Machine learning with neural networks

September 3 (2020)

B. MEHLIG

Department of Physics
University of Gothenburg
Göteborg, Sweden 2019

PREFACE

These are lecture notes for my course on *Artificial Neural Networks* that I have given at Chalmers (FFR135) and Gothenburg University (FIM720). This course describes how neural networks are used in machine learning. The lecture notes cover Hopfield nets, perceptrons, deep learning, recurrent networks, reinforcement learning, and other supervised and unsupervised machine-learning algorithms.

When I first developed my lectures, my main source was the book by Hertz, Krogh, and Palmer [1]. Other sources were the book by Haykin [2], as well as the lecture notes of Horner [3]. My main sources for the Chapter on deep learning were the book by Goodfellow, Bengio & Courville [4], and the online-book by Nielsen [5].

I am grateful to Martin Čejka who typed the first version of my hand-written lecture notes and made most of the Figures, and to Erik Werner and Hampus Linander for their interest and their help in preparing Chapter 7. I would like to thank also Johan Fries and Oleksandr Balabanov for implementing the algorithms described in Section 7.4. Johan Fries and Marina Rafajlovic made many exam questions. Finally a large number of students – past and present – pointed out misprints and errors, and suggested improvements. I thank them all.

CONTENTS

1 Introduction

The term *neural networks* refers to networks of neurons in the mammalian brain. Neurons are its fundamental units of computation, they are connected together in networks to process data. This can be a very complex task, and the dynamics of neural networks in the mammalian brain in response to external stimuli can therefore be quite intricate. Inputs and outputs of each neuron vary as functions of time, in the form of so-called *spike trains*, but also the network itself changes. We learn and improve our data-processing capacities by establishing reconnections between neurons.

Neural-network algorithms are inspired by the architecture and the dynamics of networks of neurons in the brain. The algorithms use neuron models that are highly simplified, compared with real neurons. Nevertheless, the fundamental principle is the same: artificial neural networks learn by reconnection. Such networks can perform a multitude of information-processing tasks. They can learn to recognise structures in a set of training data and generalise what they have learnt to other data sets (*supervised learning*). A *training set* contains a list of input data sets, together with a list of the corresponding *target values* that encode the properties of the input data that the network is supposed to learn. To solve such association tasks by artificial neural networks can work well – when the new data sets are governed by the same principles that gave rise to the training data.

A prime example for a problem of this type is *object recognition* in images, for instance in the sequence of camera images of a self-driving car. Recently the use of neural networks for object recognition has exploded. There are several reasons for this strong interest. It is driven, first, by the acute need for such algorithms in industry. Second, there are now much better image data bases available for training the networks. Third, there is better hardware so that networks with many layers containing many neurons can be efficiently trained (*deep learning*) [4, 6].

Another task where neural networks excel is *machine translation*. These networks are dynamical (*recurrent networks*). They take an input sequence of words or sometimes single letters. As one feeds the inputs word by word, the network outputs the words in the translated sentence. Recurrent networks can be efficiently trained on large training sets of input sentences and their translations. Google translate works in this way [7].

Artificial neural networks are good at analysing large sets of high-dimensional data where it may be difficult to determine *a priori* which properties are of interest. In this case one often relies on *unsupervised learning* algorithms where the network learns without a training set. Instead the network organises the input data in relevant ways: neural networks can detect *familiarity* and *clusters* of input patterns, and

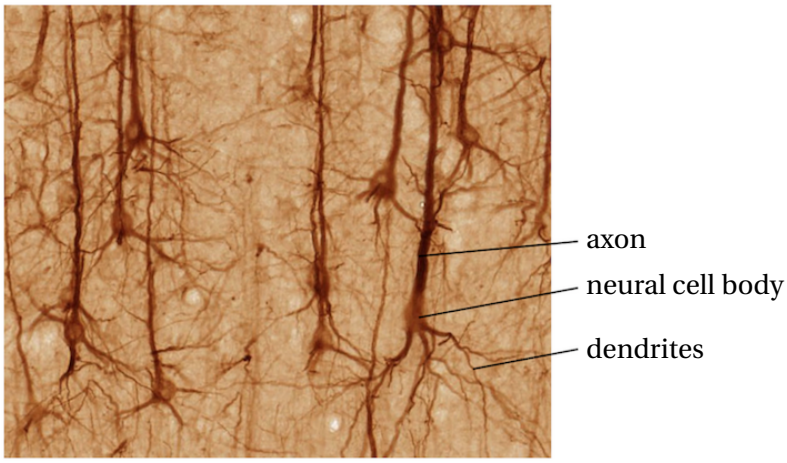


Figure 1.1: Neurons in the cerebral cortex (outer layer of the cerebrum, the largest and best developed part of the mammalian brain) of a *macaque*, an Asian monkey. Reproduced by permission of brainmaps.org [9] under the Creative Commons Attribution 3.0 License. The labels were added.

other structures in the input data. Unsupervised-learning algorithms work well when there is redundancy in the input data, and they they are particularly useful for high-dimensional data, where it is difficult to detect clusters or other data structures.

In many problems some information about targets is known, yet incomplete. For example, the network may receive a reward when it associaties the correct output with a given input, and a penalty otherwise. *Reinforcement learning*) allows the network to learn to produce outputs that receive positive feedback (reward) more frequently than those that trigger a penalty. In other words, rewarded outputs are reinforced. Related algorithms are used, for instance, in the software AlphaGo [8] that plays the game of go.

The different algorithms have much in common. They share the same building blocks: the neurons are modeled as linear threshold units (*McCulloch-Pitts neurons*), and the learning rules are similar (*Hebb's rule*). Closely related questions arise also regarding the network dynamics. A little bit of noise (not too much!) can improve the performance, and ensures that the long-time dynamics approaches a steady state. This allows to analyse the convergence of the algorithms using the *central-limit theorem*.

There are many connections to methods used in Mathematical Statistics, such as *Markov-chain Monte-Carlo* algorithms and *simulated annealing*. Certain unsupervised learning algorithms are related to *principal component analysis*, others to clustering algorithms such as *k-means clustering*.

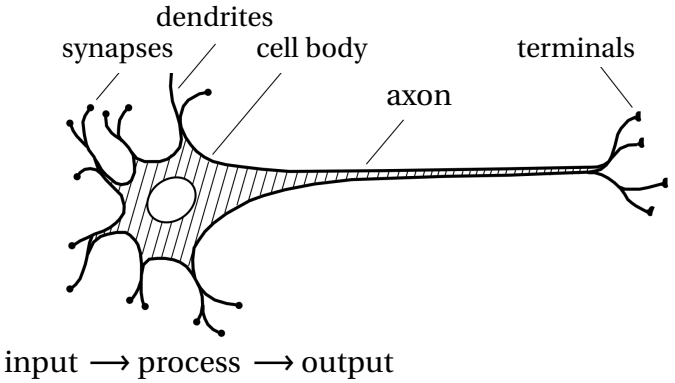


Figure 1.2: Schematic image of a neuron. Dendrites receive input in the form of electrical signals, via synapses. The signals are processed in the cell body of the neuron. The output travels from the neural cell body to other neurons through the axon.

1.1 Neural networks

The mammalian brain consists of different regions that perform different tasks. The *cerebral cortex* is the outer layer of the mammalian brain. We can think of it as a thin sheet (about 2 to 5 mm thick) that folds upon itself to increase its surface area. The cortex is the largest and best developed part of the Human brain. It contains large numbers of nerve cells, *neurons*. The Human cerebral cortex contains about 10^{10} neurons. They are linked together by nerve strands (*axons*) that branch and end in *synapses*. These synapses are the connections to other neurons. The synapses connect to *dendrites*, branched extensions from the neural cell body designed to receive input from other neurons in the form of electrical signals. A neuron in the Human brain may have thousands of synaptic connections with other neurons. The resulting network of connected neurons in the cerebral cortex is responsible for processing of visual, audio, and sensory data.

