posterity will show if they are right.

## 2.5 Transition to technical neurons: neural networks are a caricature of biology

How do we change from biological neural networks to the technical ones? Through radical simplification. I want to briefly summarize the conclusions relevant for the technical part:

We have learned that the biological neurons are linked to each other in a weighted way and when stimulated they electrically transmit their signal via the axon. From the axon they are not directly transferred to the succeeding neurons, but they first have to cross the synaptic cleft where the signal is changed again by variable chemical processes. In the receiving neuron the various inputs that have been post-processed in the synaptic cleft are summarized or accumulated to one single pulse. Depending on how the neuron is stimulated by the cumulated input, the neuron itself emits a pulse or not – thus, the output is *non-linear* and not proportional to the cumulated input. Our brief summary corresponds exactly with the few elements of biological neural networks we want to take over into the technical approximation:

**Vectorial input:** The input of technical neurons consists of many components, therefore it is a *vector*. In nature a neuron receives pulses of $10^3$ to $10^4$ other neurons on average.

**Scalar output:** The output of a neuron is a scalar, which means that the neuron only consists of one component. Several scalar outputs in turn form the vectorial input of another neuron. This particularly means that somewhere in the neuron the various input components have to be summarized in such a way that only one component remains.

**Synapses change input:** In technical neural networks the inputs are preprocessed, too. They are multiplied by a number (the weight) – they are *weighted*. The set of such weights represents the information storage of a neural network – in both biological original and technical adaptation.

**Accumulating the inputs:** In biology, the inputs are summarized to a pulse according to the chemical change, i.e., they are accumulated – on the technical side this is often realized by the weighted sum, which we will get to know later on. This means that after accumulation we continue with only *one* value, a scalar, instead of a vector.

**Non-linear characteristic:** The input of our technical neurons is also not proportional to the output.

**Adjustable weights:** The weights weighting the inputs are variable, similar to the chemical processes at the synaptic cleft. This adds a great dynamic to the network because a large part of the "knowledge" of a neural network is saved in the weights and in the form and power of the chemical processes in a synaptic cleft.

So our current, only casually formulated and very simple neuron model receives a vectorial input

$$\vec{x},$$

with components $x_i$. These are multiplied by the appropriate weights $w_i$ and accumulated:

$$\sum_i w_i x_i.$$

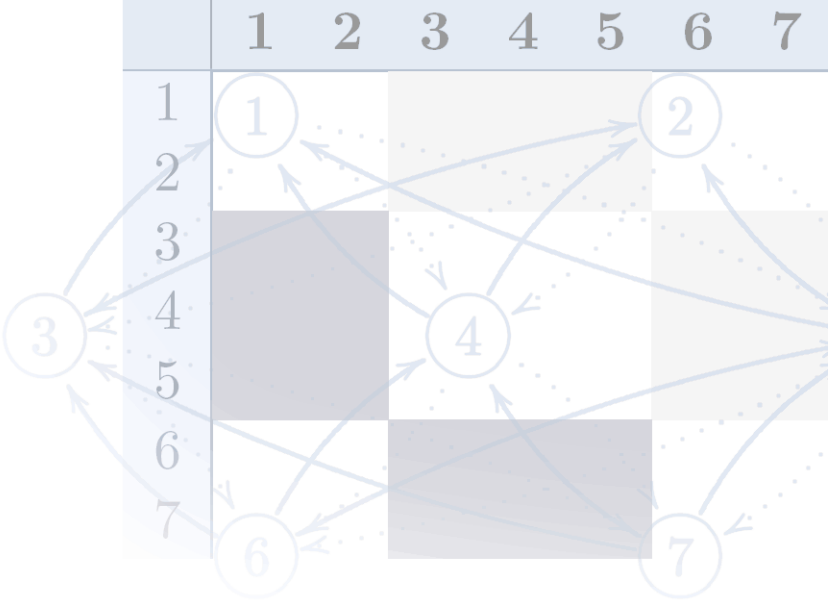
The aforementioned term is called *weighted sum*. Then the nonlinear mapping $f$ defines the scalar output $y$:

$$y = f\left(\sum_i w_i x_i\right).$$

After this transition we now want to specify more precisely our neuron model and add some odds and ends. Afterwards we will take a look at how the weights can be adjusted.

## Exercises

**Exercise 4.** It is estimated that a human brain consists of approx. $10^{11}$ nerve cells, each of which has about $10^3$ to $10^4$ synapses. For this exercise we assume $10^3$ synapses per neuron. Let us further assume that a single synapse could save 4 bits of information. Naïvely calculated: How much storage capacity does the brain have? Note: The information which neuron is connected to which other neuron is also important.




# Chapter 3

# Components of artificial neural networks

Formal definitions and colloquial explanations of the components that realize the technical adaptations of biological neural networks. Initial descriptions of how to combine these components into a neural network.

This chapter contains the formal definitions for most of the neural network components used later in the text. After this chapter you will be able to read the individual chapters of this work without having to know the preceding ones (although this would be useful).

## 3.1 The concept of time in neural networks

In some definitions of this text we use the term *time* or the number of cycles of the neural network, respectively. Time is divided into discrete time steps:

**Definition 3.1** (The concept of time)**.** The current time (present time) is referred to as $(t)$, the next ***time step*** as $(t+1)$, the preceding one as $(t-1)$. All other time steps are referred to analogously. If in the following chapters several mathematical variables (e.g. $net_j$ or $o_i$) refer to a certain point in time, the notation will be, for example, $\mathrm{net}_j(t-1)$ or $o_i(t)$.

From a biological point of view this is, of course, not very plausible (in the human brain a neuron does not wait for another one), but it significantly simplifies the implementation.

## 3.2 Components of neural networks

A technical neural network consists of simple processing units, the ***neurons***, and directed, weighted connections between those neurons. Here, the strength of a connection (or the connecting *weight*) between two neurons $i$ and $j$ is referred to as $w_{i,j}$ [1].

**Definition 3.2** (Neural network)**.** A ***neural network*** is a sorted triple $(N, V, w)$ with two sets $N$, $V$ and a function $w$, where $N$ is the set of *neurons* and $V$ a set $\{(i,j)|i,j \in \mathbb{N}\}$ whose elements are called ***connections*** between neuron $i$ and neuron $j$. The function $w : V \to \mathbb{R}$ defines the ***weights***, where $w((i,j))$, the weight of the connection between neuron $i$ and neuron $j$, is shortened to $w_{i,j}$ . Depending on the point of view it is either undefined or 0 for connections that do not exist in the network.

**SNIPE:** In Snipe, an instance of the class `NeuralNetworkDescriptor` is created in the first place. The descriptor object roughly outlines a class of neural networks, e.g. it defines the number of neuron layers in a neural network. In a second step, the descriptor object is used to instantiate an arbitrary number of `NeuralNetwork` objects. To get started with Snipe programming, the documentations of exactly these two classes are – in that order – the right thing to read. The presented layout involving descriptor and dependent neural networks is very reasonable from the implementation point of view, because it is enables to create and maintain general parameters of even very large sets of similar (but not neccessarily equal) networks.

So the weights can be implemented in a square ***weight matrix*** $W$ or, optionally, in a ***weight vector*** $W$ with the row number of the matrix indicating *where* the connection begins, and the column number of the matrix indicating, which neuron is the ***target***. Indeed, in this case the numeric 0 marks a non-existing connection. This matrix representation is also called ***Hinton diagram***[2].

The neurons and connections comprise the following components and variables (I'm following the path of the data within a neuron, which is according to fig. 3.1 on the facing page in top-down direction):

1 Note: In some of the cited literature $i$ and $j$ could be interchanged in $w_{i,j}$. Here, a consistent standard does not exist. But in this text I try to use the notation I found more frequently and in the more significant citations.

2 Note that, here again, in some of the cited literature axes and rows could be interchanged. The published literature is not consistent here, as well.

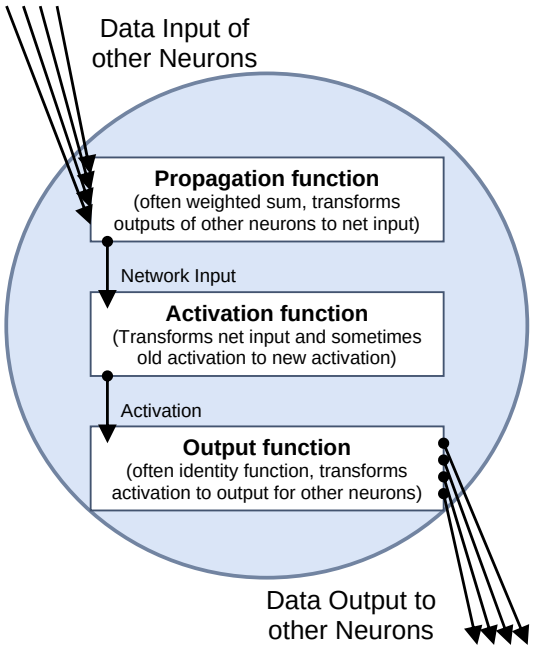



**Figure 3.1:** Data processing of a neuron. The activation function of a neuron implies the threshold value.

### 3.2.1 Connections carry information that is processed by neurons

Data are transferred between neurons via connections with the connecting weight being either excitatory or inhibitory. The definition of connections has already been included in the definition of the neural network.

**SNIPE:** Connection weights can be set using the method `NeuralNetwork.setSynapse`.

### 3.2.2 The propagation function converts vector inputs to scalar network inputs

Looking at a neuron $j$, we will usually find a lot of neurons with a connection to $j$, i.e. which transfer their output to $j$.

For a neuron $j$ the ***propagation function*** receives the outputs $o_{i_1}, \ldots, o_{i_n}$ of other neurons $i_1, i_2, \ldots, i_n$ (which are connected to $j$), and transforms them in consideration of the connecting weights $w_{i,j}$ into the *network input* $\mathrm{net}_j$ that can be further processed by the *activation function*. Thus, the ***network input*** is the result of the propagation function.

**Definition 3.3** (Propagation function and network input)**.** Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of neurons, such that $\forall z \in \{1, \ldots, n\} : \exists w_{i_z,j}$. Then the network input of $j$, called $net_j$, is calculated by the propagation function $f_{\mathrm{prop}}$ as follows:

$$net_j = f_{\mathrm{prop}}(o_{i_1}, \ldots, o_{i_n}, w_{i_1,j}, \ldots, w_{i_n,j}) \tag{3.1}$$

Here the ***weighted sum*** is very popular: The multiplication of the output of each neuron $i$ by $w_{i,j}$, and the summation of the results:

$$\mathrm{net}_j = \sum_{i \in I} (o_i \cdot w_{i,j}) \tag{3.2}$$

**SNIPE:** The propagation function in Snipe was implemented using the weighted sum.

### 3.2.3 The activation is the "switching status" of a neuron

Based on the model of nature every neuron is, to a certain extent, at all times *active*, *excited* or whatever you will call it. The reactions of the neurons to the input values depend on this *activation state*. The activation state indicates the extent of a neuron's activation and is often shortly referred to as ***activation***. Its formal definition is

included in the following definition of the *activation function*. But generally, it can be defined as follows:

**Definition 3.4** (Activation state / activation in general)**.** Let $j$ be a neuron. The activation state $a_j$, in short activation, is explicitly assigned to $j$, indicates the extent of the neuron's activity and results from the activation function.

**SNIPE:** It is possible to get and set activation states of neurons by using the methods `getActivation` or `setActivation` in the class `NeuralNetwork`.

### 3.2.4 Neurons get activated if the network input exceeds their treshold value

Near the threshold value, the activation function of a neuron reacts particularly sensitive. From the biological point of view the threshold value represents the threshold at which a neuron starts firing. The threshold value is also mostly included in the definition of the activation function, but generally the definition is the following:

**Definition 3.5** (Threshold value in general)**.** Let $j$ be a neuron. The ***threshold value*** $\Theta_j$ is uniquely assigned to $j$ and marks the position of the maximum gradient value of the activation function.

### 3.2.5 The activation function determines the activation of a neuron dependent on network input and treshold value

At a certain time – as we have already learned – the activation $a_j$ of a neuron $j$ depends on the *previous*[3] activation state of the neuron and the external input.

**Definition 3.6** (Activation function and Activation)**.** Let $j$ be a neuron. The ***activation function*** is defined as

$$a_j(t) = f_{\text{act}}(\text{net}_j(t), a_j(t-1), \Theta_j). \tag{3.3}$$

It transforms the *network input* $\text{net}_j$, as well as the *previous activation state* $a_j(t-1)$ into a *new activation state* $a_j(t)$, with the *threshold value* $\Theta$ playing an important role, as already mentioned.

3 The previous activation is not always relevant for the current – we will see examples for both variants.

Unlike the other variables within the neural network (particularly unlike the ones defined so far) the activation function is often defined *globally* for all neurons or at least for a set of neurons and only the threshold values are different for each neuron. We should also keep in mind that the threshold values can be changed, for example by a learning procedure. So it can in particular become necessary to relate the threshold value to the time and to write, for instance $\Theta_j$ as $\Theta_j(t)$ (but for reasons of clarity, I omitted this here). The activation function is also called ***transfer function***.

**SNIPE:** In Snipe, activation functions are generalized to *neuron behaviors*. Such behaviors can represent just normal activation functions, or even incorporate internal states and dynamics. Corresponding parts of Snipe can be found in the package `neuronbehavior`, which also contains some of the activation functions introduced in the next section. The interface `NeuronBehavior` allows for implementation of custom behaviors. Objects that inherit from this interface can be passed to a `NeuralNetworkDescriptor` instance. It is possible to define individual behaviors per neuron layer.

### 3.2.6 Common activation functions

The simplest activation function is the ***binary threshold function*** (fig. 3.2 on page 44), which can only take on two values (also referred to as ***Heaviside function***). If the input is above a certain threshold, the function changes from one value to another, but otherwise remains constant. This implies that the function is not differentiable at the threshold and for the rest the derivative is 0. Due to this fact, backpropagation learning, for example, is impossible (as we will see later). Also very popular is the ***Fermi function*** or ***logistic function*** (fig. 3.2)

$$\frac{1}{1+\mathrm{e}^{-x}}, \tag{3.4}$$

which maps to the range of values of $(0, 1)$ and the ***hyperbolic tangent*** (fig. 3.2) which maps to $(-1, 1)$. Both functions are differentiable. The Fermi function can be expanded by a ***temperature parameter*** $T$ into the form

$$\frac{1}{1+\mathrm{e}^{\frac{-x}{T}}}. \tag{3.5}$$

The smaller this parameter, the more does it compress the function on the $x$ axis. Thus, one can arbitrarily approximate the Heaviside function. Incidentally, there exist activation functions which are not explicitly defined but depend on the input according to a random distribution (*stochastic activation function*).

A alternative to the hypberbolic tangent that is really worth mentioning was suggested by ANGUITA et al. [APZ93], who have been tired of the slowness of the workstations back in 1993. Thinking about how to make neural network propagations faster, they quickly identified the approximation of the e-function used in the hyperbolic tangent as one of the causes of slowness. Consequently, they "engineered" an approximation to the hyperbolic tangent, just using two parabola pieces and two half-lines. At the price of delivering a slightly smaller range of values than the hyperbolic tangent ($[-0.96016; 0.96016]$ instead of $[-1; 1]$), dependent on what CPU one uses, it can be calculated 200 times faster because it just needs two multiplications and one addition. What's more, it has some other advantages that will be mentioned later.

**SNIPE:** The activation functions introduced here are implemented within the classes `Fermi` and `TangensHyperbolicus`, both of which are located in the package `neuronbehavior`. The fast hyperbolic tangent approximation is located within the class `TangensHyperbolicusAnguita`.

### 3.2.7 An output function may be used to process the activation once again

The ***output function*** of a neuron $j$ calculates the values which are transferred to the other neurons connected to $j$. More formally:

**Definition 3.7** (Output function)**.** Let $j$ be a neuron. The output function

$$f_{\text{out}}(a_j) = o_j \tag{3.6}$$

calculates the output value $o_j$ of the neuron $j$ from its *activation state* $a_j$.

Generally, the output function is defined globally, too. Often this function is the *identity*, i.e. the *activation* $a_j$ is directly output[4]:

$$f_{\text{out}}(a_j) = a_j, \text{ so } o_j = a_j \tag{3.7}$$

Unless explicitly specified differently, we will use the identity as output function within this text.

4 Other definitions of output functions may be useful if the range of values of the activation function is not sufficient.

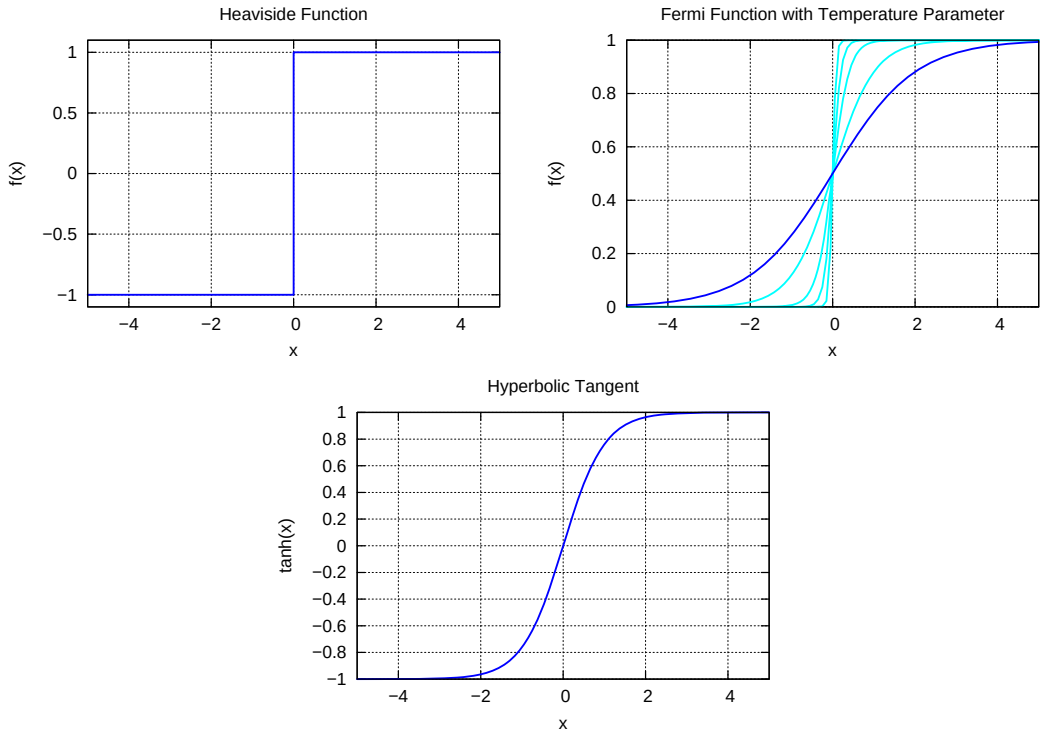



**Figure 3.2:** Various popular activation functions, from top to bottom: Heaviside or binary threshold function, Fermi function, hyperbolic tangent. The Fermi function was expanded by a temperature parameter. The original Fermi function is represented by dark colors, the temperature parameters of the modified Fermi functions are, ordered ascending by steepness, $\frac{1}{2}$, $\frac{1}{5}$, $\frac{1}{10}$ und $\frac{1}{25}$.

### 3.2.8 Learning strategies adjust a network to fit our needs

Since we will address this subject later in detail and at first want to get to know the principles of neural network structures, I will only provide a brief and general definition here:

**Definition 3.8** (General learning rule)**.** The ***learning strategy*** is an algorithm that can be used to change and thereby train the neural network, so that the network produces a desired output for a given input.

## 3.3 Network topologies

After we have become acquainted with the composition of the elements of a neural network, I want to give an overview of the usual topologies (= designs) of neural networks, i.e. to construct networks consisting of these elements. Every topology described in this text is illustrated by a map and its Hinton diagram so that the reader can immediately see the characteristics and apply them to other networks.

In the Hinton diagram the dotted weights are represented by light grey fields, the solid ones by dark grey fields. The input and output arrows, which were added for reasons of clarity, cannot be found in the Hinton diagram. In order to clarify that the connections are between the line neurons and the column neurons, I have inserted the small arrow ↱ in the upper-left cell.

**SNIPE:** Snipe is designed for realization of arbitrary network topologies. In this respect, Snipe defines different kinds of synapses depending on their source and their target. Any kind of synapse can separately be allowed or forbidden for a set of networks using the `setAllowed` methods in a `NeuralNetworkDescriptor` instance.

### 3.3.1 Feedforward networks consist of layers and connections towards each following layer

***Feedforward*** In this text feedforward networks (fig. 3.3 on the following page) are the networks we will first explore (even if we will use different topologies later). The neurons are grouped in the following ***layers***: One ***input layer***, $n$ ***hidden processing layers*** (invisible from the outside, that's why the neurons are also referred to as *hidden neurons*) and one ***output layer***. In a feedforward network each neuron in one layer has only directed connections to the neurons of the next layer (towards the output layer). In fig. 3.3 on the next page the connections permitted for a feedforward

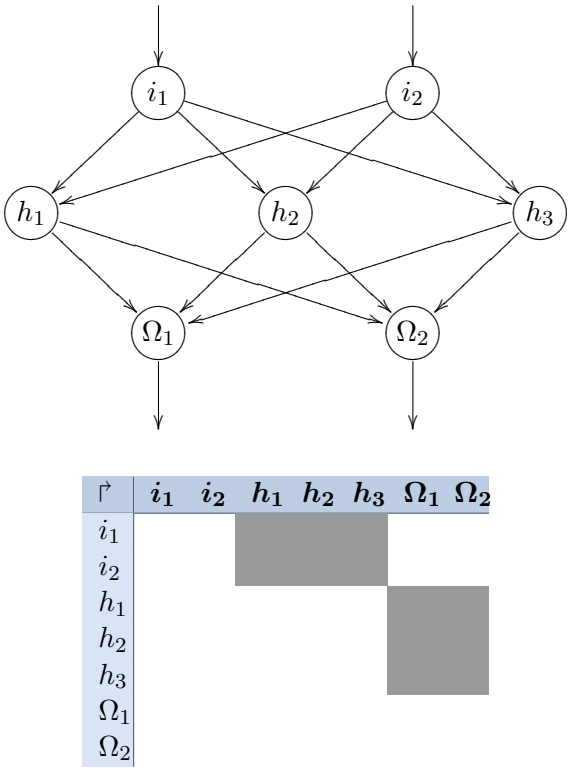



**Figure 3.3:** A feedforward network with three layers: two input neurons, three hidden neurons and two output neurons. Characteristic for the Hinton diagram of completely linked feedforward networks is the formation of blocks above the diagonal.

network are represented by solid lines. We will often be confronted with feedforward networks in which every neuron $i$ is connected to all neurons of the next layer (these layers are called ***completely linked***). To prevent naming conflicts the output neurons are often referred to as $\Omega$.

**Definition 3.9** (Feedforward network)**.** The neuron layers of a feedforward network (fig. 3.3) are clearly separated: One input layer, one output layer and one or more processing layers which are invisible from the outside (also called hidden layers). Connections are only permitted to neurons of the following layer.

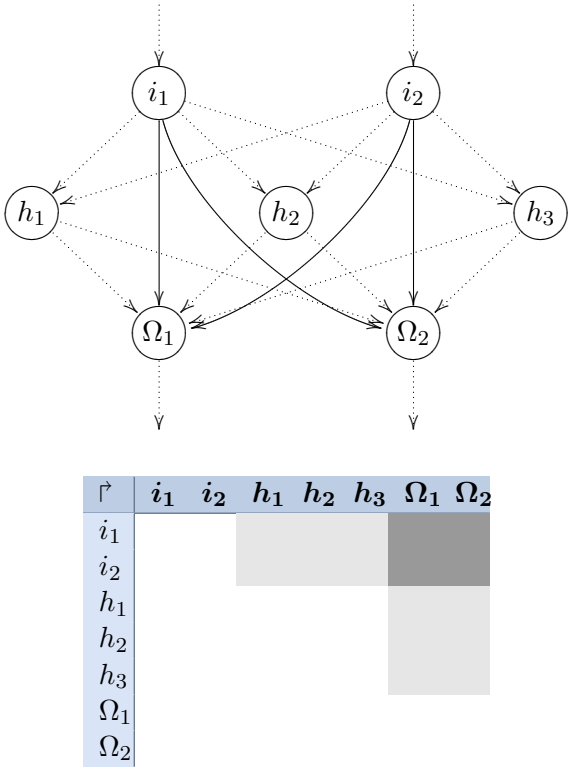



**Figure 3.4:** A feedforward network with shortcut connections, which are represented by solid lines. On the right side of the feedforward blocks new connections have been added to the Hinton diagram.

#### 3.3.1.1 Shortcut connections skip layers

Some feedforward networks permit the so-called ***shortcut connections*** (fig. 3.4): connections that skip one or more levels. These connections may only be directed towards the output layer, too.

**Definition 3.10** (Feedforward network with shortcut connections)**.** Similar to the feedforward network, but the connections may not only be directed towards the next layer but also towards any other subsequent layer.

### 3.3.2 Recurrent networks have influence on themselves

***Recurrence*** is defined as the process of a neuron influencing itself by any means or by any connection. Recurrent networks do not always have explicitly defined input or output neurons. Therefore in the figures I omitted all markings that concern this matter and only numbered the neurons.

#### 3.3.2.1 Direct recurrences start and end at the same neuron

Some networks allow for neurons to be connected to themselves, which is called ***direct recurrence*** (or sometimes *self-recurrence* (fig. 3.5 on the facing page). As a result, neurons inhibit and therefore strengthen themselves in order to reach their activation limits.

**Definition 3.11** (Direct recurrence)**.** Now we expand the feedforward network by connecting a neuron $j$ to *itself*, with the weights of these connections being referred to as $w_{j,j}$. In other words: the diagonal of the weight matrix $W$ may be different from 0.

#### 3.3.2.2 Indirect recurrences can influence their starting neuron only by making detours

If connections are allowed towards the input layer, they will be called ***indirect recurrences***. Then a neuron $j$ can use indirect forwards connections to influence itself, for example, by influencing the neurons of the next layer and the neurons of this next layer influencing $j$ (fig. 3.6 on page 50).

**Definition 3.12** (Indirect recurrence)**.** Again our network is based on a feedforward network, now with additional connections between neurons and their *preceding layer* being allowed. Therefore, below the diagonal of $W$ is different from 0.

#### 3.3.2.3 Lateral recurrences connect neurons within one layer

Connections between neurons *within one layer* are called ***lateral recurrences*** (fig. 3.7 on page 51). Here, each neuron often inhibits the other neurons of the layer and strengthens itself. As a result only the strongest neuron becomes active (***winner-takes-all scheme***).

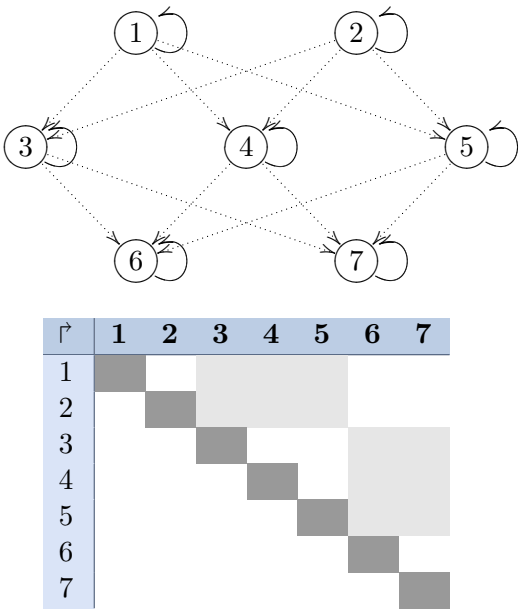
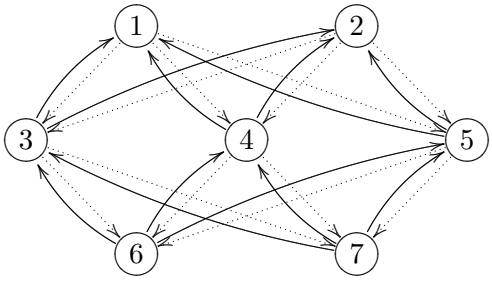



**Figure 3.5:** A network similar to a feedforward network with directly recurrent neurons. The direct recurrences are represented by solid lines and exactly correspond to the diagonal in the Hinton diagram matrix.




| ↱ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |

**Figure 3.6:** A network similar to a feedforward network with indirectly recurrent neurons. The indirect recurrences are represented by solid lines. As we can see, connections to the preceding layers can exist here, too. The fields that are symmetric to the feedforward blocks in the Hinton diagram are now occupied.

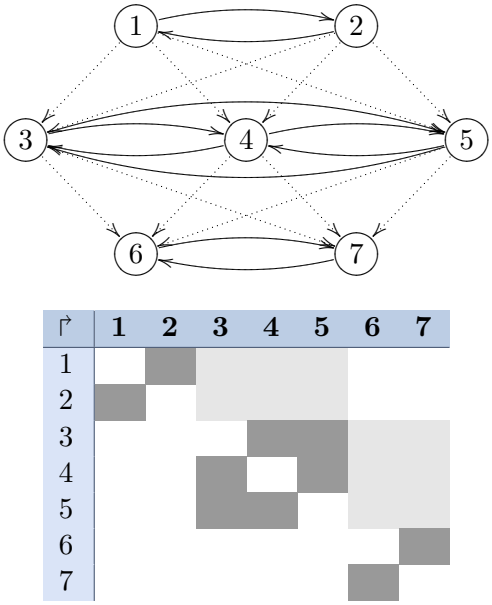



**Figure 3.7:** A network similar to a feedforward network with laterally recurrent neurons. The direct recurrences are represented by solid lines. Here, recurrences only exist within the layer. In the Hinton diagram, filled squares are concentrated around the diagonal in the height of the feedforward blocks, but the diagonal is left uncovered.

**Definition 3.13** (Lateral recurrence)**.** A laterally recurrent network permits connections *within* one layer.

### 3.3.3 Completely linked networks allow any possible connection

*Completely linked networks* permit connections between all neurons, except for direct recurrences. Furthermore, the connections must be symmetric (fig. 3.8 on the next page). A popular example are the *self-organizing maps*, which will be introduced in chapter 10.

**Definition 3.14** (Complete interconnection)**.** In this case, every neuron is always allowed to be connected to every other neuron – but as a result every neuron can become an input neuron. Therefore, direct recurrences normally cannot be applied

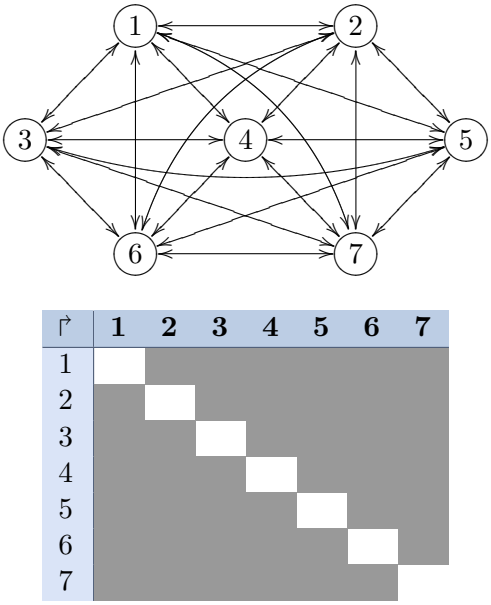



**Figure 3.8:** A completely linked network with symmetric connections and without direct recurrences. In the Hinton diagram only the diagonal is left blank.

here and clearly defined layers do not longer exist. Thus, the matrix $W$ may be unequal to 0 everywhere, except along its diagonal.

## 3.4 The bias neuron is a technical trick to consider threshold values as connection weights

By now we know that in many network paradigms neurons have a *threshold value* that indicates when a neuron becomes active. Thus, the threshold value is an activation function parameter of a neuron. From the biological point of view this sounds most plausible, but it is complicated to access the activation function at runtime in order to train the threshold value.

But threshold values $\Theta_{j_1}, \ldots, \Theta_{j_n}$ for neurons $j_1, j_2, \ldots, j_n$ can also be realized as *connecting weight of a continuously firing neuron*: For this purpose an additional bias neuron whose output value is always 1 is integrated in the network and connected to

the neurons $j_1, j_2, \ldots, j_n$. These new connections get the weights $-\Theta_{j_1}, \ldots, -\Theta_{j_n}$, i.e. they get the negative threshold values.

**Definition 3.15.** A ***bias neuron*** is a neuron whose output value is always 1 and which is represented by


.

It is used to represent neuron biases as connection weights, which enables any weight-training algorithm to train the biases at the same time.

Then the threshold value of the neurons $j_1, j_2, \ldots, j_n$ is set to 0. Now the threshold values are implemented as connection weights (fig. 3.9 on the following page) and can directly be trained together with the connection weights, which considerably facilitates the learning process.

In other words: Instead of including the threshold value in the activation function, it is now included in the propagation function. Or even shorter: The threshold value is subtracted from the network input, i.e. it is part of the network input. More formally:

Let $j_1, j_2, \ldots, j_n$ be neurons with threshold values $\Theta_{j_1}, \ldots, \Theta_{j_n}$. By inserting a bias neuron whose output value is always 1, generating connections between the said bias neuron and the neurons $j_1, j_2, \ldots, j_n$ and weighting these connections $w_{\texttt{BIAS},j_1}, \ldots, w_{\texttt{BIAS},j_n}$ with $-\Theta_{j_1}, \ldots, -\Theta_{j_n}$, we can set $\Theta_{j_1} = \ldots = \Theta_{j_n} = 0$ and receive an equivalent neural network whose threshold values are realized by connection weights.

Undoubtedly, the advantage of the bias neuron is the fact that it is much easier to implement it in the network. One disadvantage is that the representation of the network already becomes quite ugly with only a few neurons, let alone with a great number of them. By the way, a bias neuron is often referred to as ***on neuron***.

From now on, the bias neuron is omitted for clarity in the following illustrations, but we know that it exists and that the threshold values can simply be treated as weights because of it.

**SNIPE:** In Snipe, a bias neuron was implemented instead of neuron-individual biases. The neuron index of the bias neuron is 0.

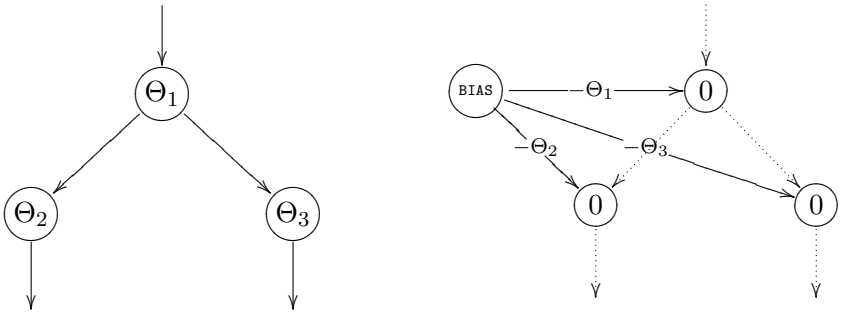



**Figure 3.9:** Two equivalent neural networks, one without bias neuron on the left, one with bias neuron on the right. The neuron threshold values can be found in the neurons, the connecting weights at the connections. Furthermore, I omitted the weights of the already existing connections (represented by dotted lines on the right side).

## 3.5 Representing neurons

We have already seen that we can either write its name or its threshold value into a neuron. Another useful representation, which we will use several times in the following, is to illustrate neurons according to their type of data processing. See fig. 3.10 on the next page for some examples without further explanation – the different types of neurons are explained as soon as we need them.

## 3.6 Take care of the order in which neuron activations are calculated

For a neural network it is very important in which ***order*** the individual neurons receive and process the input and output the results. Here, we distinguish two model classes:

### 3.6.1 Synchronous activation

All neurons change their values ***synchronously***, i.e. they simultaneously calculate network inputs, activation and output, and pass them on. Synchronous activation

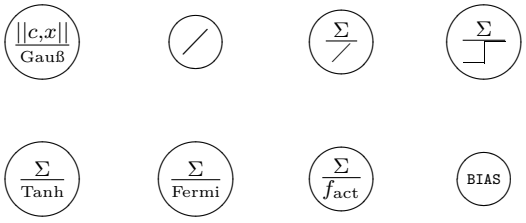



**Figure 3.10:** Different types of neurons that will appear in the following text.

corresponds closest to its biological counterpart, but it is – if to be implemented in hardware – only useful on certain parallel computers and especially not for feedforward networks. This order of activation is the most generic and can be used with networks of arbitrary topology.

**Definition 3.16** (Synchronous activation)**.** All neurons of a network calculate network inputs at the same time by means of the propagation function, activation by means of the activation function and output by means of the output function. After that the activation cycle is complete.

**SNIPE:** When implementing in software, one could model this very general activation order by every time step calculating and caching every single network input, and after that calculating all activations. This is exactly how it is done in Snipe, because Snipe has to be able to realize arbitrary network topologies.

### 3.6.2 Asynchronous activation

Here, the neurons do not change their values simultaneously but at different points of time. For this, there exist different orders, some of which I want to introduce in the following:

#### 3.6.2.1 Random order

**Definition 3.17** (Random order of activation)**.** With ***random order of activation*** a neuron $i$ is randomly chosen and its $net_i$, $a_i$ and $o_i$ are updated. For $n$ neurons a cycle is the $n$-fold execution of this step. Obviously, some neurons are repeatedly updated during one cycle, and others, however, not at all.

Apparently, this order of activation is not always useful.

#### 3.6.2.2 Random permutation

With ***random permutation*** each neuron is chosen exactly once, but in random order, during one cycle.

**Definition 3.18** (Random permutation)**.** Initially, a permutation of the neurons is calculated randomly and therefore defines the order of activation. Then the neurons are successively processed in this order.

This order of activation is as well used rarely because firstly, the order is generally useless and, secondly, it is very time-consuming to compute a new permutation for every cycle. A *Hopfield network* (chapter 8) is a topology nominally having a random or a randomly permuted order of activation. But note that in practice, for the previously mentioned reasons, a fixed order of activation is preferred.

For all orders either the previous neuron activations at time t or, if already existing, the neuron activations at time $t + 1$, for which we are calculating the activations, can be taken as a starting point.

#### 3.6.2.3 Topological order

**Definition 3.19** (Topological activation)**.** With ***topological order of activation*** the neurons are updated during one cycle and according to a fixed order. The order is defined by the *network topology*.

This procedure can only be considered for *non-cyclic*, i.e. non-recurrent, networks, since otherwise there is no order of activation. Thus, in *feedforward networks* (for which the procedure is very reasonable) the input neurons would be updated first, then the inner neurons and finally the output neurons. This may save us a lot of time: Given a synchronous activation order, a feedforward network with $n$ layers of neurons would need $n$ full propagation cycles in order to enable input data to have influence on the output of the network. Given the topological activation order, we just need one single propagation. However, not every network topology allows for finding a special activation order that enables saving time.

**SNIPE:** Those who want to use Snipe for implementing feedforward networks may save some calculation time by using the feature ***fastprop*** (mentioned within the documentation of the class `NeuralNetworkDescriptor`. Once fastprop is enabled, it will cause the data propagation to be carried out in a slightly different way. In the standard mode, all net inputs are calculated first, followed by all activations. In the fastprop mode, for every neuron, the activation is calculated right after the net input. The neuron values are calculated in ascending neuron index order. The neuron numbers are ascending from input to output layer, which provides us with the perfect topological activation order for feedforward networks.

#### 3.6.2.4 Fixed orders of activation during implementation

Obviously, ***fixed orders of activation*** can be defined as well. Therefore, when implementing, for instance, feedforward networks it is very popular to determine the order of activation *once* according to the topology and to use this order without further verification at runtime. But this is not necessarily useful for networks that are capable to change their topology.

## 3.7 Communication with the outside world: input and output of data in and from neural networks

Finally, let us take a look at the fact that, of course, many types of neural networks permit the input of data. Then these data are processed and can produce output. Let us, for example, regard the feedforward network shown in fig. 3.3 on page 46: It has two input neurons and two output neurons, which means that it also has two numerical inputs $x_1, x_2$ and outputs $y_1, y_2$. As a simplification we summarize the input and output components for $n$ input or output neurons within the vectors $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$.

**Definition 3.20** (Input vector)**.** A network with $n$ input neurons needs $n$ inputs $x_1, x_2, \ldots, x_n$. They are considered as ***input vector*** $x = (x_1, x_2, \ldots, x_n)$. As a consequence, the ***input dimension*** is referred to as $n$. Data is put into a neural network by using the components of the input vector as network inputs of the input neurons.

**Definition 3.21** (Output vector)**.** A network with $m$ output neurons provides $m$ outputs $y_1, y_2, \ldots, y_m$. They are regarded as ***output vector*** $y = (y_1, y_2, \ldots, y_m)$. Thus, the ***output dimension*** is referred to as $m$. Data is output by a neural network by the output neurons adopting the components of the output vector in their output values.

**SNIPE:** In order to propagate data through a `NeuralNetwork`-instance, the `propagate` method is used. It receives the input vector as array of doubles, and returns the output vector in the same way.

Now we have defined and closely examined the basic components of neural networks – without having seen a network in action. But first we will continue with theoretical explanations and generally describe how a neural network could learn.

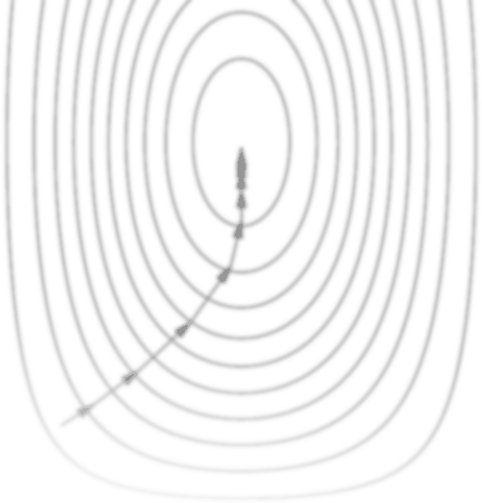
## Exercises

**Exercise 5.** Would it be useful (from your point of view) to insert one bias neuron in each layer of a layer-based network, such as a feedforward network? Discuss this in relation to the representation and implementation of the network. Will the result of the network change?

**Exercise 6.** Show for the Fermi function $f(x)$ as well as for the hyperbolic tangent $\tanh(x)$, that their derivatives can be expressed by the respective functions themselves so that the two statements

1. $f'(x) = f(x) \cdot (1 - f(x))$ and
2. $\tanh'(x) = 1 - \tanh^2(x)$

are true.



# Chapter 4

# Fundamentals on learning and training samples

Approaches and thoughts of how to teach machines. Should neural networks be corrected? Should they only be encouraged? Or should they even learn without any help? Thoughts about what we want to change during the learning procedure and how we will change it, about the measurement of errors and when we have learned enough.

---

As written above, the most interesting characteristic of neural networks is their capability to familiarize with problems by means of training and, after sufficient training, to be able to solve unknown problems of the same class. This approach is referred to as ***generalization***. Before introducing specific learning procedures, I want to propose some basic principles about the learning procedure in this chapter.

## 4.1 There are different paradigms of learning

Learning is a comprehensive term. A learning system changes itself in order to adapt to e.g. environmental changes. A neural network could learn from many things but, of course, there will always be the question of how to implement it. In principle, a neural network changes when its components are changing, as we have learned above. Theoretically, a neural network could learn by

1. developing new connections,
2. deleting existing connections,
3. changing connecting weights,

4. changing the threshold values of neurons,
5. varying one or more of the three neuron functions (remember: activation function, propagation function and output function),
6. developing new neurons, or
7. deleting existing neurons (and so, of course, existing connections).

As mentioned above, we assume the change in weight to be the most common procedure. Furthermore, deletion of connections can be realized by additionally taking care that a connection is no longer trained when it is set to 0. Moreover, we can develop further connections by setting a non-existing connection (with the value 0 in the connection matrix) to a value different from 0. As for the modification of threshold values I refer to the possibility of implementing them as weights (section 3.4). Thus, we perform any of the first four of the learning paradigms by just training synaptic weights.

The change of neuron functions is difficult to implement, not very intuitive and not exactly biologically motivated. Therefore it is not very popular and I will omit this topic here. The possibilities to develop or delete neurons do not only provide well adjusted weights during the training of a neural network, but also optimize the network topology. Thus, they attract a growing interest and are often realized by using evolutionary procedures. But, since we accept that a large part of learning possibilities can already be covered by changes in weight, they are also not the subject matter of this text (however, it is planned to extend the text towards those aspects of training).

**SNIPE:** Methods of the class `NeuralNetwork` allow for changes in connection weights, and addition and removal of both connections and neurons. Methods in `NeuralNetworkDescriptor` enable the change of neuron behaviors, respectively activation functions per layer.

Thus, we let our neural network learn by modifying the connecting weights according to rules that can be formulated as algorithms. Therefore a learning procedure is always an ***algorithm*** that can easily be implemented by means of a programming language. Later in the text I will assume that the definition of the term *desired output which is worth learning* is known (and I will define formally what a *training pattern* is) and that we have a training set of learning samples. Let a *training set* be defined as follows:

**Definition 4.1** (Training set)**.** A ***training set*** (named $P$) is a set of training patterns, which we use to train our neural net.

I will now introduce the three essential paradigms of learning by presenting the differences between their regarding training sets.

### 4.1.1 Unsupervised learning provides input patterns to the network, but no learning aides

***Unsupervised learning*** is the biologically most plausible method, but is not suitable for all problems. Only the input patterns are given; the network tries to identify similar patterns and to classify them into similar categories.

**Definition 4.2** (Unsupervised learning)**.** The training set only consists of ***input patterns***, the network tries by itself to detect similarities and to generate pattern classes.

Here I want to refer again to the popular example of Kohonen's self-organising maps (chapter 10).

### 4.1.2 Reinforcement learning methods provide feedback to the network, whether it behaves well or bad

In ***reinforcement learning*** the network receives a logical or a real value after completion of a sequence, which defines whether the result is right or wrong. Intuitively it is clear that this procedure should be more effective than unsupervised learning since the network receives specific critera for problem-solving.

**Definition 4.3** (Reinforcement learning)**.** The training set consists of *input patterns*, after completion of a sequence a value is returned to the network indicating whether the result was right or wrong and, possibly, *how* right or wrong it was.

### 4.1.3 Supervised learning methods provide training patterns together with appropriate desired outputs

In ***supervised learning*** the training set consists of input patterns as well as their correct results in the form of the precise activation of all output neurons. Thus, for each training set that is fed into the network the output, for instance, can directly be compared with the correct solution and and the network weights can be changed according to their difference. The objective is to change the weights to the effect that the network cannot only associate input and output patterns independently after the training, but can provide plausible results to unknown, similar input patterns, i.e. it *generalises*.

**Definition 4.4** (Supervised learning)**.** The training set consists of *input patterns with correct results* so that the network can receive a precise *error vector*[1] can be returned.

This learning procedure is not always biologically plausible, but it is extremely effective and therefore very practicable.

At first we want to look at the the supervised learning procedures in general, which - in this text - are corresponding to the following steps:

**Entering** the input pattern (activation of input neurons),

**Forward propagation** of the input by the network, generation of the output,

**Comparing** the output with the desired output (*teaching input*), provides error vector (difference vector),

**Corrections of the network** are calculated based on the error vector,

**Corrections** are applied.

### 4.1.4 Offline or online learning?

It must be noted that learning can be ***offline*** (a set of training samples is presented, then the weights are changed, the total error is calculated by means of a error function operation or simply accumulated - see also section 4.4) or ***online*** (after every sample presented the weights are changed). Both procedures have advantages and disadvantages, which will be discussed in the learning procedures section if necessary. Offline training procedures are also called ***batch training procedures*** since a batch of results is corrected all at once. Such a training section of a whole batch of training samples including the related change in weight values is called ***epoch***.

**Definition 4.5** (Offline learning)**.** Several training patterns are entered into the network at once, the errors are accumulated and it learns for all patterns at the same time.