Figure 1.1 shows neurons in the cerebral cortex of the *macaque*, an Asian monkey. The image displays a silver-stained cross section through the cerebral cortex. The brown and black parts are the neurons. One can distinguish the cell bodies of the neural cells, their axons, and their dendrites.

Figure 1.2 shows a more schematic view of a neuron. Information is processed from left to right. On the left are the dendrites that receive signals and connect to the cell body of the neuron where the signal is processed. The right part of the Figure shows the axon, through which the output is sent to the dendrites of other neurons.

Information is transmitted as an electrical signal. Figure 1.3 shows an example of the time series of the electric potential for a pyramidal neuron in fish [10]. The time series consists of an intermittent series of electrical-potential spikes. Quiescent

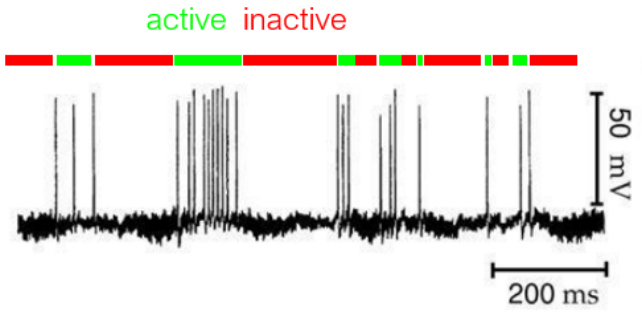


Figure 1.3: Spike train in electrosensory pyramidal neuron in fish (*eigenmannia*). Time series from Ref. [10]. Reproduced by permission of the publisher.

periods without spikes occur when the neuron is *inactive*, during spike-rich periods the neuron is *active*.

1.2 McCulloch-Pitts neurons

In artificial networks, the ways in which information is processed and signals are transferred are highly simplified. The model we use nowadays for the computational unit, the artificial neuron, goes back to McCulloch and Pitts [11]. Rosenblatt [12, 13] described how to connect such units in artificial neural networks to process information. He referred to these networks as *perceptrons*.

In its simplest form, the model for the artificial neuron has only two states, *active* or *inactive*. The model works as a linear threshold unit: it processes all input signals and computes an output. If the output exceeds a given threshold then the state of the neuron is said to be *active*, otherwise *inactive*. The model is illustrated in Figure 1.4. Neurons usually perform repeated computations, and one divides up time into discrete time steps $t = 0, 1, 2, 3, \ldots$. The state of neuron number j at time step t is denoted by

$$n_j(t) = \begin{cases} 0 & \text{inactive}, \\ 1 & \text{active}. \end{cases} \tag{1.1}$$

Given the signals $n_j(t)$, neuron number i computes

$$n_i(t+1) = \theta_{\mathrm{H}}\Big(\sum_j w_{ij} n_j(t) - \mu_i\Big). \tag{1.2}$$

As written, this computation is performed for all neurons i in parallel, and the outputs n_i are the inputs to all neurons at the next time step, therefore the outputs have the time argument $t+1$. These steps are repeated many times, resulting in

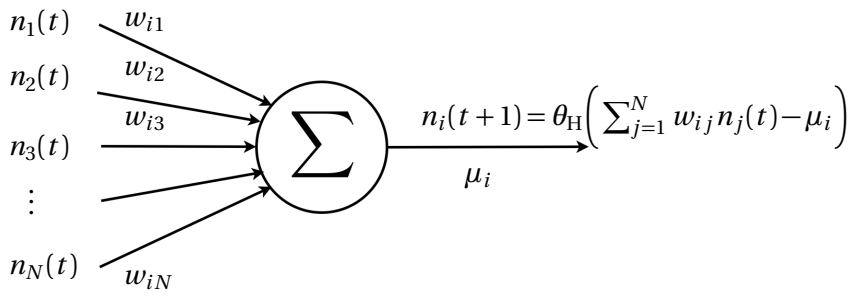


Figure 1.4: Schematic diagram of a McCulloch-Pitts neuron. The index of the neuron is i, it receives inputs from N neurons. The strength of the connection from neuron j to neuron i is denoted by w_{ij}. The *activation function* is the Heaviside function, $\theta_{\rm H}(b)$ (see text). The *threshold* value for neuron i is denoted by μ_i. The index $t = 0,1,2,3,\ldots$ labels the discrete time sequence of computation steps.

time series of the activity levels of all neurons in the network, referred to as *neural dynamics*.

Now consider the details of the computation step (1.2). The function $\theta_{\rm H}(b)$ an *activation function*. Its argument is called *local field*, $b_i(t) = \sum_j w_{ij}n_j(t) - \mu_i$. Since the neurons can only assume the states 0/1, the activation function is taken to be the *Heaviside function* (Figure 1.5):

$$\theta_{\rm H}(b) = \begin{cases} 0 & b < 0, \\ 1, & b \geq 0. \end{cases} \tag{1.3}$$

The Heaviside function is usually not defined at $b = 0$. To avoid problems in our computer algorithms we define $\theta_{\rm H}(0) = 1$.

Equation (1.2) shows that the neuron performs a weighted linear average of the inputs $n_j(t)$. The *weights* w_{ij} are called *synaptic* weights. Here the first index, i, refers to the neuron that does the computation, and j labels all neurons that connect to neuron i. The connection strengths between different pairs of neurons are in general different, reflecting different strengths of the synaptic couplings. When the value of w_{ij} is positive, we say that the coupling is called *excitatory*. When w_{ij} is negative, the connection is called *inhibitory*. When $w_{ij} = 0$ there is no connection. Finally, the threshold for neuron i is denoted by μ_i.

1.3 Other models for neural computation

The dynamics defined by Equations (1.1) and (1.2) is just a caricature of the time series of electrical signals in the cortex. For a start, the neural model described in the previous Section can only assume two states, instead of a continuous range of signal

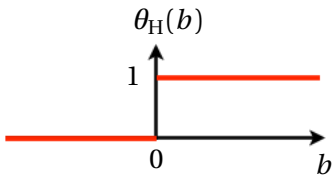


Figure 1.5: Heaviside function [Equation (1.3)].

strengths. While real neurons produce time series of spikes, the model gives rise to time sequences of zeros and ones. The two states, 0 and 1, are meant to model the inactive and active periods shown in Figure 1.3. For many computation tasks this is quite sufficient, and for our purposes it does not matter that the dynamics of real neurons is so different. The aim is not to model the neural dynamics in the brain, but to construct computation models inspired by real neural dynamics.

In the course of these lectures it will become apparent that the simplest model described above must be generalised to achieve certain tasks. Sometimes it is necessary to allow the neuron to respond continuously to its inputs. To this end one replaces Eq. (1.2) by

$$n_i(t+1) = g\Big(\sum_j w_{ij} n_j(t) - \mu_i\Big). \tag{1.4}$$

Here $g(b)$ is a continuous *activation function*. An example is shown in Figure 1.6. This choice dictates that the states assume continuous values too, not just the discrete values 0 and 1 as given in Equation (1.1).

Equations (1.2) and (1.4) are called *synchronous updating*, because all neurons are updated in parallel: at time step t all inputs $n_j(t)$ are stored. Then all neurons i are simultaneously updated using the stored inputs. An alternative is to choose one neuron, the one with index m say, and to update only this one:

$$n_i(t+1) = \begin{cases} g\big(\sum_j w_{mj} n_j(t) - \mu_m\big) & \text{for } i = m\,, \\ n_i(t) & \text{otherwise}\,. \end{cases} \tag{1.5}$$

This scheme is called *asynchronous updating*. If there are N neurons, then one synchronous step corresponds to N asynchronous steps, on average. This difference in time scales is not the only difference between synchronous and asynchronous updating. In general the two schemes yield different neural dynamics.