**Definition 4.6** (Online learning)**.** The network learns directly from the errors of each training sample.

1 The term error vector will be defined in section 4.2, where mathematical formalisation of learning is discussed.

### 4.1.5 Questions you should answer before learning

The application of such schemes certainly requires preliminary thoughts about some questions, which I want to introduce now as a check list and, if possible, answer them in the course of this text:

- ▷ Where does the learning input come from and in what form?
- ▷ How must the weights be modified to allow fast and reliable learning?
- ▷ How can the success of a learning process be measured in an objective way?
- ▷ Is it possible to determine the "best" learning procedure?
- ▷ Is it possible to predict if a learning procedure terminates, i.e. whether it will reach an optimal state after a finite time or if it, for example, will oscillate between different states?
- ▷ How can the learned patterns be stored in the network?
- ▷ Is it possible to avoid that newly learned patterns destroy previously learned associations (the so-called stability/plasticity dilemma)?

We will see that all these questions cannot be generally answered but that they have to be discussed for each learning procedure and each network topology individually.

## 4.2 Training patterns and teaching input

Before we get to know our first learning rule, we need to introduce the *teaching input*. In (this) case of supervised learning we assume a training set consisting of training patterns and the corresponding correct output values we want to see at the output neurons after the training. While the network has not finished training, i.e. as long as it is generating wrong outputs, these output values are referred to as teaching input, and that for each neuron individually. Thus, for a neuron $j$ with the incorrect output $o_j$, $t_j$ is the teaching input, which means it is the correct or desired output for a training pattern $p$.

**Definition 4.7** (Training patterns)**.** A ***training pattern*** is an input vector $p$ with the components $p_1, p_2, \ldots, p_n$ whose desired output is known. By entering the training pattern into the network we receive an output that can be compared with the teaching input, which is the desired output. The ***set of training patterns*** is called $P$. It contains a finite number of ordered pairs$(p, t)$ of training patterns with corresponding desired output.

Training ***patterns*** are often simply called patterns, that is why they are referred to as $p$. In the literature as well as in this text they are called synonymously patterns, training samples etc.

**Definition 4.8** (Teaching input)**.** Let $j$ be an output neuron. The ***teaching input*** $t_j$ is the desired and correct value $j$ should output after the input of a certain training pattern. Analogously to the vector $p$ the teaching inputs $t_1, t_2, \ldots, t_n$ of the neurons can also be combined into a vector $t$. $t$ always refers to a specific training pattern $p$ and is, as already mentioned, contained in the set $P$ of the training patterns.

**SNIPE:** Classes that are relevant for training data are located in the package `training`. The class `TrainingSampleLesson` allows for storage of training patterns and teaching inputs, as well as simple preprocessing of the training data.

**Definition 4.9** (Error vector)**.** For several output neurons $\Omega_1, \Omega_2, \ldots, \Omega_n$ the difference between output vector and teaching input under a training input $p$

$$E_p = \begin{pmatrix} t_1 - y_1 \\ \vdots \\ t_n - y_n \end{pmatrix}$$

is referred to as ***error vector***, sometimes it is also called ***difference vector***. Depending on whether you are learning offline or online, the difference vector refers to a specific training pattern, or to the error of a set of training patterns which is normalized in a certain way.

Now I want to briefly summarize the vectors we have yet defined. There is the

**input vector** $x$, which can be entered into the neural network. Depending on the type of network being used the neural network will output an

**output vector** $y$. Basically, the

**training sample** $p$ is nothing more than an input vector. We only use it for training purposes because we know the corresponding

**teaching input** $t$ which is nothing more than the desired output vector to the training sample. The

**error vector** $E_p$ is the difference between the teaching input $t$ and the actural output $y$.

So, what $x$ and $y$ are for the general network operation are $p$ and $t$ for the network training - and during training we try to bring $y$ as close to $t$ as possible. One advice concerning notation: We referred to the output values of a neuron $i$ as $o_i$. Thus, the output of an output neuron $\Omega$ is called $o_\Omega$. But the output values of a network are referred to as $y_\Omega$. Certainly, these network outputs are only neuron outputs, too, but they are outputs of output neurons. In this respect

$$y_\Omega = o_\Omega$$

is true.

## 4.3 Using training samples

We have seen how we *can* learn in principle and which steps are required to do so. Now we should take a look at the selection of training data and the learning curve. After successful learning it is particularly interesting whether the network has only ***memorized*** – i.e. whether it can use our training samples to quite exactly produce the right output but to provide wrong answers for all other problems of the same class.

Suppose that we want the network to train a mapping $\mathbb{R}^2 \to \mathbb{B}^1$ and therefor use the training samples from fig. 4.1 on the next page: Then there could be a chance that, finally, the network will exactly mark the colored areas around the training samples with the output 1 (fig. 4.1, top), and otherwise will output 0 . Thus, it has sufficient storage capacity to concentrate on the six training samples with the output 1. This implies an oversized network with too much free storage capacity.

On the other hand a network could have insufficient capacity (fig. 4.1, bottom) – this rough presentation of input data does not correspond to the good generalization performance we desire. Thus, we have to find the balance (fig. 4.1, middle).

### 4.3.1 It is useful to divide the set of training samples

An often proposed solution for these problems is to *divide*, the training set into

- one training set really used to train ,
- and one verification set to test our progress

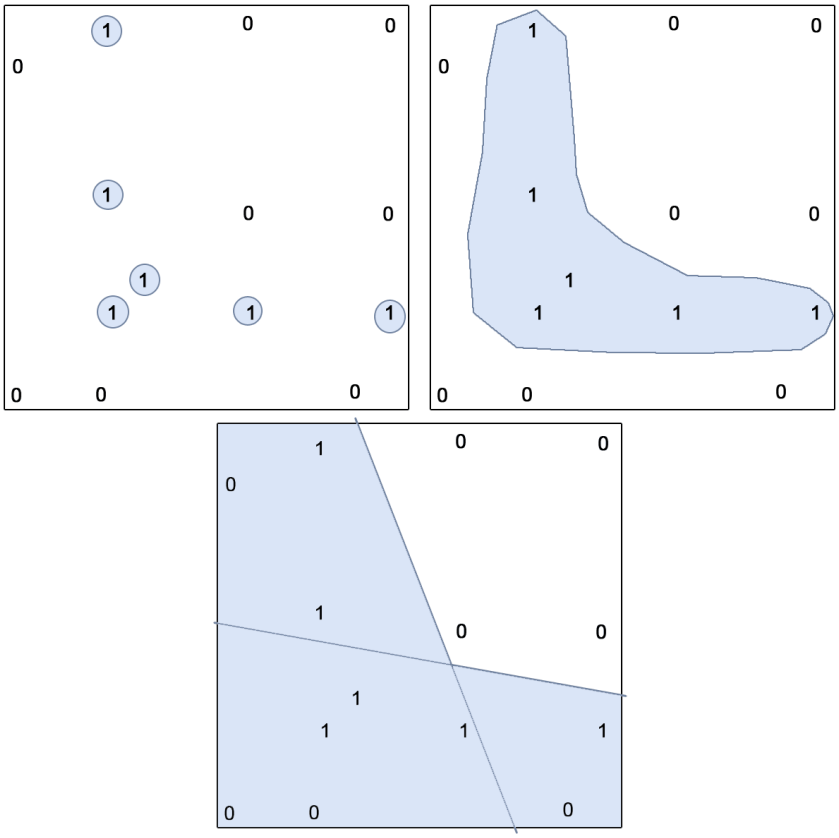


**Figure 4.1:** Visualization of training results of the same training set on networks with a capacity being too high (top), correct (middle) or too low (bottom).

– provided that there are enough training samples. The usual division relations are, for instance, 70% for training data and 30% for verification data (randomly chosen). We can finish the training when the network provides good results on the training data as well as on the verification data.

**SNIPE:** The method `splitLesson` within the class `TrainingSampleLesson` allows for splitting a TrainingSampleLesson with respect to a given ratio.

But note: If the verification data provide poor results, do not modify the network structure until these data provide good results – otherwise you run the risk of tailoring the network to the verification data. This means, that these data are included in the training, even if they are not used explicitly for the training. The solution is a third set of validation data used only for validation *after* a supposably successful training.

By training less patterns, we obviously withhold information from the network and risk to worsen the learning performance. But this text is not about 100% exact reproduction of given samples but about successful generalization and approximation of a whole function – for which it can definitely be useful to train less information into the network.

### 4.3.2 Order of pattern representation

You can find different strategies to choose the order of pattern presentation: If patterns are presented in random sequence, there is no guarantee that the patterns are learned equally well (however, this is the standard method). Always the same sequence of patterns, on the other hand, provokes that the patterns will be memorized when using recurrent networks (later, we will learn more about this type of networks). A *random permutation* would solve both problems, but it is – as already mentioned – very time-consuming to calculate such a permutation.

**SNIPE:** The method `shuffleSamples` located in the class `TrainingSampleLesson` permutes a lesson.

## 4.4 Learning curve and error measurement

The learning curve indicates the progress of the error, which can be determined in various ways. The motivation to create a learning curve is that such a curve can indicate whether the network is progressing or not. For this, the error should be

normalized, i.e. represent a distance measure between the correct and the current output of the network. For example, we can take the same pattern-specific, squared error with a prefactor, which we are also going to use to derive the backpropagation of error (let $\Omega$ be output neurons and O the set of output neurons):

$$\mathrm{Err}_p = \frac{1}{2} \sum_{\Omega \in O} (t_\Omega - y_\Omega)^2 \tag{4.1}$$

**Definition 4.10** (Specific error)**.** The ***specific error*** $\mathrm{Err}_p$ is based on a single training sample, which means it is generated online.

Additionally, the ***root mean square*** (abbreviated: ***RMS***) and the ***Euclidean distance*** are often used.

The Euclidean distance (generalization of the theorem of PYTHAGORAS) is useful for lower dimensions where we can still visualize its usefulness.

**Definition 4.11** (Euclidean distance)**.** The Euclidean distance between two vectors $t$ and $y$ is defined as

$$\mathrm{Err}_p = \sqrt{\sum_{\Omega \in O} (t_\Omega - y_\Omega)^2}. \tag{4.2}$$

Generally, the root mean square is commonly used since it considers extreme outliers to a greater extent.

**Definition 4.12** (Root mean square)**.** The root mean square of two vectors $t$ and $y$ is defined as

$$\mathrm{Err}_p = \sqrt{\frac{\sum_{\Omega \in O} (t_\Omega - y_\Omega)^2}{|O|}}. \tag{4.3}$$

As for offline learning, the total error in the course of one training epoch is interesting and useful, too:

$$\mathrm{Err} = \sum_{p \in P} \mathrm{Err}_p \tag{4.4}$$

**Definition 4.13** (Total error)**.** The ***total error*** Err is based on all training samples, that means it is generated offline.

Analogously we can generate a total RMS and a total Euclidean distance in the course of a whole epoch. Of course, it is possible to use other types of error measurement. To get used to further error measurement methods, I suggest to have a look into the technical report of Prechelt [Pre94]. In this report, both error measurement methods and sample problems are discussed (this is why there will be a simmilar suggestion during the discussion of exemplary problems).

**SNIPE:** There are several static methods representing different methods of error measurement implemented in the class `ErrorMeasurement`.

Depending on our method of error measurement our learning curve certainly changes, too. A perfect learning curve looks like a negative exponential function, that means it is proportional to $e^{-t}$ (fig. 4.2 on the following page). Thus, the representation of the learning curve can be illustrated by means of a logarithmic scale (fig. 4.2, second diagram from the bottom) – with the said scaling combination a descending line implies an exponential descent of the error.

With the network doing a good job, the problems being not too difficult and the logarithmic representation of Err you can see - metaphorically speaking - a descending line that often forms "spikes" at the bottom – here, we reach the limit of the 64-bit resolution of our computer and our network has actually learned the optimum of what it is capable of learning.

Typical learning curves can show a few flat areas as well, i.e. they can show some steps, which is no sign of a malfunctioning learning process. As we can also see in fig. 4.2, a well-suited representation can make any slightly decreasing learning curve look good – so just be cautious when reading the literature.

### 4.4.1 When do we stop learning?

Now, the big question is: When do we stop learning? Generally, the training is stopped when the user in front of the learning computer "thinks" the error was small enough. Indeed, there is no easy answer and thus I can once again only give you something to think about, which, however, depends on a more objective view on the comparison of several learning curves.

Confidence in the results, for example, is boosted, when the network always reaches nearly the same final error-rate for different random initializations – so repeated initialization and training will provide a more objective result.

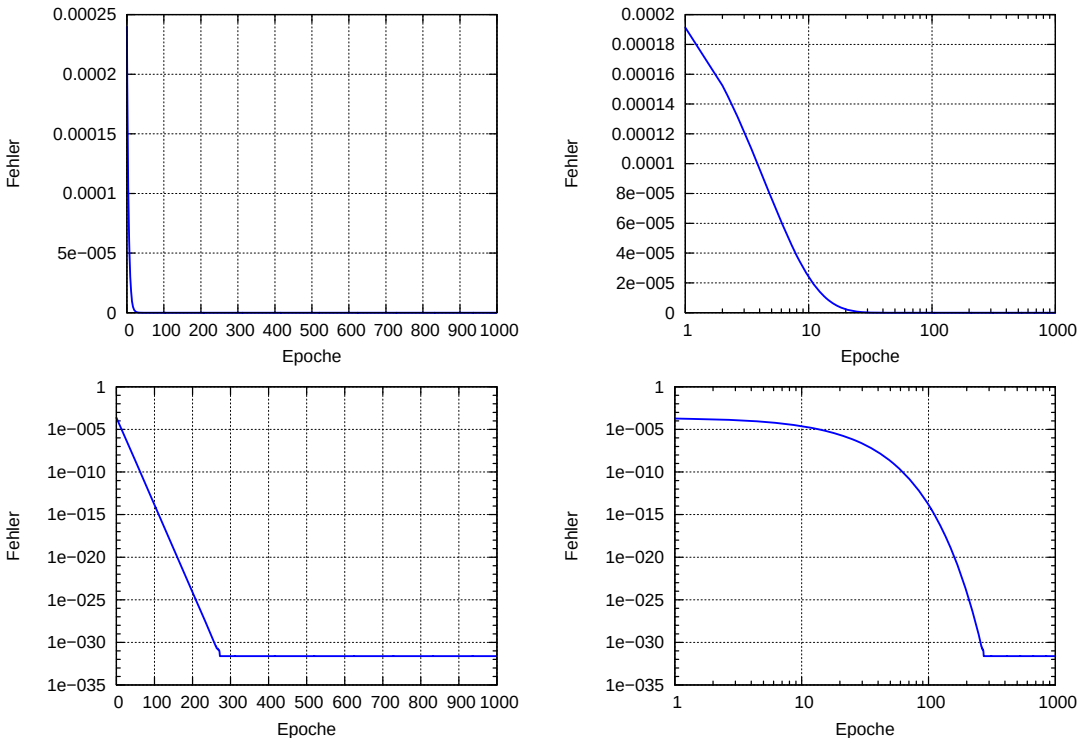



**Figure 4.2:** All four illustrations show the same (idealized, because very smooth) learning curve. Note the alternating logarithmic and linear scalings! Also note the small "inaccurate spikes" visible in the sharp bend of the curve in the first and second diagram from bottom.

On the other hand, it can be possible that a curve descending fast in the beginning can, after a longer time of learning, be overtaken by another curve: This can indicate that either the learning rate of the worse curve was too high or the worse curve itself simply got stuck in a local minimum, but was the first to find it.

Remember: Larger error values are worse than the small ones.

But, in any case, note: Many people only generate a learning curve in respect of the training data (and then they are surprised that only a few things will work) – but for reasons of objectivity and clarity it should not be forgotten to plot the verification data on a second learning curve, which generally provides values that are slightly worse and with stronger oscillation. But with good generalization the curve can decrease, too.

When the network eventually begins to memorize the samples, the shape of the learning curve can provide an indication: If the learning curve of the verification samples is suddenly and rapidly rising while the learning curve of the verification data is continuously falling, this could indicate memorizing and a generalization getting poorer and poorer. At this point it could be decided whether the network has already learned well enough at the next point of the two curves, and maybe the final point of learning is to be applied here (this procedure is called ***early stopping***).

Once again I want to remind you that they are all acting as *indicators* and not to draw If-Then conclusions.

## 4.5 Gradient optimization procedures

In order to establish the mathematical basis for some of the following learning procedures I want to explain briefly what is meant by *gradient descent*: the *backpropagation of error* learning procedure, for example, involves this mathematical basis and thus inherits the advantages and disadvantages of the gradient descent.

Gradient descent procedures are generally used where we want to maximize or minimize $n$-dimensional functions. Due to clarity the illustration (fig. 4.3 on the next page) shows only two dimensions, but principally there is no limit to the number of dimensions.

The *gradient* is a vector $g$ that is defined for any differentiable point of a function, that points from this point exactly towards the *steepest ascent* and indicates the gradient in this direction by means of its norm $|g|$. Thus, the gradient is a *generalization of the derivative for multi-dimensional functions.* Accordingly, the *negative gradient* $-g$ exactly points towards the *steepest descent.* The gradient operator $\nabla$ is referred to

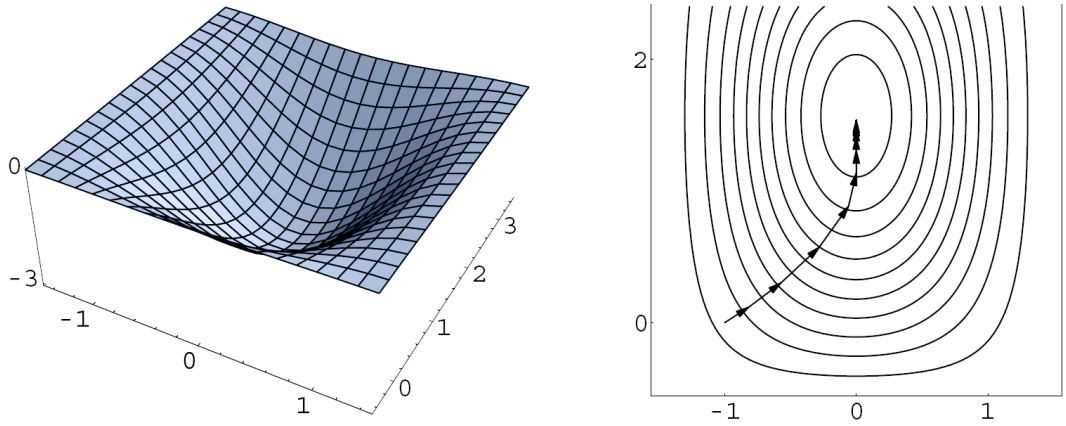



**Figure 4.3:** Visualization of the gradient descent on a two-dimensional error function. We move forward in the opposite direction of $g$, i.e. with the steepest descent towards the lowest point, with the step width being proportional to $|g|$ (the steeper the descent, the faster the steps). On the left the area is shown in 3D, on the right the steps over the contour lines are shown in 2D. Here it is obvious how a movement is made in the opposite direction of $g$ towards the minimum of the function and continuously slows down proportionally to $|g|$. **Source**: `http://webster.fhs-hagenberg.ac.at/staff/sdreisei/Teaching/WS2001-2002/PatternClassification/graddescent.pdf`

as ***nabla operator***, the overall notation of the the gradient $g$ of the point $(x, y)$ of a two-dimensional function $f$ being $g(x, y) = \nabla f(x, y)$.

**Definition 4.14** (Gradient)**.** Let $g$ be a ***gradient***. Then $g$ is a vector with $n$ components that is defined for any point of a (differential) $n$-dimensional function $f(x_1, x_2, \ldots, x_n)$. The gradient operator notation is defined as

$$g(x_1, x_2, \ldots, x_n) = \nabla f(x_1, x_2, \ldots, x_n).$$

$g$ directs from any point of $f$ towards the steepest ascent from this point, with $|g|$ corresponding to the degree of this ascent.

*Gradient descent* means to going *downhill* in small steps from any starting point of our function towards the gradient $g$ (which means, vividly speaking, the direction to which a ball would roll from the starting point), with the size of the steps being proportional to $|g|$ (the steeper the descent, the longer the steps). Therefore, we move slowly on a flat plateau, and on a steep ascent we run downhill rapidly. If we came into a valley, we would - depending on the size of our steps - jump over it or we would return into

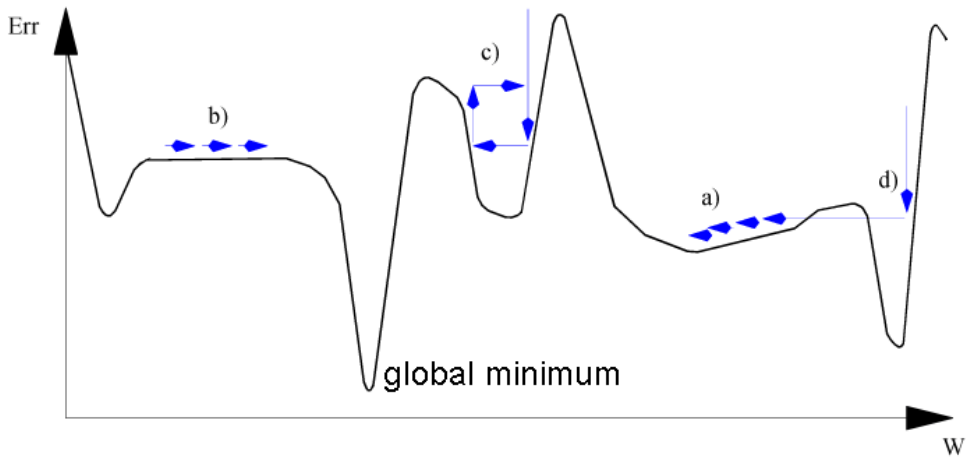



**Figure 4.4:** Possible errors during a gradient descent: a) Detecting bad minima, b) Quasi-standstill with small gradient, c) Oscillation in canyons, d) Leaving good minima.

the valley across the opposite hillside in order to come closer and closer to the deepest point of the valley by walking back and forth, similar to our ball moving within a round bowl.

**Definition 4.15** (Gradient descent)**.** Let $f$ be an $n$-dimensional function and $s = (s_1, s_2, \ldots, s_n)$ the given starting point. ***Gradient descent*** means going from $f(s)$ against the direction of $g$, i.e. towards $-g$ with steps of the size of $|g|$ towards smaller and smaller values of $f$.

Gradient descent procedures are not an errorless optimization procedure at all (as we will see in the following sections) – however, they work still well on many problems, which makes them an optimization paradigm that is frequently used. Anyway, let us have a look on their potential disadvantages so we can keep them in mind a bit.

### 4.5.1 Gradient procedures incorporate several problems

As already implied in section 4.5, the gradient descent (and therefore the backpropagation) is promising but not foolproof. One ***problem***, is that the result does not always reveal if an error has occurred.

#### 4.5.1.1 Often, gradient descents converge against suboptimal minima

Every gradient descent procedure can, for example, get stuck within a local minimum (part a of fig. 4.4 on the preceding page). This problem is increasing proportionally to the size of the error surface, and there is no universal solution. In reality, one cannot know if the optimal minimum is reached and considers a training successful, if an acceptable minimum is found.

#### 4.5.1.2 Flat plataeus on the error surface may cause training slowness

When passing a flat plateau, for instance, the gradient also becomes negligibly small because there is hardly a descent (part b of fig. 4.4), which requires many further steps. A hypothetically possible gradient of 0 would completely stop the descent.

#### 4.5.1.3 Even if good minima are reached, they may be left afterwards

On the other hand the gradient is very large at a steep slope so that large steps can be made and a good minimum can possibly be missed (part d of fig. 4.4).

#### 4.5.1.4 Steep canyons in the error surface may cause oscillations

A sudden alternation from one very strong negative gradient to a very strong positive one can even result in oscillation (part c of fig. 4.4). In nature, such an error does not occur very often so that we can think about the possibilities b and d.

## 4.6 Exemplary problems allow for testing self-coded learning strategies

We looked at learning from the formal point of view – not much yet but a little. Now it is time to look at a few exemplary problem you can later use to test implemented networks and learning rules.

| $i_1$ | $i_2$ | $i_3$ | $\Omega$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**Table 4.1:** Illustration of the parity function with three inputs.

### 4.6.1 Boolean functions

A popular example is the one that did not work in the nineteen-sixties: the XOR function ($\mathbb{B}^2 \to \mathbb{B}^1$). We need a hidden neuron layer, which we have discussed in detail. Thus, we need at least two neurons in the inner layer. Let the activation function in all layers (except in the input layer, of course) be the hyperbolic tangent. Trivially, we now expect the outputs 1.0 or $-1.0$, depending on whether the function XOR outputs 1 or 0 - and exactly here is where the first beginner's mistake occurs.

For outputs close to 1 or -1, i.e. close to the limits of the hyperbolic tangent (or in case of the Fermi function 0 or 1), we need very large network inputs. The only chance to reach these network inputs are large weights, which have to be learned: The learning process is largely extended. Therefore it is wiser to enter the teaching inputs 0.9 or $-0.9$ as desired outputs or to be satisfied when the network outputs those values instead of 1 and $-1$.

Another favourite example for singlelayer perceptrons are the boolean functions AND and OR.

### 4.6.2 The parity function

The parity function maps a set of bits to 1 or 0, depending on whether an even number of input bits is set to 1 or not. Basically, this is the function $\mathbb{B}^n \to \mathbb{B}^1$. It is characterized by easy learnability up to approx. $n = 3$ (shown in table 4.1), but the learning effort rapidly increases from $n = 4$. The reader may create a score table for the 2-bit parity function. What is conspicuous?

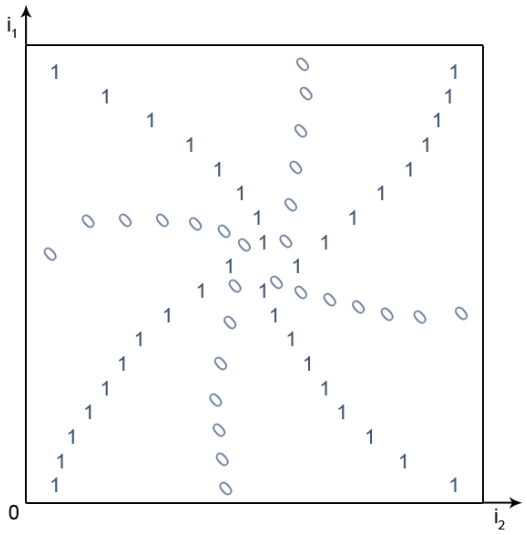



**Figure 4.5:** Illustration of the training samples of the 2-spiral problem

### 4.6.3 The 2-spiral problem

As a training sample for a function let us take two spirals coiled into each other (fig. 4.5) with the function certainly representing a mapping $\mathbb{R}^2 \to \mathbb{B}^1$. One of the spirals is assigned to the output value 1, the other spiral to 0. Here, memorizing does not help. The network has to understand the mapping itself. This example can be solved by means of an MLP, too.

### 4.6.4 The checkerboard problem

We again create a two-dimensional function of the form $\mathbb{R}^2 \to \mathbb{B}^1$ and specify checkered training samples (fig. 4.6 on the next page) with one colored field representing 1 and all the rest of them representing 0. The difficulty increases proportionally to the size of the function: While a $3 \times 3$ field is easy to learn, the larger fields are more difficult (here we eventually use methods that are more suitable for this kind of problems than the MLP).

The 2-spiral problem is very similar to the checkerboard problem, only that, mathematically speaking, the first problem is using polar coordinates instead of Cartesian coordinates. I just want to introduce as an example one last trivial case: the identity.

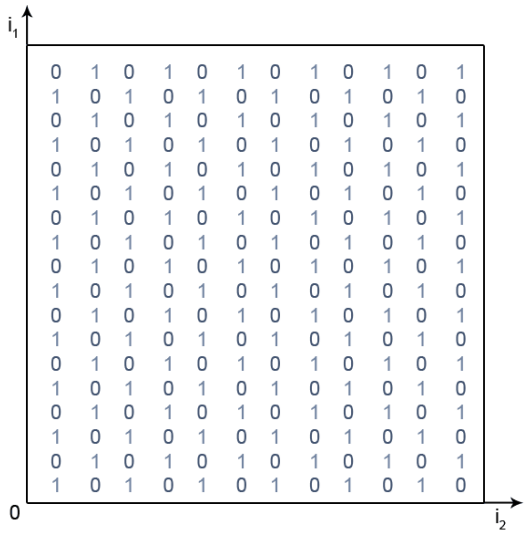



**Figure 4.6:** Illustration of training samples for the checkerboard problem

### 4.6.5 The identity function

By using linear activation functions the identity mapping from $\mathbb{R}^1$ to $\mathbb{R}^1$ (of course only within the parameters of the used activation function) is no problem for the network, but we put some obstacles in its way by using our sigmoid functions so that it would be difficult for the network to learn the identity. Just try it for the fun of it.

Now, it is time to hava a look at our first mathematical learning rule.

### 4.6.6 There are lots of other exemplary problems

For lots and lots of further exemplary problems, I want to recommend the technical report written by prechelt [Pre94] which also has been named in the sections about error measurement procedures..

## 4.7 The Hebbian learning rule is the basis for most other learning rules

In 1949, Donald O. Hebb formulated the ***Hebbian rule*** [Heb49] which is the basis for most of the more complicated learning rules we will discuss in this text. We distinguish between the original form and the more general form, which is a kind of principle for other learning rules.

### 4.7.1 Original rule

**Definition 4.16** (Hebbian rule)**.** "If neuron $j$ receives an input from neuron $i$ and if both neurons are strongly active at the same time, then increase the weight $w_{i,j}$ (i.e. the strength of the connection between $i$ and $j$)." Mathematically speaking, the rule is:

$$\Delta w_{i,j} \sim \eta o_i a_j \tag{4.5}$$

with $\Delta w_{i,j}$ being the ***change in weight*** from $i$ to $j$ , which is proportional to the following factors:

- the output $o_i$ of the predecessor neuron $i$, as well as,
- the activation $a_j$ of the successor neuron $j$,
- a constant $\eta$, i.e. the learning rate, which will be discussed in section 5.4.3.

The changes in weight $\Delta w_{i,j}$ are simply added to the weight $w_{i,j}$.

Why am I speaking twice about *activation*, but in the formula I am using $o_i$ and $a_j$, i.e. the *output* of neuron of neuron $i$ and the activation of neuron $j$? Remember that the identity is often used as output function and therefore $a_i$ and $o_i$ of a neuron are often the same. Besides, Hebb postulated his rule long before the specification of technical neurons. Considering that this learning rule was preferred in binary activations, it is clear that with the possible activations $(1, 0)$ the weights will either increase or remain constant . Sooner or later they would go ad infinitum, since they can only be corrected "upwards" when an error occurs. This can be compensated by using the activations (-1,1)[2]. Thus, the weights are decreased when the activation of the predecessor neuron dissents from the one of the successor neuron, otherwise they are increased.

2 But that is no longer the "original version" of the Hebbian rule.

### 4.7.2 Generalized form

Most of the learning rules discussed before are a specialization of the mathematically more general form [MR86] of the Hebbian rule.

**Definition 4.17** (Hebbian rule, more general)**.** The ***generalized form of the Hebbian Rule*** only specifies the proportionality of the change in weight to the product of two undefined functions, but with defined input values.

$$\Delta w_{i,j} = \eta \cdot h(o_i, w_{i,j}) \cdot g(a_j, t_j) \tag{4.6}$$

Thus, the product of the functions

- $g(a_j, t_j)$ and
- $h(o_i, w_{i,j})$
- as well as the constant learning rate $\eta$

results in the change in weight. As you can see, $h$ receives the output of the predecessor cell $o_i$ as well as the weight from predecessor to successor $w_{i,j}$ while $g$ expects the actual and desired activation of the successor $a_j$ and $t_j$ (here $t$ stands for the aforementioned *teaching input*). As already mentioned $g$ and $h$ are not specified in this general definition. Therefore, we will now return to the path of specialization we discussed before equation 4.6. After we have had a short picture of what a learning rule could look like and of our thoughts about learning itself, we will be introduced to our first network paradigm including the learning procedure.

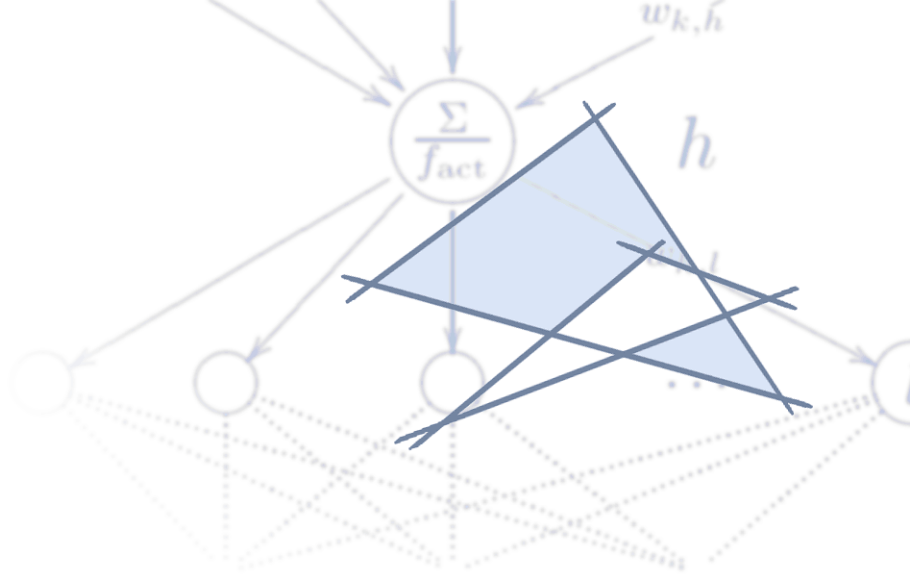
## Exercises

**Exercise 7.** Calculate the average value $\mu$ and the standard deviation $\sigma$ for the following data points.

$$p1 = (2, 2, 2)$$
$$p2 = (3, 3, 3)$$
$$p3 = (4, 4, 4)$$
$$p4 = (6, 0, 0)$$
$$p5 = (0, 6, 0)$$
$$p6 = (0, 0, 6)$$

# Part II

# Supervised learning network paradigms




# Chapter 5

# The perceptron, backpropagation and its variants

A classic among the neural networks. If we talk about a neural network, then in the majority of cases we speak about a percepton or a variation of it. Perceptrons are multilayer networks without recurrence and with fixed input and output layers. Description of a perceptron, its limits and extensions that should avoid the limitations. Derivation of learning procedures and discussion of their problems.

As already mentioned in the history of neural networks, the perceptron was described by Frank Rosenblatt in 1958 [Ros58]. Initially, Rosenblatt defined the already discussed *weighted sum* and a non-linear activation function as components of the perceptron.

There is no established definition for a perceptron, but most of the time the term is used to describe a *feedforward network with shortcut connections*. This network has a layer of scanner neurons (*retina*) with *statically* weighted connections to the following layer and is called input layer (fig. 5.1 on the next page); but the weights of all other layers are allowed to be changed. All neurons subordinate to the retina are pattern detectors. Here we initially use a binary perceptron with every output neuron having exactly two possible output values (e.g. $\{0, 1\}$ or $\{-1, 1\}$). Thus, a binary threshold function is used as activation function, depending on the threshold value $\Theta$ of the output neuron.

In a way, the binary activation function represents an IF query which can also be negated by means of negative weights. The perceptron can thus be used to accomplish true logical information processing.

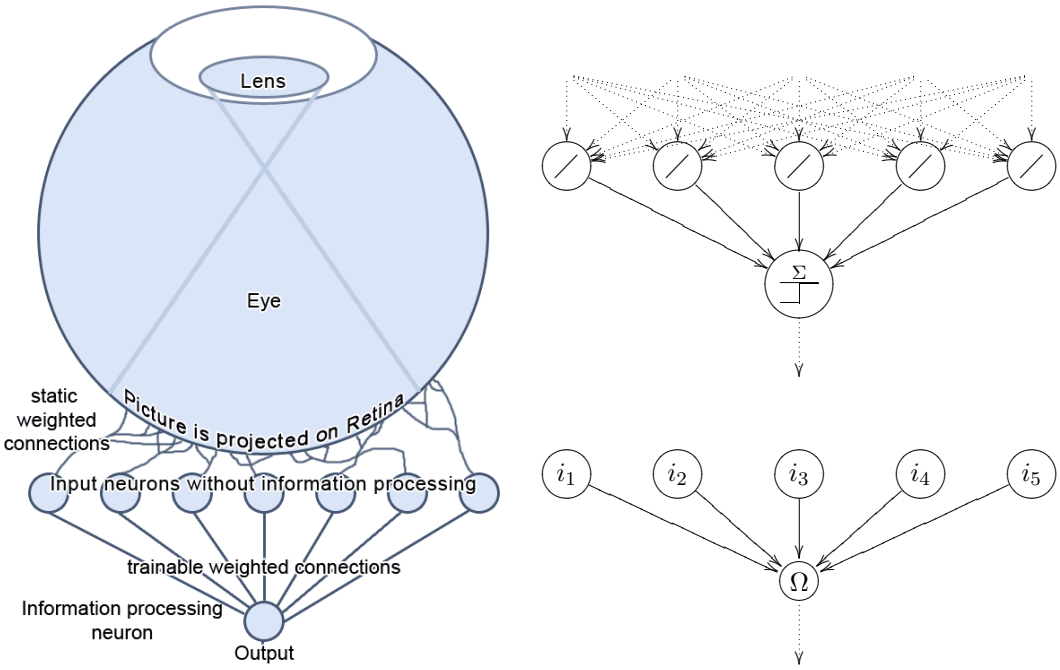



**Figure 5.1:** Architecture of a perceptron with one layer of variable connections in different views. The solid-drawn weight layer in the two illustrations on the bottom can be trained.
**Left side:** Example of scanning information in the eye.
**Right side, upper part:** Drawing of the same example with indicated fixed-weight layer using the defined designs of the functional descriptions for neurons.
**Right side, lower part:** Without indicated fixed-weight layer, with the name of each neuron corresponding to our convention. The fixed-weight layer will no longer be taken into account in the course of this work.

Whether this method is reasonable is another matter – of course, this is not the easiest way to achieve Boolean logic. I just want to illustrate that perceptrons can be used as simple logical components and that, theoretically speaking, any Boolean function can be realized by means of perceptrons being connected in series or interconnected in a sophisticated way. But we will see that this is not possible without connecting them serially. Before providing the definition of the perceptron, I want to define some types of neurons used in this chapter.

**Definition 5.1** (Input neuron)**.** An ***input neuron*** is an ***identity neuron***. It exactly forwards the information received. Thus, it represents the identity function, which

should be indicated by the symbol $\diagup$. Therefore the input neuron is represented by the symbol  .

**Definition 5.2** (Information processing neuron)**.** ***Information processing neurons*** somehow process the input information, i.e. do not represent the identity function. A ***binary neuron*** sums up all inputs by using the weighted sum as propagation function, which we want to illustrate by the sign $\Sigma$. Then the activation function of the neuron is the binary threshold function, which can be illustrated by  . This leads us to the complete depiction of information processing neurons, namely  . Other neurons that use the weighted sum as propagation function but the activation functions *hyperbolic tangent* or *Fermi function*, or with a separately defined activation function $f_{\text{act}}$, are similarly represented by


These neurons are also referred to as ***Fermi neurons*** or ***Tanh neuron***.

Now that we know the components of a perceptron we should be able to define it.

**Definition 5.3** (Perceptron)**.** The ***perceptron*** (fig. 5.1 on the facing page) is[1] a feedforward network containing a ***retina*** that is used only for data acquisition and which has fixed-weighted connections with the first neuron layer (input layer). The fixed-weight layer is followed by at least one trainable weight layer. One neuron layer is completely linked with the following layer. The first layer of the perceptron consists of the *input neurons* defined above.

A feedforward network often contains shortcuts which does not exactly correspond to the original description and therefore is not included in the definition. We can see that the retina is not included in the lower part of fig. 5.1. As a matter of fact the first neuron layer is often understood (simplified and sufficient for this method) as input layer, because this layer only forwards the input values. The retina itself and the static weights behind it are no longer mentioned or displayed, since they do not process information in any case. So, the depiction of a perceptron starts with the input neurons.

1 It may confuse some readers that I claim that there is no definition of a perceptron but then define the perceptron in the following section. I therefore suggest keeping my definition in the back of your mind and just take it for granted in the course of this work.

**SNIPE:** The methods `setSettingsTopologyFeedForward` and the variation `-WithShortcuts` in a `NeuralNetworkDescriptor`-Instance apply settings to a descriptor, which are appropriate for feedforward networks or feedforward networks with shortcuts. The respective kinds of connections are allowed, all others are not, and fastprop is activated.

## 5.1 The singlelayer perceptron provides only one trainable weight layer

Here, connections with trainable weights go from the input layer to an output neuron $\Omega$, which returns the information whether the pattern entered at the input neurons was recognized or not. Thus, a singlelayer perception (abbreviated SLP) has only one level of trainable weights (fig. 5.1 on page 84).

**Definition 5.4** (Singlelayer perceptron)**.** A ***singlelayer perceptron*** (***SLP***) is a perceptron having only one layer of variable weights and one layer of output neurons $\Omega$. The technical view of an SLP is shown in fig. 5.2 on the facing page.

Certainly, the existence of several output neurons $\Omega_1, \Omega_2, \ldots, \Omega_n$ does not considerably change the concept of the perceptron (fig. 5.3 on the next page): A perceptron with several output neurons can also be regarded as several different perceptrons with the same input.

The Boolean functions `AND` and `OR` shown in fig. 5.4 on page 88 are trivial examples that can easily be composed.

Now we want to know how to train a singlelayer perceptron. We will therefore at first take a look at the perceptron learning algorithm and then we will look at the delta rule.

### 5.1.1 Perceptron learning algorithm and convergence theorem

The original ***perceptron learning algorithm*** with binary neuron activation function is described in alg. 1. It has been proven that the algorithm converges in finite time – so in finite time the perceptron can learn anything it can represent (***perceptron convergence theorem***, [Ros62]). But please do not get your hopes up too soon! What the perceptron is capable to represent will be explored later.

During the exploration of linear separability of problems we will cover the fact that at least the singlelayer perceptron unfortunately cannot represent a lot of problems.

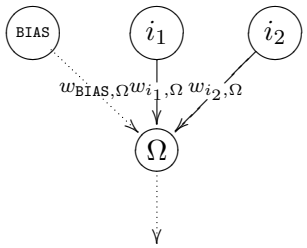



**Figure 5.2:** A singlelayer perceptron with two input neurons and one output neuron. The network returns the output by means of the arrow leaving the network. The trainable layer of weights is situated in the center (labeled). As a reminder, the bias neuron is again included here. Although the weight $w_{\mathtt{BIAS},\Omega}$ is a normal weight and also treated like this, I have represented it by a dotted line – which significantly increases the clarity of larger networks. In future, the bias neuron will no longer be included.

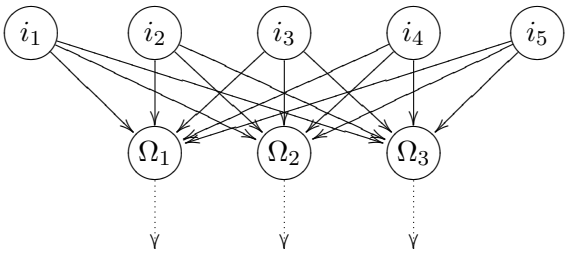



**Figure 5.3:** Singlelayer perceptron with several output neurons

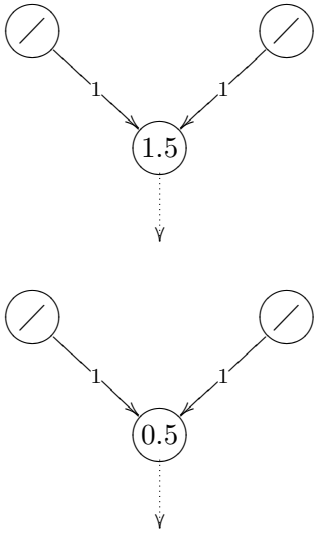



**Figure 5.4:** Two singlelayer perceptrons for Boolean functions. The upper singlelayer perceptron realizes an AND, the lower one realizes an OR. The activation function of the information processing neuron is the binary threshold function. Where available, the threshold values are written into the neurons.

```
1: while ∃p ∈ P and error too large do
2:    Input p into the network, calculate output y {P set of training patterns}
3:    for all output neurons Ω do
4:       if y_Ω = t_Ω then
5:          Output is okay, no correction of weights
6:       else
7:          if y_Ω = 0 then
8:             for all input neurons i do
9:                w_{i,Ω} := w_{i,Ω} + o_i {...increase weight towards Ω by o_i}
10:            end for
11:         end if
12:         if y_Ω = 1 then
13:            for all input neurons i do
14:               w_{i,Ω} := w_{i,Ω} − o_i {...decrease weight towards Ω by o_i}
15:            end for
16:         end if
17:      end if
18:   end for
19: end while
```

**Algorithm 1:** Perceptron learning algorithm. The perceptron learning algorithm reduces the weights to output neurons that return 1 instead of 0, and in the inverse case increases weights.

### 5.1.2 The delta rule as a gradient based learning strategy for SLPs

In the following we deviate from our binary threshold value as activation function because at least for *backpropagation of error* we need, as you will see, a *differentiable* or even a *semi-linear* activation function. For the now following delta rule (like backpropagation derived in [MR86]) it is not always necessary but useful. This fact, however, will also be pointed out in the appropriate part of this work. Compared with the aforementioned perceptron learning algorithm, the delta rule has the advantage to be suitable for non-binary activation functions and, being far away from the learning target, to automatically learn faster.

Suppose that we have a singlelayer perceptron with randomly set weights which we want to teach a function by means of training samples. The set of these training samples is called $P$. It contains, as already defined, the pairs $(p, t)$ of the training samples $p$ and the associated teaching input $t$. I also want to remind you that

- $x$ is the input vector and
- $y$ is the output vector of a neural network,
- output neurons are referred to as $\Omega_1, \Omega_2, \ldots, \Omega_{|O|}$,
- $i$ is the input and
- $o$ is the output of a neuron.

Additionally, we defined that

- the error vector $E_p$ represents the difference $(t - y)$ under a certain training sample $p$.
- Furthermore, let $O$ be the set of output neurons and
- $I$ be the set of input neurons.

Another naming convention shall be that, for example, for an output $o$ and a teaching input $t$ an additional index $p$ may be set in order to indicate that these values are pattern-specific. Sometimes this will considerably enhance clarity.

Now our learning target will certainly be, that for all training samples the output $y$ of the network is approximately the desired output $t$, i.e. formally it is true that

$$\forall p : y \approx t \quad \text{or} \quad \forall p : E_p \approx 0.$$

This means we first have to understand the total error Err as a function of the weights: The total error increases or decreases depending on how we change the weights.

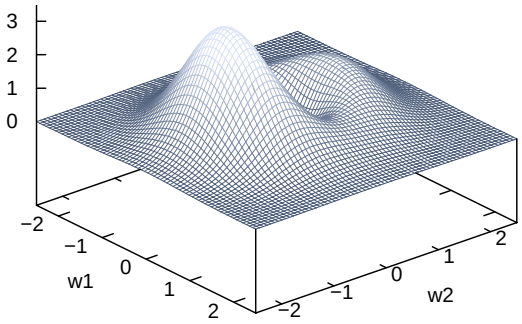



**Figure 5.5:** Exemplary error surface of a neural network with two trainable connections $w_1$ und $w_2$. Generally, neural networks have more than two connections, but this would have made the illustration too complex. And most of the time the error surface is too craggy, which complicates the search for the minimum.

**Definition 5.5** (Error function)**.** The ***error function***

$$\mathrm{Err} : W \to \mathbb{R}$$

regards the set[2] of weights $W$ as a vector and maps the values onto the normalized output error (normalized because otherwise not all errors can be mapped onto one single $e \in \mathbb{R}$ to perform a gradient descent). It is obvious that a ***specific error function*** can analogously be generated for a single pattern $p$.