Different schemes for choosing neurons are used in asynchronous updating. One possibility is to arrange the neurons into a two-dimensional array and to update them one them by one, in a certain order. In the *typewriter scheme*, for example, one updates the neurons in the top row of the array first, from left to right, then the second row from left to right, and so forth.

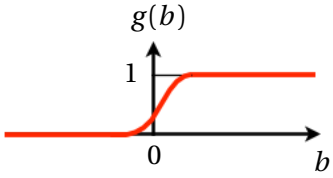


Figure 1.6: Continuous activation function.

A second possibility is to choose randomly which neuron to update. This introduces stochasticity into the neural dynamics. This is very important, and we will see that there are different ways of introducing stochasticity. Random asynchronous updating is one example. In many scientific problems it is advantageous to avoid stochasticity, when randomness is due to errors (multiplicative or additive noise) that diminish the performance of the system. In neural-network dynamics, by contrast, stochasticity is often helpful, as we shall see below.

1.4 Summary

Artificial neural networks use a highly simplified model for the fundamental computation unit, the neuron. In its simplest form, the model is just a binary threshold unit. The units are linked together by weights w_{ij}, and each unit computes a weighted average of its inputs. The network performs these computations in sequence. Usually one considers discrete sequences of computation time steps, $t = 0, 1, 2, 3, \ldots$. Either all neurons are updated simultaneously in one time step (synchronous updating), or only one chosen neuron is updated (asynchronous updating). Most neural-network algorithms are built using the model described in this Chapter.

PART I
HOPFIELD NETWORKS

The Hopfield network [14, 15] is an artificial neural network that can recognise or reconstruct images. Consider for example the binary images of digits in Figure 2.1. The images are stored in the network by assigning the weights w_{ij} in a certain way (called *Hebb's rule*). Then one feeds an image of a distorted version of one of the digits (Figure 2.2) to the network by assigning the initial states of the neurons in the network to the bits in the distorted image. The idea is that the neural-network dynamics converges to the correct undistorted digit. In this way the network can recognise the input as a distorted image of the correct digit (*retrieve* this digit). The point is that the network may recognise patterns with many bits very efficiently. This idea is quite old though. In the past such networks were used to perform pattern recognition tasks. Today there are more efficient algorithms for this purpose (Chapter 7).

Yet the first part of these lectures deals with Hopfield networks, for several reasons. First, Hopfield nets form the basis for more recent algorithms such as *restricted Boltzmann machines* [16] and *deep-belief networks* [2]. Second, all other neural-network algorithms discussed in these lectures are built from the same building blocks and use learning rules that are closely related to Hebb's rule. Third, Hopfield networks can solve optimisation problems, and the resulting algorithm is closely related to *Markov-chain Monte-Carlo algorithms* which are much used for a wide range of problems in Physics and Mathematical Statistics. Fourth, and most importantly, a certain degree of noise (not too much) can substantially improve the performance of Hopfield networks, and it is understood in detail why. The reason that so much is known about the role of noise in Hopfield networks is that they are closely related to stochastic systems studied in Physics, namely *random magnets* and *spin glasses*. The point is: understanding the effect of noise on the dynamics of Hopfield networks helps to analyse the performance of other neural-network models.

2 Deterministic Hopfield networks

2.1 Pattern recognition

As an example for a pattern-recognition task consider p images (*patterns*), each with N bits. The patterns could be the letters in the alphabet, or the digits shown in Figure 2.1. The different patterns are labeled by the index $\mu = 1, \ldots, p$. The bits of pattern μ are denoted by $x_i^{(\mu)}$. The index i labels the bits of a given pattern, it ranges from 1 to N. The bits are *binary*: they can take only the values 0 and 1, as illustrated in Figure 2.2. To determine the generic properties of the algorithm, one often turns

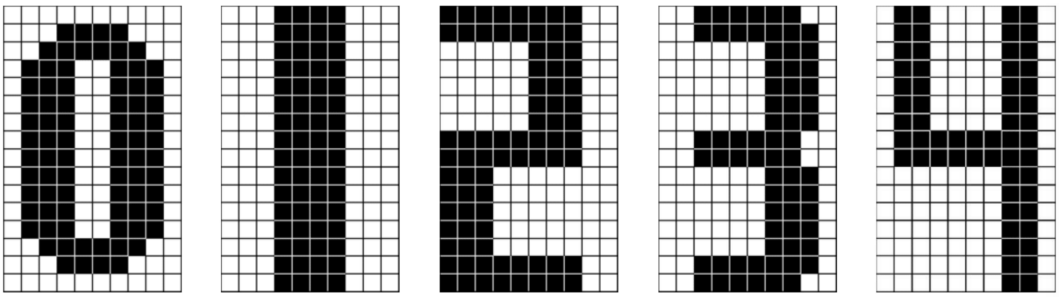


Figure 2.1: Binary representation of the digits 0 to 4. Each pattern has 10×16 pixels. Adapted from Figure 14.17 in Ref. [2]. The slightly peculiar shapes help the Hopfield net to distinguish the patterns [17].

to *random patterns* where each bit $x_i^{(\mu)}$ is chosen randomly. Each bit takes either value with probability $\frac{1}{2}$, and different bits (in the same and in different patterns) are independent. It is convenient to gather the bits of a pattern in a column vector

$$\boldsymbol{x}^{(\mu)} = \begin{bmatrix} x_1^{(\mu)} \\ x_2^{(\mu)} \\ \vdots \\ x_N^{(\mu)} \end{bmatrix}. \tag{2.1}$$

In this book, vectors are written in bold math font.

The task of the neural net is to recognise distorted patterns, to determine for instance that the pattern on the right in Figure 2.2 is a distorted version of the digit zero (left pattern in Figure 2.2). In practice one *stores p* patterns in the network, presents it with a distorted version of one of these patterns. The network retrieves the stored pattern that is most similar to the distorted one.

The formulation of the problem requires to define how similar two given patterns are to each other. One possibility is to use the *Hamming distance*. For patterns with 0/1 bits, the Hamming distance h_μ between the patterns $\boldsymbol{x}$ and $\boldsymbol{x}^{(\mu)}$ is defined as

$$h_\mu \equiv \sum_{i=1}^{N} \left[x_i^{(\mu)} (1 - x_i) + \left(1 - x_i^{(\mu)}\right) x_i \right]. \tag{2.2}$$

The Hamming distance equals the number of bits by which the patterns differ. Two patterns are identical if they have Hamming distance zero. For 0/1 patterns, Equation (2.2) is equivalent to:

$$\frac{h_\mu}{N} = \frac{1}{N} \sum_{i=1}^{N} \left(x_i^{(\mu)} - x_i \right)^2. \tag{2.3}$$

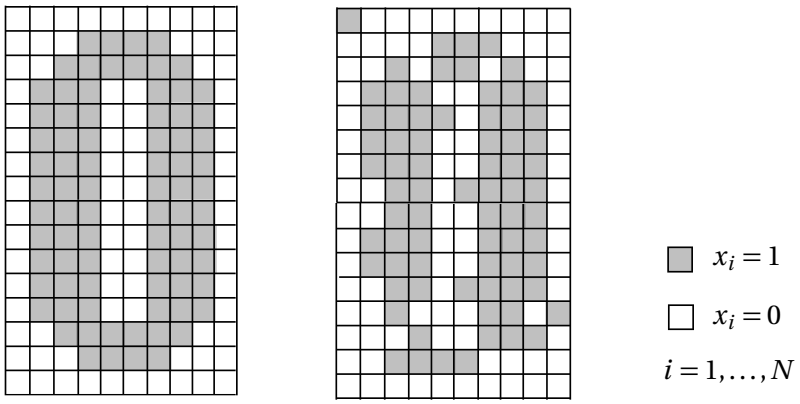


Figure 2.2: Binary image ($N = 160$) of the digit 0, and a distorted version of the same image.