As already shown in section 4.5, gradient descent procedures calculate the gradient of an arbitrary but finite-dimensional function (here: of the error function $\mathrm{Err}(W)$) and move down against the direction of the gradient until a minimum is reached. $\mathrm{Err}(W)$ is defined on the set of all weights which we here regard as the vector $W$. So we try to decrease or to minimize the error by simply tweaking the weights – thus one receives information about how to change the weights (the change in all weights is referred to as $\Delta W$) by calculating the gradient $\nabla\mathrm{Err}(W)$ of the error function $\mathrm{Err}(W)$:

$$\Delta W \sim -\nabla\mathrm{Err}(W). \tag{5.1}$$

Due to this relation there is a proportionality constant $\eta$ for which equality holds ($\eta$ will soon get another meaning and a real practical use beyond the mere meaning of a proportionality constant. I just ask the reader to be patient for a while.):

$$\Delta W = -\eta\nabla\mathrm{Err}(W). \tag{5.2}$$

2 Following the tradition of the literature, I previously defined $W$ as a weight *matrix*. I am aware of this conflict but it should not bother us here.

To simplify further analysis, we now rewrite the gradient of the error-function according to all weights as an usual partial derivative according to a single weight $w_{i,\Omega}$ (the only variable weights exists between the hidden and the output layer $\Omega$). Thus, we tweak every single weight and observe how the error function changes, i.e. we derive the error function according to a weight $w_{i,\Omega}$ and obtain the value $\Delta w_{i,\Omega}$ of how to change this weight.

$$\Delta w_{i,\Omega} = -\eta \frac{\partial \mathrm{Err}(W)}{\partial w_{i,\Omega}}. \tag{5.3}$$

Now the following question arises: How is our error function defined exactly? It is not good if many results are far away from the desired ones; the error function should then provide large values – on the other hand, it is similarly bad if many results are close to the desired ones but there exists an extremely far outlying result. The ***squared distance*** between the output vector $y$ and the teaching input $t$ appears adequate to our needs. It provides the error $\mathrm{Err}_p$ that is specific for a training sample $p$ over the output of all output neurons $\Omega$:

$$\mathrm{Err}_p(W) = \frac{1}{2} \sum_{\Omega \in O} (t_{p,\Omega} - y_{p,\Omega})^2. \tag{5.4}$$

Thus, we calculate the squared difference of the components of the vectors $t$ and $y$, given the pattern $p$, and sum up these squares. The summation of the specific errors $\mathrm{Err}_p(W)$ of all patterns $p$ then yields the definition of the error Err and therefore the definition of the error function $\mathrm{Err}(W)$:

$$\mathrm{Err}(W) = \sum_{p \in P} \mathrm{Err}_p(W) \tag{5.5}$$

$$= \frac{1}{2} \overbrace{\sum_{p \in P} \underbrace{\left( \sum_{\Omega \in O} (t_{p,\Omega} - y_{p,\Omega})^2 \right)}_{\text{sum over all } \Omega}}^{\text{sum over all } p}. \tag{5.6}$$

The observant reader will certainly wonder where the factor $\frac{1}{2}$ in equation 5.4 suddenly came from and why there is no root in the equation, as this formula looks very similar to the Euclidean distance. Both facts result from simple pragmatics: Our intention is to minimize the error. Because the root function decreases with its argument, we can simply omit it for reasons of calculation and implementation efforts, since we do not need it for minimization. Similarly, it does not matter if the term to be minimized is divided by 2: Therefore I am allowed to multiply by $\frac{1}{2}$. This is just done so that it cancels with a 2 in the course of our calculation.

Now we want to continue deriving the delta rule for linear activation functions. We have already discussed that we tweak the individual weights $w_{i,\Omega}$ a bit and see how the error $\mathrm{Err}(W)$ is changing – which corresponds to the derivative of the error function $\mathrm{Err}(W)$ according to the very same weight $w_{i,\Omega}$. This derivative corresponds to the sum of the derivatives of all specific errors $\mathrm{Err}_p$ according to this weight (since the total error $\mathrm{Err}(W)$ results from the sum of the specific errors):

$$\Delta w_{i,\Omega} = -\eta \frac{\partial \mathrm{Err}(W)}{\partial w_{i,\Omega}} \tag{5.7}$$

$$= \sum_{p \in P} -\eta \frac{\partial \mathrm{Err}_p(W)}{\partial w_{i,\Omega}}. \tag{5.8}$$

Once again I want to think about the question of how a neural network processes data. Basically, the data is only transferred through a function, the result of the function is sent through another one, and so on. If we ignore the output function, the path of the neuron outputs $o_{i_1}$ and $o_{i_2}$, which the neurons $i_1$ and $i_2$ entered into a neuron $\Omega$, initially is the propagation function (here weighted sum), from which the network input is going to be received. This is then sent through the activation function of the neuron $\Omega$ so that we receive the output of this neuron which is at the same time a component of the output vector $y$:

$$\begin{aligned} \mathrm{net}_\Omega &\rightarrow f_{\mathrm{act}} \\ &= f_{\mathrm{act}}(\mathrm{net}_\Omega) \\ &= o_\Omega \\ &= y_\Omega. \end{aligned}$$

As we can see, this output results from many nested functions:

$$o_\Omega = f_{\mathrm{act}}(\mathrm{net}_\Omega) \tag{5.9}$$

$$= f_{\mathrm{act}}(o_{i_1} \cdot w_{i_1,\Omega} + o_{i_2} \cdot w_{i_2,\Omega}). \tag{5.10}$$

It is clear that we could break down the output into the single input neurons (this is unnecessary here, since they do not process information in an SLP). Thus, we want to calculate the derivatives of equation 5.8 and due to the nested functions we can apply the *chain rule* to factorize the derivative $\frac{\partial \mathrm{Err}_p(W)}{\partial w_{i,\Omega}}$ in equation 5.8.

$$\frac{\partial \mathrm{Err}_p(W)}{\partial w_{i,\Omega}} = \frac{\partial \mathrm{Err}_p(W)}{\partial o_{p,\Omega}} \cdot \frac{\partial o_{p,\Omega}}{\partial w_{i,\Omega}}. \tag{5.11}$$

Let us take a look at the first multiplicative factor of the above equation 5.11 on the preceding page which represents the derivative of the specific error $\mathrm{Err}_p(W)$ according to the output, i.e. the change of the error $\mathrm{Err}_p$ with an output $o_{p,\Omega}$: The examination of $\mathrm{Err}_p$ (equation 5.4 on page 92) clearly shows that this change is exactly the difference between teaching input and output $(t_{p,\Omega} - o_{p,\Omega})$ (remember: Since $\Omega$ is an output neuron, $o_{p,\Omega} = y_{p,\Omega}$). The closer the output is to the teaching input, the smaller is the specific error. Thus we can replace one by the other. This difference is also called $\delta_{p,\Omega}$ (which is the reason for the name delta rule):

$$\frac{\partial \mathrm{Err}_p(W)}{\partial w_{i,\Omega}} = -(t_{p,\Omega} - o_{p,\Omega}) \cdot \frac{\partial o_{p,\Omega}}{\partial w_{i,\Omega}} \tag{5.12}$$

$$= -\delta_{p,\Omega} \cdot \frac{\partial o_{p,\Omega}}{\partial w_{i,\Omega}} \tag{5.13}$$

The second multiplicative factor of equation 5.11 on the preceding page and of the following one is the derivative of the output specific to the pattern $p$ of the neuron $\Omega$ according to the weight $w_{i,\Omega}$. So how does $o_{p,\Omega}$ change when the weight from $i$ to $\Omega$ is changed? Due to the requirement at the beginning of the derivation, we only have a linear activation function $f_{\mathrm{act}}$, therefore we can just as well look at the change of the network input when $w_{i,\Omega}$ is changing:

$$\frac{\partial \mathrm{Err}_p(W)}{\partial w_{i,\Omega}} = -\delta_{p,\Omega} \cdot \frac{\partial \sum_{i \in I}(o_{p,i} w_{i,\Omega})}{\partial w_{i,\Omega}}. \tag{5.14}$$

The resulting derivative $\frac{\partial \sum_{i \in I}(o_{p,i} w_{i,\Omega})}{\partial w_{i,\Omega}}$ can now be simplified: The function $\sum_{i \in I}(o_{p,i} w_{i,\Omega})$ to be derived consists of many summands, and only the summand $o_{p,i} w_{i,\Omega}$ contains the variable $w_{i,\Omega}$, according to which we derive. Thus, $\frac{\partial \sum_{i \in I}(o_{p,i} w_{i,\Omega})}{\partial w_{i,\Omega}} = o_{p,i}$ and therefore:

$$\frac{\partial \mathrm{Err}_p(W)}{\partial w_{i,\Omega}} = -\delta_{p,\Omega} \cdot o_{p,i} \tag{5.15}$$

$$= -o_{p,i} \cdot \delta_{p,\Omega}. \tag{5.16}$$

We insert this in equation 5.8 on the previous page, which results in our modification rule for a weight $w_{i,\Omega}$:

$$\Delta w_{i,\Omega} = \eta \cdot \sum_{p \in P} o_{p,i} \cdot \delta_{p,\Omega}. \tag{5.17}$$

However: From the very beginning the derivation has been intended as an offline rule by means of the question of how to add the errors of all patterns and how to learn them *after* all patterns have been represented. Although this approach is mathematically correct, the implementation is far more time-consuming and, as we will see later in this chapter, partially needs a lot of compuational effort during training.

The "online-learning version" of the delta rule simply omits the summation and learning is realized immediately after the presentation of each pattern, this also simplifies the notation (which is no longer necessarily related to a pattern $p$):

$$\Delta w_{i,\Omega} = \eta \cdot o_i \cdot \delta_\Omega. \tag{5.18}$$

This version of the delta rule shall be used for the following definition:

**Definition 5.6** (Delta rule)**.** If we determine, analogously to the aforementioned derivation, that the function $h$ of the Hebbian theory (equation 4.6 on page 79) only provides the output $o_i$ of the predecessor neuron $i$ and if the function $g$ is the difference between the desired activation $t_\Omega$ and the actual activation $a_\Omega$, we will receive the ***delta rule***, also known as ***Widrow-Hoff rule***:

$$\Delta w_{i,\Omega} = \eta \cdot o_i \cdot (t_\Omega - a_\Omega) = \eta o_i \delta_\Omega \tag{5.19}$$

If we use the desired output (instead of the activation) as teaching input, and therefore the output function of the output neurons does not represent an identity, we obtain

$$\Delta w_{i,\Omega} = \eta \cdot o_i \cdot (t_\Omega - o_\Omega) = \eta o_i \delta_\Omega \tag{5.20}$$

and $\delta_\Omega$ then corresponds to the difference between $t_\Omega$ and $o_\Omega$.

In the case of the delta rule, the change of all weights to an output neuron $\Omega$ is proportional

- ▷ to the difference between the current activation or output $a_\Omega$ or $o_\Omega$ and the corresponding teaching input $t_\Omega$. We want to refer to this factor as $\delta_\Omega$ , which is also referred to as "***Delta***".

Apparently the delta rule only applies for SLPs, since the formula is always related to the teaching input, and there is *no teaching input* for the inner processing layers of neurons.

| In. 1 | In. 2 | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Table 5.1:** Definition of the logical XOR. The input values are shown of the left, the output values on the right.

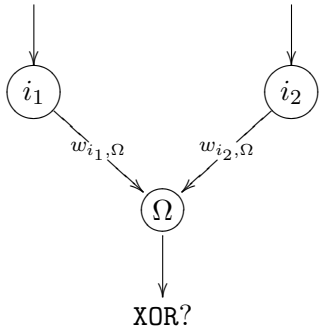



**Figure 5.6:** Sketch of a singlelayer perceptron that shall represent the XOR function - which is impossible.

## 5.2 A SLP is only capable of representing linearly separable data

Let $f$ be the XOR function which expects two binary inputs and generates a binary output (for the precise definition see table 5.1).

Let us try to represent the XOR function by means of an SLP with two input neurons $i_1, i_2$ and one output neuron $\Omega$ (fig. 5.6).

Here we use the weighted sum as propagation function, a binary activation function with the threshold value $\Theta$ and the identity as output function. Depending on $i_1$ and $i_2$, $\Omega$ has to output the value 1 if the following holds:

$$\text{net}_\Omega = o_{i_1} w_{i_1,\Omega} + o_{i_2} w_{i_2,\Omega} \geq \Theta_\Omega \tag{5.21}$$

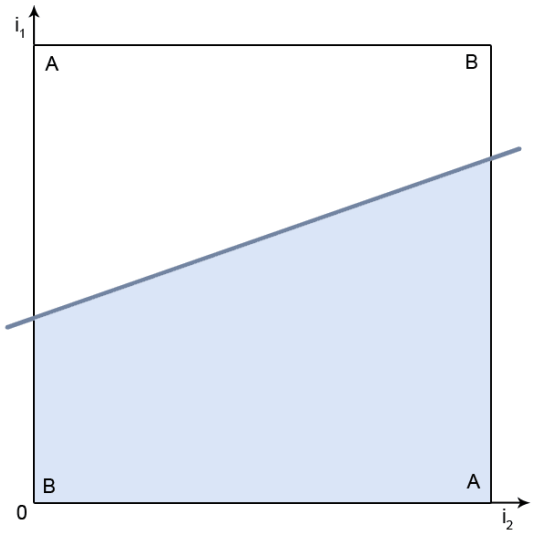



**Figure 5.7:** Linear separation of $n = 2$ inputs of the input neurons $i_1$ and $i_2$ by a 1-dimensional straight line. A and B show the corners belonging to the sets of the XOR function that are to be separated.

We assume a positive weight $w_{i_2,\Omega}$, the inequality 5.21 on the preceding page is then equivalent to

$$o_{i_1} \geq \frac{1}{w_{i_1,\Omega}}(\Theta_\Omega - o_{i_2} w_{i_2,\Omega}) \tag{5.22}$$

With a constant threshold value $\Theta_\Omega$, the right part of inequation 5.22 is a straight line through a coordinate system defined by the possible outputs $o_{i_1}$ und $o_{i_2}$ of the input neurons $i_1$ and $i_2$ (fig. 5.7).

For a (as required for inequation 5.22) positive $w_{i_2,\Omega}$ the output neuron $\Omega$ fires for input combinations lying *above* the generated straight line. For a negative $w_{i_2,\Omega}$ it would fire for all input combinations lying below the straight line. Note that only the four corners of the unit square are possible inputs because the XOR function only knows binary inputs.

In order to solve the XOR problem, we have to turn and move the straight line so that input set $A = \{(0,0),(1,1)\}$ is separated from input set $B = \{(0,1),(1,0)\}$ – this is, obviously, impossible.

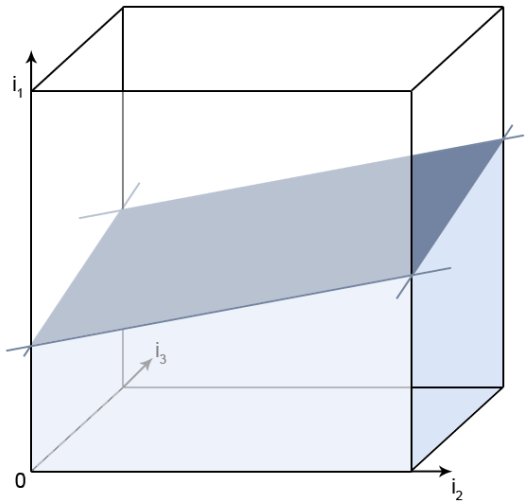



**Figure 5.8:** Linear separation of $n = 3$ inputs from input neurons $i_1$, $i_2$ and $i_3$ by 2-dimensional plane.

Generally, the input parameters of $n$ many input neurons can be represented in an $n$-dimensional cube which is separated by an SLP through an $(n-1)$-dimensional hyperplane (fig. 5.8). Only sets that can be separated by such a hyperplane, i.e. which are ***linearly separable***, can be classified by an SLP.

Unfortunately, it seems that the percentage of the linearly separable problems rapidly decreases with increasing $n$ (see table 5.2 on the facing page), which limits the functionality of the SLP. Additionally, tests for linear separability are difficult. Thus, for more difficult tasks with more inputs we need something more powerful than SLP. The XOR problem itself is one of these tasks, since a perceptron that is supposed to represent the XOR function already needs a hidden layer (fig. 5.9 on the next page).

## 5.3 A multilayer perceptron contains more trainable weight layers

A perceptron with two or more trainable weight layers (called multilayer perceptron or MLP) is more powerful than an SLP. As we know, a singlelayer perceptron can divide

| $n$ | number of binary functions | lin. separable ones | share |
|---|---|---|---|
| 1 | 4 | 4 | 100% |
| 2 | 16 | 14 | 87.5% |
| 3 | 256 | 104 | 40.6% |
| 4 | 65,536 | 1,772 | 2.7% |
| 5 | $4.3 \cdot 10^9$ | 94,572 | 0.002% |
| 6 | $1.8 \cdot 10^{19}$ | 5,028,134 | $\approx 0\%$ |

**Table 5.2:** Number of functions concerning $n$ binary inputs, and number and proportion of the functions thereof which can be linearly separated. In accordance with [Zel94, Wid89, Was89].

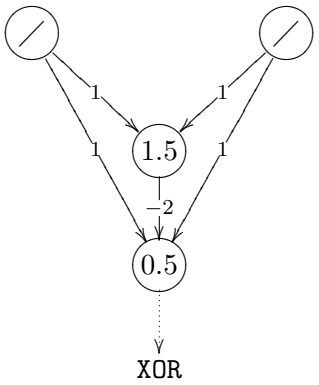



**Figure 5.9:** Neural network realizing the XOR function. Threshold values (as far as they are existing) are located within the neurons.

the input space by means of a hyperplane (in a two-dimensional input space by means of a straight line). A two-stage perceptron (two trainable weight layers, three neuron layers) can classify *convex polygons* by further processing these straight lines, e.g. in the form "recognize patterns lying above straight line 1, below straight line 2 and below straight line 3". Thus, we – metaphorically speaking - took an SLP with several output neurons and "attached" another SLP (upper part of fig. 5.10 on the facing page). A multilayer perceptron represents an ***universal function approximator***, which is proven by the *Theorem of Cybenko* [Cyb89].

Another trainable weight layer proceeds analogously, now with the convex polygons. Those can be added, subtracted or somehow processed with other operations (lower part of fig. 5.10 on the next page).

Generally, it can be mathematically proven that even a multilayer perceptron with one layer of hidden neurons can arbitrarily precisely approximate functions with only finitely many discontinuities as well as their first derivatives. Unfortunately, this proof is not constructive and therefore it is left to us to find the correct number of neurons and weights.

In the following we want to use a widespread abbreviated form for different multilayer perceptrons: We denote a two-stage perceptron with 5 neurons in the input layer, 3 neurons in the hidden layer and 4 neurons in the output layer as a 5-3-4-MLP.

**Definition 5.7** (Multilayer perceptron)**.** Perceptrons with more than one layer of variably weighted connections are referred to as ***multilayer perceptrons*** (***MLP***). An $n$-layer or $n$-stage perceptron has thereby exactly $n$ variable weight layers and $n+1$ neuron layers (the retina is disregarded here) with neuron layer 1 being the input layer.

Since three-stage perceptrons can classify sets of any form by combining and separating arbitrarily many convex polygons, another step will not be advantageous with respect to function representations. Be cautious when reading the literature: There are many different definitions of what is counted as a layer. Some sources count the neuron layers, some count the weight layers. Some sources include the retina, some the trainable weight layers. Some exclude (for some reason) the output neuron layer. In this work, I chose the definition that provides, in my opinion, the most information about the learning capabilities – and I will use it cosistently. Remember: An $n$-stage perceptron has exactly $n$ trainable weight layers. You can find a summary of which perceptrons can classify which types of sets in table 5.3 on page 102. We now want to face the challenge of training perceptrons with more than one weight layer.

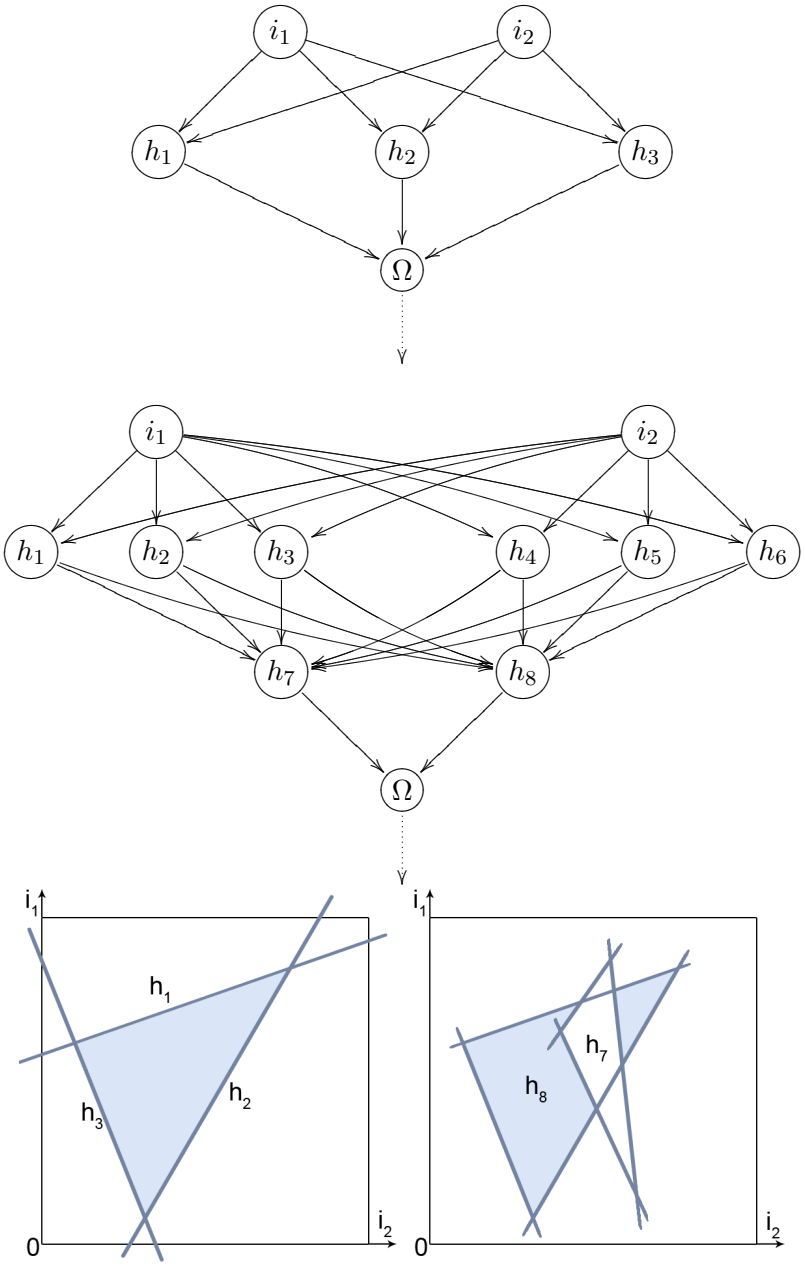



**Figure 5.10:** We know that an SLP represents a straight line. With 2 trainable weight layers, several straight lines can be combined to form convex polygons (above). By using 3 trainable weight layers several polygons can be formed into arbitrary sets (below).

| $n$ | classifiable sets |
|---|---|
| 1 | hyperplane |
| 2 | convex polygon |
| 3 | any set |
| 4 | any set as well, i.e. no advantage |

**Table 5.3:** Representation of which perceptron can classify which types of sets with $n$ being the number of trainable weight layers.

## 5.4 Backpropagation of error generalizes the delta rule to allow for MLP training

Next, I want to derive and explain the ***backpropagation of error*** learning rule (abbreviated: backpropagation, backprop or BP), which can be used to train multi-stage perceptrons with *semi-linear*[3] activation functions. Binary threshold functions and other non-differentiable functions are *no* longer supported, but that doesn't matter: We have seen that the Fermi function or the hyperbolic tangent can arbitrarily approximate the binary threshold function by means of a temperature parameter $T$. To a large extent I will follow the derivation according to [Zel94] and [MR86]. Once again I want to point out that this procedure had previously been published by Paul Werbos in [Wer74] but had consideraby less readers than in [MR86].

Backpropagation is a gradient descent procedure (including all strengths and weaknesses of the gradient descent) with the error function $\mathrm{Err}(W)$ receiving all $n$ weights as arguments (fig. 5.5 on page 91) and assigning them to the output error, i.e. being $n$-dimensional. On $\mathrm{Err}(W)$ a point of small error or even a point of the smallest error is sought by means of the gradient descent. Thus, in analogy to the delta rule, backpropagation trains the weights of the neural network. And it is exactly the delta rule or its variable $\delta_i$ for a neuron $i$ which is *expanded* from one trainable weight layer to several ones by backpropagation.

3 Semilinear functions are monotonous and differentiable – but generally they are not linear.

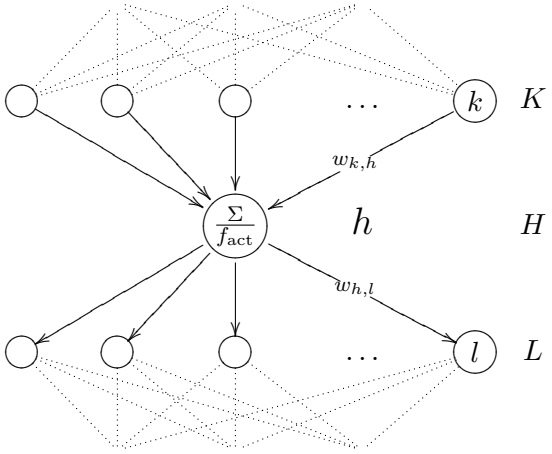



**Figure 5.11:** Illustration of the position of our neuron $h$ within the neural network. It is lying in layer $H$, the preceding layer is $K$, the subsequent layer is $L$.

### 5.4.1 The derivation is similar to the one of the delta rule, but with a generalized delta

Let us define in advance that the network input of the individual neurons $i$ results from the weighted sum. Furthermore, as with the derivation of the delta rule, let $o_{p,i}$, $\text{net}_{p,i}$ etc. be defined as the already familiar $o_i$, $\text{net}_i$, etc. under the input pattern $p$ we used for the training. Let the output function be the identity again, thus $o_i = f_{\text{act}}(\text{net}_{p,i})$ holds for any neuron $i$. Since this is a generalization of the delta rule, we use the same formula framework as with the delta rule (equation 5.20 on page 95). As already indicated, we have to generalize the variable $\delta$ for every neuron.

First of all: Where is the neuron for which we want to calculate $\delta$? It is obvious to select an arbitrary inner neuron $h$ having a set $K$ of predecessor neurons $k$ as well as a set of $L$ successor neurons $l$, which are *also inner neurons* (see fig. 5.11). It is therefore irrelevant whether the predecessor neurons are already the input neurons.

Now we perform the same derivation as for the delta rule and split functions by means the chain rule. I will not discuss this derivation in great detail, but the principal

is similar to that of the delta rule (the differences are, as already mentioned, in the generalized $\delta$). We initially derive the error function Err according to a weight $w_{k,h}$.

$$\frac{\partial \mathrm{Err}(w_{k,h})}{\partial w_{k,h}} = \underbrace{\frac{\partial \mathrm{Err}}{\partial \mathrm{net}_h}}_{=-\delta_h} \cdot \frac{\partial \mathrm{net}_h}{\partial w_{k,h}} \tag{5.23}$$

The first factor of equation 5.23 is $-\delta_h$, which we will deal with later in this text. The numerator of the second factor of the equation includes the network input, i.e. the weighted sum is included in the numerator so that we can immediately derive it. Again, all summands of the sum drop out apart from the summand containing $w_{k,h}$. This summand is referred to as $w_{k,h} \cdot o_k$. If we calculate the derivative, the output of neuron $k$ becomes:

$$\frac{\partial \mathrm{net}_h}{\partial w_{k,h}} = \frac{\partial \sum_{k \in K} w_{k,h} o_k}{\partial w_{k,h}} \tag{5.24}$$

$$= \boxed{o_k} \tag{5.25}$$

As promised, we will now discuss the $-\delta_h$ of equation 5.23, which is split up again according of the chain rule:

$$\delta_h = -\frac{\partial \mathrm{Err}}{\partial \mathrm{net}_h} \tag{5.26}$$

$$= -\frac{\partial \mathrm{Err}}{\partial o_h} \cdot \frac{\partial o_h}{\partial \mathrm{net}_h} \tag{5.27}$$

The derivation of the output according to the network input (the second factor in equation 5.27) clearly equals the derivation of the activation function according to the network input:

$$\frac{\partial o_h}{\partial \mathrm{net}_h} = \frac{\partial f_{\mathrm{act}}(\mathrm{net}_h)}{\partial \mathrm{net}_h} \tag{5.28}$$

$$= \boxed{f_{\mathrm{act}}{}'(\mathrm{net}_h)} \tag{5.29}$$

Consider this an important passage! We now analogously derive the first factor in equation 5.27. Therefore, we have to point out that the derivation of the error function according to the output of an inner neuron layer depends on the vector of all network inputs of the next following layer. This is reflected in equation 5.30:

$$-\frac{\partial \mathrm{Err}}{\partial o_h} = -\frac{\partial \mathrm{Err}(\mathrm{net}_{l_1}, \ldots, \mathrm{net}_{l_{|L|}})}{\partial o_h} \tag{5.30}$$

According to the definition of the multi-dimensional chain rule, we immediately obtain equation 5.31:

$$-\frac{\partial \mathrm{Err}}{\partial o_h} = \sum_{l \in L} \left( -\frac{\partial \mathrm{Err}}{\partial \mathrm{net}_l} \cdot \frac{\partial \mathrm{net}_l}{\partial o_h} \right) \tag{5.31}$$

The sum in equation 5.31 contains two factors. Now we want to discuss these factors being added over the subsequent layer $L$. We simply calculate the second factor in the following equation 5.33:

$$\frac{\partial \mathrm{net}_l}{\partial o_h} = \frac{\partial \sum_{h \in H} w_{h,l} \cdot o_h}{\partial o_h} \tag{5.32}$$

$$= \boxed{w_{h,l}} \tag{5.33}$$

The same applies for the first factor according to the definition of our $\delta$:

$$-\frac{\partial \mathrm{Err}}{\partial \mathrm{net}_l} = \boxed{\delta_l} \tag{5.34}$$

Now we insert:

$$\Rightarrow -\frac{\partial \mathrm{Err}}{\partial o_h} = \sum_{l \in L} \delta_l w_{h,l} \tag{5.35}$$

You can find a graphic version of the $\delta$ generalization including all splittings in fig. 5.12 on the following page.

The reader might already have noticed that some intermediate results were shown in frames. Exactly those intermediate results were highlighted in that way, which are a factor in the change in weight of $w_{k,h}$. If the aforementioned equations are combined with the highlighted intermediate results, the outcome of this will be the wanted change in weight $\Delta w_{k,h}$ to

$$\begin{aligned} \Delta w_{k,h} &= \eta o_k \delta_h \text{ with} \\ \delta_h &= f'_{\mathrm{act}}(\mathrm{net}_h) \cdot \sum_{l \in L} (\delta_l w_{h,l}) \end{aligned} \tag{5.36}$$

– of course only in case of $h$ being an inner neuron (otherweise there would not be a subsequent layer $L$).

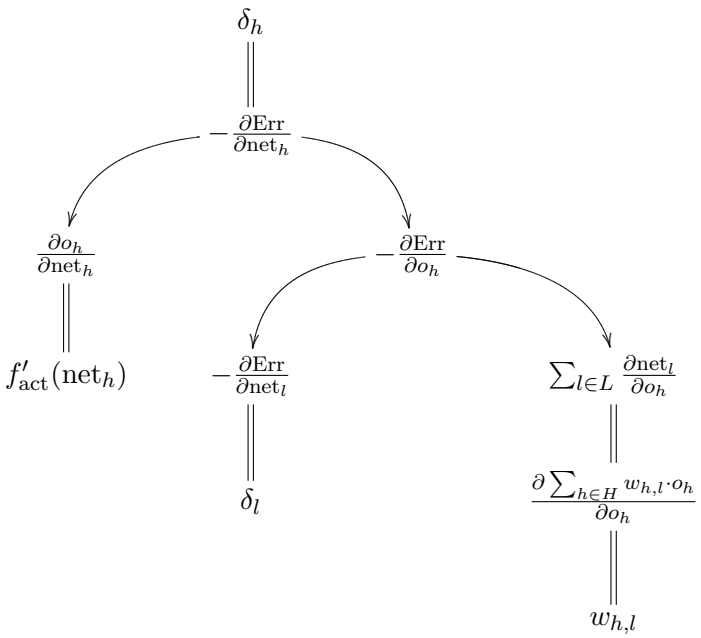



**Figure 5.12:** Graphical representation of the equations (by equal signs) and chain rule splittings (by arrows) in the framework of the backpropagation derivation. The leaves of the tree reflect the final results from the generalization of $\delta$, which are framed in the derivation.

The case of $h$ being an output neuron has already been discussed during the derivation of the delta rule. All in all, the result is the generalization of the delta rule, called *backpropagation of error*:

$$\begin{aligned} &\Delta w_{k,h} = \eta o_k \delta_h \text{ with} \\ &\delta_h = \begin{cases} f'_{\text{act}}(\text{net}_h) \cdot (t_h - y_h) & (h \text{ outside}) \\ f'_{\text{act}}(\text{net}_h) \cdot \sum_{l \in L}(\delta_l w_{h,l}) & (h \text{ inside}) \end{cases} \end{aligned} \tag{5.37}$$

In contrast to the delta rule, $\delta$ is treated differently depending on whether h is an output or an inner (i.e. hidden) neuron:

1. If $h$ is an output neuron, then

$$\delta_{p,h} = f'_{\text{act}}(\text{net}_{p,h}) \cdot (t_{p,h} - y_{p,h}) \tag{5.38}$$

   Thus, under our training pattern $p$ the weight $w_{k,h}$ from $k$ to $h$ is changed proportionally according to

   - the learning rate $\eta$,
   - the output $o_{p,k}$ of the predecessor neuron $k$,
   - the gradient of the activation function at the position of the network input of the successor neuron $f'_{\text{act}}(\text{net}_{p,h})$ and
   - the difference between teaching input $t_{p,h}$ and output $y_{p,h}$ of the successor neuron $h$.

   *In this case, backpropagation is working on two neuron layers*, the output layer with the successor neuron $h$ and the preceding layer with the predecessor neuron $k$.

2. If $h$ is an inner, hidden neuron, then

$$\delta_{p,h} = f'_{\text{act}}(\text{net}_{p,h}) \cdot \sum_{l \in L}(\delta_{p,l} \cdot w_{h,l}) \tag{5.39}$$

   holds. I want to explicitly mention *that backpropagation is now working on three layers*. Here, neuron $k$ is the predecessor of the connection to be changed with the weight $w_{k,h}$, the neuron $h$ is the successor of the connection to be changed and the neurons $l$ are lying in the layer *following* the successor neuron. Thus, according to our training pattern $p$, the weight $w_{k,h}$ from $k$ to $h$ is proportionally changed according to

   - the learning rate $\eta$,

- the output of the predecessor neuron $o_{p,k}$,
- the gradient of the activation function at the position of the network input of the successor neuron $f'_{\text{act}}(\text{net}_{p,h})$,
- as well as, and this is the difference, according to the weighted sum of the changes in weight to all neurons following $h$, $\sum_{l \in L}(\delta_{p,l} \cdot w_{h,l})$.

**Definition 5.8** (Backpropagation)**.** If we summarize formulas 5.38 on the previous page and 5.39 on the preceding page, we receive the following final formula for ***backpropagation*** (the identifiers $p$ are ommited for reasons of clarity):

$$\begin{aligned} &\Delta w_{k,h} = \eta o_k \delta_h \text{ with} \\ &\delta_h = \begin{cases} f'_{\text{act}}(\text{net}_h) \cdot (t_h - y_h) & (h \text{ outside}) \\ f'_{\text{act}}(\text{net}_h) \cdot \sum_{l \in L}(\delta_l w_{h,l}) & (h \text{ inside}) \end{cases} \end{aligned} \tag{5.40}$$

**SNIPE:** An online variant of backpropagation is implemented in the method `trainBackpropagationOfError` within the class `NeuralNetwork`.

It is obvious that backpropagation initially processes the last weight layer directly by means of the teaching input and then works backwards from layer to layer while considering each preceding change in weights. *Thus, the teaching input leaves traces in all weight layers.* Here I describe the first (delta rule) and the second part of backpropagation (generalized delta rule on more layers) in one go, which may meet the requirements of the matter but not of the research. The first part is obvious, which you will soon see in the framework of a mathematical gimmick. *Decades of development time and work lie between the first and the second, recursive part.* Like many groundbreaking inventions, it was *not until* its development that it was recognized how plausible this invention was.

### 5.4.2 Heading back: Boiling backpropagation down to delta rule

As explained above, the delta rule is a special case of backpropagation for one-stage perceptrons and linear activation functions – I want to briefly explain this circumstance and develop the delta rule out of backpropagation in order to augment the understanding of both rules. We have seen that backpropagation is defined by

$$\begin{aligned} &\Delta w_{k,h} = \eta o_k \delta_h \text{ with} \\ &\delta_h = \begin{cases} f'_{\text{act}}(\text{net}_h) \cdot (t_h - y_h) & (h \text{ outside}) \\ f'_{\text{act}}(\text{net}_h) \cdot \sum_{l \in L}(\delta_l w_{h,l}) & (h \text{ inside}) \end{cases} \end{aligned} \tag{5.41}$$

Since we only use it for one-stage perceptrons, the second part of backpropagation (light-colored) is omitted without substitution. The result is:

$$\begin{aligned} &\Delta w_{k,h} = \eta o_k \delta_h \text{ with} \\ &\delta_h = f'_{\text{act}}(\text{net}_h) \cdot (t_h - o_h) \end{aligned} \tag{5.42}$$

Furthermore, we only want to use linear activation functions so that $f'_{\text{act}}$ (light-colored) is constant. As is generally known, constants can be combined, and therefore we directly merge the constant derivative $f'_{\text{act}}$ and (being constant for at least one lerning cycle) the learning rate $\eta$ (also light-colored) in $\eta$. Thus, the result is:

$$\Delta w_{k,h} = \eta o_k \delta_h = \eta o_k \cdot (t_h - o_h) \tag{5.43}$$

This exactly corresponds to the delta rule definition.

### 5.4.3 The selection of the learning rate has heavy influence on the learning process

In the meantime we have often seen that the change in weight is, in any case, proportional to the learning rate $\eta$. Thus, the selection of $\eta$ is crucial for the behaviour of backpropagation and for learning procedures in general.

**Definition 5.9** (Learning rate)**.** Speed and accuracy of a learning procedure can always be controlled by and are always proportional to a ***learning rate*** which is written as $\eta$.

If the value of the chosen $\eta$ is too large, the jumps on the error surface are also too large and, for example, narrow valleys could simply be jumped over. Additionally, the movements across the error surface would be very uncontrolled. Thus, a small $\eta$ is the desired input, which, however, can cost a huge, often unacceptable amount of time. Experience shows that good learning rate values are in the range of

$$0.01 \leq \eta \leq 0.9.$$

The selection of $\eta$ significantly depends on the problem, the network and the training data, so that it is barely possible to give practical advise. But for instance it is popular to start with a relatively large $\eta$, e.g. 0.9, and to slowly decrease it down to 0.1. For simpler problems $\eta$ can often be kept constant.

#### 5.4.3.1 Variation of the learning rate over time

During training, another stylistic device can be a ***variable learning rate***: In the beginning, a large learning rate leads to good results, but later it results in inaccurate learning. A smaller learning rate is more time-consuming, but the result is more precise. Thus, during the learning process the learning rate needs to be decreased by one order of magnitude once or repeatedly.

A common error (which also seems to be a very neat solution at first glance) is to continually decrease the learning rate. Here it quickly happens that the descent of the learning rate is larger than the ascent of a hill of the error function we are climbing. The result is that we simply get stuck at this ascent. Solution: Rather reduce the learning rate gradually as mentioned above.

#### 5.4.3.2 Different layers – Different learning rates

The farer we move away from the output layer during the learning process, the slower backpropagation is learning. Thus, it is a good idea to select a larger learning rate for the weight layers close to the input layer than for the weight layers close to the output layer.

## 5.5 Resilient backpropagation is an extension to backpropagation of error

We have just raised two backpropagation-specific properties that can occasionally be a problem (in addition to those which are already caused by gradient descent itself): On the one hand, users of backpropagation can choose a bad learning rate. On the other hand, the further the weights are from the output layer, the slower backpropagation learns. For this reason, Martin Riedmiller et al. enhanced backpropagation and called their version ***resilient backpropagation*** (short ***Rprop***) [RB93, Rie94]. I want to compare backpropagation and Rprop, without explicitly declaring one version superior to the other. Before actually dealing with formulas, let us informally compare the two primary ideas behind Rprop (and their consequences) to the already familiar backpropagation.

**Learning rates:** Backpropagation uses by default a learning rate $\eta$, which is selected by the user, and applies to the entire network. It remains static until it is manually changed. We have already explored the disadvantages of this approach.

Here, Rprop pursues a completely different approach: there is no global learning rate. First, each weight $w_{i,j}$ has its own learning rate $\eta_{i,j}$, and second, these learning rates are not chosen by the user, but are automatically set by Rprop itself. Third, the weight changes are not static but are adapted for each time step of Rprop. To account for the temporal change, we have to correctly call it $\eta_{i,j}(t)$. This not only enables more focused learning, also the problem of an increasingly slowed down learning throughout the layers is solved in an elegant way.

**Weight change:** When using backpropagation, weights are changed proportionally to the gradient of the error function. At first glance, this is really intuitive. However, we incorporate every jagged feature of the error surface into the weight changes. It is at least questionable, whether this is always useful. Here, Rprop takes other ways as well: the amount of weight change $\Delta w_{i,j}$ simply directly corresponds to the automatically adjusted learning rate $\eta_{i,j}$. Thus the change in weight is *not* proportional to the gradient, it is only influenced by the sign of the gradient. Until now we still do not know how exactly the $\eta_{i,j}$ are adapted at run time, but let me anticipate that the resulting process looks considerably less rugged than an error function.

In contrast to backprop the weight update step is replaced and an additional step for the adjustment of the learning rate is added. Now how exactly are these ideas being implemented?

### 5.5.1 Weight changes are not proportional to the gradient

Let us first consider the change in weight. We have already noticed that the weight-specific learning rates directly serve as absolute values for the changes of the respective weights. There remains the question of where the sign comes from – this is a point at which the gradient comes into play. As with the derivation of backpropagation, we derive the error function $\mathrm{Err}(W)$ by the individual weights $w_{i,j}$ and obtain gradients $\frac{\partial \mathrm{Err}(W)}{\partial w_{i,j}}$. Now, the big difference: rather than multiplicatively incorporating the absolute value of the gradient into the weight change, we consider only the *sign* of the gradient. The gradient hence no longer determines the strength, but only the direction of the weight change.

If the sign of the gradient $\frac{\partial \mathrm{Err}(W)}{\partial w_{i,j}}$ is positive, we must decrease the weight $w_{i,j}$. So the weight is reduced by $\eta_{i,j}$. If the sign of the gradient is negative, the weight needs to be increased. So $\eta_{i,j}$ is added to it. If the gradient is exactly 0, nothing happens at all. Let us now create a formula from this colloquial description. The corresponding

terms are affixed with a $(t)$ to show that everything happens at the same time step. This might decrease clarity at first glance, but is nevertheless important because we will soon look at another formula that operates on different time steps. Instead, we shorten the gradient to: $g = \frac{\partial \mathrm{Err}(W)}{\partial w_{i,j}}$.

**Definition 5.10** (Weight change in Rprop)**.**

$$\Delta w_{i,j}(t) = \begin{cases} -\eta_{i,j}(t), & \text{if } g(t) > 0 \\ +\eta_{i,j}(t), & \text{if } g(t) < 0 \\ 0 & \text{otherwise.} \end{cases} \tag{5.44}$$

We now know how the weights are changed – now remains the question how the learning rates are adjusted. Finally, once we have understood the overall system, we will deal with the remaining details like initialization and some specific constants.

### 5.5.2 Many dynamically adjusted learning rates instead of one static

To adjust the learning rate $\eta_{i,j}$, we again have to consider the associated gradients $g$ of two time steps: the gradient that has just passed $(t-1)$ and the current one $(t)$. Again, only the sign of the gradient matters, and we now must ask ourselves: What can happen to the sign over two time steps? It can stay the same, and it can flip.

If the sign changes from $g(t-1)$ to $g(t)$, we have skipped a local minimum in the gradient. Hence, the last update was too large and $\eta_{i,j}(t)$ has to be reduced as compared to the previous $\eta_{i,j}(t-1)$. One can say, that the search needs to be more accurate. In mathematical terms, we obtain a new $\eta_{i,j}(t)$ by multiplying the old $\eta_{i,j}(t-1)$ with a constant $\eta^{\downarrow}$, which is between 1 and 0. In this case we know that in the last time step $(t-1)$ something went wrong – hence we additionally reset the weight update for the weight $w_{i,j}$ at time step $(t)$ to 0, so that it not applied at all (not shown in the following formula).

However, if the sign remains the same, one can perform a (careful!) increase of $\eta_{i,j}$ to get past shallow areas of the error function. Here we obtain our new $\eta_{i,j}(t)$ by multiplying the old $\eta_{i,j}(t-1)$ with a constant $\eta^{\uparrow}$ which is greater than 1.

**Definition 5.11** (Adaptation of learning rates in Rprop)**.**

$$\eta_{i,j}(t) = \begin{cases} \eta^{\uparrow}\eta_{i,j}(t-1), & g(t-1)g(t) > 0 \\ \eta^{\downarrow}\eta_{i,j}(t-1), & g(t-1)g(t) < 0 \\ \eta_{i,j}(t-1) & \text{otherwise.} \end{cases} \tag{5.45}$$

Caution: This also implies that Rprop is exclusively designed for offline. If the gradients do not have a certain continuity, the learning process slows down to the lowest rates (and remains there). When learning online, one changes – loosely speaking – the error function with each new epoch, since it is based on only one training pattern. This may be often well applicable in backpropagation and it is very often even faster than the offline version, which is why it is used there frequently. It lacks, however, a clear mathematical motivation, and that is exactly what we need here.

### 5.5.3 We are still missing a few details to use Rprop in practice

A few minor issues remain unanswered, namely

1. How large are $\eta^{\uparrow}$ and $\eta^{\downarrow}$ (i.e. how much are learning rates reinforced or weakened)?
2. How to choose $\eta_{i,j}(0)$ (i.e. how are the weight-specific learning rates initialized)?[4]
3. What are the upper and lower bounds $\eta_{\min}$ and $\eta_{\max}$ for $\eta_{i,j}$ set?

We now answer these questions with a quick motivation. The initial value for the learning rates should be somewhere in the order of the initialization of the weights. $\eta_{i,j}(0) = 0.1$ has proven to be a good choice. The authors of the Rprop paper explain in an obvious way that this value – as long as it is positive and without an exorbitantly high absolute value – does not need to be dealt with very critically, as it will be quickly overridden by the automatic adaptation anyway.

Equally uncritical is $\eta_{\max}$, for which they recommend, without further mathematical justification, a value of 50 which is used throughout most of the literature. One can set this parameter to lower values in order to allow only very cautious updates. Small update steps should be allowed in any case, so we set $\eta_{\min} = 10^{-6}$.

Now we have left only the parameters $\eta^{\uparrow}$ and $\eta^{\downarrow}$. Let us start with $\eta^{\downarrow}$: If this value is used, we have skipped a minimum, from which we do not know where exactly it lies on the skipped track. Analogous to the procedure of binary search, where the target object is often skipped as well, we assume it was in the middle of the skipped track. So we need to halve the learning rate, which is why the canonical choice $\eta^{\downarrow} = 0.5$ is being selected. If the value of $\eta^{\uparrow}$ is used, learning rates shall be increased with caution. Here we cannot generalize the principle of binary search and simply use the value 2.0, otherwise the learning rate update will end up consisting almost exclusively of changes in direction. Independent of the particular problems, a value of $\eta^{\uparrow} = 1.2$ has proven

4 Protipp: since the $\eta_{i,j}$ can be changed only by multiplication, 0 would be a rather suboptimal initialization :-)

to be promising. Slight changes of this value have not significantly affected the rate of convergence. This fact allowed for setting this value as a constant as well.

With advancing computational capabilities of computers one can observe a more and more widespread distribution of networks that consist of a big number of layers, i.e. ***deep networks*** . For such networks it is crucial to prefer Rprop over the original backpropagation, because backprop, as already indicated, learns very slowly at weights wich are far from the output layer. For problems with a smaller number of layers, I would recommend testing the more widespread backpropagation (with both offline and online learning) and the less common Rprop equivalently.

**SNIPE:** In Snipe resilient backpropagation is supported via the method `trainResilientBackpropagation` of the class `NeuralNetwork`. Furthermore, you can also use an additional improvement to resilient propagation, which is, however, not dealt with in this work. There are getters and setters for the different parameters of Rprop.

## 5.6 Backpropagation has often been extended and altered besides Rprop

Backpropagation has often been extended. Many of these extensions can simply be implemented as optional features of backpropagation in order to have a larger scope for testing. In the following I want to briefly describe some of them.

### 5.6.1 Adding momentum to learning

Let us assume to descent a steep slope on skis - what prevents us from immediately stopping at the edge of the slope to the plateau? Exactly - our *momentum*. With backpropagation the *momentum term* [RHW86b] is responsible for the fact that a kind of *moment of inertia* (***momentum***) is added to every step size (fig. 5.13 on the next page), by always adding a fraction of the previous change to every new change in weight:

$$(\Delta_p w_{i,j})_{\text{now}} = \eta o_{p,i} \delta_{p,j} + \alpha \cdot (\Delta_p w_{i,j})_{\text{previous}}.$$

Of course, this notation is only used for a better understanding. Generally, as already defined by the concept of time, when referring to the current cycle as $(t)$, then the previous cycle is identified by $(t-1)$, which is continued successively. And now we come to the formal definition of the momentum term:

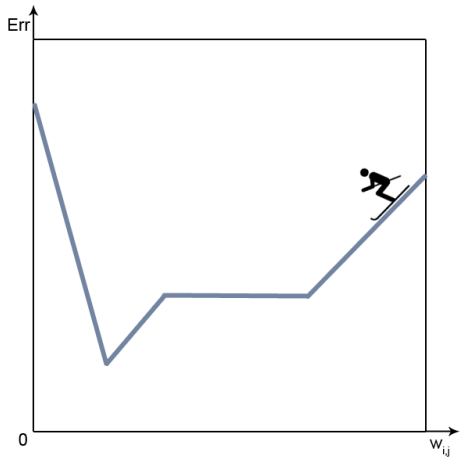



**Figure 5.13:** We want to execute the gradient descent like a skier crossing a slope, who would hardly stop immediately at the edge to the plateau.

**Definition 5.12** (Momentum term)**.** The variation of backpropagation by means of the ***momentum term*** is defined as follows:

$$\Delta w_{i,j}(t) = \eta o_i \delta_j + \alpha \cdot \Delta w_{i,j}(t-1) \tag{5.46}$$

We accelerate on plateaus (avoiding quasi-standstill on plateaus) and slow down on craggy surfaces (preventing oscillations). Moreover, the effect of inertia can be varied via the prefactor $\alpha$, common values are between 0.6 und 0.9. Additionally, the momentum enables the positive effect that our skier swings back and forth several times in a minimum, and finally lands in the minimum. Despite its nice one-dimensional appearance, the otherwise very rare error of leaving good minima unfortunately occurs more frequently because of the momentum term – which means that this is again no optimal solution (but we are by now accustomed to this condition).

### 5.6.2 Flat spot elimination prevents neurons from getting stuck

It must be pointed out that with the *hyperbolic tangent* as well as with the *Fermi function* the derivative outside of the close proximity of $\Theta$ is nearly 0. This results in the fact that it becomes very difficult to move neurons away from the limits of the activation (*flat spots*) , which could extremely extend the learning time. This problem

can be dealt with by modifying the derivative, for example by adding a constant (e.g. 0.1), which is called ***flat spot elimination*** or – more colloquial – ***fudging***.

It is an interesting observation, that success has also been achieved by using derivatives defined as constants [Fah88]. A nice example making use of this effect is the fast hyperbolic tangent approximation by Anguita et al. introduced in section 3.2.6 on page 42. In the outer regions of it's (as well approximated and accelerated) derivative, it makes use of a small constant.

### 5.6.3 The second derivative can be used, too

According to David Parker [Par87], ***Second order backpropagation*** also usese the second gradient, i.e. the second multi-dimensional derivative of the error function, to obtain more precise estimates of the correct $\Delta w_{i,j}$. Even higher derivatives only rarely improve the estimations. Thus, less training cycles are needed but those require much more computational effort.

In general, we use further derivatives (i.e. Hessian matrices, since the functions are multidimensional) for higher order methods. As expected, the procedures reduce the number of learning epochs, but significantly increase the computational effort of the individual epochs. So in the end these procedures often need more learning time than backpropagation.

The ***quickpropagation*** learning procedure [Fah88] uses the second derivative of the error propagation and locally understands the error function to be a parabola. We analytically determine the vertex (i.e. the lowest point) of the said parabola and directly jump to this point. Thus, this learning procedure is a second-order procedure. Of course, this does not work with error surfaces that cannot locally be approximated by a parabola (certainly it is not always possible to directly say whether this is the case).

### 5.6.4 Weight decay: Punishment of large weights

The weight decay according to Paul Werbos [Wer88] is a modification that extends the error by a term punishing large weights. So the error under weight decay

$$\mathrm{Err}_{\mathrm{WD}}$$

does not only increase proportionally to the actual error but also proportionally to the square of the weights. As a result the network is keeping the weights small during learning.

$$\mathrm{Err}_{\mathrm{WD}} = \mathrm{Err} + \underbrace{\beta \cdot \frac{1}{2} \sum_{w \in W} (w)^2}_{\text{punishment}} \tag{5.47}$$

This approach is inspired by nature where synaptic weights cannot become infinitely strong as well. Additionally, due to these small weights, the error function often shows weaker fluctuations, allowing easier and more controlled learning.

The prefactor $\frac{1}{2}$ again resulted from simple pragmatics. The factor $\beta$ controls the strength of punishment: Values from 0.001 to 0.02 are often used here.

### 5.6.5 Cutting networks down: Pruning and Optimal Brain Damage

If we have executed the weight decay long enough and notice that for a neuron in the input layer all successor weights are 0 or close to 0, we can remove the neuron, hence losing this neuron and some weights and thereby reduce the possibility that the network will memorize. This procedure is called ***pruning***.

Such a method to detect and delete unnecessary weights and neurons is referred to as ***optimal brain damage*** [lCDS90]. I only want to describe it briefly: The mean error per output neuron is composed of two competing terms. While one term, as usual, considers the difference between output and teaching input, the other one tries to "press" a weight towards 0. If a weight is strongly needed to minimize the error, the first term will win. If this is not the case, the second term will win. Neurons which only have zero weights can be pruned again in the end.

There are many other variations of backprop and whole books only about this subject, but since my aim is to offer an overview of neural networks, I just want to mention the variations above as a motivation to read on.

For some of these extensions it is obvious that they cannot only be applied to feedforward networks with backpropagation learning procedures.

We have gotten to know backpropagation and feedforward topology – now we have to learn how to build a neural network. It is of course impossible to fully communicate this experience in the framework of this work. To obtain at least some of this knowledge, I now advise you to deal with some of the exemplary problems from 4.6.

## 5.7 Getting started – Initial configuration of a multilayer perceptron

After having discussed the backpropagation of error learning procedure and knowing how to train an existing network, it would be useful to consider how to implement such a network.

### 5.7.1 Number of layers: Two or three may often do the job, but more are also used

Let us begin with the trivial circumstance that a network should have one layer of input neurons and one layer of output neurons, which results in at least two layers.

Additionally, we need – as we have already learned during the examination of linear separability – at least one hidden layer of neurons, if our problem is not linearly separable (which is, as we have seen, very likely).

It is possible, as already mentioned, to mathematically prove that this MLP with one hidden neuron layer is already capable of approximating arbitrary functions with any accuracy [5] – but it is necessary not only to discuss the ***representability*** of a problem by means of a perceptron but also the ***learnability***. Representability means that a perceptron can, in principle, realize a mapping - but learnability means that we are also able to teach it.

In this respect, experience shows that two hidden neuron layers (or three trainable weight layers) can be very useful to solve a problem, since many problems can be represented by a hidden layer but are very difficult to learn.

One should keep in mind that any additional layer generates additional sub-minima of the error function in which we can get stuck. All these things considered, a promising way is to try it with one hidden layer at first and if that fails, retry with two layers. Only if that fails, one should consider more layers. However, given the increasing calculation power of current computers, ***deep networks*** with a lot of layers are also used with success.

5 Note: We have not indicated the number of neurons in the hidden layer, we only mentioned the hypothetical possibility.

### 5.7.2 The number of neurons has to be tested

The number of neurons (apart from input and output layer, where the number of input and output neurons is already defined by the problem statement) principally corresponds to the number of free parameters of the problem to be represented.

Since we have already discussed the network capacity with respect to memorizing or a too imprecise problem representation, it is clear that our goal is to have as *few* free parameters as possible but as *many* as necessary.

But we also know that there is no standard solution for the question of how many neurons should be used. Thus, the most useful approach is to initially train with only a few neurons and to repeatedly train new networks with more neurons until the result significantly improves and, particularly, the generalization performance is not affected (*bottom-up approach*).

### 5.7.3 Selecting an activation function

Another very important parameter for the way of information processing of a neural network is the ***selection of an activation function***. The activation function for input neurons is fixed to the identity function, since they do not process information.

The first question to be asked is whether we actually want to use the same activation function in the hidden layer and in the ouput layer – no one prevents us from choosing different functions. Generally, the activation function is the same for all hidden neurons as well as for the output neurons respectively.

For tasks of ***function approximation*** it has been found reasonable to use the hyperbolic tangent (left part of fig. 5.14 on the next page) as activation function of the hidden neurons, while a linear activation function is used in the output. The latter is absolutely necessary so that we do not generate a limited output intervall. Contrary to the input layer which uses linear activation functions as well, the output layer still processes information, because it has threshold values. However, linear activation functions in the output can also cause huge learning steps and jumping over good minima in the error surface. This can be avoided by setting the learning rate to very small values in the output layer.

An unlimited output interval is not essential for ***pattern recognition*** tasks[6]. If the hyperbolic tangent is used in any case, the output interval will be a bit larger. Unlike

6 Generally, pattern recognition is understood as a special case of function approximation with a few discrete output possibilities.

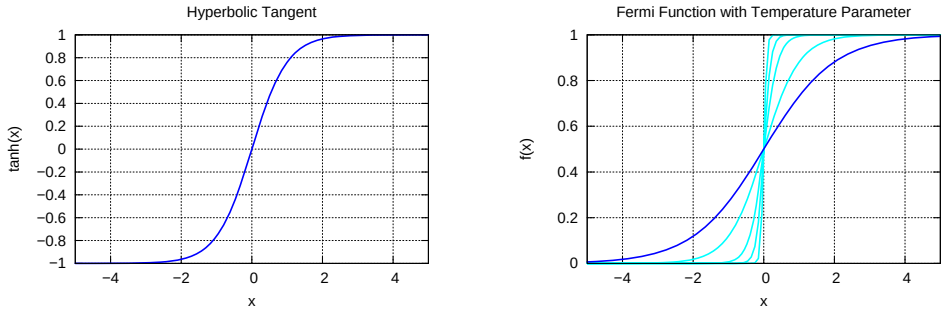



**Figure 5.14:** As a reminder the illustration of the hyperbolic tangent (left) and the Fermi function (right). The Fermi function was expanded by a temperature parameter. The original Fermi function is thereby represented by dark colors, the temperature parameter of the modified Fermi functions are, ordered ascending by steepness, $\frac{1}{2}$, $\frac{1}{5}$, $\frac{1}{10}$ and $\frac{1}{25}$.

with the hyperbolic tangent, with the Fermi function (right part of fig. 5.14) it is difficult to learn something far from the threshold value (where its result is close to 0). However, here a lot of freedom is given for selecting an activation function. But generally, the disadvantage of sigmoid functions is the fact that they hardly learn something for values far from thei threshold value, unless the network is modified.

### 5.7.4 Weights should be initialized with small, randomly chosen values

The initialization of weights is not as trivial as one might think. If they are simply initialized with 0, there will be no change in weights at all. If they are all initialized by the same value, they will all change equally during training. The simple solution of this problem is called ***symmetry breaking***, which is the initialization of weights with small random values. The range of random values could be the interval $[-0.5; 0.5]$ not including 0 or values very close to 0. This random initialization has a nice side effect: Chances are that the average of network inputs is close to 0, a value that hits (in most activation functions) the region of the greatest derivative, allowing for strong learning impulses right from the start of learning.

**SNIPE:** In Snipe, weights are initialized randomly (if a synapse initialization is wanted). The maximum absolute weight value of a synapse initialized at random can be set in a `NeuralNetworkDescriptor` using the method `setSynapseInitialRange`.

## 5.8 The 8-3-8 encoding problem and related problems

The 8-3-8 encoding problem is a classic among the multilayer perceptron test training problems. In our MLP we have an input layer with eight neurons $i_1, i_2, \ldots, i_8$, an output layer with eight neurons $\Omega_1, \Omega_2, \ldots, \Omega_8$ and one hidden layer with three neurons. Thus, this network represents a function $\mathbb{B}^8 \to \mathbb{B}^8$. Now the training task is that an input of a value 1 into the neuron $i_j$ should lead to an output of a value 1 from the neuron $\Omega_j$ (only one neuron should be activated, which results in 8 training samples.

During the analysis of the trained network we will see that the network with the 3 hidden neurons represents some kind of binary encoding and that the above mapping is possible (assumed training time: $\approx 10^4$ epochs). Thus, our network is a machine in which the input is first encoded and afterwards decoded again.

Analogously, we can train a 1024-10-1024 encoding problem. But is it possible to improve the efficiency of this procedure? Could there be, for example, a 1024-9-1024- or an 8-2-8-encoding network?

Yes, even that is possible, since the network does not depend on binary encodings: Thus, an 8-2-8 network is sufficient for our problem. But the encoding of the network is far more difficult to understand (fig. 5.15 on the next page) and the training of the networks requires a lot more time.

**SNIPE:** The static method `getEncoderSampleLesson` in the class `TrainingSampleLesson` allows for creating simple training sample lessons of arbitrary dimensionality for encoder problems like the above.

An 8-1-8 network, however, does not work, since the possibility that the output of one neuron is compensated by another one is essential, and if there is only one hidden neuron, there is certainly no compensatory neuron.

## Exercises

**Exercise 8.** Fig. 5.4 on page 88 shows a small network for the boolean functions `AND` and `OR`. Write tables with all computational parameters of neural networks (e.g. network input, activation etc.). Perform the calculations for the four possible inputs of the networks and write down the values of these variables for each input. Do the same for the `XOR` network (fig. 5.9 on page 99).

**Exercise 9.**

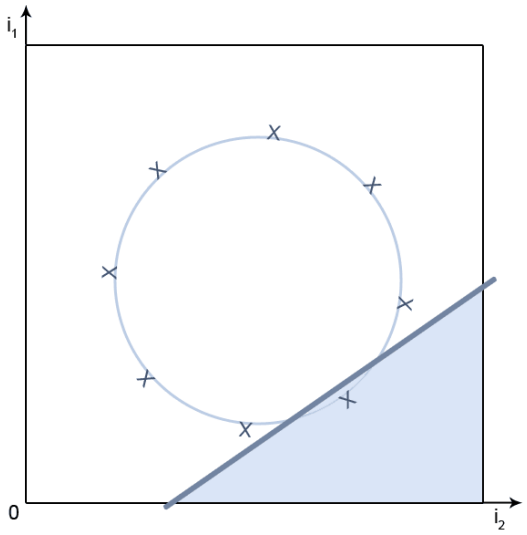



**Figure 5.15:** Illustration of the functionality of 8-2-8 network encoding. The marked points represent the vectors of the inner neuron activation associated to the samples. As you can see, it is possible to find inner activation formations so that each point can be separated from the rest of the points by a straight line. The illustration shows an exemplary separation of one point.

1. List all boolean functions $\mathbb{B}^3 \to \mathbb{B}^1$, that are linearly separable and characterize them exactly.

2. List those that are not linearly separable and characterize them exactly, too.

**Exercise 10.** A simple 2-1 network shall be trained with one single pattern by means of backpropagation of error and $\eta = 0.1$. Verify if the error

$$\mathrm{Err} = \mathrm{Err}_p = \frac{1}{2}(t - y)^2$$

converges and if so, at what value. How does the error curve look like? Let the pattern $(p, t)$ be defined by $p = (p_1, p_2) = (0.3, 0.7)$ and $t_\Omega = 0.4$. Randomly initalize the weights in the interval $[1; -1]$.

**Exercise 11.** A one-stage perceptron with two input neurons, bias neuron and binary threshold function as activation function divides the two-dimensional space into two regions by means of a straight line $g$. Analytically calculate a set of weight values for

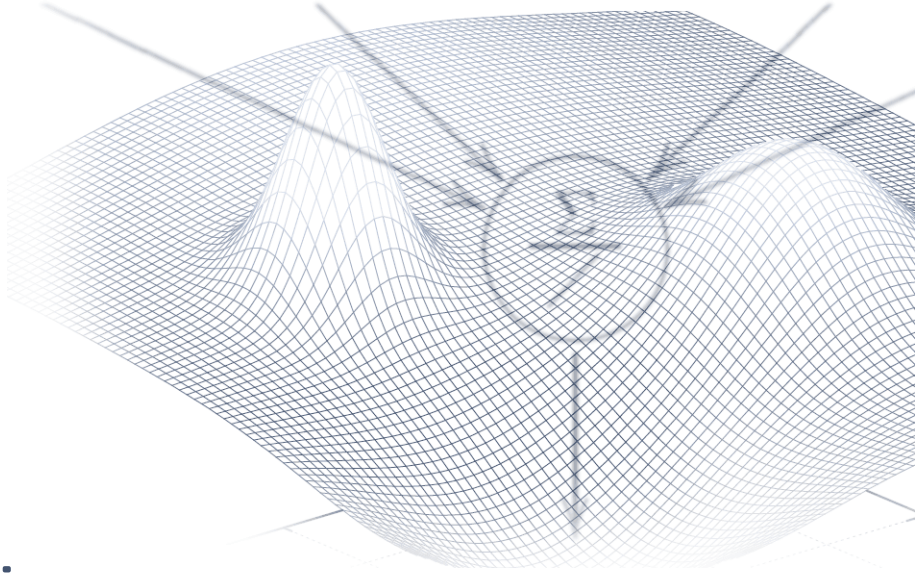
such a perceptron so that the following set $P$ of the 6 patterns of the form $(p_1, p_2, t_\Omega)$ with $\varepsilon \ll 1$ is correctly classified.

$$\begin{aligned} P = \{&(0, 0, -1); \\ &(2, -1, 1); \\ &(7 + \varepsilon, 3 - \varepsilon, 1); \\ &(7 - \varepsilon, 3 + \varepsilon, -1); \\ &(0, -2 - \varepsilon, 1); \\ &(0 - \varepsilon, -2, -1)\} \end{aligned}$$

**Exercise 12.** Calculate in a comprehensible way one vector $\Delta W$ of all changes in weight by means of the *backpropagation of error* procedure with $\eta = 1$. Let a 2-2-1 MLP with bias neuron be given and let the pattern be defined by

$$p = (p_1, p_2, t_\Omega) = (2, 0, 0.1).$$

For all weights with the target $\Omega$ the initial value of the weights should be 1. For all other weights the initial value should be 0.5. What is conspicuous about the changes?



# Chapter 6

# Radial basis functions

RBF networks approximate functions by stretching and compressing Gaussian bells and then summing them spatially shifted. Description of their functions and their learning process. Comparison with multilayer perceptrons.

According to Poggio and Girosi [PG89] radial basis function networks (RBF networks) are a paradigm of neural networks, which was developed considerably later than that of perceptrons. Like perceptrons, the RBF networks are built in layers. But in this case, they have exactly three layers, i.e. only one single layer of hidden neurons.

Like perceptrons, the networks have a feedforward structure and their layers are completely linked. Here, the input layer again does not participate in information processing. The RBF networks are - like MLPs - universal function approximators.

Despite all things in common: What is the difference between RBF networks and perceptrons? The difference lies in the information processing itself and in the computational rules within the neurons outside of the input layer. So, in a moment we will define a so far unknown type of neurons.

## 6.1 Components and structure of an RBF network

Initially, we want to discuss colloquially and then define some concepts concerning RBF networks.

**Output neurons:** In an RBF network the output neurons only contain the identity as activation function and one weighted sum as propagation function. Thus, they do little more than adding all input values and returning the sum.

**Hidden neurons** are also called RBF neurons (as well as the layer in which they are located is referred to as RBF layer). As propagation function, each hidden neuron calculates a norm that represents the distance between the input to the network and the so-called position of the neuron (center). This is inserted into a radial activation function which calculates and outputs the activation of the neuron.

**Definition 6.1** (RBF input neuron)**.** Definition and representation is identical to the definition 5.1 on page 84 of the input neuron.

**Definition 6.2** (Center of an RBF neuron)**.** The ***center*** $c_h$ of an RBF neuron $h$ is the point in the input space where the RBF neuron is located . In general, the closer the input vector is to the center vector of an RBF neuron, the higher is its activation.

**Definition 6.3** (RBF neuron)**.** The so-called ***RBF neurons*** $h$ have a propagation function $f_{\text{prop}}$ that determines the *distance* between the *center* $c_h$ of a neuron and the input vector $y$. This distance represents the network input. Then the network input is sent through a radial basis function $f_{\text{act}}$ which returns the activation or the output of the neuron. RBF neurons are represented by the symbol 

.

**Definition 6.4** (RBF output neuron)**.** ***RBF output neurons*** $\Omega$ use the weighted sum as propagation function $f_{\text{prop}}$, and the identity as activation function $f_{\text{act}}$. They are represented by the symbol 

.

**Definition 6.5** (RBF network)**.** An ***RBF network*** has exactly three layers in the following order: The input layer consisting of input neurons, the hidden layer (also called RBF layer) consisting of RBF neurons and the output layer consisting of RBF output neurons. Each layer is completely linked with the following one, shortcuts do not exist (fig. 6.1 on the next page) – it is a feedforward topology. The connections between input layer and RBF layer are unweighted, i.e. they only transmit the input. The connections between RBF layer and output layer are weighted. The original definition of an RBF network only referred to an output neuron, but – in analogy to the perceptrons – it is apparent that such a definition can be generalized. A bias neuron is not used in RBF networks. The set of input neurons shall be represented by $I$, the set of hidden neurons by $H$ and the set of output neurons by $O$.

Therefore, the inner neurons are called radial basis neurons because from their definition follows directly that all input vectors with the same distance from the center of a neuron also produce the same output value (fig. 6.2 on the facing page).

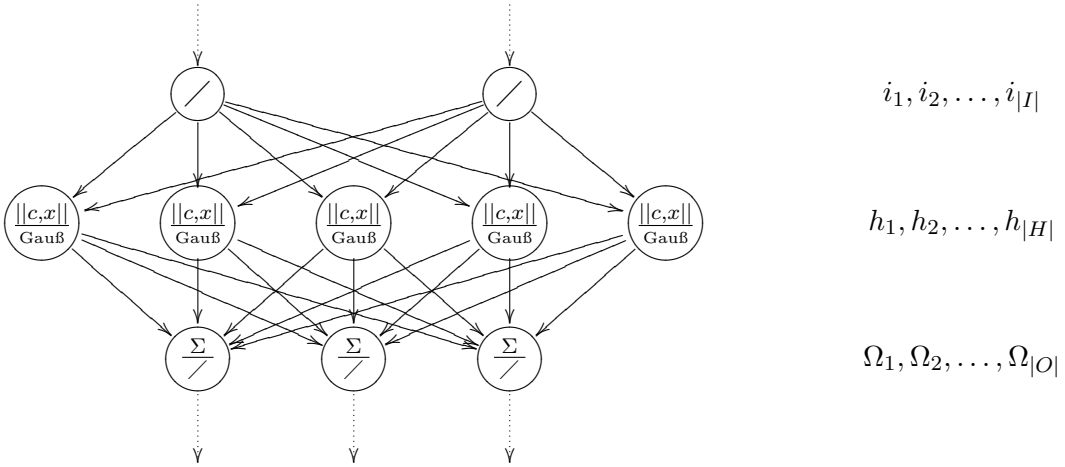



**Figure 6.1:** An exemplary RBF network with two input neurons, five hidden neurons and three output neurons. The connections to the hidden neurons are not weighted, they only transmit the input. Right of the illustration you can find the names of the neurons, which coincide with the names of the MLP neurons: Input neurons are called $i$, hidden neurons are called $h$ and output neurons are called $\Omega$. The associated sets are referred to as $I$, $H$ and $O$.

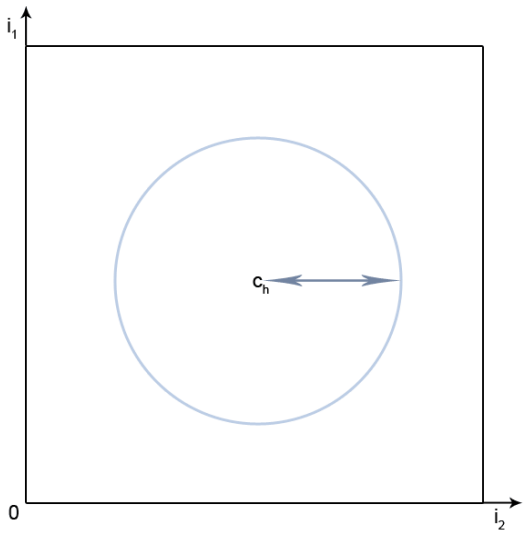



**Figure 6.2:** Let $c_h$ be the center of an RBF neuron $h$. Then the activation function $f_{\text{act}_h}$ is radially symmetric around $c_h$.

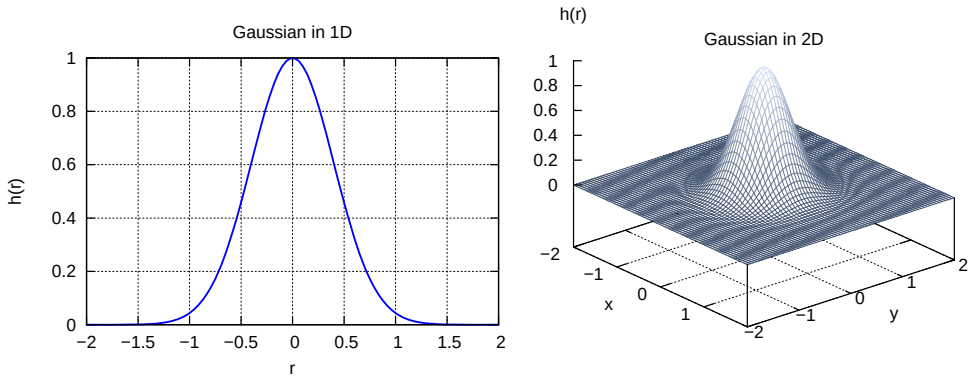



**Figure 6.3:** Two individual one- or two-dimensional Gaussian bells. In both cases $\sigma = 0.4$ holds and the centers of the Gaussian bells lie in the coordinate origin. The distance $r$ to the center $(0, 0)$ is simply calculated according to the Pythagorean theorem: $r = \sqrt{x^2 + y^2}$.

## 6.2 Information processing of an RBF network

Now the question is, what can be realized by such a network and what is its purpose. Let us go over the RBF network from top to bottom: An RBF network receives the input by means of the unweighted connections. Then the input vector is sent through a norm so that the result is a scalar. This scalar (which, by the way, can only be positive due to the norm) is processed by a radial basis function, for example by a Gaussian bell (fig. 6.3) .

The output values of the different neurons of the RBF layer or of the different Gaussian bells are added within the third layer: basically, in relation to the whole input space, Gaussian bells are added here.

Suppose that we have a second, a third and a fourth RBF neuron and therefore four differently located centers. Each of these neurons now measures another distance from the input to its own center and de facto provides different values, even if the Gaussian bell is the same. Since these values are finally simply accumulated in the output layer, one can easily see that any surface can be shaped by dragging, compressing and removing Gaussian bells and subsequently accumulating them. Here, the parameters for the superposition of the Gaussian bells are in the weights of the connections between the RBF layer and the output layer.

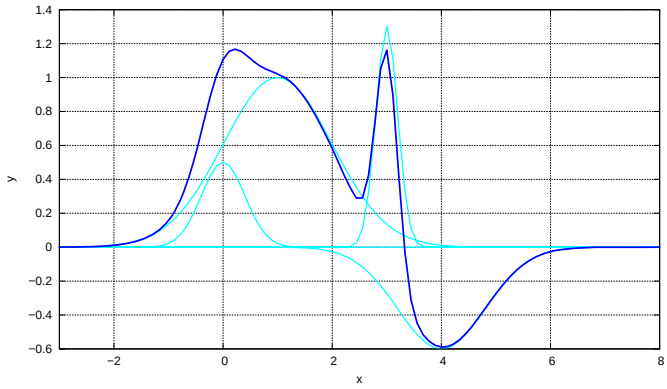



**Figure 6.4:** Four different Gaussian bells in one-dimensional space generated by means of RBF neurons are added by an output neuron of the RBF network. The Gaussian bells have different heights, widths and positions. Their centers $c_1, c_2, \ldots, c_4$ are located at $0, 1, 3, 4$, the widths $\sigma_1, \sigma_2, \ldots, \sigma_4$ at $0.4, 1, 0.2, 0.8$. You can see a two-dimensional example in fig. 6.5 on the following page.

Furthermore, the network architecture offers the possibility to freely define or train height and width of the Gaussian bells – due to which the network paradigm becomes even more versatile. We will get to know methods and approches for this later.

### 6.2.1 Information processing in RBF neurons

RBF neurons process information by using norms and radial basis functions

At first, let us take as an example a simple 1-4-1 RBF network. It is apparent that we will receive a one-dimensional output which can be represented as a function (fig. 6.4). Additionally, the network includes the centers $c_1, c_2, \ldots, c_4$ of the four inner neurons $h_1, h_2, \ldots, h_4$, and therefore it has Gaussian bells which are finally added within the output neuron $\Omega$. The network also possesses four values $\sigma_1, \sigma_2, \ldots, \sigma_4$ which influence the width of the Gaussian bells. On the contrary, the height of the Gaussian bell is influenced by the subsequent weights, since the individual output values of the bells are multiplied by those weights.

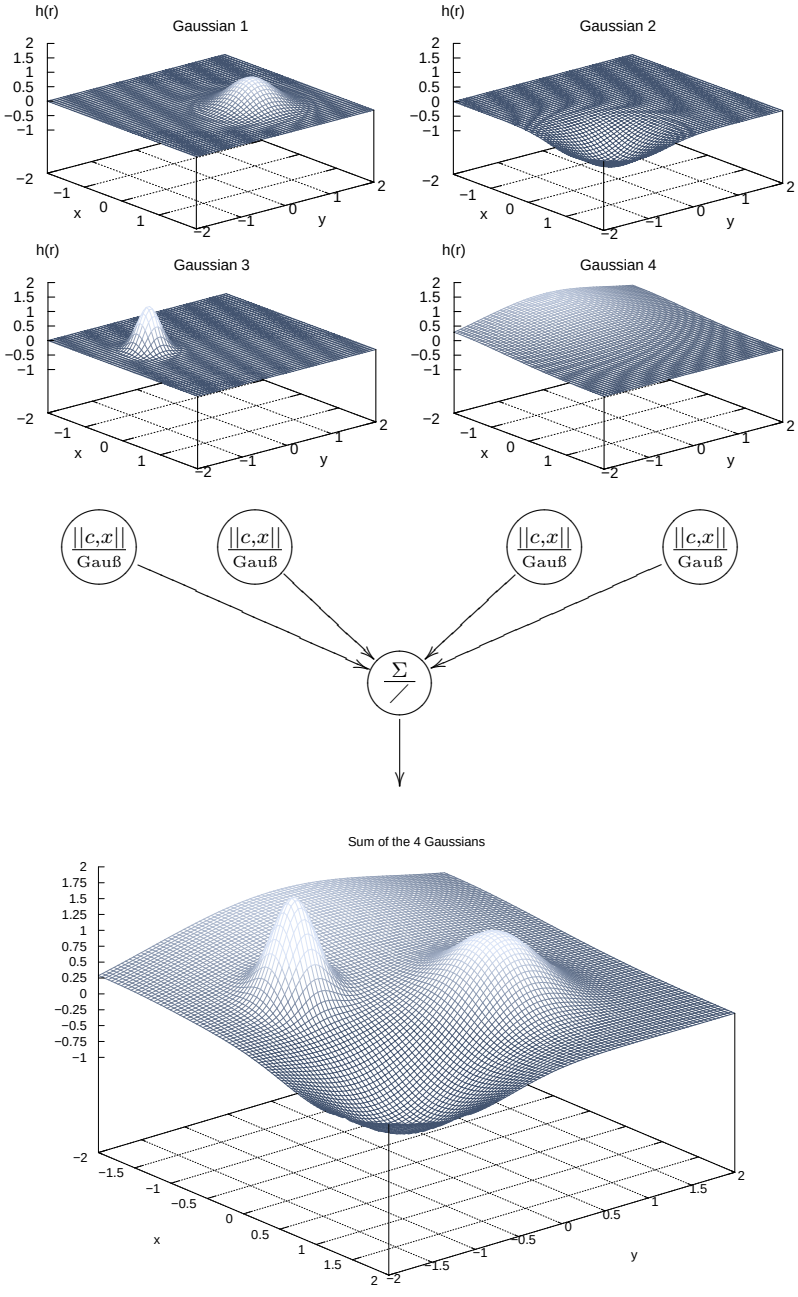



**Figure 6.5:** Four different Gaussian bells in two-dimensional space generated by means of RBF neurons are added by an output neuron of the RBF network. Once again $r = \sqrt{x^2 + y^2}$ applies for the distance. The heights $w$, widths $\sigma$ and centers $c = (x, y)$ are: $w_1 = 1, \sigma_1 = 0.4, c_1 = (0.5, 0.5)$, $w_2 = -1, \sigma_2 = 0.6, c_2 = (1.15, -1.15)$, $w_3 = 1.5, \sigma_3 = 0.2, c_3 = (-0.5, -1)$, $w_4 = 0.8, \sigma_4 = 1.4, c_4 = (-2, 0)$.

Since we use a *norm* to calculate the distance between the input vector and the center of a neuron $h$, we have different choices: Often the Euclidian norm is chosen to calculate the distance:

$$r_h = ||x - c_h|| \tag{6.1}$$

$$= \sqrt{\sum_{i \in I} (x_i - c_{h,i})^2} \tag{6.2}$$

Remember: The input vector was referred to as $x$. Here, the index $i$ runs through the input neurons and thereby through the input vector components and the neuron center components. As we can see, the Euclidean distance generates the squared differences of all vector components, adds them and extracts the root of the sum. In two-dimensional space this corresponds to the Pythagorean theorem. From the definition of a norm directly follows that the distance can only be positive. Strictly speaking, we hence only use the positive part of the activation function. By the way, activation functions other than the Gaussian bell are possible. Normally, functions that are monotonically decreasing over the interval $[0; \infty]$ are chosen.

Now that we know the ***distance*** $r_h$ between the input vector $x$ and the center $c_h$ of the RBF neuron $h$, this distance has to be passed through the activation function. Here we use, as already mentioned, a Gaussian bell:

$$f_{\text{act}}(r_h) = \mathrm{e}^{\left(\frac{-r_h^2}{2\sigma_h^2}\right)} \tag{6.3}$$

It is obvious that both the center $c_h$ and the width $\sigma_h$ can be seen as part of the activation function $f_{\text{act}}$, and hence the activation functions should not be referred to as $f_{\text{act}}$ simultaneously. One solution would be to number the activation functions like $f_{\text{act}1}, f_{\text{act}2}, \ldots, f_{\text{act}|H|}$ with $H$ being the set of hidden neurons. But as a result the explanation would be very confusing. So I simply use the name $f_{\text{act}}$ for all activation functions and regard $\sigma$ and $c$ as variables that are defined for individual neurons but no directly included in the activation function.

The reader will certainly notice that in the literature the Gaussian bell is often normalized by a multiplicative factor. We can, however, avoid this factor because we are multiplying anyway with the subsequent weights and consecutive multiplications, first by a normalization factor and then by the connections' weights, would only yield different factors there. We do not need this factor (especially because for our purpose the integral of the Gaussian bell must not always be 1) and therefore simply leave it out.

### 6.2.2 Some analytical thoughts prior to the training

The output $y_\Omega$ of an RBF output neuron $\Omega$ results from combining the functions of an RBF neuron to

$$y_\Omega = \sum_{h \in H} w_{h,\Omega} \cdot f_{\text{act}} \left( ||x - c_h|| \right). \tag{6.4}$$

Suppose that similar to the multilayer perceptron we have a set $P$, that contains $|P|$ training samples $(p, t)$. Then we obtain $|P|$ functions of the form

$$y_\Omega = \sum_{h \in H} w_{h,\Omega} \cdot f_{\text{act}} \left( ||p - c_h|| \right), \tag{6.5}$$

i.e. one function for each training sample.

Of course, with this effort we are aiming at letting the output $y$ for all training patterns $p$ converge to the corresponding teaching input $t$.

#### 6.2.2.1 Weights can simply be computed as solution of a system of equations

Thus, we have $|P|$ equations. Now let us assume that the widths $\sigma_1, \sigma_2, \ldots, \sigma_k$, the centers $c_1, c_2, \ldots, c_k$ and the training samples $p$ including the teaching input $t$ are given. We are looking for the weights $w_{h,\Omega}$ with $|H|$ weights for one output neuron $\Omega$. Thus, our problem can be seen as a system of equations since the only thing we want to change at the moment are the weights.

This demands a distinction of cases concerning the number of training samples $|P|$ and the number of RBF neurons $|H|$:

**$|P| = |H|$:** If the number of RBF neurons equals the number of patterns, i.e. $|P| = |H|$, the equation can be reduced to a matrix multiplication

$$T = M \cdot G \tag{6.6}$$
$$\Leftrightarrow \quad M^{-1} \cdot T = M^{-1} \cdot M \cdot G \tag{6.7}$$
$$\Leftrightarrow \quad M^{-1} \cdot T = E \cdot G \tag{6.8}$$
$$\Leftrightarrow \quad M^{-1} \cdot T = G, \tag{6.9}$$

where

- $T$ is the vector of the teaching inputs for all training samples,
- $M$ is the $|P| \times |H|$ matrix of the outputs of all $|H|$ RBF neurons to $|P|$ samples (remember: $|P| = |H|$, the matrix is squared and we can therefore attempt to invert it),
- $G$ is the vector of the desired weights and
- $E$ is a unit matrix with the same size as $G$.

Mathematically speaking, we can simply calculate the weights: In the case of $|P| = |H|$ there is exactly one RBF neuron available per training sample. This means, that the network exactly meets the $|P|$ existing nodes after having calculated the weights, i.e. it performs a ***precise interpolation***. To calculate such an equation we certainly do not need an RBF network, and therefore we can proceed to the next case.

Exact interpolation must not be mistaken for the memorizing ability mentioned with the MLPs: First, we are not talking about the training of RBF networks at the moment. Second, it could be advantageous for us and might in fact be intended if the network exactly interpolates between the nodes.

**$|P| < |H|$:** The system of equations is under-determined, there are more RBF neurons than training samples, i.e. $|P| < |H|$. Certainly, this case normally does not occur very often. In this case, there is a huge variety of solutions which we do not need in such detail. We can select one set of weights out of many obviously possible ones.

**$|P| > |H|$:** But most interesting for further discussion is the case if there are significantly more training samples than RBF neurons, that means $|P| > |H|$. Thus, we again want to use the generalization capability of the neural network.

If we have more training samples than RBF neurons, we cannot assume that every training sample is exactly hit. So, if we cannot exactly hit the points and therefore cannot just *interpolate* as in the aforementioned ideal case with $|P| = |H|$, we must try to find a function that ***approximates*** our training set $P$ as closely as possible: As with the MLP we try to reduce the sum of the squared error to a minimum.

How do we continue the calculation in the case of $|P| > |H|$? As above, to solve the system of equations, we have to find the solution $M$ of a matrix multiplication

$$T = M \cdot G. \tag{6.10}$$

The problem is that this time we cannot invert the $|P| \times |H|$ matrix $M$ because it is not a square matrix (here, $|P| \neq |H|$ is true). Here, we have to use the ***Moore-Penrose pseudo inverse*** $M^+$ which is defined by

$$M^+ = (M^T \cdot M)^{-1} \cdot M^T \tag{6.11}$$

Although the Moore-Penrose pseudo inverse is not the inverse of a matrix, it can be used similarly in this case[1]. We get equations that are very similar to those in the case of $|P| = |H|$:

$$T = M \cdot G \tag{6.12}$$

$$\Leftrightarrow \quad M^+ \cdot T = M^+ \cdot M \cdot G \tag{6.13}$$

$$\Leftrightarrow \quad M^+ \cdot T = E \cdot G \tag{6.14}$$

$$\Leftrightarrow \quad M^+ \cdot T = G \tag{6.15}$$

Another reason for the use of the Moore-Penrose pseudo inverse is the fact that it minimizes the squared error (which is our goal): The estimate of the vector $G$ in equation 6.15 corresponds to the ***Gauss-Markov model*** known from statistics, which is used to minimize the squared error. In the aforementioned equations 6.11 and the following ones please do not mistake the $T$ in $M^T$ (of the *transpose* of the matrix $M$) for the $T$ of the vector of all teaching inputs.

#### 6.2.2.2 The generalization on several outputs is trivial and not quite computationally expensive

We have found a mathematically exact way to directly calculate the weights. What will happen if there are several output neurons, i.e. $|O| > 1$, with $O$ being, as usual, the set of the output neurons $\Omega$? In this case, as we have already indicated, it does not change much: The additional output neurons have their own set of weights while we do not change the $\sigma$ and $c$ of the RBF layer. Thus, in an RBF network it is easy for given $\sigma$ and $c$ to realize a lot of output neurons since we only have to calculate the individual vector of weights

$$G_\Omega = M^+ \cdot T_\Omega \tag{6.16}$$

for every new output neuron $\Omega$, whereas the matrix $M^+$, which generally requires a lot of computational effort, always stays the same: So it is quite inexpensive – at least concerning the computational complexity – to add more output neurons.

1 Particularly, $M^+ = M^{-1}$ is true if $M$ is invertible. I do not want to go into detail of the reasons for these circumstances and applications of $M^+$ - they can easily be found in literature for linear algebra.

#### 6.2.2.3 Computational effort and accuracy

For realistic problems it normally applies that there are *considerably* more training samples than RBF neurons, i.e. $|P| \gg |H|$: You can, without any difficulty, use $10^6$ training samples, if you like. Theoretically, we could find the terms for the mathematically correct solution on the blackboard (after a very long time), but such calculations often seem to be imprecise and very time-consuming (matrix inversions require a lot of computational effort).

Furthermore, our Moore-Penrose pseudo-inverse is, in spite of numeric stability, no guarantee that the output vector corresponds to the teaching vector, because such extensive computations can be prone to many inaccuracies, even though the calculation is mathematically correct: Our computers can only provide us with (nonetheless very good) *approximations* of the pseudo-inverse matrices. This means that we also get only approximations of the correct weights (maybe with a lot of accumulated numerical errors) and therefore only an approximation (maybe very rough or even unrecognizable) of the desired output.

If we have enough computing power to analytically determine a weight vector, we should use it nevertheless only as an initial value for our learning process, which leads us to the real *training methods* – but otherwise it would be boring, wouldn't it?

## 6.3 Combinations of equation system and gradient strategies are useful for training

Analogous to the MLP we perform a gradient descent to find the suitable weights by means of the already well known *delta rule*. Here, backpropagation is unnecessary since we only have to train one single weight layer – which requires less computing time.

We know that the delta rule is

$$\Delta w_{h,\Omega} = \eta \cdot \delta_\Omega \cdot o_h, \tag{6.17}$$

in which we now insert as follows:

$$\Delta w_{h,\Omega} = \eta \cdot (t_\Omega - y_\Omega) \cdot f_{\text{act}}(||p - c_h||) \tag{6.18}$$

Here again I explicitly want to mention that it is very popular to divide the training into two phases by analytically computing a set of weights and then refining it by training with the delta rule.

There is still the question whether to learn offline or online. Here, the answer is similar to the answer for the multilayer perceptron: Initially, one often trains online (faster movement across the error surface). Then, after having approximated the solution, the errors are once again accumulated and, for a more precise approximation, one trains offline in a third learning phase. However, similar to the MLPs, you can be successful by using many methods.

As already indicated, in an RBF network not only the weights between the hidden and the output layer can be optimized. So let us now take a look at the possibility to vary $\sigma$ and $c$.

### 6.3.1 It is not always trivial to determine centers and widths of RBF neurons

It is obvious that the approximation accuracy of RBF networks can be increased by adapting the widths and positions of the Gaussian bells in the input space to the problem that needs to be approximated. There are several methods to deal with the centers $c$ and the widths $\sigma$ of the Gaussian bells:

**Fixed selection:** The centers and widths can be selected in a fixed manner and regardless of the training samples – this is what we have assumed until now.

**Conditional, fixed selection:** Again centers and widths are selected fixedly, but we have previous knowledge about the functions to be approximated and comply with it.

**Adaptive to the learning process:** This is definitely the most elegant variant, but certainly the most challenging one, too. A realization of this approach will not be discussed in this chapter but it can be found in connection with another network topology (section 10.6.1).

#### 6.3.1.1 Fixed selection

In any case, the goal is to cover the input space as evenly as possible. Here, widths of $\frac{2}{3}$ of the distance between the centers can be selected so that the Gaussian bells overlap by approx. "one third"[2] (fig. 6.6 on the next page). The closer the bells are set the more precise but the more time-consuming the whole thing becomes.

2 It is apparent that a Gaussian bell is mathematically infinitely wide, therefore I ask the reader to apologize this sloppy formulation.

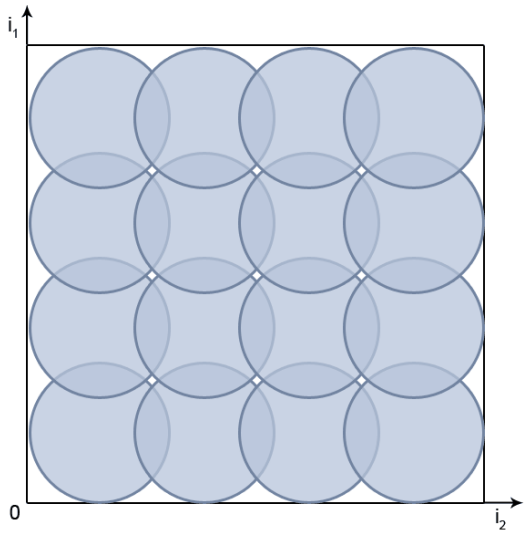



**Figure 6.6:** Example for an even coverage of a two-dimensional input space by applying radial basis functions.

This may seem to be very inelegant, but in the field of function approximation we cannot avoid even coverage. Here it is useless if the function to be approximated is precisely represented at some positions but at other positions the return value is only 0. However, the high input dimension requires a great many RBF neurons, which increases the computational effort exponentially with the dimension – and is responsible for the fact that six- to ten-dimensional problems in RBF networks are already called "high-dimensional" (an MLP, for example, does not cause any problems here).