This means that the Hamming distance is given by the mean-squared error, summed over all bits. Note that the Hamming distance does not refer to distortions by translations, rotations, or shearing. An improved version of the distance involves taking the minimum distance between the patterns subject to all possible translations, rotations, and so forth.

In summary, the task is to find the index ν for which the Hamming distance h_ν is minimal, $h_\nu \leq h_\mu$ for all $\mu = 1, \ldots, p$. How can one solve this task using a neural network? One feeds the distorted pattern $\boldsymbol{x}$ with bits x_i into the network by assigning $n_i(t=0) = x_i$. Assume that $\boldsymbol{x}$ is a distorted version of $\boldsymbol{x}^{(\nu)}$. Now the idea is to find a set of weights w_{ij} so that the network dynamics converges to the correct stored pattern:

$$n_i(t) \to x_i^{(\nu)} \qquad \text{as} \quad t \to \infty\,. \tag{2.4}$$

Which weights to choose depends on the patterns $\boldsymbol{x}^{(\mu)}$, so the weights must be functions of $\boldsymbol{x}^{(\mu)}$. We say that we *store* these patterns in the network by choosing the appropriate weights. If the network converges as in Equation (2.4), the pattern $\boldsymbol{x}^{(\nu)}$ is said to be an *attractor* of the dynamics.

2.2 Hopfield networks

Hopfield nets [14, 15] are networks of McCulloch-Pitts neurons designed to solve the pattern-recognition task described in the previous Section. The states of neuron i in the Hopfield network take the values

$$s_i = \begin{cases} -1 & \text{inactive}\,, \\ 1 & \text{active}\,, \end{cases} \tag{2.5}$$

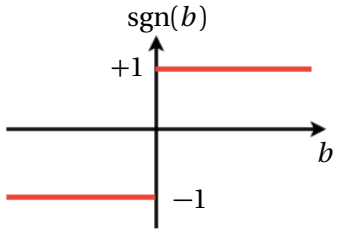


Figure 2.3: Signum function [Equation (2.7)].

instead of 0/1, because this simplifies the mathematical analysis as we shall see. The transformation from n_i to s_i is simply $s_i = 2n_i - 1$. The state or *configuration* of the net is represented by the vector

$$\boldsymbol{s} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_N \end{bmatrix}. \tag{2.6}$$

The space of all possible configurations (or states) of this network is called *configuration space* or *state space*. For the bits of the patterns we keep the symbol $x_i^{(\mu)}$, but it is important to remember that now $x_i^{(\mu)} = \pm 1$. To ensure that the s_i can only take the values ± 1, the activation function is taken to be the signum function (Figure 2.3).

$$\mathrm{sgn}(b) = \begin{cases} -1, & b < 0, \\ +1, & b \geq 0. \end{cases} \tag{2.7}$$

Usually the signum function is not defined at $b = 0$. To avoid problems in our computer algorithms we define $\mathrm{sgn}(0) = 1$.

s_i' (it could either be equal to or different from the state before the update, s_i). The asynchronous update rule takes the form

$$s_i(t+1) = \begin{cases} \mathrm{sgn}\left(\sum_j w_{mj} s_j(t) - \theta_m\right) & \text{for } i = m, \\ s_i(t) & \text{otherwise}. \end{cases} \tag{2.8}$$

The thresholds $\theta_i = 2\mu_i - \sum_j w_{ij}$ are chosen so that the update rules (1.5) and (2.8) are equivalent. A short-hand notation for the argument of the signum function is

$$b_m(t) = \sum_j w_{mj} s_j(t) - \theta_m, \tag{2.9}$$

sometimes called the *local field*. The synchronous update rule reads

$$s_i(t+1) = \mathrm{sgn}[b_i(t)] \tag{2.10}$$

where all bits are updated in parallel. Again, $b_i(t) = \sum_j w_{ij} s_j(t) - \theta_i$ is the local field.

Now we need a strategy for choosing the weights w_{ij}, so that the patterns $\boldsymbol{x}^{(\mu)}$ are attractors. If one feeds a pattern $\boldsymbol{x}$ close to $\boldsymbol{x}^{(\nu)}$ to the network, we want the network to converge to $\boldsymbol{x}^{(\nu)}$

$$\boldsymbol{s}(t=0) = \boldsymbol{x} \approx \boldsymbol{x}^{(\nu)}; \quad \boldsymbol{s}(t) \to \boldsymbol{x}^{(\nu)} \quad \text{as} \quad t \to \infty . \tag{2.11}$$

This means that the network succeeds in correcting a small number of errors. If the number of errors is too large, the network may converge to another pattern. The region in configuration space around pattern $\boldsymbol{x}^{(\nu)}$ in which all patterns converge to $\boldsymbol{x}^{(\nu)}$ is called the *region of attraction* of $\boldsymbol{x}^{(\nu)}$.

However, we shall see that it is in general very difficult to prove convergence according to Eq. (2.11). Therefore we try to answer a different question first: if one feeds one of the undistorted patterns $\boldsymbol{x}^{(\nu)}$, does the network recognise that it is one of the stored, undistorted patterns? The network should not make any changes to $\boldsymbol{x}^{(\nu)}$ because all bits are correct:

$$\boldsymbol{s}(t=0) = \boldsymbol{x}^{(\nu)}; \quad \boldsymbol{s}(t) = \boldsymbol{x}^{(\nu)} \quad \text{for all} \quad t = 0, 1, 2, \ldots . \tag{2.12}$$

Even this question is in general difficult to answer. We therefore consider a simple limit of the problem first, namely $p = 1$. There is only one pattern to recognize, $\boldsymbol{x}^{(1)}$. A suitable choice of weights w_{ij} is given by *Hebb's rule*

$$w_{ij} = \frac{1}{N} x_i^{(1)} x_j^{(1)} \quad \text{and} \quad \theta_i = 0 . \tag{2.13}$$

We say that the pattern $\boldsymbol{x}^{(1)}$ is *stored* in the network by assigning the weights w_{ij} using the rule (2.13). Note that the weights are symmetric, $w_{ij} = w_{ji}$. To check that the rule (2.13) does the trick, feed the pattern to the network by assigning $s_j(t=0) = x_j^{(1)}$, and evaluate Equation (2.8):

$$\sum_{j=1}^{N} w_{ij} x_j^{(1)} = \frac{1}{N} \sum_{j=1}^{N} x_i^{(1)} x_j^{(1)} x_j^{(1)} = \frac{1}{N} \sum_{j=1}^{N} x_i^{(1)} . \tag{2.14}$$

The last equality follows because $x_j^{(1)}$ can only take the values ± 1. The sum evaluates to N, so that

$$\operatorname{sgn}\left(\sum_{j=1}^{N} w_{ij} x_j^{(1)} \right) = x_i^{(1)} . \tag{2.15}$$

Recall that $x_i^{(\mu)} = \pm 1$, so that $\operatorname{sgn}(x_i^{(\mu)}) = x_i^{(\mu)}$. Comparing Equation (2.15) with the update rule (2.8) shows that the bits $x_j^{(1)}$ of the pattern $\boldsymbol{x}^{(1)}$ remain unchanged under

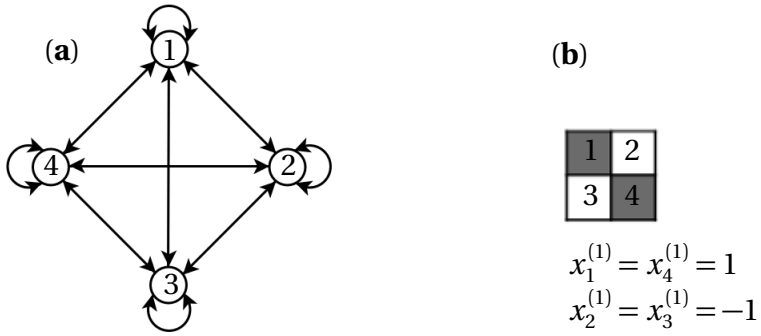


Figure 2.4: Hopfield network with $N = 4$ neurons. (**a**) Network layout. Neurons are represented as ◯. Arrows indicate symmetric connections. (**b**) Pattern $\boldsymbol{x}^{(1)\mathsf{T}} = [1,-1,-1,1]^{\mathsf{T}}$. Here T denotes the transpose of the column vector $\boldsymbol{x}^{(1)}$.

the update, as required by Eq. (2.12). The network recognises the pattern as a stored one, so Hebb's rule (2.13) does what we asked for.