#### 6.3.1.2 Conditional, fixed selection

Suppose that our training samples are not evenly distributed across the input space. It then seems obvious to arrange the centers and sigmas of the RBF neurons by means of the pattern distribution. So the training patterns can be analyzed by statistical techniques such as a *cluster analysis*, and so it can be determined whether there are statistical factors according to which we should distribute the centers and sigmas (fig. 6.7 on the following page).

A more trivial alternative would be to set $|H|$ centers on positions randomly selected from the set of patterns. So this method would allow for every training pattern $p$ to

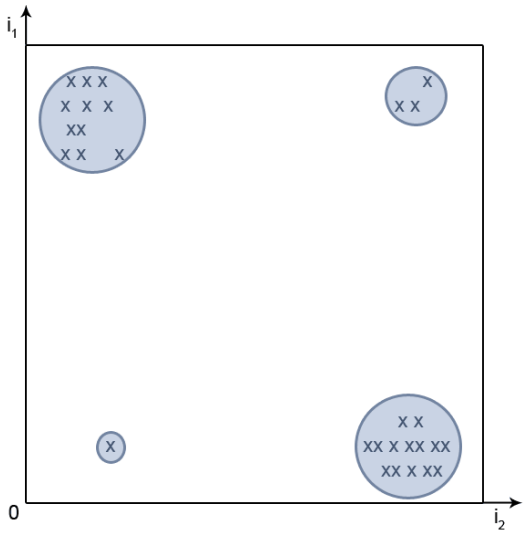



**Figure 6.7:** Example of an uneven coverage of a two-dimensional input space, of which we have previous knowledge, by applying radial basis functions.

be directly in the center of a neuron (fig. 6.8 on the next page). This is not yet very elegant but a good solution when time is an issue. Generally, for this method the widths are fixedly selected.

If we have reason to believe that the set of training samples is clustered, we can use clustering methods to determine them. There are different methods to determine clusters in an arbitrarily dimensional set of points. We will be introduced to some of them in excursus A. One neural clustering method are the so-called ROLFs (section A.5), and self-organizing maps are also useful in connection with determining the position of RBF neurons (section 10.6.1). Using ROLFs, one can also receive indicators for useful radii of the RBF neurons. *Learning vector quantisation* (chapter 9) has also provided good results. All these methods have nothing to do with the RBF networks themselves but are only used to generate some previous knowledge. Therefore we will not discuss them in this chapter but independently in the indicated chapters.

Another approach is to use the approved methods: We could slightly move the positions of the centers and observe how our error function Err is changing – a gradient descent,

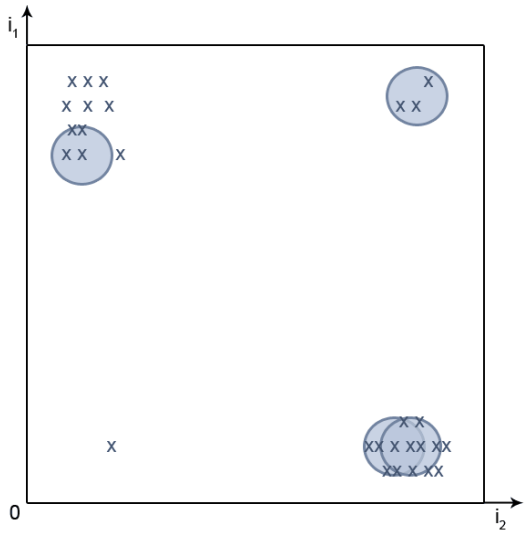



**Figure 6.8:** Example of an uneven coverage of a two-dimensional input space by applying radial basis functions. The widths were fixedly selected, the centers of the neurons were randomly distributed throughout the training patterns. This distribution can certainly lead to slightly unrepresentative results, which can be seen at the single data point down to the left.

as already known from the MLPs. In a similar manner we could look how the error depends on the values $\sigma$. Analogous to the derivation of backpropagation we derive

$$\frac{\partial \mathrm{Err}(\sigma_h c_h)}{\partial \sigma_h} \quad \text{and} \quad \frac{\partial \mathrm{Err}(\sigma_h c_h)}{\partial c_h}.$$

Since the derivation of these terms corresponds to the derivation of backpropagation we do not want to discuss it here.

But experience shows that no convincing results are obtained by regarding how the error behaves depending on the centers and sigmas. Even if mathematics claim that such methods are promising, the gradient descent, as we already know, leads to problems with very craggy error surfaces.

And that is the crucial point: Naturally, RBF networks generate *very* craggy error surfaces because, if we considerably change a $c$ or a $\sigma$, we will significantly change the appearance of the error function.

## 6.4 Growing RBF networks automatically adjust the neuron density

In ***growing RBF networks***, the number $|H|$ of RBF neurons is not constant. A certain number $|H|$ of neurons as well as their centers $c_h$ and widths $\sigma_h$ are previously selected (e.g. by means of a clustering method) and then extended or reduced. In the following text, only simple mechanisms are sketched. For more information, I refer to [Fri94].

### 6.4.1 Neurons are added to places with large error values

After generating this initial configuration the vector of the weights $G$ is analytically calculated. Then all specific errors $\mathrm{Err}_p$ concerning the set $P$ of the training samples are calculated and the maximum specific error

$$\max_P(\mathrm{Err}_p)$$

is sought.

The extension of the network is simple: We replace this maximum error with a new RBF neuron. Of course, we have to exercise care in doing this: IF the $\sigma$ are small, the neurons will only influence each other if the distance between them is short. But if the $\sigma$ are large, the already exisiting neurons are considerably influenced by the new neuron because of the overlapping of the Gaussian bells.

So it is obvious that we will adjust the already existing RBF neurons when adding the new neuron.

To put it simply, this adjustment is made by moving the centers $c$ of the other neurons away from the new neuron and reducing their width $\sigma$ a bit. Then the current output vector $y$ of the network is compared to the teaching input $t$ and the weight vector $G$ is improved by means of training. Subsequently, a new neuron can be inserted if necessary. This method is particularly suited for function approximations.

### 6.4.2 Limiting the number of neurons

Here it is mandatory to see that the network will not grow ad infinitum, which can happen very fast. Thus, it is very useful to previously define a maximum number for neurons $|H|_{\max}$.

### 6.4.3 Less important neurons are deleted

Which leads to the question whether it is possible to continue learning when this limit $|H|_{\max}$ is reached. The answer is: this would not stop learning. We only have to look for the "most unimportant" neuron and delete it. A neuron is, for example, unimportant for the network if there is another neuron that has a similar function: It often occurs that two Gaussian bells exactly overlap and at such a position, for instance, one single neuron with a higher Gaussian bell would be appropriate.

But to develop automated procedures in order to find less relevant neurons is highly problem dependent and we want to leave this to the programmer.

With RBF networks and multilayer perceptrons we have already become acquainted with and extensivley discussed two network paradigms for similar problems. Therefore we want to compare these two paradigms and look at their advantages and disadvantages.

## 6.5 Comparing RBF networks and multilayer perceptrons

We will compare multilayer perceptrons and RBF networks with respect to different aspects.

**Input dimension:** We must be careful with RBF networks in high-dimensional functional spaces since the network could very quickly require huge memory storage and computational effort. Here, a multilayer perceptron would cause less problems because its number of neuons does not grow exponentially with the input dimension.

**Center selection:** However, selecting the centers c for RBF networks is (despite the introduced approaches) still a major problem. Please use *any* previous knowledge you have when applying them. Such problems do not occur with the MLP.

**Output dimension:** The advantage of RBF networks is that the training is not much influenced when the output dimension of the network is high. For an MLP, a learning procedure such as backpropagation thereby will be very time-consuming.

**Extrapolation:** *Advantage as well as disadvantage* of RBF networks is the lack of extrapolation capability: An RBF network returns the result 0 far away from the centers of the RBF layer. On the one hand it does not extrapolate, unlike the MLP it cannot be used for extrapolation (whereby we could never know if the extrapolated values of the MLP are reasonable, but experience shows that

MLPs are suitable for that matter). On the other hand, unlike the MLP the network is capable to use this 0 to tell us "I don't know", which could be an advantage.

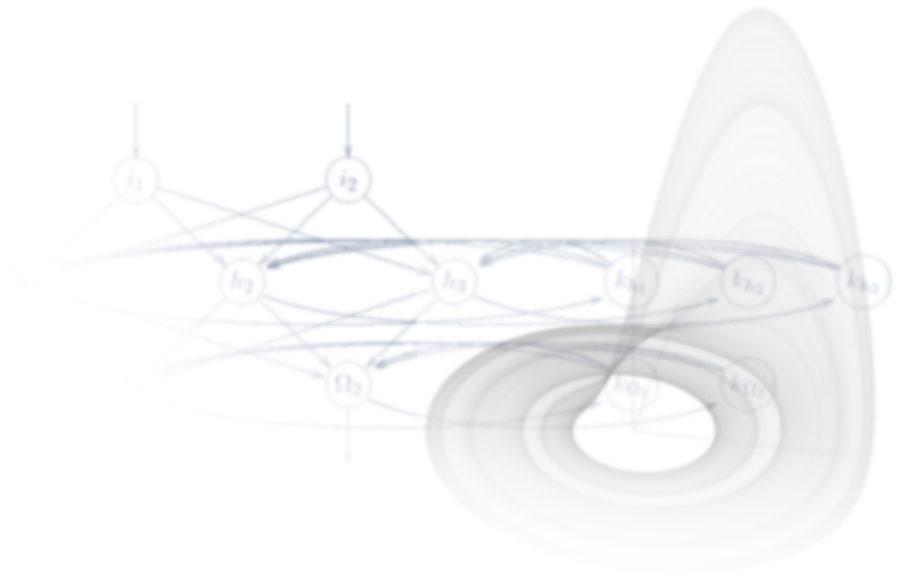
**Lesion tolerance:** For the output of an MLP, it is no so important if a weight or a neuron is missing. It will only worsen a little in total. If a weight or a neuron is missing in an RBF network then large parts of the output remain practically uninfluenced. But one part of the output is heavily affected because a Gaussian bell is directly missing. Thus, we can choose between a strong local error for lesion and a weak but global error.

**Spread:** Here the MLP is "advantaged" since RBF networks are used considerably less often – which is not always understood by professionals (at least as far as low-dimensional input spaces are concerned). The MLPs seem to have a considerably longer tradition and they are working too good to take the effort to read some pages of this work about RBF networks) :-).

## Exercises

**Exercise 13.** An $|I|$-$|H|$-$|O|$ RBF network with fixed widths and centers of the neurons should approximate a target function $u$. For this, $|P|$ training samples of the form $(p, t)$ of the function $u$ are given. Let $|P| > |H|$ be true. The weights should be analytically determined by means of the *Moore-Penrose pseudo inverse*. Indicate the running time behavior regarding $|P|$ and $|O|$ as precisely as possible.

Note: There are methods for matrix multiplications and matrix inversions that are more efficient than the canonical methods. For better estimations, I recommend to look for such methods (and their complexity). In addition to your complexity calculations, please indicate the used methods together with their complexity.



# Chapter 7

# Recurrent perceptron-like networks

Some thoughts about networks with internal states.

Generally, recurrent networks are networks that are capable of influencing themselves by means of ***recurrences***, e.g. by including the network output in the following computation steps. There are many types of recurrent networks of nearly arbitrary form, and nearly all of them are referred to as ***recurrent neural networks***. As a result, for the few paradigms introduced here I use the name ***recurrent multilayer perceptrons***.

Apparently, such a recurrent network is capable to compute more than the ordinary MLP: If the recurrent weights are set to 0, the recurrent network will be reduced to an ordinary MLP. Additionally, the recurrence generates different network-internal states so that different inputs can produce different outputs in the context of the network state.

Recurrent networks in themselves have a great dynamic that is mathematically difficult to conceive and has to be discussed extensively. The aim of this chapter is only to briefly discuss how recurrences can be structured and how network-internal states can be generated. Thus, I will briefly introduce two paradigms of recurrent networks and afterwards roughly outline their training.

With a recurrent network an input $x$ that is constant over time may lead to different results: On the one hand, the network could converge, i.e. it could transform itself into a fixed state and at some time return a fixed output value $y$. On the other hand, it could never converge, or at least not until a long time later, so that it can no longer be recognized, and as a consequence, $y$ constantly changes.

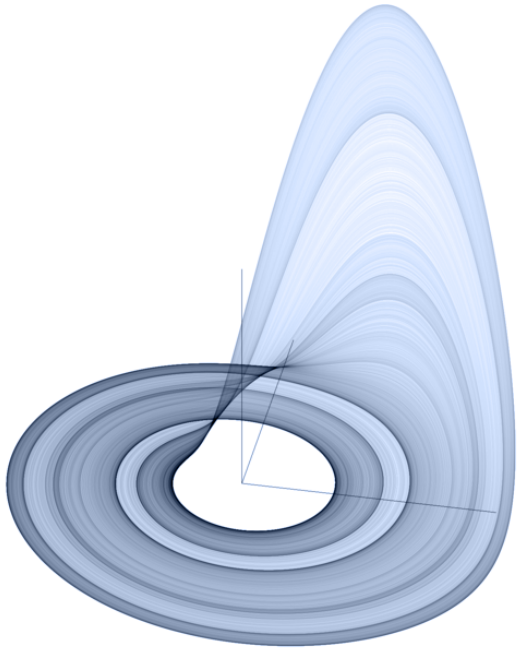


**Figure 7.1:** The Roessler attractor

If the network does not converge, it is, for example, possible to check if ***periodicals*** or ***attractors*** (fig. 7.1) are returned. Here, we can expect the complete variety of ***dynamical systems***. That is the reason why I particularly want to refer to the literature concerning dynamical systems.

Further discussions could reveal what will happen if the input of recurrent networks is changed.

In this chapter the related paradigms of recurrent networks according to Jordan and Elman will be introduced.

## 7.1 Jordan networks

A ***Jordan network*** [Jor86] is a multilayer perceptron with a set $K$ of so-called ***context neurons*** $k_1, k_2, \ldots, k_{|K|}$. There is one context neuron per output neuron (fig. 7.2 on the next page). In principle, a context neuron just memorizes an output until it can be

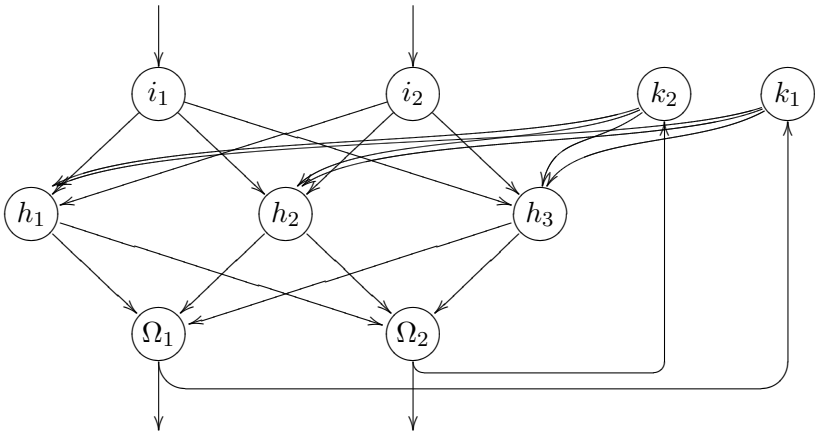



**Figure 7.2:** Illustration of a Jordan network. The network output is buffered in the context neurons and with the next time step it is entered into the network together with the new input.

processed in the next time step. Therefore, there are weighted connections between each output neuron and one context neuron. The stored values are returned to the actual network by means of complete links between the context neurons and the input layer.

In the originial definition of a Jordan network the context neurons are also recurrent to themselves via a connecting weight $\lambda$. But most applications omit this recurrence since the Jordan network is already very dynamic and difficult to analyze, even without these additional recurrences.

**Definition 7.1** (Context neuron)**.** A context neuron $k$ receives the output value of another neuron $i$ at a time $t$ and then reenters it into the network at a time $(t + 1)$.

**Definition 7.2** (Jordan network)**.** A Jordan network is a multilayer perceptron with one context neuron per output neuron. The set of context neurons is called $K$. The context neurons are completely linked toward the input layer of the network.

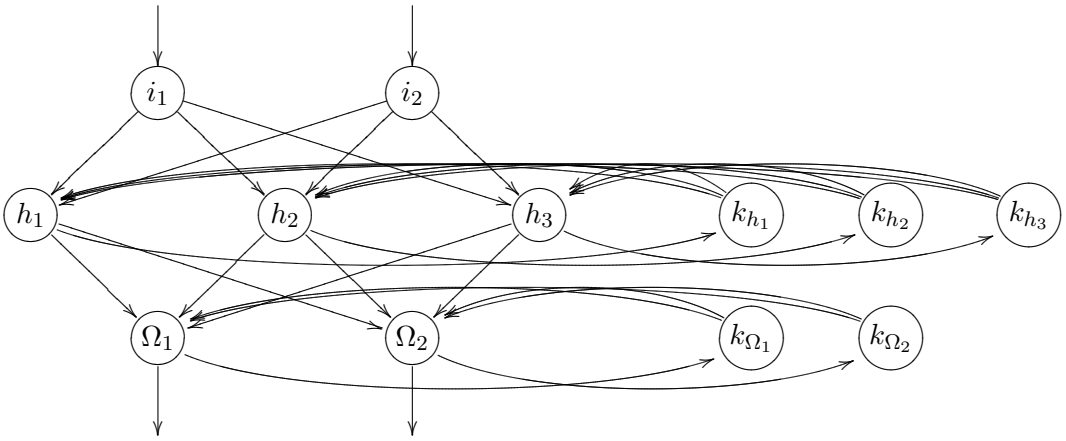



**Figure 7.3:** Illustration of an Elman network. The entire information processing part of the network exists, in a way, twice. The output of each neuron (except for the output of the input neurons) is buffered and reentered into the associated layer. For the reason of clarity I named the context neurons on the basis of their models in the actual network, but it is not mandatory to do so.

## 7.2 Elman networks

The ***Elman networks*** (a variation of the Jordan networks) [Elm90] have context neurons, too, but one layer of context neurons per information processing neuron layer (fig. 7.3). Thus, the outputs of each hidden neuron or output neuron are led into the associated context layer (again exactly one context neuron per neuron) and from there it is reentered into the complete neuron layer during the next time step (i.e. again a complete link on the way back). So the complete information processing part[1] of the MLP exists a second time as a "context version" – which once again considerably increases dynamics and state variety.

Compared with Jordan networks the Elman networks often have the advantage to act more purposeful since every layer can access its own context.

**Definition 7.3** (Elman network)**.** An Elman network is an MLP with one context neuron per information processing neuron. The set of context neurons is called $K$. This

1 Remember: The input layer does not process information.

means that there exists one context layer per information processing neuron layer with exactly the same number of context neurons. Every neuron has a weighted connection to exactly one context neuron while the context layer is completely linked towards its original layer.

Now it is interesting to take a look at the training of recurrent networks since, for instance, ordinary backpropagation of error cannot work on recurrent networks. Once again, the style of the following part is rather informal, which means that I will not use any formal definitions.

## 7.3 Training recurrent networks

In order to explain the training as comprehensible as possible, we have to agree on some simplifications that do not affect the learning principle itself.

So for the training let us assume that in the beginning the context neurons are initiated with an input, since otherwise they would have an undefined input (this is no simplification but reality).

Furthermore, we use a Jordan network without a hidden neuron layer for our training attempts so that the output neurons can directly provide input. This approach is a strong simplification because generally more complicated networks are used. But this does not change the learning principle.

### 7.3.1 Unfolding in time

Remember our actual learning procedure for MLPs, the *backpropagation of error*, which backpropagates the delta values. So, in case of recurrent networks the delta values would backpropagate cyclically through the network again and again, which makes the training more difficult. On the one hand we cannot know which of the many generated delta values for a weight should be selected for training, i.e. which values are useful. On the other hand we cannot definitely know when learning should be stopped. The advantage of recurrent networks are great state dynamics within the network; the disadvantage of recurrent networks is that these dynamics are also granted to the training and therefore make it difficult.

One learning approach would be the attempt to unfold the temporal states of the network (fig. 7.4 on page 149): Recursions are deleted by putting a similar network above the context neurons, i.e. the context neurons are, as a manner of speaking, the output

neurons of the attached network. More generally spoken, we have to backtrack the recurrences and place "'earlier"' instances of neurons in the network – thus creating a larger, but forward-oriented network without recurrences. This enables training a recurrent network with any training strategy developed for non-recurrent ones. Here, the input is entered as teaching input into every "copy" of the input neurons. This can be done for a discrete number of time steps. These training paradigms are called ***unfolding in time*** [MP69]. After the unfolding a training by means of backpropagation of error is possible.

But obviously, for one weight $w_{i,j}$ several changing values $\Delta w_{i,j}$ are received, which can be treated differently: accumulation, averaging etc. A simple accumulation could possibly result in enormous changes per weight if all changes have the same sign. Hence, also the average is not to be underestimated. We could also introduce a *discounting factor*, which weakens the influence of $\Delta w_{i,j}$ of the past.

Unfolding in time is particularly useful if we receive the impression that the closer past is more important for the network than the one being further away. The reason for this is that backpropagation has only little influence in the layers farther away from the output (remember: the farther we are from the output layer, the smaller the influence of backpropagation).

Disadvantages: the training of such an unfolded network will take a long time since a large number of layers could possibly be produced. A problem that is no longer negligible is the limited computational accuracy of ordinary computers, which is exhausted very fast because of so many nested computations (the farther we are from the output layer, the smaller the influence of backpropagation, so that this limit is reached). Furthermore, with several levels of context neurons this procedure could produce very large networks to be trained.

### 7.3.2 Teacher forcing

Other procedures are the equivalent ***teacher forcing*** and ***open loop learning***. They detach the recurrence during the learning process: We simply pretend that the recurrence does not exist and apply the teaching input to the context neurons during the training. So, backpropagation becomes possible, too. Disadvantage: with Elman networks a teaching input for non-output-neurons is not given.

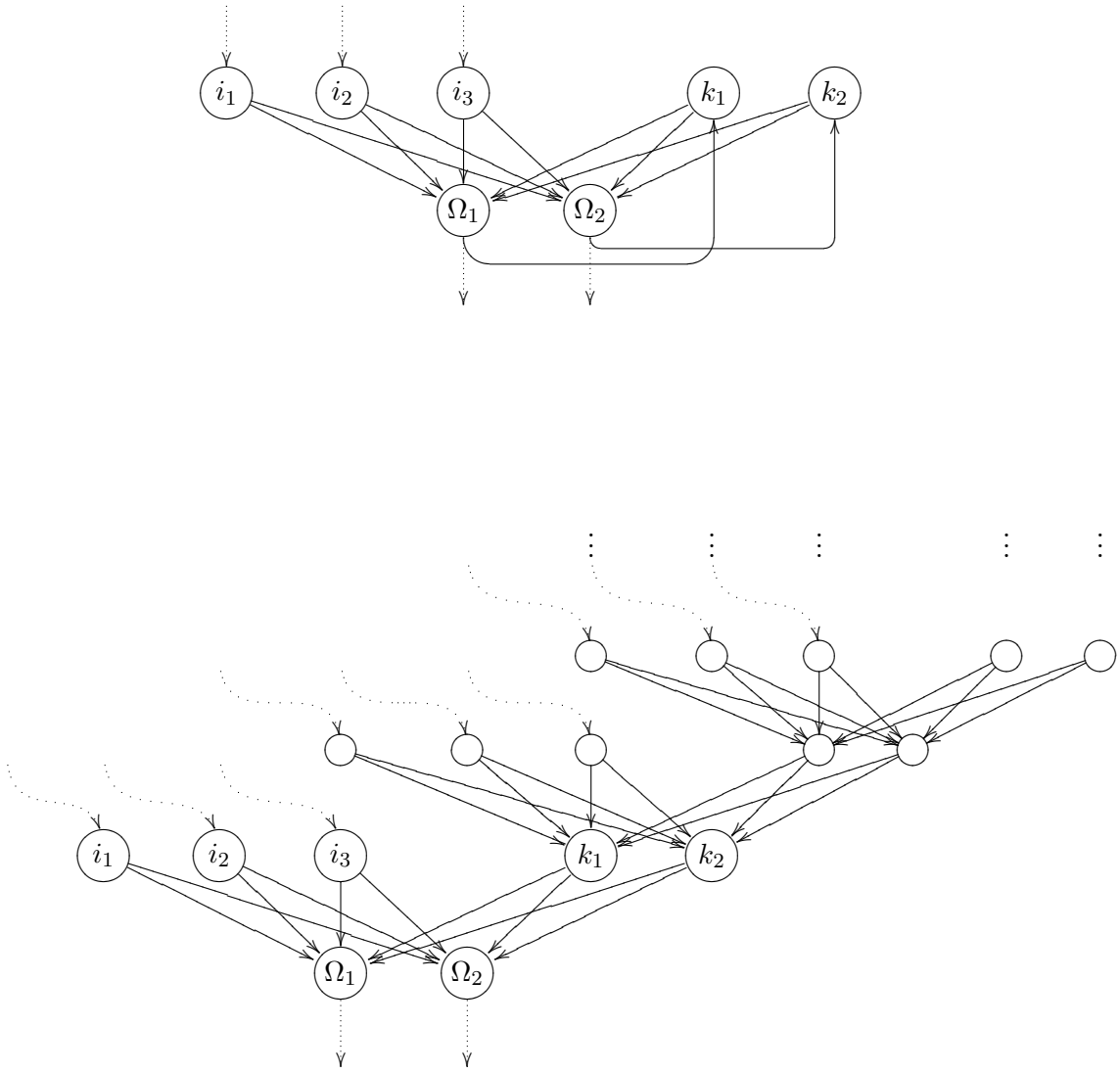



**Figure 7.4:** Illustration of the unfolding in time with a small exemplary recurrent MLP. **Top:** The recurrent MLP. **Bottom:** The unfolded network. For reasons of clarity, I only added names to the lowest part of the unfolded network. Dotted arrows leading into the network mark the inputs. Dotted arrows leading out of the network mark the outputs. Each "network copy" represents a time step of the network with the most recent time step being at the bottom.

### 7.3.3 Recurrent backpropagation

Another popular procedure without limited time horizon is the ***recurrent backpropagation*** using methods of differential calculus to solve the problem [Pin87].

### 7.3.4 Training with evolution

Due to the already long lasting training time, ***evolutionary algorithms*** have proved to be of value, especially with recurrent networks. One reason for this is that they are not only unrestricted with respect to recurrences but they also have other advantages when the mutation mechanisms are chosen suitably: So, for example, neurons and weights can be adjusted and the network topology can be optimized (of course the result of learning is not necessarily a Jordan or Elman network). With ordinary MLPs, however, evolutionary strategies are less popular since they certainly need a lot more time than a directed learning procedure such as backpropagation.

# Chapter 8

# Hopfield networks

In a magnetic field, each particle applies a force to any other particle so that all particles adjust their movements in the energetically most favorable way. This natural mechanism is copied to adjust noisy inputs in order to match their real models.

---

Another supervised learning example of the wide range of neural networks was developed by John Hopfield: the so-called ***Hopfield networks*** [Hop82]. Hopfield and his physically motivated networks have contributed a lot to the renaissance of neural networks.

## 8.1 Hopfield networks are inspired by particles in a magnetic field

The idea for the Hopfield networks originated from the behavior of particles in a magnetic field: Every particle "communicates" (by means of magnetic forces) with every other particle (completely linked) with each particle trying to reach an energetically favorable state (i.e. *a minimum of the energy function*). As for the neurons this state is known as activation. Thus, all particles or neurons rotate and thereby encourage each other to continue this rotation. As a manner of speaking, our neural network is a cloud of particles

Based on the fact that the particles automatically detect the minima of the energy function, Hopfield had the idea to use the "spin" of the particles to process information: Why not letting the particles search minima on arbitrary functions? Even if we only use two of those ***spins***, i.e. a *binary activation*, we will recognize that the developed Hopfield network shows considerable dynamics.

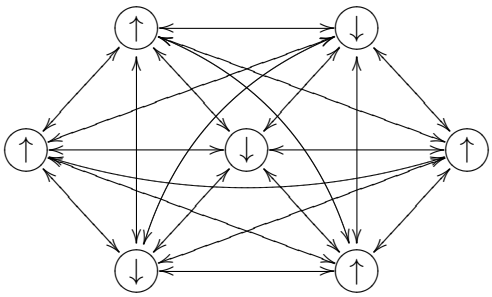


**Figure 8.1:** Illustration of an exemplary Hopfield network. The arrows ↑ and ↓ mark the binary "spin". Due to the completely linked neurons the layers cannot be separated, which means that a Hopfield network simply includes a set of neurons.

## 8.2 In a hopfield network, all neurons influence each other symmetrically

Briefly speaking, a Hopfield network consists of a set $K$ of completely linked neurons with binary activation (since we only use two spins), with the weights being symmetric between the individual neurons and without any neuron being *directly connected* to itself (fig. 8.1). Thus, the *state* of $|K|$ neurons with two possible states $\in \{-1, 1\}$ can be described by a string $x \in \{-1, 1\}^{|K|}$.

The complete link provides a full square matrix of weights between the neurons. The meaning of the weights will be discussed in the following. Furthermore, we will soon recognize according to which rules the neurons are spinning, i.e. are changing their state.

Additionally, the complete link leads to the fact that we do not know any input, output or hidden neurons. Thus, we have to think about how we can input something into the $|K|$ neurons.

**Definition 8.1** (Hopfield network)**.** A Hopfield network consists of a set $K$ of completely linked neurons without direct recurrences. The activation function of the neurons is the binary threshold function with outputs $\in \{1, -1\}$.

**Definition 8.2** (State of a Hopfield network)**.** The state of the network consists of the activation states of all neurons. Thus, the state of the network can be understood as a binary string $z \in \{-1, 1\}^{|K|}$.

### 8.2.1 Input and output of a Hopfield network are represented by neuron states

We have learned that a network, i.e. a set of $|K|$ particles, that is in a state is automatically looking for a minimum. An input pattern of a Hopfield network is exactly such a state: A binary string $x \in \{-1, 1\}^{|K|}$ that initializes the neurons. Then the network is looking for the minimum to be taken (which we have previously defined by the input of training samples) on its energy surface.

But when do we know that the minimum has been found? This is simple, too: when the network stops. It can be proven that a Hopfield network with a symmetric weight matrix that has zeros on its diagonal *always converges* [CG88] , i.e. at some point it will stand still. Then the output is a binary string $y \in \{-1, 1\}^{|K|}$, namely the state string of the network that has found a minimum.

Now let us take a closer look at the contents of the weight matrix and the rules for the state change of the neurons.

**Definition 8.3** (Input and output of a Hopfield network)**.** The input of a Hopfield network is binary string $x \in \{-1, 1\}^{|K|}$ that initializes the state of the network. After the convergence of the network, the output is the binary string $y \in \{-1, 1\}^{|K|}$ generated from the new network state.

### 8.2.2 Significance of weights

We have already said that the neurons change their states, i.e. their direction, from $-1$ to 1 or vice versa. These spins occur dependent on the current states of the other neurons and the associated weights. Thus, the weights are capable to control the complete change of the network. The weights can be positive, negative, or 0. Colloquially speaking, for a weight $w_{i,j}$ between two neurons $i$ and $j$ the following holds:

**If $w_{i,j}$ is positive,** it will try to force the two neurons to become equal – the larger they are, the harder the network will try. If the neuron $i$ is in state 1 and the neuron $j$ is in state $-1$, a high positive weight will advise the two neurons that it is energetically more favorable to be equal.

**If $w_{i,j}$ is negative,** its behavior will be analoguous only that $i$ and $j$ are urged to be different. A neuron $i$ in state $-1$ would try to urge a neuron $j$ into state 1.

**Zero weights** lead to the two involved neurons not influencing each other.

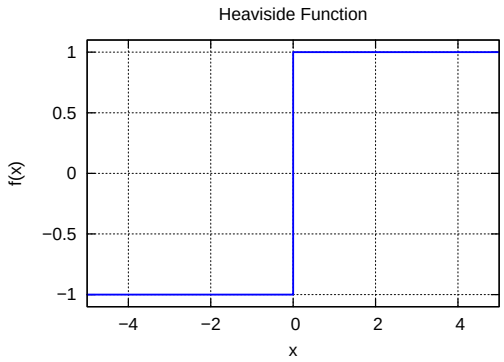



**Figure 8.2:** Illustration of the binary threshold function.

The weights as a whole apparently take the way from the current state of the network towards the next minimum of the energy function. We now want to discuss how the neurons follow this way.

### 8.2.3 A neuron changes its state according to the influence of the other neurons

Once a network has been trained and initialized with some starting state, the change of state $x_k$ of the individual neurons $k$ occurs according to the scheme

$$x_k(t) = f_{\text{act}}\left(\sum_{j \in K} w_{j,k} \cdot x_j(t-1)\right) \tag{8.1}$$

in each time step, where the function $f_{\text{act}}$ generally is the binary threshold function (fig. 8.2) with threshold 0. Colloquially speaking: a neuron $k$ calculates the sum of $w_{j,k} \cdot x_j(t-1)$, which indicates how strong and into which direction the neuron $k$ is forced by the other neurons $j$. Thus, the new state of the network (time $t$) results from the state of the network at the previous time $t-1$. This sum is the direction into which the neuron $k$ is pushed. Depending on the sign of the sum the neuron takes state 1 or $-1$.

Another difference between Hopfield networks and other already known network topologies is the *asynchronous update*: A neuron $k$ is randomly chosen every time, which then

recalculates the activation. Thus, the new activation of the previously changed neurons immediately influences the network, i.e. one time step indicates the change of a single neuron.

Regardless of the aforementioned random selection of the neuron, a Hopfield network is often much easier to implement: The neurons are simply processed one after the other and their activations are recalculated until no more changes occur.

**Definition 8.4** (Change in the state of a Hopfield network)**.** The change of state of the neurons occurs asynchronously with the neuron to be updated being randomly chosen and the new state being generated by means of this rule:

$$x_k(t) = f_{\text{act}}\left(\sum_{j \in J} w_{j,k} \cdot x_j(t-1)\right).$$

Now that we know how the weights influence the changes in the states of the neurons and force the entire network towards a minimum, then there is the question of how to teach the weights to force the network towards a *certain* minimum.

## 8.3 The weight matrix is generated directly out of the training patterns

The aim is to generate minima on the mentioned energy surface, so that at an input the network can converge to them. As with many other network paradigms, we use a set $P$ of training patterns $p \in \{1, -1\}^{|K|}$, representing the minima of our energy surface.

Unlike many other network paradigms, we do not look for the minima of an unknown error function but define minima on such a function. The purpose is that the network shall automatically take the closest minimum when the input is presented. For now this seems unusual, but we will understand the whole purpose later.

Roughly speaking, the training of a Hopfield network is done by training each training pattern *exactly once* using the rule described in the following (***Single Shot Learning***), where $p_i$ and $p_j$ are the states of the neurons $i$ and $j$ under $p \in P$:

$$w_{i,j} = \sum_{p \in P} p_i \cdot p_j \tag{8.2}$$

This results in the weight matrix $W$. Colloquially speaking: We initialize the network by means of a training pattern and then process weights $w_{i,j}$ one after another. For each of these weights we verify: Are the neurons $i, j$ n the same state or do the states vary? In the first case we add 1 to the weight, in the second case we add $-1$.

This we repeat for each training pattern $p \in P$. Finally, the values of the weights $w_{i,j}$ are high when $i$ and $j$ corresponded with many training patterns. Colloquially speaking, this high value tells the neurons: "Often, it is energetically favorable to hold the same state". The same applies to negative weights.

Due to this training we can store a certain fixed number of patterns $p$ in the weight matrix. At an input $x$ the network will converge to the stored pattern that is closest to the input $p$.

Unfortunately, the number of the maximum storable and reconstructible patterns $p$ is limited to

$$|P|_{\text{MAX}} \approx 0.139 \cdot |K|, \tag{8.3}$$

which in turn only applies to orthogonal patterns. This was shown by precise (and time-consuming) mathematical analyses, which we do not want to specify now. If more patterns are entered, already stored information will be destroyed.

**Definition 8.5** (Learning rule for Hopfield networks)**.** The individual elements of the weight matrix $W$ are defined by a single processing of the learning rule

$$w_{i,j} = \sum_{p \in P} p_i \cdot p_j,$$

where the diagonal of the matrix is covered with zeros. Here, no more than $|P|_{\text{MAX}} \approx 0.139 \cdot |K|$ training samples can be trained and at the same time maintain their function.

Now we know the functionality of Hopfield networks but nothing about their practical use.

## 8.4 Autoassociation and traditional application

Hopfield networks, like those mentioned above, are called ***autoassociators***. An autoassociator $a$ exactly shows the aforementioned behavior: Firstly, when a known pattern $p$ is entered, exactly this known pattern is returned. Thus,

$$a(p) = p,$$

with $a$ being the associative mapping. Secondly, and that is the practical use, this also works with inputs that are close to a pattern:

$$a(p + \varepsilon) = p.$$

Afterwards, the autoassociator is, in any case, in a stable state, namely in the state $p$.

If the set of patterns $P$ consists of, for example, letters or other characters in the form of pixels, the network will be able to correctly recognize deformed or noisy letters with high probability (fig. 8.3 on the following page).

The primary fields of application of Hopfield networks are ***pattern recognition*** and pattern completion, such as the zip code recognition on letters in the eighties. But soon the Hopfield networks were replaced by other systems in most of their fields of application, for example by OCR systems in the field of letter recognition. Today Hopfield networks are virtually no longer used, they have not become established in practice.

## 8.5 Heteroassociation and analogies to neural data storage

So far we have been introduced to Hopfield networks that converge from an arbitrary input into the closest minimum of a static energy surface.

Another variant is a dynamic energy surface: Here, the appearance of the energy surface depends on the current state and we receive a ***heteroassociator*** instead of an autoassociator. For a heteroassociator

$$a(p + \varepsilon) = p$$

is no longer true, but rather

$$h(p + \varepsilon) = q,$$

which means that a pattern is mapped onto another one. $h$ is the heteroassociative mapping. Such heteroassociations are achieved by means of an asymmetric weight matrix $V$.

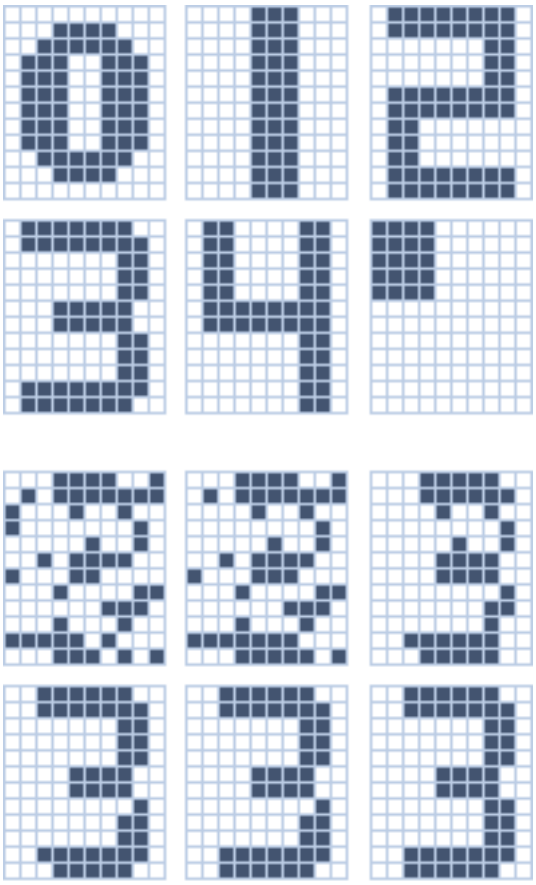


**Figure 8.3:** Illustration of the convergence of an exemplary Hopfield network. Each of the pictures has $10 \times 12 = 120$ binary pixels. In the Hopfield network each pixel corresponds to one neuron. The upper illustration shows the training samples, the lower shows the convergence of a heavily noisy 3 to the corresponding training sample.

Heteroassociations connected in series of the form

$$\begin{aligned} h(p+\varepsilon) &= q \\ h(q+\varepsilon) &= r \\ h(r+\varepsilon) &= s \\ &\vdots \\ h(z+\varepsilon) &= p \end{aligned}$$

can provoke a fast cycle of states

$$p \to q \to r \to s \to \ldots \to z \to p,$$

whereby a single pattern is never completely accepted: Before a pattern is entirely completed, the heteroassociation already tries to generate the successor of this pattern. Additionally, the network would never stop, since after having reached the last state $z$, it would proceed to the first state $p$ again.

### 8.5.1 Generating the heteroassociative matrix

We generate the matrix $V$ by means of elements $v$ very similar to the autoassociative matrix with $p$ being (per transition) the training sample before the transition and $q$ being the training sample to be generated from $p$:

$$v_{i,j} = \sum_{p,q \in P, p \neq q} p_i q_j \tag{8.4}$$

The diagonal of the matrix is again filled with zeros. The neuron states are, as always, adapted during operation. Several transitions can be introduced into the matrix by a simple addition, whereby the said limitation exists here, too.

**Definition 8.6** (Learning rule for the heteroassociative matrix)**.** For two training samples $p$ being predecessor and $q$ being successor of a heteroassociative transition the weights of the heteroassociative matrix $V$ result from the learning rule

$$v_{i,j} = \sum_{p,q \in P, p \neq q} p_i q_j,$$

with several heteroassociations being introduced into the network by a simple addition.

### 8.5.2 Stabilizing the heteroassociations

We have already mentioned the problem that the patterns are not completely generated but that the next pattern is already beginning before the generation of the previous pattern is finished.

This problem can be avoided by not only influencing the network by means of the heteroassociative matrix $V$ but also by the already known autoassociative matrix $W$.

Additionally, the neuron adaptation rule is changed so that competing terms are generated: One term autoassociating an existing pattern and one term trying to convert the very same pattern into its successor. The associative rule provokes that the network stabilizes a pattern, remains there for a while, goes on to the next pattern, and so on.

$$x_i(t+1) = \tag{8.5}$$
$$f_{\text{act}}\left(\underbrace{\sum_{j\in K} w_{i,j}x_j(t)}_{\text{autoassociation}} + \underbrace{\sum_{k\in K} v_{i,k}x_k(t-\Delta t)}_{\text{heteroassociation}}\right)$$

Here, the value $\Delta t$ causes, descriptively speaking, the influence of the matrix $V$ to be delayed, since it only refers to a network being $\Delta t$ versions behind. The result is a change in state, during which the individual states are stable for a short while. If $\Delta t$ is set to, for example, twenty steps, then the asymmetric weight matrix will realize any change in the network only twenty steps later so that it initially works with the autoassociative matrix (since it still perceives the predecessor pattern of the current one), and only after that it will work against it.

### 8.5.3 Biological motivation of heterassociation

From a biological point of view the transition of stable states into other stable states is highly motivated: At least in the beginning of the nineties it was assumed that the Hopfield modell will achieve an approximation of the state dynamics in the brain, which realizes much by means of state chains: When I would ask you, dear reader, to recite the alphabet, you generally will manage this better than (please try it immediately) to answer the following question:

*Which letter in the alphabet follows the letter P?*

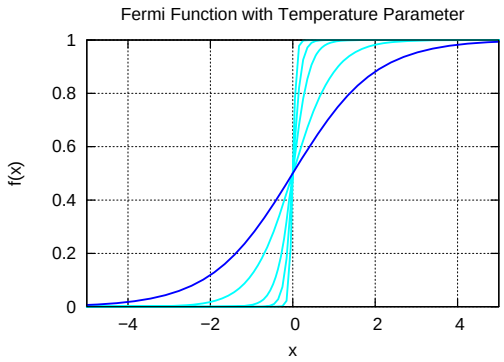



**Figure 8.4:** The already known Fermi function with different temperature parameter variations.

Another example is the phenomenon that one cannot remember a situation, but the place at which one memorized it the last time is perfectly known. If one returns to this place, the forgotten situation often comes back to mind.

## 8.6 Continuous Hopfield networks

So far, we only have discussed Hopfield networks with binary activations. But Hopfield also described a version of his networks with continuous activations [Hop84], which we want to cover at least briefly: ***continuous Hopfield networks***. Here, the activation is no longer calculated by the binary threshold function but by the Fermi function with temperature parameters (fig. 8.4).

Here, the network is stable for symmetric weight matrices with zeros on the diagonal, too.

Hopfield also stated, that continuous Hopfield networks can be applied to find acceptable solutions for the NP-hard travelling salesman problem [HT85]. According to some verification trials [Zel94] this statement can't be kept up any more. But today there are faster algorithms for handling this problem and therefore the Hopfield network is no longer used here.

## Exercises

**Exercise 14.** Indicate the storage requirements for a Hopfield network with $|K| = 1000$ neurons when the weights $w_{i,j}$ shall be stored as integers. Is it possible to limit the value range of the weights in order to save storage space?

**Exercise 15.** Compute the weights $w_{i,j}$ for a Hopfield network using the training set

$$\begin{aligned} P = \{&(-1,-1,-1,-1,-1,1);\\ &(-1,1,1,-1,-1,-1);\\ &(1,-1,-1,1,-1,1)\}. \end{aligned}$$

# Chapter 9

# Learning vector quantization

Learning Vector Quantization is a learning procedure with the aim to represent the vector training sets divided into predefined classes as well as possible by using a few representative vectors. If this has been managed, vectors which were unkown until then could easily be assigned to one of these classes.

Slowly, part II of this text is nearing its end – and therefore I want to write a last chapter for this part that will be a smooth transition into the next one: A chapter about the ***learning vector quantization*** (abbreviated ***LVQ***) [Koh89] described by Teuvo Kohonen, which can be characterized as being related to the *self organizing feature maps*. These SOMs are described in the next chapter that already belongs to part III of this text, since SOMs learn unsupervised. Thus, after the exploration of LVQ I want to bid farewell to supervised learning.

Previously, I want to announce that there are different variations of LVQ, which will be mentioned but not exactly represented. The goal of this chapter is rather to analyze the underlying principle.

## 9.1 About quantization

In order to explore the *learning vector quantization* we should at first get a clearer picture of what ***quantization*** (which can also be referred to as ***discretization***) is.

Everybody knows the sequence of discrete numbers

$$\mathbb{N} = \{1, 2, 3, \ldots\},$$

which contains the natural numbers. ***Discrete*** means, that this sequence consists of *separated* elements that are not interconnected. The elements of our example are exactly such numbers, because the natural numbers do not include, for example, numbers between 1 and 2. On the other hand, the sequence of real numbers $\mathbb{R}$, for instance, is ***continuous***: It does not matter how close two selected numbers are, there will always be a number between them.

Quantization means that a continuous space is divided into discrete sections: By deleting, for example, all decimal places of the real number 2.71828, it could be assigned to the natural number 2. Here it is obvious that any other number having a 2 in front of the comma would also be assigned to the natural number 2, i.e. 2 would be some kind of *representative* for all real numbers within the interval $[2; 3)$.

It must be noted that a sequence can be irregularly quantized, too: For instance, the timeline for a week could be quantized into working days and weekend.

A special case of quantization is ***digitization***: In case of digitization we always talk about *regular* quantization of a continuous space into a number system with respect to a certain ***basis***. If we enter, for example, some numbers into the computer, these numbers will be digitized into the binary system (basis 2).

**Definition 9.1** (Quantization)**.** Separation of a continuous space into discrete sections.

**Definition 9.2** (Digitization)**.** Regular quantization.

## 9.2 LVQ divides the input space into separate areas

Now it is almost possible to describe by means of its name what LVQ should enable us to do: A set of representatives should be used to divide an input space into classes that reflect the input space as well as possible (fig. 9.1 on the facing page). Thus, each element of the input space should be assigned to a vector as a representative, i.e. to a class, where the set of these representatives should represent the entire input space as precisely as possible. Such a vector is called ***codebook vector***. A codebook vector is the representative of exactly those input space vectors lying closest to it, which divides the input space into the said discrete areas.

It is to be emphasized that we have to know in advance how many classes we have and which training sample belongs to which class. Furthermore, it is important that the classes must not be disjoint, which means they may overlap.

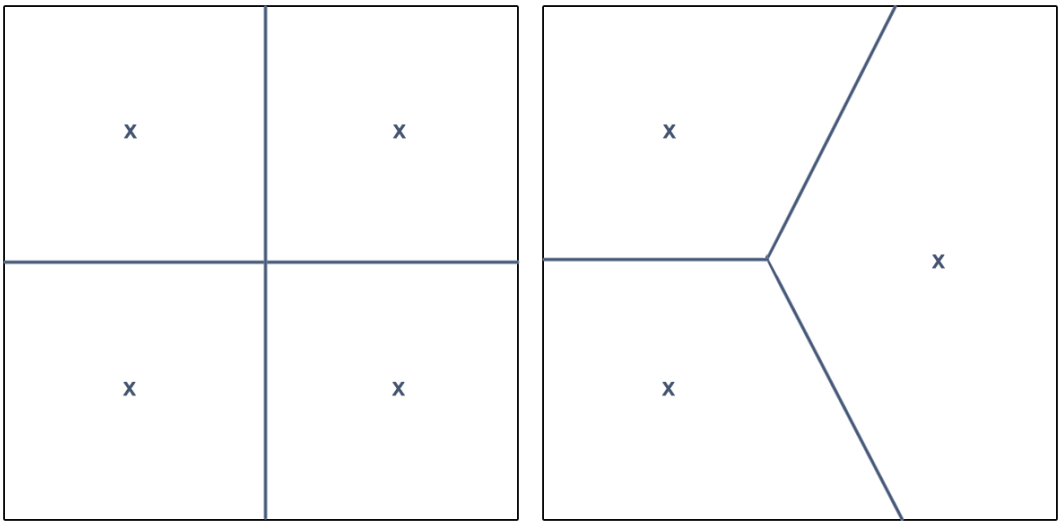


**Figure 9.1:** BExamples for quantization of a two-dimensional input space. DThe lines represent the class limit, the $\times$ mark the codebook vectors.

Such separation of data into classes is interesting for many problems for which it is useful to explore only some characteristic representatives instead of the possibly huge set of all vectors – be it because it is less time-consuming or because it is sufficiently precise.

## 9.3 Using codebook vectors: the nearest one is the winner

The use of a prepared set of codebook vectors is very simple: For an input vector $y$ the class association is easily decided by considering which codebook vector is the closest – so, the codebook vectors build a ***voronoi diagram*** out of the set. Since each codebook vector can clearly be associated to a class, each input vector is associated to a class, too.

## 9.4 Adjusting codebook vectors

As we have already indicated, the LVQ is a supervised learning procedure. Thus, we have a teaching input that tells the learning procedure whether the classification of the input pattern is right or wrong: In other words, we have to know in advance the number of classes to be represented or the number of codebook vectors.

Roughly speaking, it is the aim of the learning procedure that training samples are used to cause a previously defined number of randomly initialized codebook vectors to reflect the training data as precisely as possible.

### 9.4.1 The procedure of learning

Learning works according to a simple scheme. We have (since learning is supervised) a set $P$ of $|P|$ training samples. Additionally, we already know that classes are predefined, too, i.e. we also have a set of classes $C$. A codebook vector is clearly assigned to each class. Thus, we can say that the set of classes $|C|$ contains many codebook vectors $C_1, C_2, \ldots, C_{|C|}$.

This leads to the structure of the training samples: They are of the form $(p, c)$ and therefore contain the training input vector $p$ and its class affiliation $c$. For the class affiliation

$$c \in \{1, 2, \ldots, |C|\}$$

holds, which means that it clearly assigns the training sample to a class or a codebook vector.

Intuitively, we could say about learning: "Why a learning procedure? We calculate the average of all class members and place their codebook vectors there – and that's it." But we will see soon that our learning procedure can do a lot more.

I only want to briefly discuss the steps of the fundamental LVQ learning procedure:

**Initialization:** We place our set of codebook vectors on random positions in the input space.

**Training sample:** A training sample $p$ of our training set $P$ is selected and presented.

**Distance measurement:** We measure the distance $||p - C||$ between all codebook vectors $C_1, C_2, \ldots, C_{|C|}$ and our input $p$.

**Winner:** The closest codebook vector wins, i.e. the one with

$$\min_{C_i \in C} ||p - C_i||.$$

**Learning process:** The learning process takes place according to the rule

$$\Delta C_i = \eta(t) \cdot h(p, C_i) \cdot (p - C_i) \tag{9.1}$$

$$C_i(t+1) = C_i(t) + \Delta C_i, \tag{9.2}$$

which we now want to break down.

- ▷ We have already seen that the first factor $\eta(t)$ is a time-dependent learning rate allowing us to differentiate between large learning steps and fine tuning.
- ▷ The last factor $(p - C_i)$ is obviously the *direction* toward which the codebook vector is moved.
- ▷ But the function $h(p, C_i)$ is the core of the rule: It implements a distinction of cases.

  **Assignment is correct:** The winner vector is the codebook vector of the class that includes $p$. In this case, the function provides positive values and the codebook vector moves towards $p$.

  **Assignment is wrong:** The winner vector does not represent the class that includes $p$. Therefore it moves away from $p$.

We can see that our definition of the function $h$ was not precise enough. With good reason: From here on, the LVQ is divided into different nuances, dependent of how exactly $h$ and the learning rate should be defined (called ***LVQ1***, ***LVQ2***, ***LVQ3***, ***OLVQ***, etc). The differences are, for instance, in the strength of the codebook vector movements. They are not all based on the same principle described here, and as announced I don't want to discuss them any further. Therefore I don't give any formal definition regarding the aforementioned learning rule and LVQ.

## 9.5 Connection to neural networks

Until now, in spite of the learning process, the question was what LVQ has to do with neural networks. The codebook vectors can be understood as neurons with a fixed position within the input space, similar to RBF networks. Additionally, in nature it

often occurs that in a group one neuron may fire (a winner neuron, here: a codebook vector) and, in return, inhibits all other neurons.

I decided to place this brief chapter about learning vector quantization here so that this approach can be continued in the following chapter about self-organizing maps: We will classify further inputs by means of neurons distributed throughout the input space, only that this time, we do not know which input belongs to which class.

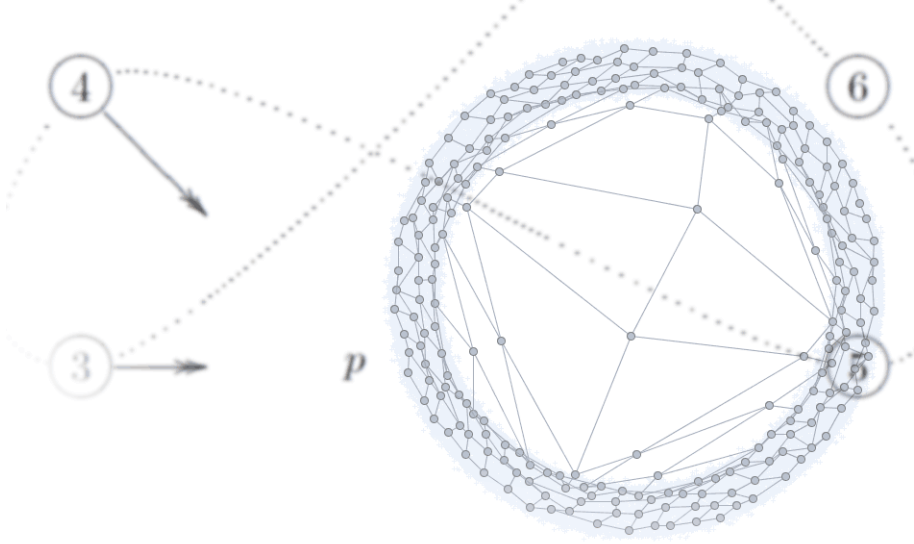
Now let us take a look at the *unsupervised learning networks*!

## Exercises

**Exercise 16.** Indicate a quantization which equally distributes all vectors $H \in \mathcal{H}$ in the five-dimensional unit cube $\mathcal{H}$ into one of 1024 classes.

# Part III

# Unsupervised learning network paradigms




# Chapter 10

# Self-organizing feature maps

A paradigm of unsupervised learning neural networks, which maps an input space by its fixed topology and thus independently looks for simililarities. Function, learning procedure, variations and neural gas.

---

If you take a look at the concepts of biological neural networks mentioned in the introduction, one question will arise: How does our brain store and recall the impressions it receives every day. Let me point out that the brain does not have any training samples and therefore no "desired output". And while already considering this subject we realize that there is no output in this sense at all, too. Our brain responds to external input by changes in state. These are, so to speak, its output.

Based on this principle and exploring the question of how biological neural networks organize themselves, Teuvo Kohonen developed in the Eighties his *self-organizing feature maps* [Koh82, Koh98], shortly referred to as ***self-organizing maps*** or ***SOMs***. A paradigm of neural networks where the output is the state of the network, which learns completely unsupervised, i.e. without a teacher.

Unlike the other network paradigms we have already got to know, for SOMs it is unnecessary to ask what the neurons calculate. We only ask *which neuron is active at the moment*. Biologically, this is very motivated: If in biology the neurons are connected to certain muscles, it will be less interesting to know how strong a certain muscle is contracted but *which* muscle is activated. In other words: We are not interested in the exact output of the neuron but in knowing *which neuron* provides output. Thus, SOMs are considerably more related to biology than, for example, the feedforward networks, which are increasingly used for calculations.

## 10.1 Structure of a self-organizing map

Typically, SOMs have – like our brain – the task to map a high-dimensional input ($N$ dimensions) onto areas in a low-dimensional ***grid*** of cells ($G$ dimensions) to draw a map of the high-dimensional space, so to speak. To generate this map, the SOM simply obtains arbitrary many points of the input space. During the input of the points the SOM will try to cover as good as possible the positions on which the points appear by its neurons. This particularly means, that every neuron can be assigned to a certain position in the input space.

At first, these facts seem to be a bit confusing, and it is recommended to briefly reflect about them. There are two spaces in which SOMs are working:

- The $N$-dimensional input space and
- the $G$-dimensional grid on which the neurons are lying and which indicates the neighborhood relationships between the neurons and therefore the *network topology*.

In a one-dimensional grid, the neurons could be, for instance, like pearls on a string. Every neuron would have exactly two neighbors (except for the two end neurons). A two-dimensional grid could be a square array of neurons (fig. 10.1 on the next page). Another possible array in two-dimensional space would be some kind of honeycomb shape. Irregular topologies are possible, too, but not very often. Topolgies with more dimensions and considerably more neighborhood relationships would also be possible, but due to their lack of visualization capability they are not employed very often.

Even if $N = G$ is true, the two spaces are not equal and have to be distinguished. In this special case they only have the same dimension.

Initially, we will briefly and formally regard the functionality of a self-organizing map and then make it clear by means of some examples.

**Definition 10.1** (SOM neuron)**.** Similar to the neurons in an RBF network a ***SOM neuron*** $k$ does not occupy a fixed position $c_k$ (a ***center***) in the input space.

**Definition 10.2** (Self-organizing map)**.** A self-organizing map is a set $K$ of SOM neurons. If an input vector is entered, exactly that neuron $k \in K$ is activated which is closest to the input pattern in the input space. The dimension of the input space is referred to as $N$.

**Definition 10.3** (Topology)**.** The neurons are interconnected by neighborhood relationships. These neighborhood relationships are called ***topology***. The training of

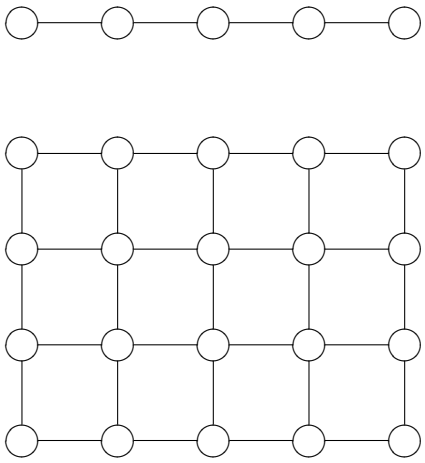


**Figure 10.1:** Example topologies of a self-organizing map. Above we can see a one-dimensional topology, below a two-dimensional one.

a SOM is highly influenced by the topology. It is defined by the *topology function* $h(i, k, t)$, where $i$ is the winner neuron[1] ist, $k$ the neuron to be adapted (which will be discussed later) and $t$ the timestep. The dimension of the topology is referred to as $G$.

## 10.2 SOMs always activate the neuron with the least distance to an input pattern

Like many other neural networks, the SOM has to be trained before it can be used. But let us regard the very simple functionality of a complete self-organizing map before training, since there are many analogies to the training. Functionality consists of the following steps:

**Input** of an arbitrary value $p$ of the input space $\mathbb{R}^N$.

**Calculation of the distance** between every neuron $k$ and $p$ by means of a norm, i.e. calculation of $||p - c_k||$.

1 We will learn soon what a winner neuron is.

**One neuron becomes active,** namely such neuron $i$ with the shortest calculated distance to the input. All other neurons remain inactive.This paradigm of activity is also called *winner-takes-all scheme*. The output we expect due to the input of a SOM shows *which* neuron becomes active.

In many literature citations, the description of SOMs is more formal: Often an input layer is described that is completely linked towards an SOM layer. Then the input layer ($N$ neurons) forwards all inputs to the SOM layer. The SOM layer is laterally linked in itself so that a winner neuron can be established and inhibit the other neurons. I think that this explanation of a SOM is not very descriptive and therefore I tried to provide a clearer description of the network structure.

Now the question is which neuron is activated by which input – and the answer is given by the network itself during training.

## 10.3 Training

[Training makes the SOM topology cover the input space] The training of a SOM is nearly as straightforward as the functionality described above. Basically, it is structured into five steps, which partially correspond to those of functionality.

**Initialization:** The network starts with random neuron centers $c_k \in \mathbb{R}^N$ from the input space.

**Creating an input pattern:** A ***stimulus***, i.e. a point $p$, is selected from the input space $\mathbb{R}^N$. Now this stimulus is entered into the network.

**Distance measurement:** Then the distance $||p - c_k||$ is determined for every neuron $k$ in the network.

**Winner takes all:** The ***winner neuron*** $i$ is determined, which has the smallest distance to $p$, i.e. which fulfills the condition

$$||p - c_i|| \leq ||p - c_k|| \quad \forall\ k \neq i$$

. You can see that from several winner neurons one can be selected at will.

**Adapting the centers:** The neuron centers are moved within the input space according to the rule[2]

$$\Delta c_k = \eta(t) \cdot h(i,k,t) \cdot (p - c_k),$$

where the values $\Delta c_k$ are simply added to the existing centers. The last factor shows that the change in position of the neurons $k$ is proportional to the distance to the input pattern $p$ and, as usual, to a time-dependent learning rate $\eta(t)$. The above-mentioned network *topology* exerts its influence by means of the function $h(i,k,t)$, which will be discussed in the following.

**Definition 10.4** (SOM learning rule)**.** A SOM is trained by presenting an input pattern and determining the associated *winner neuron*. The winner neuron and its neighbor neurons, which are defined by the topology function, then adapt their centers according to the rule

$$\Delta c_k = \eta(t) \cdot h(i,k,t) \cdot (p - c_k), \tag{10.1}$$

$$c_k(t+1) = c_k(t) + \Delta c_k(t). \tag{10.2}$$

### 10.3.1 The topology function defines, how a learning neuron influences its neighbors

The ***topology function*** $h$ is not defined on the input space but *on the grid* and represents the neighborhood relationships between the neurons, i.e. the topology of the network. It can be time-dependent (which it often is) – which explains the parameter $t$. The parameter $k$ is the index running through all neurons, and the parameter $i$ is the index of the winner neuron.

In principle, the function shall take a large value if $k$ is the neighbor of the winner neuron or even the winner neuron itself, and small values if not. SMore precise definition: The topology function must be *unimodal*, i.e. it must have exactly one maximum. This maximum must be next to the winner neuron $i$, for which the distance to itself certainly is 0.

Additionally, the time-dependence enables us, for example, to reduce the neighborhood in the course of time.

2 Note: In many sources this rule is written $\eta h(p - c_k)$, which wrongly leads the reader to believe that $h$ is a constant. This problem can easily be solved by not omitting the multiplication dots $\cdot$.

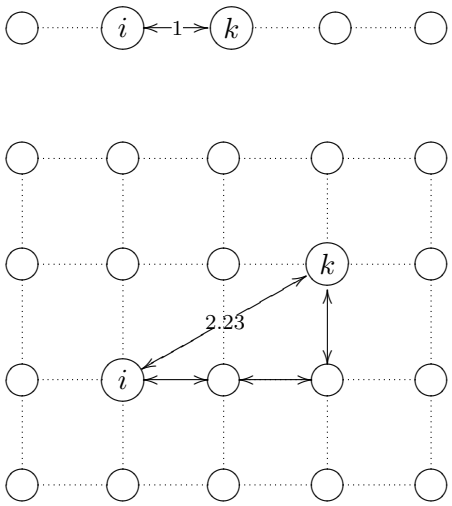



**Figure 10.2:** Example distances of a one-dimensional SOM topology (above) and a two-dimensional SOM topology (below) between two neurons $i$ and $k$. In the lower case the Euclidean distance is determined (in two-dimensional space equivalent to the Pythagoream theorem). In the upper case we simply count the discrete path length between $i$ and $k$. To simplify matters I required a fixed grid edge length of 1 in both cases.

In order to be able to output large values for the neighbors of $i$ and small values for non-neighbors, the function $h$ needs some kind of *distance notion* on the grid because from somewhere it has to know *how far* $i$ and $k$ are apart from each other on the grid. There are different methods to calculate this distance.

On a two-dimensional grid we could apply, for instance, the *Euclidean distance* (lower part of fig. 10.2) or on a one-dimensional grid we could simply use the number of the connections between the neurons $i$ and $k$ (upper part of the same figure).

**Definition 10.5** (Topology function)**.** The topology function $h(i, k, t)$ describes the neighborhood relationships in the topology. It can be any unimodal function that reaches its maximum when $i = k$ gilt. Time-dependence is optional, but often used.

#### 10.3.1.1 Introduction of common distance and topology functions

A common distance function would be, for example, the already known ***Gaussian bell*** (see fig. 10.3 on the next page). It is unimodal with a maximum close to 0. Additionally, its width can be changed by applying its parameter $\sigma$ , which can be used to realize the neighborhood being reduced in the course of time: We simply relate the time-dependence to the $\sigma$ and the result is a monotonically decreasing $\sigma(t)$. Then our topology function could look like this:

$$h(i,k,t) = \mathrm{e}^{\left(-\frac{||g_i - c_k||^2}{2\cdot\sigma(t)^2}\right)}, \tag{10.3}$$

where $g_i$ and $g_k$ represent the neuron positions *on the grid*, not the neuron positions in the input space, which would be referred to as $c_i$ and $c_k$.

Other functions that can be used instead of the Gaussian function are, for instance, the ***cone function***, the ***cylinder function*** or the ***Mexican hat function*** (fig. 10.3 on the following page). Here, the Mexican hat function offers a particular biological motivation: Due to its negative digits it rejects some neurons close to the winner neuron, a behavior that has already been observed in nature. This can cause sharply separated map areas – and that is exactly why the Mexican hat function has been suggested by Teuvo Kohonen himself. But this adjustment characteristic is not necessary for the functionality of the map, it could even be possible that the map would diverge, i.e. it could virtually explode.

### 10.3.2 Learning rates and neighborhoods can decrease monotonically over time

To avoid that the later training phases forcefully pull the entire map towards a new pattern, the SOMs often work with temporally monotonically decreasing learning rates and neighborhood sizes. At first, let us talk about the learning rate: Typical sizes of the target value of a learning rate are two sizes smaller than the initial value, e.g

$$0.01 < \eta < 0.6$$

could be true. But this size must also depend on the network topology or the size of the neighborhood.

As we have already seen, a decreasing neighborhood size can be realized, for example, by means of a time-dependent, monotonically decreasing $\sigma$ with the Gaussin bell being used in the topology function.

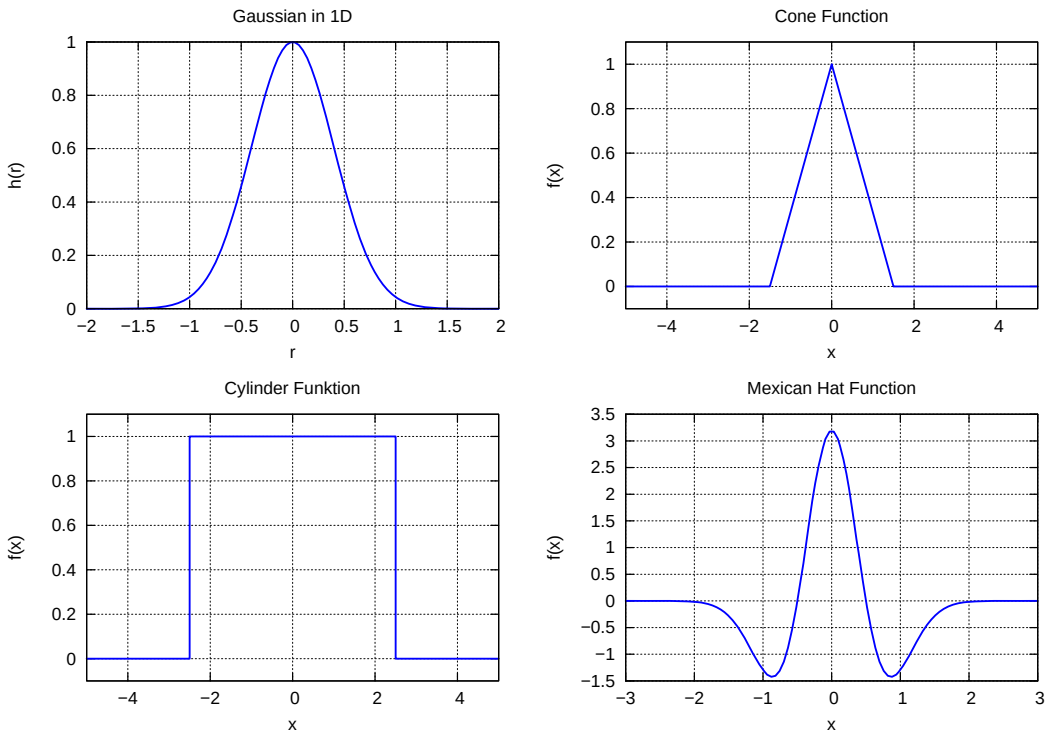



**Figure 10.3:** Gaussian bell, cone function, cylinder function and the Mexican hat function suggested by Kohonen as examples for topology functions of a SOM..

The advantage of a decreasing neighborhood size is that in the beginning a moving neuron "pulls along" many neurons in its vicinity, i.e. the randomly initialized network can unfold fast and properly in the beginning. In the end of the learning process, only a few neurons are influenced at the same time which stiffens the network as a whole but enables a good "fine tuning" of the individual neurons.

It must be noted that

$$h \cdot \eta \leq 1$$

must always be true, since otherwise the neurons would constantly miss the current training sample.

But enough of theory – let us take a look at a SOM in action!

## 10.4 Examples for the functionality of SOMs

Let us begin with a simple, mentally comprehensible example.

In this example, we use a two-dimensional input space, i.e. $N = 2$ is true. Let the grid structure be one-dimensional ($G = 1$). Furthermore, our example SOM should consist of 7 neurons and the learning rate should be $\eta = 0.5$.

The neighborhood function is also kept simple so that we will be able to mentally comprehend the network:

$$h(i,k,t) = \begin{cases} 1 & k \text{ direct neighbor of } i, \\ 1 & k = i, \\ 0 & \text{otherw.} \end{cases} \tag{10.4}$$

Now let us take a look at the above-mentioned network with random initialization of the centers (fig. 10.4 on the next page) and enter a training sample $p$. Obviously, in our example the input pattern is closest to neuron 3, i.e. this is the winning neuron.

We remember the learning rule for SOMs

$$\Delta c_k = \eta(t) \cdot h(i,k,t) \cdot (p - c_k)$$

and process the three factors from the back:

**Learning direction:** Remember that the neuron centers $c_k$ are vectors in the input space, as well as the pattern $p$. Thus, the factor $(p - c_k)$ indicates the vector of the neuron $k$ to the pattern $p$. This is now multiplied by different scalars:

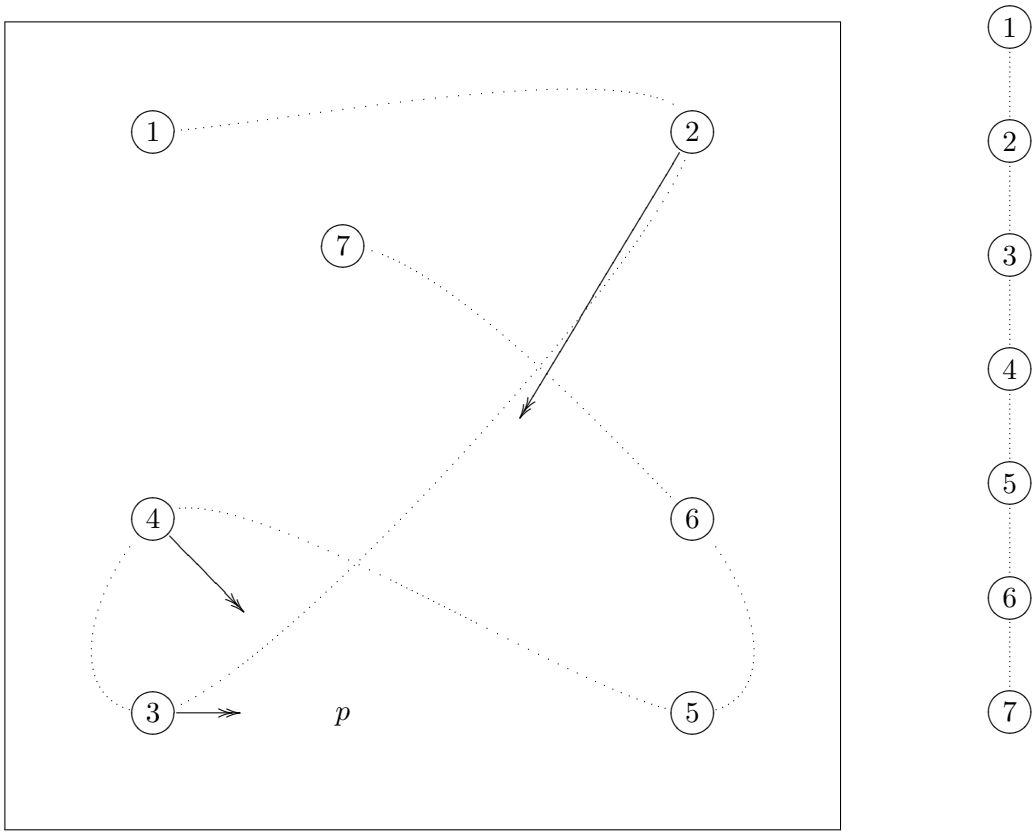



**Figure 10.4:** Illustration of the two-dimensional input space (left) and the one-dimensional topolgy space (right) of a self-organizing map. Neuron 3 is the winner neuron since it is closest to $p$. In the topology, the neurons 2 and 4 are the neighbors of 3. The arrows mark the movement of the winner neuron and its neighbors towards the training sample $p$.

To illustrate the one-dimensional topology of the network, it is plotted into the input space by the dotted line. The arrows mark the movement of the winner neuron and its neighbors towards the pattern.

**Our topology function** $h$ indicates that only the winner neuron and its two closest neighbors (here: 2 and 4) are allowed to learn by returning 0 for all other neurons. A time-dependence is not specified. Thus, our vector $(p - c_k)$ is multiplied by either 1 or 0.

**The learning rate** indicates, as always, the strength of learning. As already mentioned, $\eta = 0.5$, i. e. all in all, the result is that the winner neuron and its neighbors (here: 2, 3 and 4) approximate the pattern p half the way (in the figure marked by arrows).

Although the center of neuron 7 – seen from the input space – is considerably closer to the input pattern $p$ than neuron 2, neuron 2 is learning and neuron 7 is not. I want to remind that the network topology specifies which neuron is allowed to learn *and not its position in the input space.* This is exactly the mechanism by which a topology can significantly cover an input space without having to be related to it by any sort.

After the adaptation of the neurons 2, 3 and 4 the next pattern is applied, and so on. Another example of how such a one-dimensional SOM can develop in a two-dimensional input space with uniformly distributed input patterns in the course of time can be seen in figure 10.5 on the following page.

End states of one- and two-dimensional SOMs with differently shaped input spaces can be seen in figure 10.6 on page 183. As we can see, not every input space can be neatly covered by every network topology. There are so called *exposed* neurons – neurons which are located in an area where no input pattern has ever been occurred. A one-dimensional topology generally produces less exposed neurons than a two-dimensional one: For instance, during training on circularly arranged input patterns it is nearly impossible with a two-dimensional squared topology to avoid the exposed neurons in the center of the circle. These are pulled in every direction during the training so that they finally remain in the center. But this does not make the one-dimensional topology an optimal topology since it can only find less complex neighborhood relationships than a multi-dimensional one.

### 10.4.1 Topological defects are failures in SOM unfolding

During the unfolding of a SOM it could happen that a ***topological defect*** (fig. 10.7 on page 184) occurs, i.e. the SOM does not unfold correctly. A topological defect can be described at best by means of the word "knotting".

A remedy for topological defects could be to increase the initial values for the neighborhood size, because the more complex the topology is (or the more neighbors each

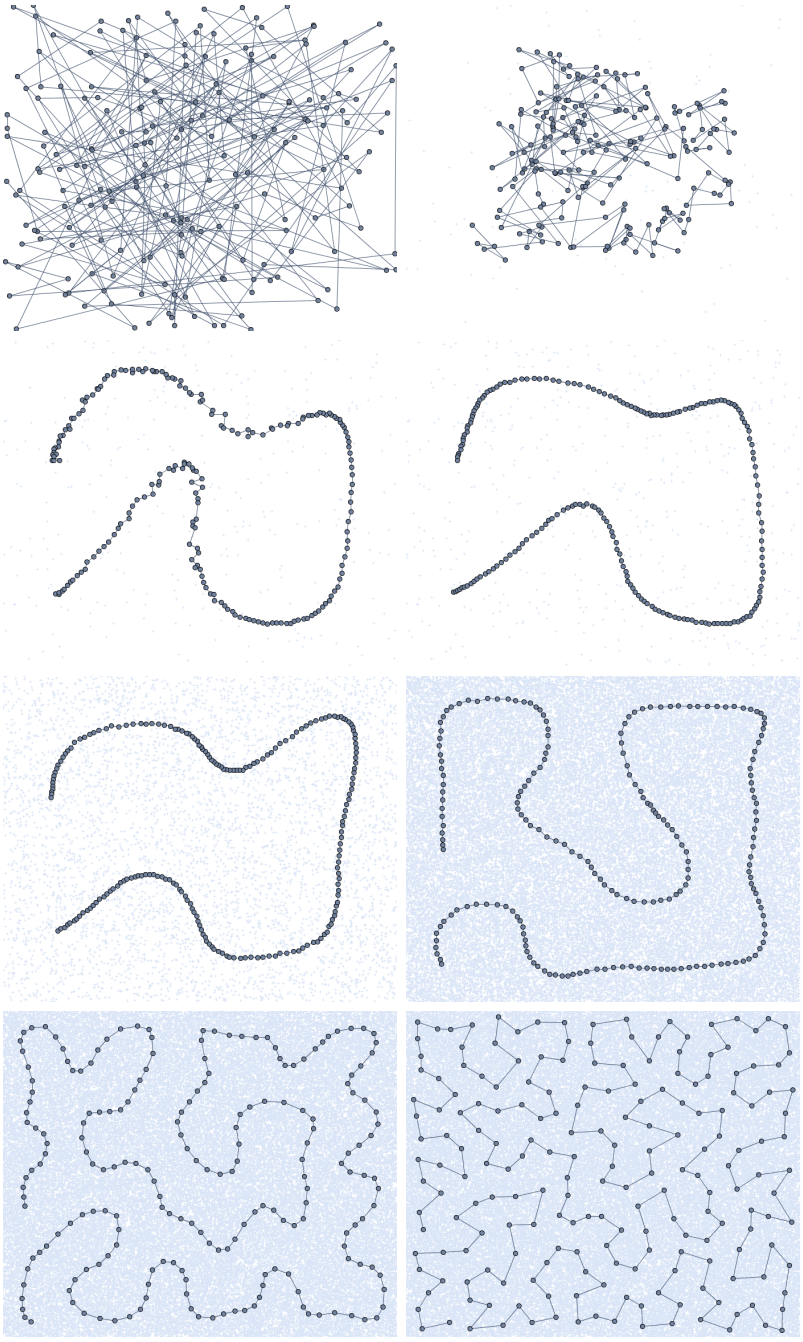


**Figure 10.5:** Behavior of a SOM with one-dimensional topology ($G = 1$) after the input of 0, 100, 300, 500, 5000, 50000, 70000 and 80000 randomly distributed input patterns $p \in \mathbb{R}^2$. During the training $\eta$ decreased from $1.0$ to $0.1$, the $\sigma$ parameter of the Gauss function decreased from $10.0$ to $0.2$.

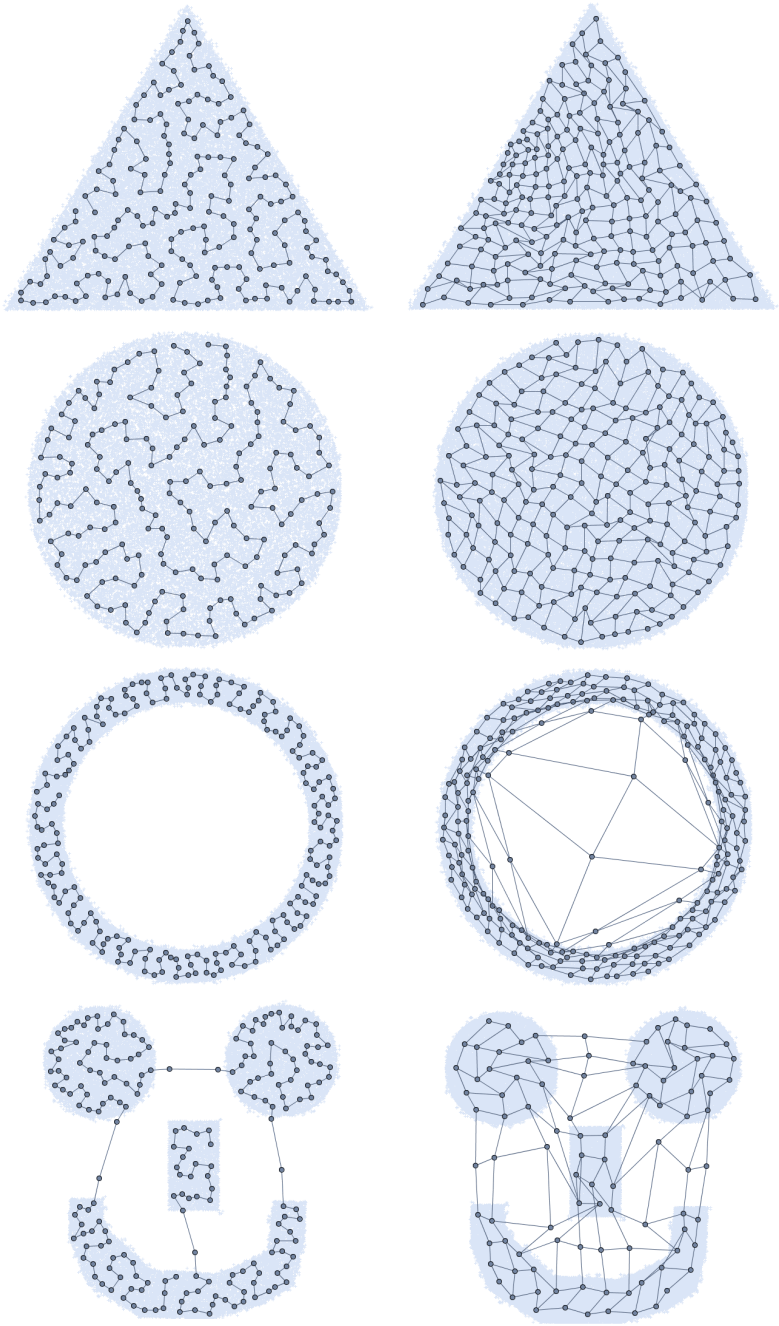


**Figure 10.6:** End states of one-dimensional (left column) and two-dimensional (right column) SOMs on different input spaces. 200 neurons were used for the one-dimensional topology, $10 \times 10$ neurons for the two-dimensionsal topology and 80.000 input patterns for all maps.

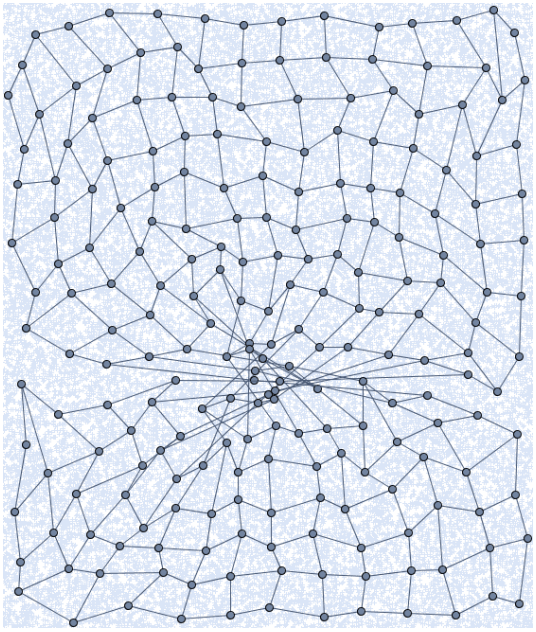


**Figure 10.7:** A topological defect in a two-dimensional SOM.

neuron has, respectively, since a three-dimensional or a honeycombed two-dimensional topology could also be generated) the more difficult it is for a randomly initialized map to unfold.

## 10.5 It is possible to adjust the resolution of certain areas in a SOM

We have seen that a SOM is trained by entering input patterns of the input space $\mathbb{R}^N$ one after another, again and again so that the SOM will be aligned with these patterns and *map* them. It could happen that we want a certain subset $U$ of the input space to be mapped more precise than the other ones.

This problem can easily be solved by means of SOMs: During the training disproportionally many input patterns of the area $U$ are presented to the SOM. If the number of training patterns of $U \subset \mathbb{R}^N$ presented to the SOM exceeds the number of those pat-

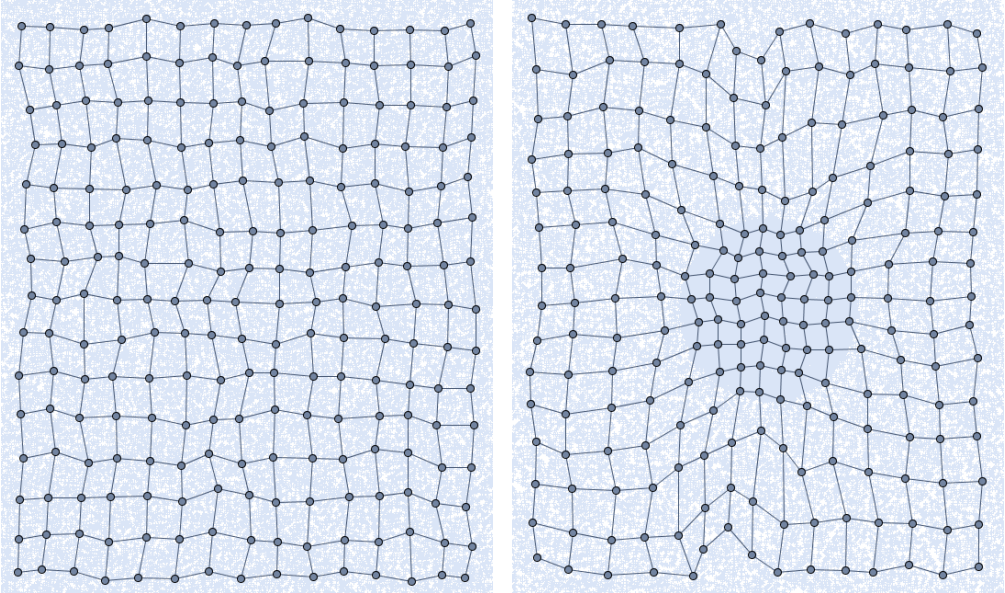


**Figure 10.8:** Training of a SOM with $G = 2$ on a two-dimensional input space. On the left side, the chance to become a training pattern was equal for each coordinate of the input space. On the right side, for the central circle in the input space, this chance is more than ten times larger than for the remaining input space (visible in the larger pattern density in the background). In this circle the neurons are obviously more crowded and the remaining area is covered less dense but in both cases the neurons are still evenly distributed. The two SOMS were trained by means of 80.000 training samples and decreasing $\eta$ ($1 \to 0.2$) as well as decreasing $\sigma$ ($5 \to 0.5$).

terns of the remaining $\mathbb{R}^N \setminus U$, then more neurons will group there while the remaining neurons are sparsely distributed on $\mathbb{R}^N \setminus U$ (fig. 10.8).

As you can see in the illustration, the edge of the SOM could be deformed. This can be compensated by assigning to the edge of the input space a slightly higher probability of being hit by training patterns (an often applied approach for reaching every corner with the SOMs).

Also, a higher learning rate is often used for edge and corner neurons, since they are only pulled into the center by the topology. This also results in a significantly improved corner coverage.

## 10.6 Application of SOMs

Regarding the biologically inspired ***associative data storage***, there are many fields of application for self-organizing maps and their variations.

For example, the different phonemes of the finnish language have successfully been mapped onto a SOM with a two dimensional discrete grid topology and therefore neighborhoods have been found (a SOM does nothing else than finding neighborhood relationships). So one tries once more to break down a high-dimensional space into a low-dimensional space (the topology), looks if some structures have been developed – et voilà: clearly defined areas for the individual phenomenons are formed.

Teuvo Kohonen himself made the effort to search many papers mentioning his SOMs in their keywords. In this large input space the individual papers now individual positions, depending on the occurrence of keywords. Then Kohonen created a SOM with $G = 2$ and used it to map the high-dimensional "paper space" developed by him.

Thus, it is possible to enter any paper into the completely trained SOM and look which neuron in the SOM is activated. It will be likely to discover that the *neighbored* papers in the topology are interesting, too. This type of brain-like ***context-based search*** also works with many other input spaces.

It is to be noted that the system itself defines what is neighbored, i.e. *similar*, within the topology – and that's why it is so interesting.

This example shows that the position $c$ of the neurons in the input space is not significant. It is rather interesting to see which neuron is activated when an unknown input pattern is entered. Next, we can look at which of the previous inputs this neuron was also activated – and will immediately discover a group of very similar inputs. The more the inputs within the topology are diverging, the less things they have in common. Virtually, the topology generates a map of the input characteristics – reduced to descriptively few dimensions in relation to the input dimension.

Therefore, the topology of a SOM often is two-dimensional so that it can be easily visualized, while the input space can be very high-dimensional.

### 10.6.1 SOMs can be used to determine centers for RBF neurons

SOMs arrange themselves exactly towards the positions of the outgoing inputs. As a result they are used, for example, to select the centers of an RBF network. We have already been introduced to the paradigm of the RBF network in chapter 6.

As we have already seen, it is possible to control which areas of the input space should be covered with higher resolution - or, in connection with RBF networks, on which areas of our function should the RBF network work with more neurons, i.e. work more exactly. As a further useful feature of the combination of RBF networks with SOMs one can use the topology obtained through the SOM: During the final training of a RBF neuron it can be used to influence neighboring RBF neurons in different ways.

For this, many neural network simulators offer an additional so-called *SOM layer* in connection with the simulation of RBF networks.

## 10.7 Variations of SOMs

There are different variations of SOMs for different variations of representation tasks:

### 10.7.1 A neural gas is a SOM without a static topology

The ***neural gas*** is a variation of the self-organizing maps of Thomas Martinetz [MBS93], which has been developed from the difficulty of mapping complex input information that partially only occur in the subspaces of the input space or even change the subspaces (fig. 10.9 on the following page).

The idea of a neural gas is, roughly speaking, to realize a SOM without a grid structure. Due to the fact that they are derived from the SOMs the learning steps are very similar to the SOM learning steps, but they include an additional intermediate step:

- again, random initialization of $c_k \in \mathbb{R}^n$
- selection and presentation of a pattern of the input space $p \in \mathbb{R}^n$
- neuron distance measurement
- identification of the winner neuron $i$
- *Intermediate step:* generation of a list $L$ of neurons sorted in ascending order by their distance to the winner neuron. Thus, the first neuron in the list $L$ is the neuron that is *closest* to the winner neuron.
- changing the centers by means of the known rule but with the slightly modified topology function

$$h_L(i, k, t).$$

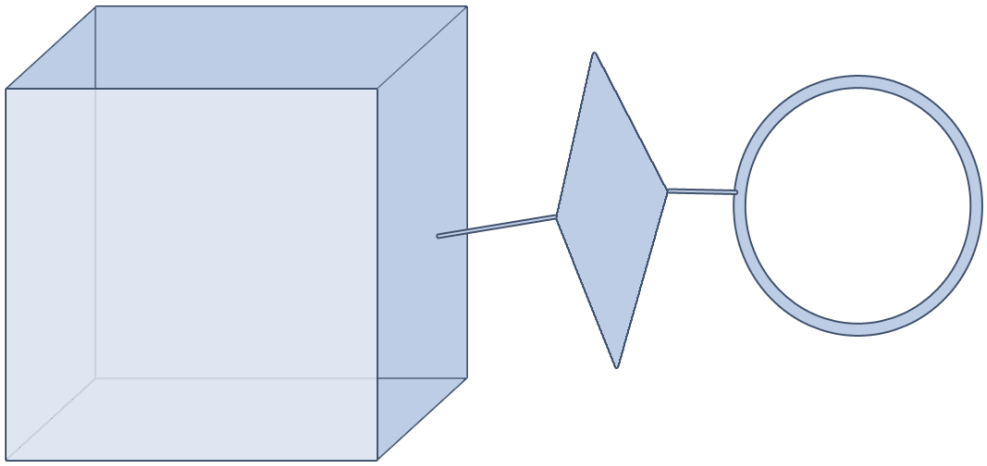


**Figure 10.9:** A figure filling different subspaces of the actual input space of different positions therefore can hardly be filled by a SOM.

The function $h_L(i, k, t)$, which is slightly modified compared with the original function $h(i, k, t)$, now regards the first elements of the list as the neighborhood of the winner neuron $i$. The direct result is that – similar to the free-floating molecules in a gas – the neighborhood relationships between the neurons can change anytime, and the number of neighbors is almost arbitrary, too. The distance within the neighborhood is now represented by the distance within the input space.

The bulk of neurons can become as stiffened as a SOM by means of a constantly decreasing neighborhood size. It does not have a fixed dimension but it can take the dimension that is locally needed at the moment, which can be very advantageous.

A disadvantage could be that there is no fixed grid forcing the input space to become regularly covered, and therefore wholes can occur in the cover or neurons can be isolated.

In spite of all practical hints, it is as always the user's responsibility not to understand this text as a catalog for easy answers but to explore all advantages and disadvantages himself.

Unlike a SOM, the neighborhood of a neural gas must initially refer to all neurons since otherwise some outliers of the random initialization may never reach the remaining group. To forget this is a popular error during the implementation of a neural gas.

With a neural gas it is possible to learn a kind of complex input such as in fig. 10.9 on the preceding page since we are not bound to a fixed-dimensional grid. But some computational effort could be necessary for the permanent sorting of the list (here, it could be effective to store the list in an ordered data structure right from the start).

**Definition 10.6** (Neural gas)**.** A neural gas differs from a SOM by a completely dynamic neighborhood function. With every learning cycle it is decided anew which neurons are the neigborhood neurons of the winner neuron. Generally, the criterion for this decision is the distance between the neurosn and the winner neuron in the input space.