But does the network correct small errors? In other words, is the pattern $\boldsymbol{x}^{(1)}$ an attractor [Eq. (2.11)]? This question cannot be answered in general. Yet in practice Hopfield nets work often very well! It is a fundamental insight that neural networks may work well although it is impossible to strictly prove that their dynamics converges to the correct solution.

To illustrate the difficulties consider an example, a Hopfield network with $p = 1$ and $N = 4$ (Figure 2.4). Store the pattern $\boldsymbol{x}^{(1)}$ shown in Figure 2.4 by assigning the weights w_{ij} using Hebb's rule (2.13). Now feed a distorted pattern $\boldsymbol{x}$ to the network that has a non-zero distance to $\boldsymbol{x}^{(1)}$:

$$h_1 = \frac{1}{4}\sum_{i=1}^{4}\left(x_i - x_i^{(1)}\right)^2 > 0\,. \tag{2.16}$$

The factor $\frac{1}{4}$ takes into account that the patterns take the values ± 1 and not $0/1$ as in Section 2.1. To feed the pattern to the network, one sets $s_i(t=0) = x_i$. Now iterate the dynamics using synchronous updating (2.10). Results for different distorted patterns are shown in Figure 2.5. We see that the first two distorted patterns (distance 1) converge to the stored pattern, cases (**a**) and (**b**). But the third distorted pattern does not [case (**c**)].

To understand this behaviour it is most convenient to analyse the synchronous dynamics using the *weight matrix*

$$\mathbb{W} = \frac{1}{N}\boldsymbol{x}^{(1)}\boldsymbol{x}^{(1)\mathsf{T}}\,. \tag{2.17}$$

Here $\boldsymbol{x}^{(1)\mathsf{T}}$ denotes the *transpose* of the column vector $\boldsymbol{x}^{(1)}$, so that $\boldsymbol{x}^{(1)\mathsf{T}}$ is a row vector. The standard rules for matrix multiplication apply also to column and row vectors,

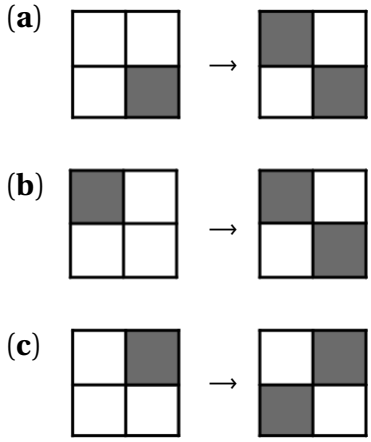


Figure 2.5: Reconstruction of a distorted image. Under synchronous updating (2.10) the first two distorted images (**a**) and (**b**) converge to the stored pattern $\boldsymbol{x}^{(1)}$, but pattern (**c**) does not.

they are just $N \times 1$ and $1 \times N$ matrices. This means that the product on the r.h.s. of Equation (2.17) is an $N \times N$ matrix. In the following, matrices with elements A_{ij} or B_{ij} are written as $\mathbb{A}$, $\mathbb{B}$, and so forth. The product in Equation (2.17) is also referred to as an *outer product*. The product

$$\boldsymbol{x}^{(1)\mathsf{T}}\boldsymbol{x}^{(1)} = \sum_{j=1}^{N} [x_j^{(1)}]^2 = N\,, \tag{2.18}$$

by contrast, is just a number (equal to N). The product (2.18) is called *scalar product*. It also is denoted by $\boldsymbol{x}^{(1)} \cdot \boldsymbol{x}^{(1)} = \boldsymbol{x}^{(1)\mathsf{T}}\boldsymbol{x}^{(1)}$. An excellent source for those not familiar with these terms from Linear Algebra (Figure 2.6) is Chapter 6 of the book by Mathews and Walker [18].

Using Equation (2.18) we see that $\mathbb{W}$ *projects* onto the vector $\boldsymbol{x}^{(1)}$,

$$\mathbb{W}\boldsymbol{x}^{(1)} = \boldsymbol{x}^{(1)}\,. \tag{2.19}$$

It follows from Equation (2.17) that the matrix $\mathbb{W}$ is *idempotent*:

$$\mathbb{W}^n = \mathbb{W} \quad \text{for } n = 1, 2, 3, \ldots\,. \tag{2.20}$$

Equations (2.19) and (2.20) mean that the network recognises the pattern $\boldsymbol{x}^{(1)}$ as the stored one. The pattern is not updated [Eq. (2.12)]. This example illustrates the general proof, Equations (2.14) and (2.15).

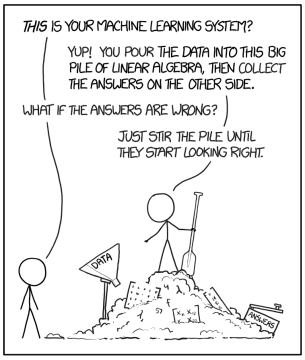


Figure 2.6: Reproduced from xkcd.com/1838 under the creative commons attribution-noncommercial 2.5 license.

Now consider the distorted pattern (**a**) in Figure 2.5. We feed this pattern to the network by assigning

$$\boldsymbol{s}(t=0)=\begin{bmatrix}-1\\-1\\-1\\1\end{bmatrix}. \tag{2.21}$$

To compute one step in the synchronous dynamics (2.10) we simply apply $\mathbb{W}$ to $\boldsymbol{s}(t=0)$. This is done in two steps, using the outer-product form (2.17) of the weight matrix. We first multiply $\boldsymbol{s}(t=0)$ with $\boldsymbol{x}^{(1)\mathsf{T}}$ from the left

$$\boldsymbol{x}^{(1)\mathsf{T}}\boldsymbol{s}(t=0)=\begin{bmatrix}1, & -1, & -1, & 1\end{bmatrix}\begin{bmatrix}-1\\-1\\-1\\1\end{bmatrix}=2, \tag{2.22}$$

and then we multiply this result with $\boldsymbol{x}^{(1)}$. This gives:

$$\mathbb{W}\boldsymbol{s}(t=0)=\tfrac{1}{2}\boldsymbol{x}^{(1)}. \tag{2.23}$$

The signum of the i-th component of the vector $\mathbb{W}\boldsymbol{s}(t=0)$ yields $s_i(t=1)$:

$$s_i(t=1)=\operatorname{sgn}\Big(\sum_{j=1}^{N} w_{ij}s_j(t=0)\Big)=x_i^{(1)}. \tag{2.24}$$

This means that the state of the network converges to the stored pattern, in one synchronous update. Since $\mathbb{W}$ is idempotent, the network stays there: the pattern $\boldsymbol{x}^{(1)}$ is an attractor. Case (**b**) in Figure 2.5 works in a similar way.

Now look at case (**c**), where the network fails to converge to the stored pattern. We feed this pattern to the network by assigning $\boldsymbol{s}(t=0)=[-1,1,-1,-1]^{\mathsf{T}}$. For one iteration of the synchronous dynamics we first evaluate

$$\boldsymbol{x}^{(1)\mathsf{T}}\boldsymbol{s}(0)=\begin{bmatrix}1, & -1, & -1, & 1\end{bmatrix}\begin{bmatrix}-1\\1\\-1\\-1\end{bmatrix}=-2\,. \tag{2.25}$$

It follows that

$$\mathbb{W}\boldsymbol{s}(t=0)=-\tfrac{1}{2}\boldsymbol{x}^{(1)}\,. \tag{2.26}$$

Using the update rule (2.10) we find

$$\boldsymbol{s}(t=1)=-\boldsymbol{x}^{(1)}\,. \tag{2.27}$$

Equation (2.20) implies that

$$\boldsymbol{s}(t)=-\boldsymbol{x}^{(1)}\quad\text{for}\quad t\geq 1\,. \tag{2.28}$$

Thus the network shown in Figure 2.4 has two attractors, the pattern $\boldsymbol{x}^{(1)}$ as well as the *inverted* pattern $-\boldsymbol{x}^{(1)}$. This is a general property of McCulloch-Pitts dynamics with Hebb's rule: if $\boldsymbol{x}^{(1)}$ is an attractor, then the pattern $-\boldsymbol{x}^{(1)}$ is an attractor too. But one ends up in the correct pattern $\boldsymbol{x}^{(1)}$ when more than half of the bits in $\boldsymbol{s}(t=0)$ are correct.

In summary we have shown that Hebb's rule (2.13) allows the Hopfield network to recognise a stored pattern: if we feed the stored pattern without any distortions to the network, then it does not change the bits. This does not mean, however, that the network recognises distorted patterns. It may or may not converge to the correct pattern. We expect that convergence is more likely when the number of wrong bits is small. If all distorted patterns near the stored pattern $\boldsymbol{x}^{(1)}$ converge to $\boldsymbol{x}^{(1)}$ then we say that $\boldsymbol{x}^{(1)}$ is an attractor. If $\boldsymbol{x}^{(1)}$ is an attractor, then $-\boldsymbol{x}^{(1)}$ is too.

When there are more than one patterns then Hebb's rule (2.13) must be generalised. A guess is to simply sum Equation (2.13) over the stored patterns:

$$w_{ij}=\frac{1}{N}\sum_{\mu=1}^{p}x_i^{(\mu)}x_j^{(\mu)}\quad\text{and}\quad\theta_i=0 \tag{2.29}$$

(Hebb's rule for $p>1$ patterns). As for $p=1$ the weight matrix is symmetric, $\mathbb{W}=\mathbb{W}^{\mathsf{T}}$, so that $w_{ij}=w_{ji}$. The diagonal weights are not zero in general. An alternative version of Hebb's rule [2] defines the diagonal weights to zero:

$$w_{ij}=\frac{1}{N}\sum_{\mu=1}^{p}x_i^{(\mu)}x_j^{(\mu)}\quad\text{for}\quad i\neq j\,,\quad w_{ii}=0\,,\quad\text{and}\quad\theta_i=0\,. \tag{2.30}$$

If we store only one pattern, $p = 1$, this modified rule Hebb's rule (2.30) satisfies Equation (2.12). In this Section we use Equation (2.30).

If we assign the weights according to Equation (2.30), does the network recognise distorted patterns? We saw in the previous Section that this question is difficult to answer in general, even for $p = 1$. Therefore we ask, first, whether the network recognises the stored pattern $\boldsymbol{x}^{(\nu)}$. The question is whether

$$\underbrace{\operatorname{sgn}\Big(\frac{1}{N}\sum_{j\neq i}\sum_{\mu} x_i^{(\mu)} x_j^{(\mu)} x_j^{(\nu)}\Big)}_{\equiv b_i^{(\nu)}} \stackrel{?}{=} x_i^{(\nu)}. \tag{2.31}$$

To check whether Equation (2.31) holds or not, we must repeat the calculation described on page 14. As a first step we evaluate the argument of the signum function,

$$b_i^{(\nu)} = \big(1-\tfrac{1}{N}\big) x_i^{(\nu)} + \frac{1}{N}\sum_{j\neq i}\sum_{\mu\neq\nu} x_i^{(\mu)} x_j^{(\mu)} x_j^{(\nu)}. \tag{2.32}$$

Here we have split the sum over the patterns into two contributions. The first term corresponds to $\mu = \nu$, where ν refers to the pattern that was fed to the network, the one that we want the network to recognise. The second term in Equation (2.32) contains the sum over the remaining patterns. For large N we can approximate $(1-\frac{1}{N}) \approx 1$. It follows that condition (2.31) is satisfied if the second term in (2.32) does not affect the sign of the r.h.s. of this Equation. This second term is called *cross-talk* term.

Whether adding the cross-talk term to $\boldsymbol{x}^{(\nu)}$ changes the signum of the r.h.s. of Equation (2.32) or not, depends on the stored patterns. Since the cross-talk term contains a sum over μ we may expect that this term does not matter if p is small enough. If this is true for all i and ν then all p stored patterns are recognised. Furthermore, by analogy with the example described in the previous Section, it is plausible that the stored patterns are then also attractors, so that slightly distorted patterns converge to the correct stored pattern. In other words, patterns close to $\boldsymbol{x}^{(\nu)}$ converge to $\boldsymbol{x}^{(\nu)}$ under the network dynamics (but this is not guaranteed).

For a more quantitative analysis of the effect of the cross-talk term we store patterns with random bits (*random patterns*). Different bits (different values of i and/or μ) are assigned ± 1 independently with equal probability:

$$\text{Prob}(x_i^{(\nu)} = \pm 1) = \tfrac{1}{2}. \tag{2.33}$$

This means that different patterns are *uncorrelated* because their *covariance* vanishes:

$$\langle x_i^{(\mu)} x_j^{(\nu)} \rangle = \delta_{ij}\delta_{\mu\nu}. \tag{2.34}$$

Here $\langle \cdots \rangle$ denotes an average over many realisations of random patterns, and δ_{ij} is the *Kronecker delta*, equal to unity if $i = j$ but zero otherwise. Note that it follows from Equation (2.33) that $\langle x_j^{(\mu)} \rangle = 0$.

We now ask: what is the probability that the cross-talk term changes the signum of the r.h.s. of Equation (2.32)? In other words, what is the probability that the network produces a wrong bit in one asynchronous update, if all bits were initially correct? The magnitude of the cross-talk term does not matter when it has the same sign as $x_i^{(\nu)}$. If it has a different sign, then the cross-talk term may matter. It does if its magnitude is larger than unity (the magnitude of $x_i^{(\nu)}$). To simplify the analysis one wants to avoid having to distinguish between the two cases, whether or not the cross-talk term has the same sign as $x_i^{(\nu)}$. To this end one defines:

$$C_i^{(\nu)} \equiv -x_i^{(\nu)} \underbrace{\frac{1}{N} \sum_{j \neq i} \sum_{\mu \neq \nu} x_i^{(\mu)} x_j^{(\mu)} x_j^{(\nu)}}_{\text{cross-talk term}} . \tag{2.35}$$

If $C_i^{(\nu)} < 0$ then the cross-talk term has same sign as $x_i^{(\nu)}$, so that the cross-talk term does not matter, adding it does not change the sign of $x_i^{(\nu)}$. If $0 < C_i^{(\nu)} < 1$ it does not matter either, only when $C_i^{(\nu)} > 1$. The network produces an error in updating neuron i if $C_i^{(\nu)} > 1$ for particular values i and ν: if initially $s_i(0) = x_i^{(\nu)}$, then the sign of the bit changes under the update although it should not – so that an error results.