### 10.7.2 A Multi-SOM consists of several separate SOMs

In order to present another variant of the SOMs, I want to formulate an extended problem: What do we do with input patterns from which we know that they are confined in different (maybe disjoint) areas?

Here, the idea is to use not only one SOM but several ones: A ***multi-self-organizing map***, shortly referred to as ***M-SOM*** [GKE01b, GKE01a, GS06]. It is unnecessary that the SOMs have the same topology or size, an M-SOM is just a combination of $M$ SOMs.

This learning process is analog to that of the SOMs. However, only the neurons belonging to the winner SOM of each training step are adapted. Thus, it is easy to represent two disjoint clusters of data by means of two SOMs, even if one of the clusters is not represented in every dimension of the input space $\mathbb{R}^N$. Actually, the individual SOMs exactly reflect these clusters.

**Definition 10.7** (Multi-SOM)**.** A multi-SOM is nothing more than the simultaneous use of $M$ SOMs.

### 10.7.3 A multi-neural gas consists of several separate neural gases

Analogous to the multi-SOM, we also have a set of $M$ neural gases: a ***multi-neural gas*** [GS06, SG06]. This construct behaves analogous to neural gas and M-SOM: Again, only the neurons of the winner gas are adapted.

The reader certainly wonders what advantage is there to use a multi-neural gas since an individual neural gas is already capable to divide into clusters and to work on complex input patterns with changing dimensions. Basically, this is correct, but a multi-neural gas has two serious advantages over a simple neural gas.

1. With several gases, we can directly tell which neuron belongs to which gas. This is particularly important for clustering tasks, for which multi-neural gases have been used recently. Simple neural gases can also find and cover clusters, but now we cannot recognize which neuron belongs to which cluster.

2. A lot of computational effort is saved when large original gases are divided into several smaller ones since (as already mentioned) the sorting of the list $L$ could use a lot of computational effort while the sorting of several smaller lists $L_1, L_2, \ldots, L_M$ is less time-consuming – even if these lists in total contain the same number of neurons.

As a result we will only obtain local instead of global sortings, but in most cases these local sortings are sufficient.

Now we can choose between two extreme cases of multi-neural gases: One extreme case is the ordinary neural gas $M = 1$, i.e. we only use one single neural gas. Interesting enough, the other extreme case (very large $M$, a few or only one neuron per gas) behaves analogously to the K-means clustering (for more information on clustering procedures see excursus A).

**Definition 10.8** (Multi-neural gas)**.** A multi-neural gas is nothing more than the simultaneous use of $M$ neural gases.

### 10.7.4 Growing neural gases can add neurons to themselves

A ***growing neural gas*** is a variation of the aforementioned neural gas to which more and more neurons are added according to certain rules. Thus, this is an attempt to work against the isolation of neurons or the generation of larger wholes in the cover.

Here, this subject should only be mentioned but not discussed.

To build a growing SOM is more difficult because new neurons have to be integrated in the neighborhood.

## Exercises

**Exercise 17.** A regular, two-dimensional grid shall cover a two-dimensional surface as "well" as possible.

1. Which grid structure would suit best for this purpose?

2. Which criteria did you use for "well" and "best"?

The very imprecise formulation of this exercise is intentional.


# Chapter 11

# Adaptive resonance theory

An ART network in its original form shall classify binary input vectors, i.e. to assign them to a 1-out-of-$n$ output. Simultaneously, the so far unclassified patterns shall be recognized and assigned to a new class.

As in the other smaller chapters, we want to try to figure out the basic idea of the ***adaptive resonance theory*** (abbreviated: ***ART***) without discussing its theory profoundly.

In several sections we have already mentioned that it is difficult to use neural networks for the learning of new information in addition to but without destroying the already existing information. This circumstance is called ***stability / plasticity dilemma***.

In 1987, Stephen Grossberg and Gail Carpenter published the first version of their ART network [Gro76] in order to alleviate this problem. This was followed by a whole family of ART improvements (which we want to discuss briefly, too).

It is the idea of unsupervised learning, whose aim is the (initially binary) pattern recognition, or more precisely the categorization of patterns into classes. But additionally an ART network shall be capable to find new classes.

## 11.1 Task and structure of an ART network

An ART network comprises exactly two layers: the input layer $I$ and the recognition layer $O$ with the input layer being completely linked towards the recognition layer. This complete link induces a ***top-down weight matrix*** $W$ that contains the weight values of the connections between each neuron in the input layer and each neuron in the recognition layer (fig. 11.1 on the following page).

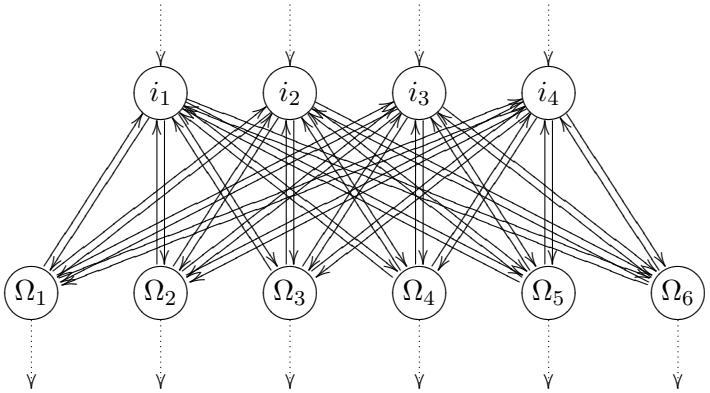



**Figure 11.1:** Simplified illustration of the ART network structure. Top: the input layer, bottom: the recognition layer. In this illustration the lateral inhibition of the recognition layer and the control neurons are omitted.

Simple binary patterns are entered into the input layer and transferred to the recognition layer while the recognition layer shall return a 1-out-of-$|O|$ encoding, i.e. it should follow the winner-takes-all scheme. For instance, to realize this 1-out-of-$|O|$ encoding the principle of *lateral inhibition* can be used – or in the implementation the most activated neuron can be searched. For practical reasons an IF query would suit this task best.

### 11.1.1 Resonance takes place by activities being tossed and turned

But there also exists a ***bottom-up weight matrix*** $V$, which propagates the activities within the recognition layer back into the input layer. Now it is obvious that these activities are bounced forth and back again and again, a fact that leads us to ***resonance***. Every activity within the input layer causes an activity within the recognition layer while in turn in the recognition layer every activity causes an activity within the input layer.

In addition to the two mentioned layers, in an ART network also exist a few neurons that exercise control functions such as signal enhancement. But we do not want to discuss this theory further since here only the basic principle of the ART network should

become explicit. I have only mentioned it to explain that in spite of the recurrences, the ART network will achieve a stable state after an input.

## 11.2 The learning process of an ART network is divided to top-down and bottom-up learning

The trick of adaptive resonance theory is not only the configuration of the ART network but also the two-piece learning procedure of the theory: On the one hand we train the top-down matrix $W$, on the other hand we train the bottom-up matrix $V$ (fig. 11.2 on the next page).

### 11.2.1 Pattern input and top-down learning

When a pattern is entered into the network it causes - as already mentioned - an activation at the output neurons and the strongest neuron wins. Then the weights of the matrix $W$ going towards the output neuron are changed such that the output of the strongest neuron $\Omega$ is still enhanced, i.e. the class affiliation of the input vector to the class of the output neuron $\Omega$ becomes enhanced.

### 11.2.2 Resonance and bottom-up learning

The training of the backward weights of the matrix $V$ is a bit tricky: Only the weights of the respective winner neuron are trained towards the input layer and our current input pattern is used as teaching input. Thus, the network is trained to enhance input vectors.

### 11.2.3 Adding an output neuron

Of course, it could happen that the neurons are nearly equally activated or that several neurons are activated, i.e. that the network is indecisive. In this case, the mechanisms of the control neurons activate a signal that adds a new output neuron. Then the current pattern is assigned to this output neuron and the weight sets of the new neuron are trained as usual.

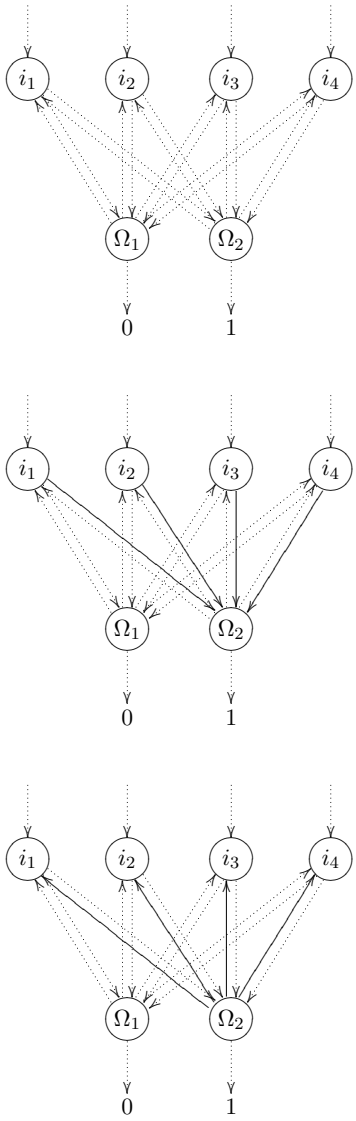



**Figure 11.2:** Simplified illustration of the two-piece training of an ART network: The trained weights are represented by solid lines. Let us assume that a pattern has been entered into the network and that the numbers mark the outputs. **Top:** We can see that $\Omega_2$ is the winner neuron. **Middle:** So the weights are trained towards the winner neuron and **(below)** the weights of the winner neuron are trained towards the input layer.

Thus, the advantage of this system is not only to divide inputs into classes and to find new classes, it can also tell us after the activation of an output neuron what a typical representative of a class looks like - which is a significant feature.

Often, however, the system can only moderately distinguish the patterns. The question is when a new neuron is permitted to become active and when it should learn. In an ART network there are different additional control neurons which answer this question according to different mathematical rules and which are responsible for intercepting special cases.

At the same time, one of the largest objections to an ART is the fact that an ART network uses a special distinction of cases, similar to an IF query, that has been forced into the mechanism of a neural network.

## 11.3 Extensions

As already mentioned above, the ART networks have often been extended.

***ART-2*** [CG87] is extended to continuous inputs and additionally offers (in an extension called ***ART-2A***) enhancements of the learning speed which results in additional control neurons and layers.

***ART-3*** [CG90] 3 improves the learning ability of ART-2 by adapting additional biological processes such as the chemical processes within the synapses[1].

Apart from the described ones there exist many other extensions.

1 Because of the frequent extensions of the adaptive resonance theory wagging tongues already call them "ART-$n$ networks".

# Part IV

# Excursi, appendices and registers

# Appendix A

# Excursus: Cluster analysis and regional and online learnable fields

In Grimm's dictionary the extinct German word "Kluster" is described by "was dicht und dick zusammensitzet (a thick and dense group of sth.)". In static cluster analysis, the formation of groups within point clouds is explored. Introduction of some procedures, comparison of their advantages and disadvantages. Discussion of an adaptive clustering method based on neural networks. A regional and online learnable field models from a point cloud, possibly with a lot of points, a comparatively small set of neurons being representative for the point cloud.

As already mentioned, many problems can be traced back to problems in ***cluster analysis***. Therefore, it is necessary to research procedures that examine whether groups (so-called ***clusters***) exist within point clouds.

Since cluster analysis procedures need a notion of distance between two points, a ***metric*** must be defined on the space where these points are situated.

We briefly want to specify what a metric is.

**Definition A.1** (Metric)**.** A relation $\text{dist}(x_1, x_2)$ defined for two objects $x_1, x_2$ is referred to as metric if each of the following criteria applies:

1. $\text{dist}(x_1, x_2) = 0$ if and only if $x_1 = x_2$,
2. $\text{dist}(x_1, x_2) = \text{dist}(x_2, x_1)$, i.e. symmetry,
3. $\text{dist}(x_1, x_3) \leq \text{dist}(x_1, x_2) + \text{dist}(x_2, x_3)$, i.e. the triangle inequality holds.

Colloquially speaking, a metric is a tool for determining distances between points in any space. Here, the distances have to be symmetrical, and the distance between to points may only be 0 if the two points are equal. Additionally, the triangle inequality must apply.

Metrics are provided by, for example, the ***squared distance*** and the ***Euclidean distance***, which have already been introduced. Based on such metrics we can define a clustering procedure that uses a metric as distance measure.

Now we want to introduce and briefly discuss different clustering procedures.

## A.1 k-means clustering allocates data to a predefined number of clusters

***k-means clustering*** according to J. MacQueen [Mac67] is an algorithm that is often used because of its low computation and storage complexity and which is regarded as "inexpensive and good". The operation sequence of the k-means clustering algorithm is the following:

1. Provide data to be examined.
2. Define $k$, which is the number of cluster centers.
3. Select $k$ random vectors for the cluster centers (also referred to as ***codebook vectors***).
4. Assign each data point to the next codebook vector[1]
5. Compute cluster centers for all clusters.
6. Set codebook vectors to new cluster centers.
7. Continue with 4 until the assignments are no longer changed.

Step 2 already shows one of the great questions of the k-means algorithm: The number $k$ of the cluster centers has to be determined in advance. This cannot be done by the algorithm. The problem is that it is not necessarily known in advance how $k$ can be determined best. Another problem is that the procedure can become quite instable if the codebook vectors are badly initialized. But since this is random, it is often useful to restart the procedure. This has the advantage of not requiring much computational effort. If you are fully aware of those weaknesses, you will receive quite good results.

1 The name *codebook vector* was created because the often used name *cluster vector* was too unclear.

However, complex structures such as "clusters in clusters" cannot be recognized. If $k$ is high, the outer ring of the construction in the following illustration will be recognized as many single clusters. If $k$ is low, the ring with the small inner clusters will be recognized as one cluster.

For an illustration see the upper right part of fig. A.1 on page 205.

## A.2 k-nearest neighboring looks for the k nearest neighbors of each data point

The ***k-nearest neighboring procedure*** [CH67] connects each data point to the $k$ closest neighbors, which often results in a division of the groups. Then such a group builds a cluster. The advantage is that the number of clusters occurs all by itself. The disadvantage is that a large storage and computational effort is required to find the next neighbor (the distances between all data points must be computed and stored).

There are some special cases in which the procedure combines data points belonging to different clusters, if $k$ is too high. (see the two small clusters in the upper right of the illustration). Clusters consisting of only one single data point are basically conncted to another cluster, which is not always intentional.

Furthermore, it is not mandatory that the links between the points are symmetric.

But this procedure allows a recognition of rings and therefore of "clusters in clusters", which is a clear advantage. Another advantage is that the procedure adaptively responds to the distances in and between the clusters.

For an illustration see the lower left part of fig. A.1.

## A.3 $\varepsilon$-nearest neighboring looks for neighbors within the radius $\varepsilon$ for each data point

Another approach of neighboring: here, the neighborhood detection does not use a fixed number $k$ of neighbors but a radius $\varepsilon$, which is the reason for the name ***epsilon-nearest neighboring***. Points are neigbors if they are at most $\varepsilon$ apart from each other. Here, the storage and computational effort is obviously very high, which is a disadvantage.

But note that there are some special cases: Two separate clusters can easily be connected due to the unfavorable situation of a single data point. This can also happen with $k$-nearest neighboring, but it would be more difficult since in this case the number of neighbors per point is limited.

An advantage is the symmetric nature of the neighborhood relationships. Another advantage is that the combination of minimal clusters due to a fixed number of neighbors is avoided.

On the other hand, it is necessary to skillfully initialize $\varepsilon$ in order to be successful, i.e. smaller than half the smallest distance between two clusters. With variable cluster and point distances within clusters this can possibly be a problem.

For an illustration see the lower right part of fig. A.1.

## A.4 The silhouette coefficient determines how accurate a given clustering is

As we can see above, there is no easy answer for clustering problems. Each procedure described has very specific disadvantages. In this respect it is useful to have a criterion to decide how good our cluster division is. This possibility is offered by the ***silhouette coefficient*** according to [Kau90]. This coefficient measures how well the clusters are delimited from each other and indicates if points may be assigned to the wrong clusters.

Let $P$ be a point cloud and $p$ a point in $P$. Let $c \subseteq P$ be a cluster within the point cloud and $p$ be part of this cluster, i.e. $p \in c$. The set of clusters is called $C$. Summary:

$$p \in c \subseteq P$$

applies.

To calculate the silhouette coefficient, we initially need the average distance between point $p$ and all its cluster neighbors. This variable is referred to as $a(p)$ and defined as follows:

$$a(p) = \frac{1}{|c| - 1} \sum_{q \in c, q \neq p} \text{dist}(p, q) \tag{A.1}$$

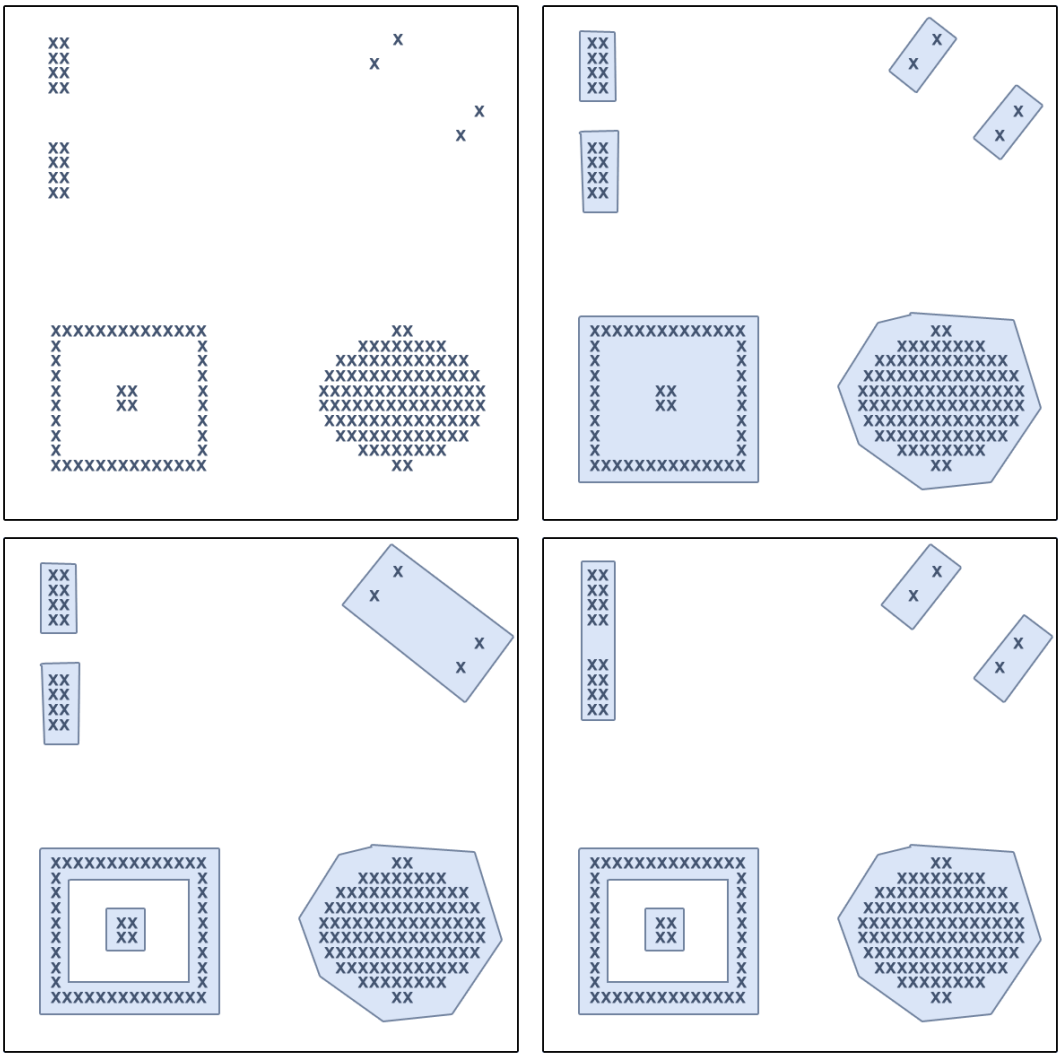


**Figure A.1: Top left: our set of points.** We will use this set to explore the different clustering methods. **Top right: $k$-means clustering.** Using this procedure we chose $k = 6$. As we can see, the procedure is not capable to recognize "clusters in clusters" (bottom left of the illustration). Long "lines" of points are a problem, too: They would be recognized as many small clusters (if $k$ is sufficiently large). **Bottom left: $k$-nearest neighboring.** If $k$ is selected too high (higher than the number of points in the smallest cluster), this will result in cluster combinations shown in the upper right of the illustration. **Bottom right: $\varepsilon$-nearest neighboring.** This procedure will cause difficulties if $\varepsilon$ is selected larger than the minimum distance between two clusters (see upper left of the illustration), which will then be combined.

Furthermore, let $b(p)$ be the average distance between our point $p$ and all points of the next cluster ($g$ represents all clusters except for $c$):

$$b(p) = \min_{g \in C, g \neq c} \frac{1}{|g|} \sum_{q \in g} \text{dist}(p, q) \tag{A.2}$$

The point $p$ is classified well if the distance to the center of the own cluster is minimal and the distance to the centers of the other clusters is maximal. In this case, the following term provides a value close to 1:

$$s(p) = \frac{b(p) - a(p)}{\max\{a(p), b(p)\}} \tag{A.3}$$

Apparently, the whole term $s(p)$ can only be within the interval $[-1; 1]$. A value close to -1 indicates a bad classification of $p$.

The silhouette coefficient $S(P)$ results from the average of all values $s(p)$:

$$S(P) = \frac{1}{|P|} \sum_{p \in P} s(p). \tag{A.4}$$

As above the total quality of the cluster division is expressed by the interval $[-1; 1]$.

As different clustering strategies with different characteristics have been presented now (lots of further material is presented in [DHS01]), as well as a measure to indicate the quality of an existing arrangement of given data into clusters, I want to introduce a clustering method based on an unsupervised learning neural network [SGE05] which was published in 2005. Like all the other methods this one may not be perfect but it eliminates large standard weaknesses of the known clustering methods

## A.5 Regional and online learnable fields are a neural clustering strategy

The paradigm of neural networks, which I want to introduce now, are the ***regional and online learnable fields***, shortly referred to as ***ROLFs***.

### A.5.1 ROLFs try to cover data with neurons

Roughly speaking, the regional and online learnable fields are a set $K$ of neurons which try to cover a set of points as well as possible by means of their distribution in the input space. For this, neurons are added, moved or changed in their size during training if necessary. The parameters of the individual neurons will be discussed later.

**Definition A.2** (Regional and online learnable field)**.** A regional and online learnable field (abbreviated ROLF or ROLF network) is a set $K$ of neurons that are trained to cover a certain set in the input space as well as possible.

#### A.5.1.1 ROLF neurons feature a position and a radius in the input space

Here, a ***ROLF neuron*** $k \in K$ has two parameters: Similar to the RBF networks, it has a ***center*** $c_k$, i.e. a position in the input space.

But it has yet another parameter: The radius $\sigma$, which defines the radius of the ***perceptive surface*** surrounding the neuron[2]. A neuron covers the part of the input space that is situated within this radius.

$c_k$ and $\sigma_k$ are locally defined for each neuron. This particularly means that the neurons are capable to cover surfaces of different sizes.

The radius of the perceptive surface is specified by $r = \rho \cdot \sigma$ (fig. A.2 on the next page) with the multiplier $\rho$ being *globally* defined and previously specified for all neurons. Intuitively, the reader will wonder what this multiplicator is used for. Its significance will be discussed later. Furthermore, the following has to be observed: It is not necessary for the perceptive surface of the different neurons to be of the same size.

**Definition A.3** (ROLF neuron)**.** The parameters of a ROLF neuron $k$ are a center $c_k$ and a radius $\sigma_k$.

**Definition A.4** (Perceptive surface)**.** The perceptive surface of a ROLF neuron $k$ consists of all points within the radius $\rho \cdot \sigma$ in the input space.

### A.5.2 A ROLF learns unsupervised by presenting training samples online

Like many other paradigms of neural networks our ROLF network learns by receiving many training samples $p$ of a training set $P$. The learning is unsupervised. For each training sample $p$ entered into the network two cases can occur:

1. There is one accepting neuron $k$ for $p$ or
2. there is no accepting neuron at all.

If in the first case several neurons are suitable, then there will be *exactly one* ***accepting neuron*** insofar as the closest neuron is the accepting one. For the accepting neuron $k$ $c_k$ and $\sigma_k$ are adapted.

2 I write "defines" and not "is" because the actual radius is specified by $\sigma \cdot \rho$.

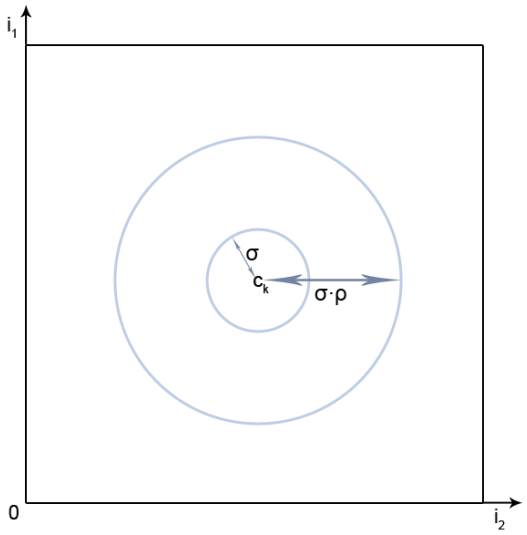



**Figure A.2:** Structure of a ROLF neuron.

**Definition A.5** (Accepting neuron)**.** The criterion for a ROLF neuron $k$ to be an accepting neuron of a point $p$ is that the point $p$ must be located within the perceptive surface of $k$. If $p$ is located in the perceptive surfaces of several neurons, then the closest neuron will be the accepting one. If there are several closest neurons, one can be chosen randomly.

#### A.5.2.1 Both positions and radii are adapted throughout learning

Let us assume that we entered a training sample $p$ into the network and that there is an accepting neuron $k$. Then the radius moves towards $||p - c_k||$ (i.e. towards the distance between $p$ and $c_k$) and the center $c_k$ towards $p$. Additionally, let us define the two learning rates $\eta_\sigma$ and $\eta_c$ for radii and centers.

$$c_k(t+1) = c_k(t) + \eta_c(p - c_k(t))$$
$$\sigma_k(t+1) = \sigma_k(t) + \eta_\sigma(||p - c_k(t)|| - \sigma_k(t))$$

Note that here $\sigma_k$ is a scalar while $c_k$ is a vector in the input space.

**Definition A.6** (Adapting a ROLF neuron)**.** A neuron $k$ accepted by a point $p$ is adapted according to the following rules:

$$c_k(t+1) = c_k(t) + \eta_c(p - c_k(t)) \tag{A.5}$$

$$\sigma_k(t+1) = \sigma_k(t) + \eta_\sigma(||p - c_k(t)|| - \sigma_k(t)) \tag{A.6}$$

#### A.5.2.2 The radius multiplier allows neurons to be able not only to shrink

Now we can understand the function of the multiplier $\rho$: Due to this multiplier the perceptive surface of a neuron includes more than only all points surrounding the neuron in the radius $\sigma$. This means that due to the aforementioned learning rule $\sigma$ cannot only decrease but also increase.

**Definition A.7** (Radius multiplier)**.** The radius multiplier $\rho > 1$ is globally defined and expands the perceptive surface of a neuron $k$ to a multiple of $\sigma_k$. So it is ensured that the radius $\sigma_k$ cannot only decrease but also increase.

Generally, the radius multiplier is set to values in the lower one-digit range, such as 2 or 3.

So far we only have discussed the case in the ROLF training that there is an accepting neuron for the training sample $p$.

#### A.5.2.3 As required, new neurons are generated

This suggests to discuss the approach for the case that there is no accepting neuron.

In this case a new accepting neuron $k$ is *generated* for our training sample. The result is of course that $c_k$ and $\sigma_k$ have to be initialized.

The initialization of $c_k$ can be understood intuitively: The center of the new neuron is simply set on the training sample, i.e.

$$c_k = p.$$

We generate a new neuron because there is no neuron close to $p$ – for logical reasons, we place the neuron exactly on $p$.

But how to set a $\sigma$ when a new neuron is generated? For this purpose there exist different options:

**Init-$\sigma$:** We always select a predefined static $\sigma$.

**Minimum $\sigma$:** We take a look at the $\sigma$ of each neuron and select the minimum.

**Maximum $\sigma$:** We take a look at the $\sigma$ of each neuron and select the maximum.

**Mean $\sigma$:** We select the mean $\sigma$ of all neurons.

Currently, the mean-$\sigma$ variant is the favorite one although the learning procedure also works with the other ones. In the minimum-$\sigma$ variant the neurons tend to cover less of the surface, in the maximum-$\sigma$ variant they tend to cover more of the surface.

**Definition A.8** (Generating a ROLF neuron)**.** If a new ROLF neuron $k$ is generated by entering a training sample $p$, then $c_k$ is intialized with $p$ and $\sigma_k$ according to one of the aforementioned strategies (init-$\sigma$, minimum-$\sigma$, maximum-$\sigma$, mean-$\sigma$).

The training is complete when after repeated randomly permuted pattern presentation no new neuron has been generated in an epoch and the positions of the neurons barely change.

### A.5.3 Evaluating a ROLF

The result of the training algorithm is that the training set is gradually covered well and precisely by the ROLF neurons and that a high concentration of points on a spot of the input space does not automatically generate more neurons. Thus, a possibly very large point cloud is reduced to very few representatives (based on the input set).

Then it is very easy to define the number of clusters: Two neurons are (according to the definition of the ROLF) connected when their perceptive surfaces overlap (i.e. some kind of *nearest neighboring* is executed with the variable perceptive surfaces). A cluster is a group of connected neurons or a group of points of the input space covered by these neurons (fig. A.3 on the facing page).

Of course, the complete ROLF network can be evaluated by means of other clustering methods, i.e. the neurons can be searched for clusters. Particularly with clustering methods whose storage effort grows quadratic to $|P|$ the storage effort can be reduced dramatically since generally there are considerably less ROLF neurons than original data points, but the neurons represent the data points quite well.

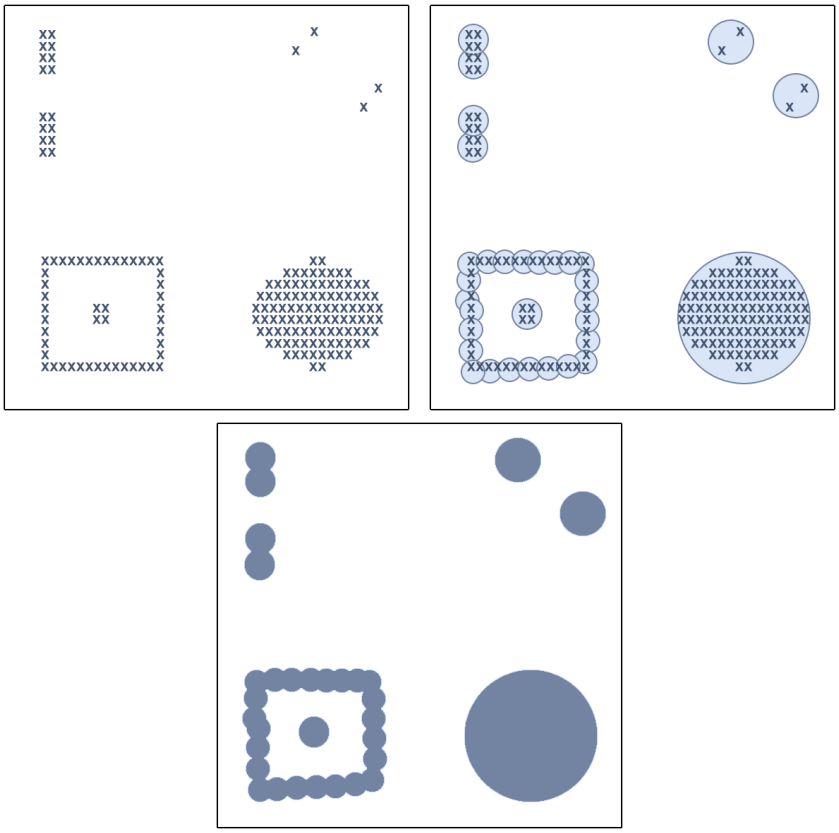


**Figure A.3:** The clustering process. **Top:** the input set, **middle:** the input space covered by ROLF neurons, **bottom:** the input space only covered by the neurons (representatives).

### A.5.4 Comparison with popular clustering methods

It is obvious, that storing the neurons rather than storing the input points takes the biggest part of the storage effort of the ROLFs. This is a great advantage for huge point clouds with a lot of points.

Since it is unnecessary to store the entire point cloud, our ROLF, as a neural clustering method, has the capability to *learn online*, which is definitely a great advantage. Furthermore, it can (similar to $\varepsilon$ nearest neighboring or $k$ nearest neighboring) distinguish clusters from enclosed clusters – but due to the online presentation of the data without a quadratically growing storage effort, which is by far the greatest disadvantage of the two neighboring methods.

Additionally, the issue of the size of the individual clusters proportional to their distance from each other is addressed by using variable perceptive surfaces - which is also not always the case for the two mentioned methods.

The ROLF compares favorably with $k$-means clustering, as well: Firstly, it is unnecessary to previously know the number of clusters and, secondly, $k$-means clustering recognizes clusters enclosed by other clusters as separate clusters.

### A.5.5 Initializing radii, learning rates and multiplier is not trivial

Certainly, the disadvantages of the ROLF shall not be concealed: It is not always easy to select the appropriate initial value for $\sigma$ and $\rho$. The previous knowledge about the data set can so to say be included in $\rho$ and the initial value of $\sigma$ of the ROLF: Fine-grained data clusters should use a small $\rho$ and a small $\sigma$ initial value. But the smaller the $\rho$ the smaller, the chance that the neurons will grow if necessary. Here again, there is no easy answer, just like for the learning rates $\eta_c$ and $\eta_\sigma$.

For $\rho$ the multipliers in the lower single-digit range such as 2 or 3 are very popular. $\eta_c$ and $\eta_\sigma$ successfully work with values about 0.005 to 0.1, variations during run-time are also imaginable for this type of network. Initial values for $\sigma$ generally depend on the cluster and data distribution (i.e. they often have to be tested). But compared to wrong initializations – at least with the mean-$\sigma$ strategy – they are relatively robust after some training time.

As a whole, the ROLF is on a par with the other clustering methods and is particularly very interesting for systems with low storage capacity or huge data sets.

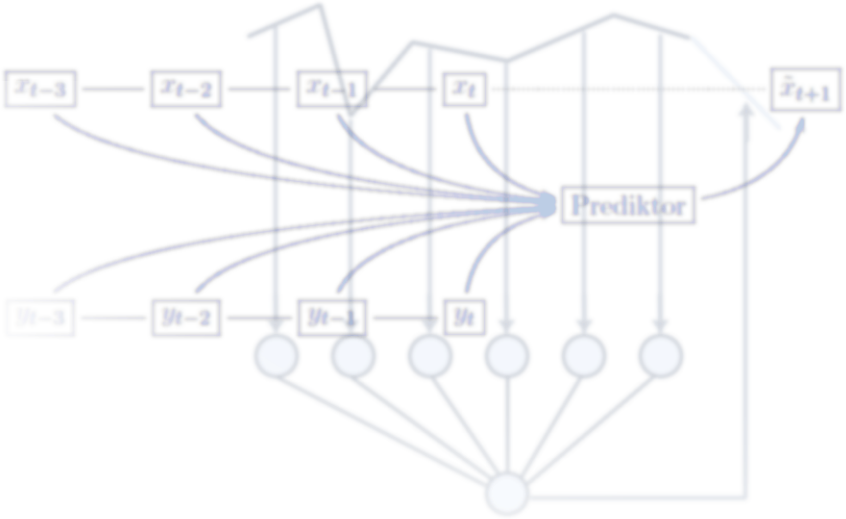
### A.5.6 Application examples

A first application example could be finding color clusters in RGB images. Another field of application directly described in the ROLF publication is the recognition of words transferred into a 720-dimensional feature space. Thus, we can see that ROLFs are relatively robust against higher dimensions. Further applications can be found in the field of analysis of attacks on network systems and their classification.

## Exercises

**Exercise 18.** Determine at least four adaptation steps for one single ROLF neuron $k$ if the four patterns stated below are presented one after another in the indicated order. Let the initial values for the ROLF neuron be $c_k = (0.1, 0.1)$ and $\sigma_k = 1$. Furthermore, let $\eta_c = 0.5$ and $\eta_\sigma = 0$. Let $\rho = 3$.

$$\begin{aligned} P = \{&(0.1, 0.1); \\ &= (0.9, 0.1); \\ &= (0.1, 0.9); \\ &= (0.9, 0.9)\}. \end{aligned}$$




# Appendix B

# Excursus: neural networks used for prediction

Discussion of an application of neural networks: a look ahead into the future of time series.

After discussing the different paradigms of neural networks it is now useful to take a look at an application of neural networks which is brought up often and (as we will see) is also used for fraud: The application of ***time series prediction***. This excursus is structured into the description of time series and estimations about the requirements that are actually needed to predict the values of a time series. Finally, I will say something about the range of software which should predict share prices or other economic characteristics by means of neural networks or other procedures.

This chapter should not be a detailed description but rather indicate some approaches for time series prediction. In this respect I will again try to avoid formal definitions.

## B.1 About time series

A ***time series*** is a series of values discretized in time. For example, daily measured temperature values or other meteorological data of a specific site could be represented by a time series. Share price values also represent a time series. Often the measurement of time series is timely equidistant, and in many time series the future development of their values is very interesting, e.g. the daily weather forecast.

Time series can also be values of an actually continuous function read in a certain distance of time $\Delta t$ (fig. B.1 on the next page).

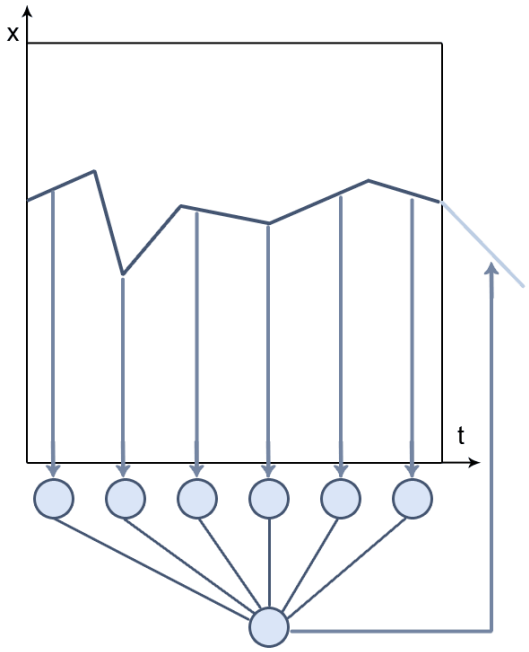



**Figure B.1:** A function $x$ that depends on the time is sampled at discrete time steps (time discretized), this means that the result is a time series. The sampled values are entered into a neural network (in this example an SLP) which shall learn to predict the future values of the time series.

If we want to *predict* a time series, we will look for a neural network that maps the previous series values to future developments of the time series, i.e. if we know longer sections of the time series, we will have enough training samples. Of course, these are not examples for the future to be predicted but it is tried to generalize and to extrapolate the past by means of the said samples.

But before we begin to predict a time series we have to answer some questions about this time series we are dealing with and ensure that it fulfills some requirements.

1. Do we have any evidence which suggests that future values depend in any way on the past values of the time series? Does the past of a time series include information about its future?

2. Do we have enough past values of the time series that can be used as training patterns?

3. In case of a prediction of a continuous function: What must a useful $\Delta t$ look like?

Now these questions shall be explored in detail.

How much information about the future is included in the past values of a time series? This is the most important question to be answered for any time series that should be mapped into the future. If the future values of a time series, for instance, do not depend on the past values, then a time series prediction based on them will be impossible.

In this chapter, we assume systems whose future values can be deduced from their states – the deterministic systems. This leads us to the question of what a system state is.

A system state *completely* describes a system for a certain point of time. The future of a deterministic system would be clearly defined by means of the complete description of its current state.

The problem in the real world is that such a state concept includes all things that influence our system by any means.

In case of our weather forecast for a specific site we could definitely determine the temperature, the atmospheric pressure and the cloud density as the meteorological state of the place at a time $t$. But the whole state would include significantly more information. Here, the worldwide phenomena that control the weather would be interesting as well as small local pheonomena such as the cooling system of the local power plant.

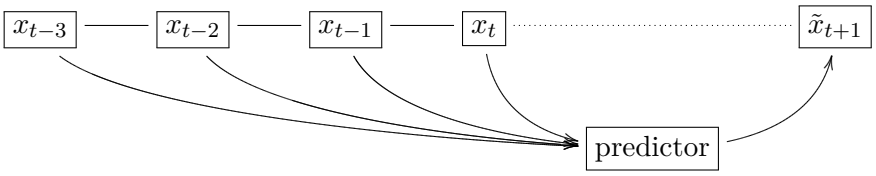



**Figure B.2:** Representation of the one-step-ahead prediction. It is tried to calculate the future value from a series of past values. The predicting element (in this case a neural network) is referred to as predictor.

So we shall note that the system state is desirable for prediction but not always possible to obtain. Often only fragments of the current states can be acquired, e.g. for a weather forecast these fragments are the said weather data.

However, we can partially overcome these weaknesses by using not only one single state (the last one) for the prediction, but by using several past states. From this we want to derive our first prediction system:

## B.2 One-step-ahead prediction

The first attempt to predict the next future value of a time series out of past values is called ***one-step-ahead prediction*** (fig. B.2).

Such a predictor system receives the last $n$ observed state parts of the system as input and outputs the prediction for the next state (or state part). The idea of a state space with predictable states is called ***state space forecasting***.

The aim of the predictor is to realize a function

$$f(x_{t-n+1}, \ldots, x_{t-1}, x_t) = \tilde{x}_{t+1}, \tag{B.1}$$

which receives exactly $n$ past values in order to predict the future value. Predicted values shall be headed by a tilde (e.g. $\tilde{x}$) to distinguish them from the actual future values.

The most intuitive and simplest approach would be to find a linear combination

$$\tilde{x}_{i+1} = a_0 x_i + a_1 x_{i-1} + \ldots + a_j x_{i-j} \tag{B.2}$$

that approximately fulfills our conditions.

Such a construction is called ***digital filter***. Here we use the fact that time series usually have a lot of past values so that we can set up a series of equations[1]:

$$\begin{aligned} x_t &= a_0 x_{t-1} + \ldots + a_j x_{t-1-(n-1)} \\ x_{t-1} &= a_0 x_{t-2} + \ldots + a_j x_{t-2-(n-1)} \\ &\vdots \\ x_{t-n} &= a_0 x_{t-n} + \ldots + a_j x_{t-n-(n-1)} \end{aligned} \tag{B.3}$$

Thus, $n$ equations could be found for $n$ unknown coefficients and solve them (if possible). Or another, better approach: we could use $m > n$ equations for $n$ unknowns in such a way that the sum of the mean squared errors of the already known prediction is minimized. This is called ***moving average procedure***.

But this linear structure corresponds to a singlelayer perceptron with a linear activation function which has been trained by means of data from the past (The experimental setup would comply with fig. B.1 on page 216). In fact, the training by means of the delta rule provides results very close to the analytical solution.

Even if this approach often provides satisfying results, we have seen that many problems cannot be solved by using a singlelayer perceptron. Additional layers with linear activation function are useless, as well, since a multilayer perceptron with only linear activation functions can be reduced to a singlelayer perceptron. Such considerations lead to a non-linear approach.

The multilayer perceptron and non-linear activation functions provide a universal non-linear function approximator, i.e. we can use an $n$-$|H|$-1-MLP for $n$ n inputs out of the past. An RBF network could also be used. But remember that here the number $n$ has to remain low since in RBF networks high input dimensions are very complex to realize. So if we want to include many past values, a multilayer perceptron will require considerably less computational effort.

## B.3 Two-step-ahead prediction

What approaches can we use to to see farther into the future?

1 Without going into detail, I want to remark that the prediction becomes easier the more past values of the time series are available. I would like to ask the reader to read up on the *Nyquist-Shannon sampling theorem*

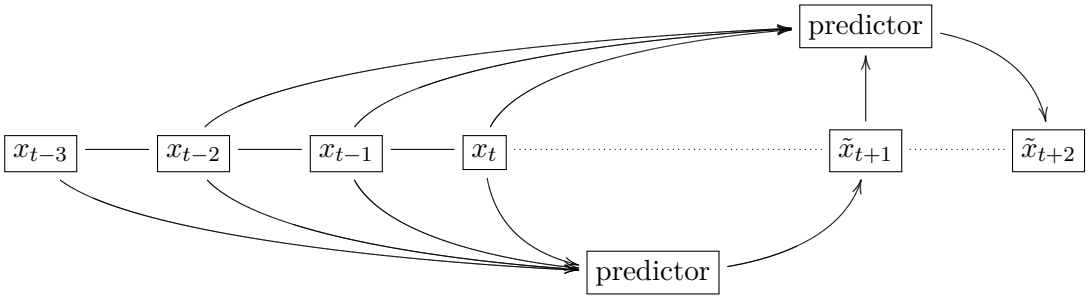



**Figure B.3:** Representation of the two-step-ahead prediction. Attempt to predict the second future value out of a past value series by means of a second predictor and the involvement of an already predicted value.

### B.3.1 Recursive two-step-ahead prediction

In order to extend the prediction to, for instance, two time steps into the future, we could perform two one-step-ahead predictions in a row (fig. B.3), i.e. a recursive ***two-step-ahead prediction***. Unfortunately, the value determined by means of a one-step-ahead prediction is generally imprecise so that errors can be built up, and the more predictions are performed in a row the more imprecise becomes the result.

### B.3.2 Direct two-step-ahead prediction

We have already guessed that there exists a better approach: Just like the system can be trained to predict the next value, we can certainly train it to predict the next but one value. This means we directly train, for example, a neural network to look two time steps ahead into the future, which is referred to as ***direct two-step-ahead prediction*** (fig. B.4 on the next page). Obviously, the direct two-step-ahead prediction is technically identical to the one-step-ahead prediction. The only difference is the training.

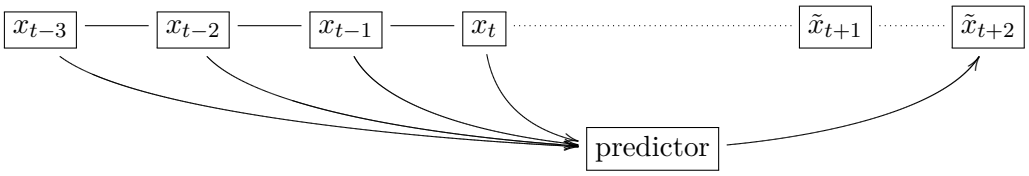



**Figure B.4:** Representation of the direct two-step-ahead prediction. Here, the second time step is predicted directly, the first one is omitted. Technically, it does not differ from a one-step-ahead prediction.

## B.4 Additional optimization approaches for prediction

The possibility to predict values far away in the future is not only important because we try to look farther ahead into the future. There can also be periodic time series where other approaches are hardly possible: If a lecture begins at 9 a.m. every Thursday, it is not very useful to know how many people sat in the lecture room on Monday to predict the number of lecture participants. The same applies, for example, to periodically occurring commuter jams.

### B.4.1 Changing temporal parameters

Thus, it can be useful to intentionally leave gaps in the future values as well as in the past values of the time series, i.e. to introduce the parameter $\Delta t$ which indicates which past value is used for prediction. Technically speaking, we still use a one-step-ahead prediction only that we extend the input space or train the system to predict values lying farther away.