The one step *error probability* $P_{\text{error}}^{t=1}$ is defined as the probability that an error occurs in one asynchronous attempt to update a single bit, given that initially all bits are correct. Therefore $P_{\text{error}}^{t=1}$ is given by:

$$P_{\text{error}}^{t=1} = \text{Prob}(C_i^{(\nu)} > 1) . \tag{2.36}$$

Since patterns and bits are identically distributed, $\text{Prob}(C_i^{(\nu)} > 1)$ does not depend on i or ν. Therefore $P_{\text{error}}^{t=1}$ does not carry any indices.

How does $P_{\text{error}}^{t=1}$ depend on the parameters of the problem, p and N? When both p and N are large we can use the *central-limit theorem* to answer this question. Since different bits/patterns are independent, we can think of $C_i^{(\nu)}$ as a sum of independent random numbers c_m that take the values -1 and $+1$ with equal probabilities,

$$C_i^{(\nu)} = -\frac{1}{N} \sum_{j \neq i} \sum_{\mu \neq \nu} x_i^{(\mu)} x_j^{(\mu)} x_j^{(\nu)} x_i^{(\nu)} = -\frac{1}{N} \sum_{m=1}^{(N-1)(p-1)} c_m . \tag{2.37}$$

There are $M = (N-1)(p-1)$ terms in the sum on the r.h.s. because terms with $\mu = \nu$ are excluded, and also those with $j = i$ [Equation (2.30)]. If we use Equation (2.29)

instead, then there is a correction to Equation (2.37) from the diagonal weights. For $p \ll N$ this correction is small.

When p and N are large, then the sum $\sum_m c_m$ contains a large number of independently identically distributed random numbers with mean zero and variance unity. It follows from the central-limit theorem that $\frac{1}{N}\sum_m c_m$ is Gaussian distributed with mean zero, and with variance

$$\sigma_C^2 = \frac{1}{N^2}\Big\langle \Big(\sum_{m=1}^{M} c_m\Big)^2 \Big\rangle = \frac{1}{N^2}\sum_{n=1}^{M}\sum_{m=1}^{M}\langle c_n c_m\rangle . \tag{2.38}$$

Here $\langle\cdots\rangle$ denotes an average over realisations of c_m. Since the random numbers c_m are independent for different indices, $\langle c_n c_m\rangle = \delta_{nm}$. So only the diagonal terms in the double sum contribute, summing up to $M \approx Np$. Therefore

$$\sigma_C^2 \approx \frac{p}{N} . \tag{2.39}$$

One way of showing that the distribution of $\sum_m c_m$ is approximately Gaussian distributed is to represent it in terms of *Bernoulli trials*. The sum $\sum_{m=1}^{M} c_m$ equals $2k-M$ where k is the number of occurrences $+1$ in the sum. Since the probability of $c_m = \pm 1$ is $\frac{1}{2}$, the probability of drawing k times $+1$ and $M-k$ times -1 is

$$P_{k,M} = \binom{M}{k}\left(\tfrac{1}{2}\right)^k\left(\tfrac{1}{2}\right)^{M-k} . \tag{2.40}$$

Here $\binom{M}{k} = M!/[k!(M-k)!]$ denotes the number of ways in which k occurrences of $+1$ can be distributed over M places. We expect that the quantity $2k-M$ is Gaussian distributed with mean zero and variance M. To demonstrate this, it is convenient to use the variable $z = (2k-M)/\sqrt{M}$, because it should then be Gaussian with mean zero and unit variance. To check whether this is the case, we substitute $k = \frac{M}{2} + \frac{\sqrt{M}}{2}z$ into Equation (2.40) and take the limit of large M using Stirling's approximation

$$n! \approx \mathrm{e}^{n\log n - n + \frac{1}{2}\log 2\pi n} . \tag{2.41}$$

Expanding $P_{k,M}$ to leading order in M^{-1} assuming that z remains of order unity gives $P_{k,M} = \sqrt{2/(\pi M)}\exp(-z^2/2)$. From $P(z)\mathrm{d}z = P(k)\mathrm{d}k$ it follows that $P(z) = (\sqrt{M}/2)P(k)$, so that $P(z) = (2\pi)^{-1/2}\exp(-z^2/2)$. So the distribution of z is Gaussian with zero mean and unit variance, as we intended to show.

In summary, the distribution of C is Gaussian

$$P(C) = (2\pi\sigma_C^2)^{-1/2}\exp[-C^2/(2\sigma_C^2)] \tag{2.42}$$

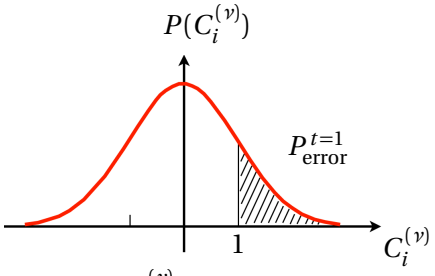


Figure 2.7: Gaussian distribution of $C_i^{(\nu)}$. The hashed area equals the one-step error probability $P_{\text{error}}^{t=1}$.

with mean zero and variance $\sigma_C^2 \approx \frac{p}{N}$, as illustrated in Figure 2.7. To determine $P_{\text{error}}^{t=1}$ [Equation (2.36)] we must integrate this distribution from 1 to ∞:

$$P_{\text{error}}^{t=1} = \frac{1}{\sqrt{2\pi}\sigma_C} \int_1^\infty \mathrm{d}C \; \mathrm{e}^{-\frac{C^2}{2\sigma_C^2}} = \frac{1}{2}\left[1 - \mathrm{erf}\left(\sqrt{\frac{N}{2p}}\right)\right]. \tag{2.43}$$

Here erf is the *error function* defined as [19]

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z \mathrm{d}x \; \mathrm{e}^{-x^2}. \tag{2.44}$$

Since erf(z) increases monotonically as z increases we conclude that $P_{\text{error}}^{t=1}$ increases as p increases, or as N decreases. This is expected: it is more difficult for the network to distinguish stored patterns when there are more of them. On the other hand, it is easier to differentiate stored patterns if they have more bits. We also see that the one-step error probability depends on p and N only through the combination

$$\alpha \equiv \frac{p}{N}. \tag{2.45}$$

The parameter α is called the *storage capacity* of the network. Figure 2.8 shows how $P_{\text{error}}^{t=1}$ depends on the storage capacity. For $\alpha = 0.2$ for example. the one-step error probability is slightly larger than 1%.

In the derivation of Equation (2.43) we assumed that the stored patterns are random with independent bits. Realistic patterns are not random. We nevertheless expect that $P_{\text{error}}^{t=1}$ describes the typical one-step error probability of the Hopfield network when p and N are large. However, it is straightforward to construct counter examples. Consider for example *orthogonal patterns*:

$$\boldsymbol{x}^{(\mu)} \cdot \boldsymbol{x}^{(\nu)} = 0 \quad \text{for} \quad \mu \neq \nu. \tag{2.46}$$

The cross-talk term vanishes in this case, so that $P_{\text{error}}^{t=1} = 0$.

More importantly, the error probability defined in this Section refers only to the initial update, the first iteration. What happens in the next iteration, and after many

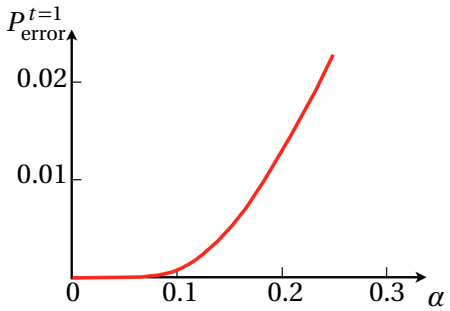


Figure 2.8: Dependence of the one-step error probability on the storage capacity α according to Equation (2.43), schematic.

iterations? Numerical experiments show that the error probability can be much higher in later iterations, because more errors tend to increase the probability of making another error. So the estimate $P_{\text{error}}^{t=1}$ is only a lower bound.