It is also possible to combine different $\Delta t$: In case of the traffic jam prediction for a Monday the values of the last few days could be used as data input *in addition* to the values of the previous Mondays. Thus, we use the last values of several periods, in this case the values of a weekly and a daily period. We could also include an annual period in the form of the beginning of the holidays (for sure, everyone of us has already spent a lot of time on the highway because he forgot the beginning of the holidays).

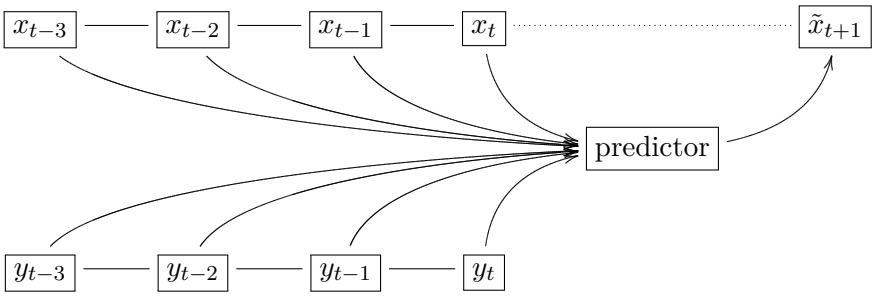



**Figure B.5:** Representation of the heterogeneous one-step-ahead prediction. Prediction of a time series under consideration of a second one.

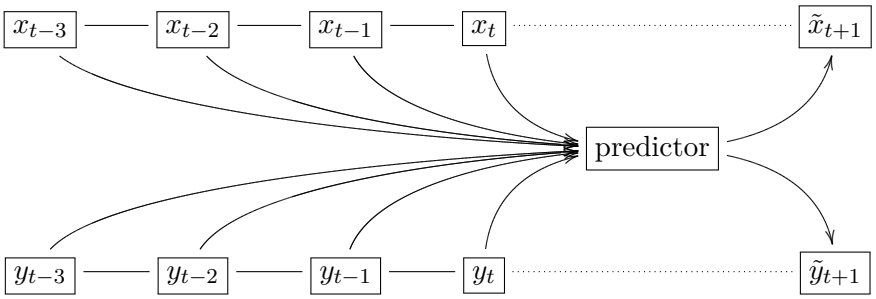



**Figure B.6:** Heterogeneous one-step-ahead prediction of two time series at the same time.

### B.4.2 Heterogeneous prediction

Another prediction approach would be to predict the future values of a single time series out of several time series, if it is assumed that the additional time series is related to the future of the first one (***heterogeneous one-step-ahead prediction***, fig. B.5).

If we want to predict two outputs of two related time series, it is certainly possible to perform two parallel one-step-ahead predictions (analytically this is done very often because otherwise the equations would become very confusing); or in case of the neural networks an additional output neuron is attached and the knowledge of both time series is used for both outputs (fig. B.6).

You'll find more and more general material on time series in [WG94].

## B.5 Remarks on the prediction of share prices

Many people observe the changes of a share price in the past and try to conclude the future from those values in order to benefit from this knowledge. Share prices are discontinuous and therefore they are principally difficult functions. Furthermore, the functions can only be used for discrete values – often, for example, in a daily rhythm (including the maximum and minimum values per day, if we are lucky) with the daily variations certainly being eliminated. But this makes the whole thing even more difficult.

There are *chartists*, i.e. people who look at many diagrams and decide by means of a lot of background knowledge and decade-long experience whether the equities should be bought or not (and often they are very successful).

Apart from the share prices it is very interesting to predict the exchange rates of currencies: If we exchange 100 Euros into Dollars, the Dollars into Pounds and the Pounds back into Euros it could be possible that we will finally receive 110 Euros. But once found out, we would do this more often and thus we would change the exchange rates into a state in which such an increasing circulation would no longer be possible (otherwise we could produce money by generating, so to speak, a financial perpetual motion machine.

At the stock exchange, successful stock and currency brokers raise or lower their thumbs – and thereby indicate whether in their opinion a share price or an exchange rate will increase or decrease. Mathematically speaking, they indicate the first bit (sign) of the first derivative of the exchange rate. In that way excellent worldclass brokers obtain success rates of about 70%.

In Great Britain, the heterogeneous one-step-ahead prediction was successfully used to increase the accuracy of such predictions to 76%: In addition to the time series of the values *indicators* such as the oil price in Rotterdam or the US national debt were included.

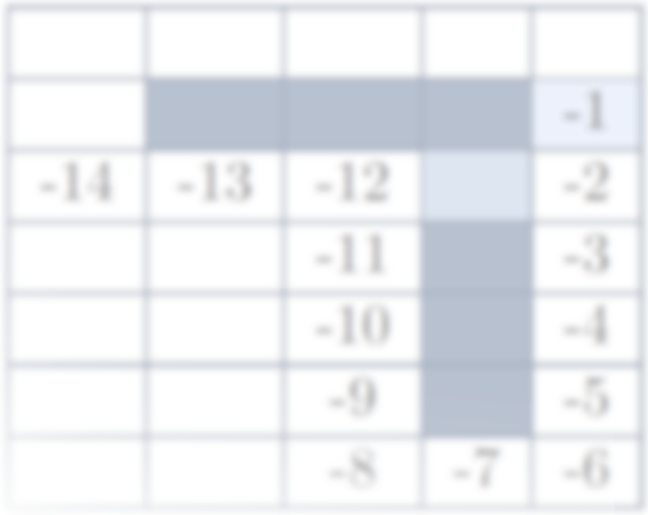
This is just an example to show the magnitude of the accuracy of stock-exchange evaluations, since we are still talking only about the first bit of the first derivation! We still do not know how strong the expected increase or decrease will be and also whether the effort will pay off: Probably, one wrong prediction could nullify the profit of one hundred correct predictions.

How can neural networks be used to predict share prices? Intuitively, we assume that future share prices are a function of the previous share values.

But this assumption is wrong: Share prices are no function of their past values, but a *function of their assumed future value*. We do not buy shares because their values have been increased during the last days, but because we *believe* that they will futher increase tomorrow. If, as a consequence, many people buy a share, they will boost the price. Therefore their assumption was right – a ***self-fulfilling prophecy*** has been generated, a phenomenon long known in economics.

The same applies the other way around: We sell shares because we believe that *tomorrow* the prices will decrease. This will beat down the prices the next day and generally even more the day after the next.

Again and again some software appears which uses scientific key words such as "neural networks" to purport that it is capable to predict where share prices are going. Do not buy such software! In addition to the aforementioned scientific exclusions there is one simple reason for this: If these tools work – why should the manufacturer sell them? Normally, useful economic knowledge is kept secret. If we knew a way to definitely gain wealth by means of shares, we would earn our millions by using this knowledge instead of selling it for 30 euros, wouldn't we?




# Appendix C

# Excursus: reinforcement learning

What if there were no training samples but it would nevertheless be possible to evaluate how well we have learned to solve a problem? Let us examine a learning paradigm that is situated between supervised and unsupervised learning.

I now want to introduce a more exotic approach of learning – just to leave the usual paths. We know learning procedures in which the network is exactly told what to do, i.e. we provide exemplary output values. We also know learning procedures like those of the self-organizing maps, into which only input values are entered.

Now we want to explore something in-between: The learning paradigm of reinforcement learning – *reinforcement learning* according to Sutton and Barto [SB98].

Reinforcement learning in itself is no neural network but only one of the three learning paradigms already mentioned in chapter 4. In some sources it is counted among the supervised learning procedures since a feedback is given. Due to its very rudimentary feedback it is reasonable to separate it from the supervised learning procedures – apart from the fact that there are no training samples at all.

While it is generally known that procedures such as backpropagation cannot work in the human brain itself, reinforcement learning is usually considered as being biologically more motivated.

The term ***reinforcement learning*** comes from cognitive science and psychology and it describes the learning system of carrot and stick, which occurs everywhere in nature, i.e. learning by means of good or bad experience, reward and punishment. But there is no learning aid that exactly explains what we have to do: We only receive a total result for a process (Did we win the game of chess or not? And how sure was this victory?), but no results for the individual intermediate steps.

For example, if we ride our bike with worn tires and at a speed of exactly $21,5\frac{km}{h}$ through a turn over some sand with a grain size of 0.1mm, on the average, then nobody could tell us exactly which handlebar angle we have to adjust or, even worse, how strong the great number of muscle parts in our arms or legs have to contract for this. Depending on whether we reach the end of the curve unharmed or not, we soon have to face the *learning experience*, a feedback or a *reward*, be it good or bad. Thus, the reward is very simple - but on the other hand it is considerably easier to obtain. If we now have tested different velocities and turning angles often enough and received some rewards, we will get a feel for what works and what does not. The aim of reinforcement learning is to maintain exactly this feeling.

Another example for the quasi-impossibility to achieve a sort of cost or utility function is a tennis player who tries to maximize his athletic success on the long term by means of complex movements and ballistic trajectories in the three-dimensional space including the wind direction, the importance of the tournament, private factors and many more.

To get straight to the point: Since we receive only little feedback, reinforcement learning often means *trial and error* – and therefore it is very slow.

## C.1 System structure

Now we want to briefly discuss different sizes and components of the system. We will define them more precisely in the following sections. Broadly speaking, reinforcement learning represents the mutual interaction between an *agent* and an *environmental system* (fig. C.2).

The agent shall solve some problem. He could, for instance, be an autonomous robot that shall avoid obstacles. The agent performs some actions within the environment and in return receives a feedback from the environment, which in the following is called *reward*. This cycle of action and reward is characteristic for reinforcement learning. The agent influences the system, the system provides a reward and then changes.

The reward is a real or discrete scalar which describes, as mentioned above, how well we achieve our aim, but it does not give any guidance *how* we can achieve it. The aim is always to make the sum of rewards as high as possible on the long term.

### C.1.1 The gridworld

As a learning example for reinforcement learning I would like to use the so-called ***gridworld***. We will see that its structure is very simple and easy to figure out and therefore reinforcement is actually not necessary. However, it is very suitable for representing the approach of reinforcement learning. Now let us exemplary define the individual components of the reinforcement system by means of the gridworld. Later, each of these components will be examined more exactly.

**Environment:** The gridworld (fig. C.1 on the following page) is a simple, discrete world in two dimensions which in the following we want to use as *environmental system*.

**Agent:** As an *Agent* we use a simple robot being situated in our gridworld.

**State space:** As we can see, our gridworld has $5 \times 7$ fields with 6 fields being unaccessible. Therefore, our agent can occupy 29 positions in the grid world. These positions are regarded as *states* for the agent.

**Action space:** The *actions* are still missing. We simply define that the robot could move one field up or down, to the right or to the left (as long as there is no obstacle or the edge of our gridworld).

**Task:** Our agent's task is to leave the gridworld. The exit is located on the right of the light-colored field.

**Non-determinism:** The two obstacles can be connected by a "door". When the door is closed (lower part of the illustration), the corresponding field is inaccessible. The position of the door cannot change during a cycle but only between the cycles.

We now have created a small world that will accompany us through the following learning strategies and illustrate them.

### C.1.2 Agent und environment

Our aim is that the agent learns what happens by means of the reward. Thus, it is trained over, of and by means of a dynamic system, the ***environment***, in order to reach an aim. But what does learning mean in this context?

The ***agent*** shall learn *a mapping of situations* to actions (called *policy*), i.e. it shall learn what to do in which situation to achieve a certain (given) aim. The aim is simply shown to the agent by giving an award for the achievement.

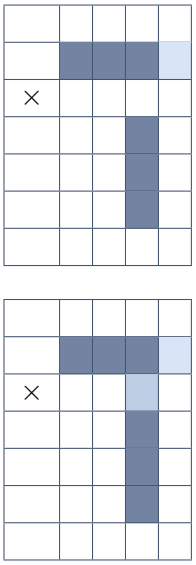


**Figure C.1:** A graphical representation of our gridworld. Dark-colored cells are obstacles and therefore inaccessible. The exit is located on the right side of the light-colored field. The symbol $\times$ marks the starting position of our agent. In the upper part of our figure the door is open, in the lower part it is closed.

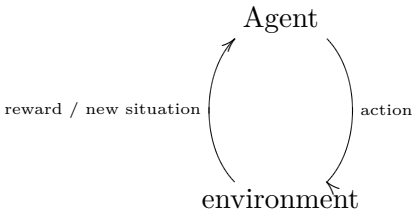



**Figure C.2:** The agent performs some actions within the environment and in return receives a reward.

Such an award must not be mistaken for the reward – on the agent's way to the solution it may sometimes be useful to receive a smaller award or a punishment when in return the longterm result is maximum (similar to the situation when an investor just sits out the downturn of the share price or to a pawn sacrifice in a chess game). So, if the agent is heading into the right direction towards the target, it receives a positive reward, and if not it receives no reward at all or even a negative reward (punishment). The award is, so to speak, the final sum of all rewards – which is also called *return*.

After having colloquially named all the basic components, we want to discuss more precisely which components can be used to make up our abstract reinforcement learning system.

**In the gridworld:** In the gridworld, the agent is a simple robot that should find the exit of the gridworld. The environment is the gridworld itself, which is a discrete gridworld.

**Definition C.1** (Agent)**.** In reinforcement learning the agent can be formally described as a mapping of the situation space $S$ into the action space $A(s_t)$. The meaning of situations $s_t$ will be defined later and should only indicate that the action space depends on the current situation.

$$\text{Agent: } S \to A(s_t) \tag{C.1}$$

**Definition C.2** (Environment)**.** The environment represents a stochastic mapping of an action $A$ in the current situation $s_t$ to a reward $r_t$ and a new situation $s_{t+1}$.

$$\text{Environment: } S \times A \to P(S \times r_t) \tag{C.2}$$

### C.1.3 States, situations and actions

As already mentioned, an agent can be in different ***states***: In case of the gridworld, for example, it can be in different positions (here we get a two-dimensional state vector).

For an agent is ist not always possible to realize all information about its current state so that we have to introduce the term ***situation***. A situation is a state *from the agent's point of view*, i.e. only a more or less precise *approximation of a state*.

Therefore, situations generally do not allow to clearly "predict" successor situations – even with a completely deterministic system this may not be applicable. If we knew all states and the transitions between them exactly (thus, the complete system), it would be possible to plan optimally and also easy to find an optimal policy (methods are provided, for example, by dynamic programming).

Now we know that reinforcement learning is an interaction between the agent and the system including ***actions*** $a_t$ and situations $s_t$. The agent cannot determine by itself whether the current situation is good or bad: This is exactly the reason why it receives the said reward from the environment.

**In the gridworld:** States are positions where the agent can be situated. Simply said, the situations equal the states in the gridworld. Possible actions would be to move towards north, south, east or west.

Situation and action can be vectorial, the reward is always a scalar (in an extreme case even only a binary value) since the aim of reinforcement learning is to get along with little feedback. A complex vectorial reward would equal a real teaching input.

By the way, the cost function should be minimized, which would not be possible, however, with a vectorial reward since we do not have any intuitive order relations in multi-dimensional space, i.e. we do not directly know what is better or worse.

**Definition C.3** (State)**.** Within its environment the agent is in a state. States contain any information about the agent within the environmental system. Thus, it is theoretically possible to clearly predict a successor state to a performed action within a deterministic system out of this godlike state knowledge.

**Definition C.4** (Situation)**.** Situations $s_t$ (here at time $t$) of a ***situation space*** $S$ are the agent's limited, approximate knowledge about its state. This approximation (about which the agent cannot even know how good it is) makes clear predictions impossible.

**Definition C.5** (Action)**.** Actions $a_t$ can be performed by the agent (whereupon it could be possible that depending on the situation another ***action space*** $A(S)$ exists). They cause state transitions and therefore a new situation from the agent's point of view.

### C.1.4 Reward and return

As in real life it is our aim to receive an award that is as high as possible, i.e. to maximize the sum of the expected ***rewards*** $r$, called ***return*** $R$, on the long term. For finitely many time steps[1] the rewards can simply be added:

$$R_t = r_{t+1} + r_{t+2} + \dots \tag{C.3}$$

$$= \sum_{x=1}^{\infty} r_{t+x} \tag{C.4}$$

Certainly, the return is only estimated here (if we knew all rewards and therefore the return completely, it would no longer be necessary to learn).

**Definition C.6** (Reward)**.** A reward $r_t$ is a scalar, real or discrete (even sometimes only binary) reward or punishment which the environmental system returns to the agent as reaction to an action.

**Definition C.7** (Return)**.** The return $R_t$ is the accumulation of all received rewards until time $t$.

#### C.1.4.1 Dealing with long periods of time

However, not every problem has an explicit target and therefore a finite sum (e.g. our agent can be a robot having the task to drive around again and again and to avoid obstacles). In order not to receive a diverging sum in case of an infinite series of reward estimations a weakening factor $0 < \gamma < 1$ is used, which weakens the influence of future rewards. This is not only useful if there exists no *target* but also if the target is very far away:

$$R_t = r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \dots \tag{C.5}$$

$$= \sum_{x=1}^{\infty} \gamma^{x-1} r_{t+x} \tag{C.6}$$

The farther the reward is away, the smaller is the influence it has in the agent's decisions.

1 In practice, only finitely many time steps will be possible, even though the formulas are stated with an infinite sum in the first place

Another possibility to handle the return sum would be a limited ***time horizon*** $\tau$ so that only $\tau$ many following rewards $r_{t+1}, \ldots, r_{t+\tau}$ are regarded:

$$R_t = r_{t+1} + \ldots + \gamma^{\tau-1} r_{t+\tau} \tag{C.7}$$

$$= \sum_{x=1}^{\tau} \gamma^{x-1} r_{t+x} \tag{C.8}$$

Thus, we divide the timeline into ***episodes***. Usually, one of the two methods is used to limit the sum, if not both methods together.

As in daily living we try to approximate our current situation to a *desired state*. Since it is not mandatory that only the next expected reward but the expected *total sum* decides what the agent will do, it is also possible to perform actions that, on short notice, result in a negative reward (e.g. the pawn sacrifice in a chess game) but will pay off later.

### C.1.5 The policy

After having considered and formalized some system components of reinforcement learning the actual aim is still to be discussed:

During reinforcement learning the agent learns a ***policy***

$$\Pi : S \to P(A),$$

Thus, it continuously adjusts a mapping of the situations to the probabilities $P(A)$, with which any action $A$ is performed in any situation $S$. A policy can be defined as a *strategy to select actions that would maximize the reward in the long term.*

**In the gridworld:** In the gridworld the policy is the strategy according to which the agent tries to exit the gridworld.

**Definition C.8** (Policy)**.** The policy $\Pi$ s a mapping of situations to probabilities to perform every action out of the action space. So it can be formalized as

$$\Pi : S \to P(A). \tag{C.9}$$

Basically, we distinguish between two policy paradigms: An ***open loop policy*** represents an open control chain and creates out of an initial situation $s_0$ a sequence of actions $a_0, a_1, \ldots$ with $a_i \neq a_i(s_i); i > 0$. Thus, in the beginning the agent develops a plan and consecutively executes it to the end without considering the intermediate situations (therefore $a_i \neq a_i(s_i)$, actions after $a_0$ do not depend on the situations).

**In the gridworld:** In the gridworld, an open-loop policy would provide a precise direction towards the exit, such as the way from the given starting position to (in abbreviations of the directions) `EEEEN`.

So an open-loop policy is a sequence of actions without interim feedback. A sequence of actions is generated out of a starting situation. If the system is known well and truly, such an open-loop policy can be used successfully and lead to useful results. But, for example, to know the chess game well and truly it would be necessary to try every possible move, which would be very time-consuming. Thus, for such problems we have to find an alternative to the open-loop policy, which incorporates the current situations into the action plan:

A ***closed loop policy*** is a closed loop, a function

$$\Pi : s_i \to a_i \text{ with } a_i = a_i(s_i),$$

in a manner of speaking. Here, the environment influences our action or the agent responds to the input of the environment, respectively, as already illustrated in fig. C.2. A closed-loop policy, so to speak, is a reactive plan to map current situations to actions to be performed.

**In the gridworld:** A closed-loop policy would be responsive to the current position and choose the direction according to the action. In particular, when an obstacle appears dynamically, such a policy is the better choice.

When selecting the actions to be performed, again two basic strategies can be examined.

#### C.1.5.1 Exploitation vs. exploration

As in real life, during reinforcement learning often the question arises whether the exisiting knowledge is only willfully exploited or new ways are also explored. Initially, we want to discuss the two extremes:

A ***greedy policy*** always chooses the way of the highest reward that can be determined in advance, i.e. the way of the highest known reward. This policy represents the ***exploitation approach*** and is very promising when the used system is already known.

In contrast to the exploitation approach it is the aim of the ***exploration approach*** to explore a system as detailed as possible so that also such paths leading to the target can be found which may be not very promising at first glance but are in fact very successful.

Let us assume that we are looking for the way to a restaurant, a safe policy would be to always take the way we already know, not matter how unoptimal and long it may be, and not to try to explore better ways. Another approach would be to explore shorter ways every now and then, even at the risk of taking a long time and being unsuccessful, and therefore finally having to take the original way and arrive too late at the restaurant.

In reality, often a combination of both methods is applied: In the beginning of the learning process it is researched with a higher probability while at the end more existing knowledge is exploited. Here, a static probability distribution is also possible and often applied.

**In the gridworld:** For finding the way in the gridworld, the restaurant example applies equally.

## C.2 Learning process

Let us again take a look at daily life. Actions can lead us from one situation into different subsituations, from each subsituation into further sub-subsituations. In a sense, we get a ***situation tree*** where links between the nodes must be considered (often there are several ways to reach a situation – so the tree could more accurately be referred to as a *situation graph*). he leaves of such a tree are the end situations of the system. The exploration approach would search the tree as thoroughly as possible and become acquainted with all leaves. The exploitation approach would unerringly go to the best known leave.

Analogous to the situation tree, we also can create an action tree. Here, the rewards for the actions are within the nodes. Now we have to adapt from daily life how we learn exactly.

### C.2.1 Rewarding strategies

Interesting and very important is the question for what a reward and what kind of reward is awarded since the design of the reward significantly controls system behavior. As we have seen above, there generally are (again as in daily life) various actions that can be performed in any situation. There are different strategies to evaluate the selected situations and to learn which series of actions would lead to the target. First of all, this principle should be explained in the following.

We now want to indicate some extreme cases as design examples for the reward:

A rewarding similar to the rewarding in a chess game is referred to as ***pure delayed reward***: We only receive the reward at the end of and not during the game. This method is always advantageous when we finally can say whether we were succesful or not, but the interim steps do not allow an estimation of our situation. If we win, then

$$r_t = 0 \ \forall t < \tau \tag{C.10}$$

as well as $r_\tau = 1$. If we lose, then $r_\tau = -1$. With this rewarding strategy a reward is only returned by the leaves of the situation tree.

***Pure negative reward***: Here,

$$r_t = -1 \ \forall t < \tau. \tag{C.11}$$

This system finds the most rapid way to reach the target because this way is automatically the most favorable one in respect of the reward. The agent receives punishment for anything it does – even if it does nothing. As a result it is the most inexpensive method for the agent to reach the target fast.

Another strategy is the ***avoidance strategy***: Harmful situations are avoided. Here,

$$r_t \in \{0, -1\}, \tag{C.12}$$

Most situations do not receive any reward, only a few of them receive a negative reward. The agent agent will avoid getting too close to such negative situations

Warning: Rewarding strategies can have unexpected consequences. A robot that is told "have it your own way but if you touch an obstacle you will be punished" will simply stand still. If standing still is also punished, it will drive in small circles. Reconsidering this, we will understand that this behavior optimally fulfills the return of the robot but unfortunately was not intended to do so.

Furthermore, we can show that especially small tasks can be solved better by means of negative rewards while positive, more differentiated rewards are useful for large, complex tasks.

For our gridworld we want to apply the pure negative reward strategy: The robot shall find the exit as fast as possible.

| | | | | |
|---|---|---|---|---|
| -6 | -5 | -4 | -3 | -2 |
| -7 | | | | -1 |
| -6 | -5 | -4 | -3 | -2 |
| -7 | -6 | -5 | | -3 |
| -8 | -7 | -6 | | -4 |
| -9 | -8 | -7 | | -5 |
| -10 | -9 | -8 | -7 | -6 |

| | | | | |
|---|---|---|---|---|
| -6 | -5 | -4 | -3 | -2 |
| -7 | | | | -1 |
| -8 | -9 | -10 | | -2 |
| -9 | -10 | -11 | | -3 |
| -10 | -11 | -10 | | -4 |
| -11 | -10 | -9 | | -5 |
| -10 | -9 | -8 | -7 | -6 |

**Figure C.3:** Representation of each optimal return per field in our gridworld by means of pure negative reward awarding, at the top with an open and at the bottom with a closed door.

### C.2.2 The state-value function

Unlike our agent we have a godlike view of our gridworld so that we can swiftly determine which robot starting position can provide which optimal return.

In figure C.3 these optimal returns are applied per field.

**In the gridworld:** The state-value function for our gridworld exactly represents such a function per situation (= position) with the difference being that here the function is unknown and has to be learned.

Thus, we can see that it would be more practical for the robot to be capable to *evaluate* the current and future situations. So let us take a look at another system component of reinforcement learning: the ***state-value function*** $V(s)$, which with regard to a policy $\Pi$ is often called $V_{\Pi}(s)$. Because whether a situation is bad often depends on the general behavior $\Pi$ of the agent.

A situation being bad under a policy that is searching risks and checking out limits would be, for instance, if an agent on a bicycle turns a corner and the front wheel begins to slide out. And due to its daredevil policy the agent would not brake in this

situation. With a risk-aware policy the same situations would look much better, thus it would be evaluated higher by a good state-value function

$V_\Pi(s)$ simply returns the value the current situation $s$ has for the agent under policy $\Pi$. Abstractly speaking, according to the above definitions, the value of the state-value function corresponds to the return $R_t$ (the expected value) of a situation $s_t$. $E_\Pi$ denotes the set of the expected returns under $\Pi$ and the current situation $s_t$.

$$V_\Pi(s) = E_\Pi\{R_t|s = s_t\}$$

**Definition C.9** (State-value function)**.** The state-value function $V_\Pi(s)$ has the task of determining the value of situations under a policy, i.e. to answer the agent's question of whether a situation $s$ is good or bad or how good or bad it is. For this purpose it returns the expectation of the return under the situation:

$$V_\Pi(s) = E_\Pi\{R_t|s = s_t\} \tag{C.13}$$

The optimal state-value function is called $V^*_\Pi(s)$.

Unfortunaely, unlike us our robot does not have a godlike view of its environment. It does not have a table with optimal returns like the one shown above to orient itself. The aim of reinforcement learning is that the robot generates its state-value function bit by bit on the basis of the returns of many trials and approximates the optimal state-value function $V^*$ (if there is one).

In this context I want to introduce two terms closely related to the cycle between state-value function and policy:

#### C.2.2.1 Policy evaluation

***Policy evaluation*** is the approach to try a policy a few times, to provide many rewards that way and to gradually accumulate a state-value function by means of these rewards.

#### C.2.2.2 Policy improvement

***Policy improvement*** means to improve a policy itself, i.e. to turn it into a new and better one. In order to improve the policy we have to aim at the return finally having a larger value than before, i.e. until we have found a shorter way to the restaurant and have walked it successfully

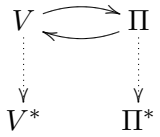



**Figure C.4:** The cycle of reinforcement learning which ideally leads to optimal $\Pi^*$ and $V^*$.

The principle of reinforcement learning is to realize an interaction. It is tried to evaluate how good a policy is in individual situations. The changed state-value function provides information about the system with which we again improve our policy. These two values lift each other, which can mathematically be proved, so that the final result is an optimal policy $\Pi^*$ and an optimal state-value function $V^*$ (fig. C.4). This cycle sounds simple but is very time-consuming.

At first, let us regard a simple, random policy by which our robot could slowly fulfill and improve its state-value function without any previous knowledge.

### C.2.3 Monte Carlo method

The easiest approach to accumulate a state-value function is mere trial and error. Thus, we select a randomly behaving policy which does not consider the accumulated state-value function for its random decisions. It can be proved that at some point we will find the exit of our gridworld by chance.

Inspired by random-based games of chance this approach is called ***Monte Carlo method***.

If we additionally assume a *pure negative reward*, it is obvious that we can receive an optimum value of $-6$ for our starting field in the state-value function. Depending on the random way the random policy takes values other (smaller) than $-6$ can occur for the starting field. Intuitively, we want to memorize only the *better* value for one state (i.e. one field). But here caution is advised: In this way, the learning procedure would *work only with deterministic systems*. Our door, which can be open or closed during a cycle, would produce oscillations for all fields and such oscillations would influence their shortest way to the target.

With the Monte Carlo method we prefer to use the learning rule[2]

$$V(s_t)_{\text{new}} = V(s_t)_{\text{alt}} + \alpha(R_t - V(s_t)_{\text{alt}}),$$

in which the update of the state-value function is obviously influenced by both the old state value and the received return ($\alpha$ is the learning rate). Thus, the agent gets some kind of memory, new findings always change the situation value just a little bit. An exemplary learning step is shown in fig. C.5 on the next page.

In this example, the computation of the state value was applied for only one single state (our initial state). It should be obvious that it is possible (and often done) to train the values for the states visited in-between (in case of the gridworld our ways to the target) at the same time. The result of such a calculation related to our example is illustrated in fig. C.6 on page 241.

The Monte Carlo method seems to be suboptimal and usually it is significantly slower than the following methods of reinforcement learning. But this method is the only one for which it can be mathematically proved that it works and therefore it is very useful for theoretical considerations.

**Definition C.10** (Monte Carlo learning)**.** Actions are randomly performed regardless of the state-value function and in the long term an expressive state-value function is accumulated by means of the following learning rule.

$$V(s_t)_{\text{new}} = V(s_t)_{\text{alt}} + \alpha(R_t - V(s_t)_{\text{alt}}),$$

### C.2.4 Temporal difference learning

Most of the learning is the result of experiences; e.g. walking or riding a bicycle without getting injured (or not), even mental skills like mathematical problem solving benefit a lot from experience and simple *trial and error*. Thus, we initialize our policy with arbitrary values – we try, learn and improve the policy due to *experience* (fig. C.7). In contrast to the Monte Carlo method we want to do this in a more directed manner.

Just as we learn from experience to react on different situations in different ways the ***temporal difference*** learning (abbreviated: ***TD learning***), does the same by training $V_{\Pi}(s)$ (i.e. the agent learns to estimate which situations are worth a lot and which are not). Again the current situation is identified with $s_t$, the following situations

2 The learning rule is, among others, derived by means of the *Bellman equation*, but this derivation is not discussed in this chapter.

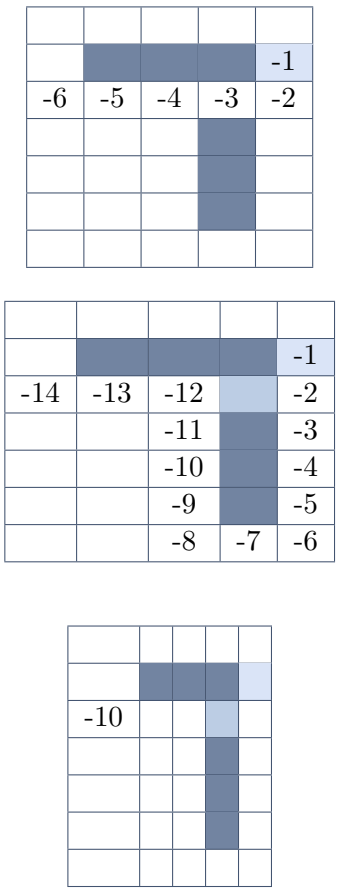



**Figure C.5:** Application of the Monte Carlo learning rule with a learning rate of $\alpha = 0.5$. Top: two exemplary ways the agent randomly selects are applied (one with an open and one with a closed door). Bottom: The result of the learning rule for the value of the initial state considering both ways. Due to the fact that in the course of time many different ways are walked given a random policy, a very expressive state-value function is obtained.

| | | | | |
|---|---|---|---|---|
| | | | | -1 |
| -10 | -9 | -8 | -3 | -2 |
| | | -11 | | -3 |
| | | -10 | | -4 |
| | | -9 | | -5 |
| | | -8 | -7 | -6 |

**Figure C.6:** Extension of the learning example in fig. C.5 in which the returns for intermediate states are also used to accumulate the state-value function. Here, the low value on the door field can be seen very well: If this state is possible, it must be very positive. If the door is closed, this state is impossible.

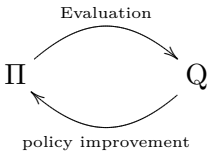



**Figure C.7:** We try different actions within the environment and as a result we learn and improve the policy.

with $s_{t+1}$ and so on. Thus, the learning formula for the state-value function $V_\Pi(s_t)$ is

$$\begin{aligned} V(s_t)_{\text{new}} =& V(s_t) \\ &+ \underbrace{\alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))}_{\text{change of previous value}} \end{aligned}$$

We can see that the change in value of the current situation $s_t$, which is proportional to the learning rate $\alpha$, is influenced by

- ▷ the received reward $r_{t+1}$,
- ▷ the previous return weighted with a factor $\gamma$ of the following situation $V(s_{t+1})$,
- ▷ the previous value of the situation $V(s_t)$.

**Definition C.11** (Temporal difference learning)**.** Unlike the Monte Carlo method, TD learning looks ahead by regarding the following situation $s_{t+1}$. Thus, the learning rule is given by

$$\begin{aligned} V(s_t)_{\text{new}} =& V(s_t) \\ &+ \underbrace{\alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))}_{\text{change of previous value}}. \end{aligned} \tag{C.14}$$

### C.2.5 The action-value function

Analogous to the state-value function $V_\Pi(s)$, the ***action-value function*** $Q_\Pi(s, a)$ is another system component of reinforcement learning, which evaluates a certain action $a$ under a certain situation $s$ and the policy $\Pi$.

**In the gridworld:** In the gridworld, the action-value function tells us how good it is to move from a certain field into a certain direction (fig. C.8 on the next page).

**Definition C.12** (Action-value function)**.** Like the state-value function, the action-value function $Q_\Pi(s_t, a)$ evaluates certain actions on the basis of certain situations under a policy. The optimal action-value function is called $Q^*_\Pi(s_t, a)$.

As shown in fig. C.9, the actions are performed until a target situation (here referred to as $s_\tau$) is achieved (if there exists a target situation, otherwise the actions are simply performed again and again).

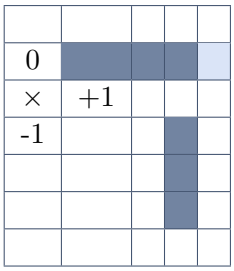



**Figure C.8:** Exemplary values of an action-value function for the position $\times$. Moving right, one remains on the fastest way towards the target, moving up is still a quite fast way, moving down is not a good way at all (provided that the door is open for all cases).

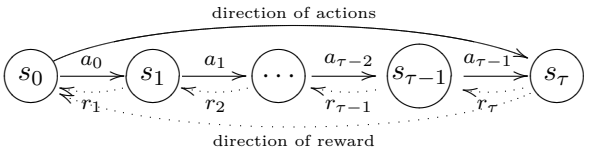



**Figure C.9:** Actions are performed until the desired target situation is achieved. Attention should be paid to numbering: Rewards are numbered beginning with 1, actions and situations beginning with 0 (This has simply been adopted as a convention).

### C.2.6 Q learning

This implies $Q_\Pi(s,a)$ as learning fomula for the action-value function, and – analogously to TD learning – its application is called ***Q learning***:

$$Q(s_t,a)_{\text{new}} = Q(s_t,a) + \underbrace{\alpha(r_{t+1} + \gamma \underbrace{\max_a Q(s_{t+1},a)}_{\text{greedy strategy}} - Q(s_t,a))}_{\text{change of previous value}}.$$

Again we break down the change of the current action value (proportional to the learning rate $\alpha$) under the current situation. It is influenced by

- the received reward $r_{t+1}$,
- the maximum action over the following actions weighted with $\gamma$ (Here, a greedy strategy is applied since it can be assumed that the best known action is selected. With TD learning, on the other hand, we do not mind to always get into the best known next situation.),
- the previous value of the action under our situation $s_t$ known as $Q(s_t,a)$ (remember that this is also weighted by means of $\alpha$).

Usually, the action-value function learns considerably faster than the state-value function. But we must not disregard that reinforcement learning is generally quite slow: The system has to find out itself what is good. But the advantage of Q learning is: $\Pi$ can be initialized arbitrarily, and by means of Q learning the result is *always* $Q^*$.

**Definition C.13** (Q learning)**.** Q learning trains the action-value function by means of the learning rule

$$Q(s_t,a)_{\text{new}} = Q(s_t,a) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a)). \tag{C.15}$$

and thus finds $Q^*$ in any case.

## C.3 Example applications

### C.3.1 TD gammon

***TD gammon*** is a very successful backgammon game based on TD learning invented by Gerald Tesauro. The situation here is the current configuration of the board. Anyone who has ever played backgammon knows that the situation space is huge (approx. $10^{20}$ situations). As a result, the state-value functions cannot be computed explicitly (particularly in the late eighties when TD gammon was introduced). The selected rewarding strategy was the *pure delayed reward*, i.e. the system receives the reward not before the end of the game and at the same time the reward is the return. Then the system was allowed to practice itself (initially against a backgammon program, then against an entity of itself). The result was that it achieved the highest ranking in a computer-backgammon league and strikingly disproved the theory that a computer programm is not capable to master a task better than its programmer.

### C.3.2 The car in the pit

Let us take a look at a car parking on a one-dimensional road at the bottom of a deep pit without being able to get over the slope on both sides straight away by means of its engine power in order to leave the pit. Trivially, the executable actions here are the possibilities to drive forwards and backwards. The intuitive solution we think of immediately is to move backwards, to gain momentum at the opposite slope and oscillate in this way several times to dash out of the pit.

The actions of a reinforcement learning system would be "full throttle forward", "full reverse" and "doing nothing".

Here, "everything costs" would be a good choice for awarding the reward so that the system learns fast how to leave the pit and realizes that our problem cannot be solved by means of mere forward directed engine power. So the system will slowly build up the movement.

The policy can no longer be stored as a table since the state space is hard to discretize. As policy a function has to be generated.

### C.3.3 The pole balancer

The ***pole balancer*** was developed by Barto, Sutton and Anderson.

Let be given a situation including a vehicle that is capable to move either to the right at full throttle or to the left at full throttle (bang bang control). *Only* these two actions can be performed, standing still is impossible. On the top of this car is hinged an upright pole that could tip over to both sides. The pole is built in such a way that it always tips over to one side so it never stands still (let us assume that the pole is rounded at the lower end).

The angle of the pole relative to the vertical line is referred to as $\alpha$. Furthermore, the vehicle always has a fixed position $x$ an our one-dimensional world and a velocity of $\dot{x}$. Our one-dimensional world is limited, i.e. there are maximum values and minimum values $x$ can adopt.

The aim of our system is to learn to steer the car in such a way that it can balance the pole, to prevent the pole from tipping over. This is achieved best by an avoidance strategy: As long as the pole is balanced the reward is 0. If the pole tips over, the reward is -1.

Interestingly, the system is soon capable to keep the pole balanced by tilting it sufficiently fast and with small movements. At this the system mostly is in the center of the space since this is farthest from the walls which it understands as negative (if it touches the wall, the pole will tip over).

#### C.3.3.1 Swinging up an inverted pendulum

More difficult for the system is the following initial situation: the pole initially hangs down, has to be swung up over the vehicle and finally has to be stabilized. In the literature this task is called ***swing up an inverted pendulum***.

## C.4 Reinforcement learning in connection with neural networks

Finally, the reader would like to ask why a text on "neural networks" includes a chapter about reinforcement learning.

The answer is very simple. We have already been introduced to supervised and unsupervised learning procedures. Although we do not always have an omniscient teacher

who makes unsupervised learning possible, this does not mean that we do not receive any feedback at all. There is often something in between, some kind of criticism or school mark. Problems like this can be solved by means of reinforcement learning.

But not every problem is that easily solved like our gridworld: In our backgammon example we have approx. $10^{20}$ situations and the situation tree has a large branching factor, let alone other games. Here, the tables used in the gridworld can no longer be realized as state- and action-value functions. Thus, we have to find approximators for these functions.

And which learning approximators for these reinforcement learning components come immediately into our mind? Exactly: neural networks.

## Exercises

**Exercise 19.** A robot control system shall be persuaded by means of reinforcement learning to find a strategy in order to exit a maze as fast as possible.

- ▷ What could an appropriate state-value function look like?
- ▷ How would you generate an appropriate reward?

Assume that the robot is capable to avoid obstacles and at any time knows its position $(x, y)$ and orientation $\phi$.

**Exercise 20.** Describe the function of the two components *ASE* and *ACE* as they have been proposed by Barto, Sutton and Anderson to control the *pole balancer*.

Bibliography: [BSA83].

**Exercise 21.** Indicate several "classical" problems of informatics which could be solved efficiently by means of reinforcement learning. Please give reasons for your answers.

# Bibliography


[And72] James A. Anderson. A simple neural network generating an interactive memory. *Mathematical Biosciences*, 14:197–220, 1972.

[APZ93] D. Anguita, G. Parodi, and R. Zunino. Speed improvement of the back-propagation on current-generation workstations. In *WCNN'93, Portland: World Congress on Neural Networks, July 11-15, 1993, Oregon Convention Center, Portland, Oregon*, volume 1. Lawrence Erlbaum, 1993.

[BSA83] A. Barto, R. Sutton, and C. Anderson. Neuron-like adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):834–846, September 1983.

[CG87] G. A. Carpenter and S. Grossberg. ART2: Self-organization of stable category recognition codes for analog input patterns. *Applied Optics*, 26:4919–4930, 1987.

[CG88] M.A. Cohen and S. Grossberg. Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *Computer Society Press Technology Series Neural Networks*, pages 70–81, 1988.

[CG90] G. A. Carpenter and S. Grossberg. ART 3: Hierarchical search using chemical transmitters in self-organising pattern recognition architectures. *Neural Networks*, 3(2):129–152, 1990.

[CH67] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[CR00] N.A. Campbell and JB Reece. *Biologie. Spektrum.* Akademischer Verlag, 2000.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

[DHS01] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern classification.* Wiley New York, 2001.

[Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, April 1990.

[Fah88] S. E. Fahlman. An empirical sudy of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, CMU, 1988.

[FMI83] K. Fukushima, S. Miyake, and T. Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):826–834, September/October 1983.

[Fri94] B. Fritzke. Fast learning with incremental RBF networks. *Neural Processing Letters*, 1(1):2–5, 1994.

[GKE01a] N. Goerke, F. Kintzler, and R. Eckmiller. Self organized classification of chaotic domains from a nonlinearattractor. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, volume 3, 2001.

[GKE01b] N. Goerke, F. Kintzler, and R. Eckmiller. Self organized partitioning of chaotic attractors for control. *Lecture notes in computer science*, pages 851–856, 2001.

[Gro76] S. Grossberg. Adaptive pattern classification and universal recoding, I: Parallel development and coding of neural feature detectors. *Biological Cybernetics*, 23:121–134, 1976.

[GS06] Nils Goerke and Alexandra Scherbart. Classification using multi-soms and multi-neural gas. In *IJCNN*, pages 3895–3902, 2006.

[Heb49] Donald O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Wiley, New York, 1949.

[Hop82] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proc. of the National Academy of Science, USA*, 79:2554–2558, 1982.

[Hop84] JJ Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences*, 81(10):3088–3092, 1984.

[HT85] JJ Hopfield and DW Tank. Neural computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

[Jor86] M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Conference of the Cognitive Science Society*, pages 531–546. Erlbaum, 1986.

[Kau90] L. Kaufman. Finding groups in data: an introduction to cluster analysis. In *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York, 1990.

[Koh72] T. Kohonen. Correlation matrix memories. *IEEEtC*, C-21:353–359, 1972.

[Koh82] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69, 1982.

[Koh89] Teuvo Kohonen. *Self-Organization and Associative Memory*. Springer-Verlag, Berlin, third edition, 1989.

[Koh98] T. Kohonen. The self-organizing map. *Neurocomputing*, 21(1-3):1–6, 1998.

[KSJ00] E.R. Kandel, J.H. Schwartz, and T.M. Jessell. *Principles of neural science*. Appleton & Lange, 2000.

[lCDS90] Y. le Cun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan Kaufmann, 1990.

[Mac67] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematics, Statistics and Probability, Vol. 1*, pages 281–296, 1967.

[MBS93] Thomas M. Martinetz, Stanislav G. Berkovich, and Klaus J. Schulten. 'Neural-gas' network for vector quantization and its application to time-series prediction. *IEEE Trans. on Neural Networks*, 4(4):558–569, 1993.

[MBW+10] K.D. Micheva, B. Busse, N.C. Weiler, N. O'Rourke, and S.J. Smith. Single-synapse analysis of a diverse synapse population: proteomic imaging methods and markers. *Neuron*, 68(4):639–653, 2010.

[MP43] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133, 1943.

[MP69] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, Mass, 1969.

[MR86] J. L. McClelland and D. E. Rumelhart. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 2. MIT Press, Cambridge, 1986.

[Par87] David R. Parker. Optimal algorithms for adaptive networks: Second order back propagation, second order direct propagation, and second order hebbian learning. In Maureen Caudill and Charles Butler, editors, *IEEE First International Conference on Neural Networks (ICNN'87)*, volume II, pages II–593–II–600, San Diego, CA, June 1987. IEEE.

[PG89] T. Poggio and F. Girosi. *A theory of networks for approximation and learning*. MIT Press, Cambridge Mass., 1989.

[Pin87] F. J. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59:2229–2232, 1987.

[PM47] W. Pitts and W.S. McCulloch. How we know universals the perception of auditory and visual forms. *Bulletin of Mathematical Biology*, 9(3):127–147, 1947.

[Pre94] L. Prechelt. Proben1: A set of neural network benchmark problems and benchmarking rules. *Technical Report*, 21:94, 1994.

[RB93] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[RD05] G. Roth and U. Dicke. Evolution of the brain and intelligence. *Trends in Cognitive Sciences*, 9(5):250–257, 2005.

[RHW86a] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, October 1986.

[RHW86b] David E. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, and the PDP research group., editors, *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1: Foundations*. MIT Press, 1986.

[Rie94] M. Riedmiller. Rprop - description and implementation details. Technical report, University of Karlsruhe, 1994.

[Ros58] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[Ros62] F. Rosenblatt. *Principles of Neurodynamics*. Spartan, New York, 1962.

[SB98] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[SG06] A. Scherbart and N. Goerke. Unsupervised system for discovering patterns in time-series, 2006.

[SGE05] Rolf Schatten, Nils Goerke, and Rolf Eckmiller. Regional and online learnable fields. In Sameer Singh, Maneesha Singh, Chidanand Apté, and Petra Perner, editors, *ICAPR (2)*, volume 3687 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2005.

[Ste61] K. Steinbuch. Die lernmatrix. *Kybernetik (Biological Cybernetics)*, 1:36–45, 1961.

[vdM73] C. von der Malsburg. Self-organizing of orientation sensitive cells in striate cortex. *Kybernetik*, 14:85–100, 1973.

[Was89] P. D. Wasserman. *Neural Computing Theory and Practice*. New York : Van Nostrand Reinhold, 1989.

[Wer74] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[Wer88] P. J. Werbos. Backpropagation: Past and future. In *Proceedings ICNN-88, San Diego*, pages 343–353, 1988.

[WG94] A.S. Weigend and N.A. Gershenfeld. *Time series prediction*. Addison-Wesley, 1994.

[WH60] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *Proceedings WESCON*, pages 96–104, 1960.

[Wid89] R. Widner. Single-stage logic. AIEE Fall General Meeting, 1960. *Wasserman, P. Neural Computing, Theory and Practice, Van Nostrand Reinhold*, 1989.

[Zel94] Andreas Zell. *Simulation Neuronaler Netze*. Addison-Wesley, 1994. German.

# List of Figures

# Index

## *



## A



## B



## C

## D



## E



## F

## G



## H



## I



## J



## K



## L

## M

## N



## O



## P

## Q



## R

## S



## T

## U



## V



## W