2.3 Energy function

Consider the long-time limit $t \to \infty$. Does the algorithm converge, as required by Equation (2.11)? This is perhaps the most important question in the analysis of neural-net algorithms, because an algorithm that does not converge to a meaningful solution is useless.

The standard way of analysing convergence of neural-net algorithms is to define an *energy function* that has a minimum at the desired solution, $\boldsymbol{s} = \boldsymbol{x}^{(\nu)}$ say. We monitor how the energy function changes as we iterate, and keep track of the smallest values of H encountered, to find the minimum. If we store only one pattern, $p = 1$, then a suitable energy function is

$$H = -\frac{1}{2N}\left(\sum_{i=1}^{N} s_i x_i^{(1)}\right)^2 . \tag{2.47}$$

This function is minimal when $\boldsymbol{s} = \boldsymbol{x}^{(1)}$, when $s_i = x_i^{(1)}$ for all i. It is customary to insert the factor $1/(2N)$, it does not change the fact that H is minimal at $\boldsymbol{s} = \boldsymbol{x}^{(1)}$.

A crucial point is that the McCulloch-Pitts dynamics (2.8) *converges* to the minimum. This follows from the fact that H cannot increase under (2.8). To prove this important property of the McCulloch-Pitts dynamics, we begin by evaluating the expression on the r.h.s. of Equation (2.47):

$$H = -\frac{1}{2N}\left(\sum_i s_i x_i^{(1)}\right)\left(\sum_j s_j x_j^{(1)}\right) = -\frac{1}{2}\sum_{ij}^{N} \underbrace{\left(\frac{1}{N} x_i^{(1)} x_j^{(1)}\right)}_{=w_{ij}} s_i s_j . \tag{2.48}$$

Using Hebb's rule (2.13) we find that the energy function (2.47) takes the form

$$H = -\frac{1}{2}\sum_{ij} w_{ij} s_i s_j \,. \tag{2.49}$$

This function has the same form as the energy function (or *Hamiltonian*) for certain physical models of magnetic systems consisting of interacting spins [20], where the interaction energy between spins s_i and s_j is $\frac{1}{2}(w_{ij}+w_{ji})s_i s_j$. Note that Hebb's rule (2.13) yields symmetric weights, $w_{ij} = w_{ji}$, and $w_{ii} > 0$. Setting the diagonal weights to zero does not change the fact that H is minimal at $\boldsymbol{s} = \boldsymbol{x}^{(1)}$ because $s_i^2 = 1$. This implies that the diagonal weights just give a constant contribution to H, independent of $\boldsymbol{s}$.

The second step is to show that H cannot increase under the McCulloch-Pitts dynamics (2.8). In this case we say that the energy function is a *Lyapunov function* of the McCulloch-Pitts dynamics. To demonstrate that the energy function is a Lyapunov function, choose a neuron m and update it according to Equation (2.8). We denote the updated state of neuron m by s'_m:

$$s'_m = \operatorname{sgn}\Big(\sum_j w_{mj} s_j\Big), \tag{2.50}$$

all other neurons remain unchanged. There are two possibilities, either $s'_m = s_m$ or $s'_m = -s_m$. In the first case H remains unchanged, $H' = H$. Here H' refers to the value of the energy function after the update (2.50). The other case is $s'_m = -s_m$. In this case the energy function changes by the amount

$$\begin{aligned} H' - H &= -\frac{1}{2}\sum_{j\neq m}(w_{mj}+w_{jm})(s'_m s_j - s_m s_j) - \frac{1}{2} w_{mm}(s'_m s'_m - s_m s_m) \\ &= \sum_{j\neq m}(w_{mj}+w_{jm}) s_m s_j \,. \end{aligned} \tag{2.51}$$

The sum goes over all neurons j that are connected to the neuron m, the one to be updated in Equation (2.50). Now if the weights are symmetric, $H'-H$ equals

$$H' - H = 2\sum_{j\neq m} w_{mj} s_m s_j = 2\sum_j w_{mj} s_m s_j - 2 w_{mm} \,. \tag{2.52}$$

Since the sign of $\sum_j w_{mj} s_j$ is that of $s'_m = -s_m$, and since $w_{mm} > 0$ it follows that

$$H' - H < 0 \,. \tag{2.53}$$

So either H remains constant under the McCulloch-Pits dynamics, or its value decreases. This means, in particular, that the McCulloch-Pitts dynamics must

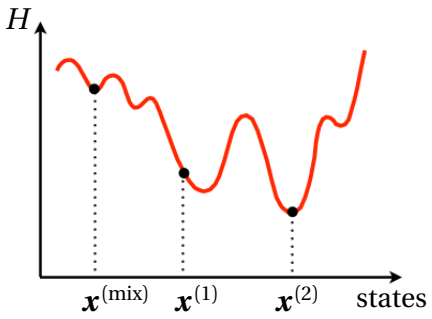


Figure 2.9: Minima in the energy function are attractors in state space. Not all minima correspond to stored patterns, and stored patterns need not correspond to minima.

converge to minima of the energy function. For the energy function () this means that the dynamics must either converge to the stored pattern or to its inverse. Both are attractors.[1] We assumed the thresholds to vanish, but the proof also works when the thresholds are not zero, in this case for the energy function

$$H = -\frac{1}{2}\sum_{ij} w_{ij} s_i s_j + \sum_i \theta_i s_i \,. \tag{2.54}$$

Up to now we considered only one stored pattern, $p = 1$. If we store more than one pattern [Hebb's rule (2.29)], the proof that (2.49) cannot increase under the McCulloch-Pitts dynamics works in the same way because no particular form of the weights w_{ij} was assumed, only that they must be symmetric, and that the diagonal weights must not be negative. In this case it follows that minima of the energy function must correspond to attractors, as illustrated schematically in Figure 2.9. The state space of the network – corresponding to all possible choices of $(s_1, \ldots s_N)$ – is illustrated drawn as a single axis, the x-axis. But when N is large, the state space is really very high dimensional.

However, when $p > 1$ some stored patterns may not be attractors. This follows from our analysis of the cross-talk term in Section 2.2 . If the cross-talk term causes errors for a certain stored pattern that is fed into the network, then this pattern is not located at a minimum of the energy function. To see this, note that Equations (2.29) and (2.49) give

$$H = -\frac{1}{2N}\sum_{\mu=1}^{p}\left(\sum_{i=1}^{N} s_i x_i^{(\mu)}\right)^2 . \tag{2.55}$$

[1]The derivation outlined here did not use the specific form of Hebb's rule (2.13), only that the weights are symmetric, and that $w_{mm} > 0$. It is clear that the argument works also when $w_{mm} = 0$. However, it does not work when $w_{mm} < 0$. In this case it is still true that H assumes a minimum at $\boldsymbol{s} = \boldsymbol{x}^{(1)}$, but H can increase under the update rule, so that convergence is not guaranteed. We must therefore require that the diagonal weights are not negative.

While the energy function defined in Equation (2.47) has a minimum at $\boldsymbol{x}^{(1)}$, Equation (2.55) need not have a minimum at $\boldsymbol{x}^{(1)}$ (or at any other stored pattern), because a maximal value of $\left(\sum_{i=1}^{N} s_i x_i^{(1)}\right)^2$ may be compensated by terms stemming from other patterns. This happens rarely when p is small (Section 2.2).

Conversely there may be minima that do not correspond to stored patterns. Such states are referred to as *spurious states*. The network may converge to spurious states, this is undesirable but inevitable. It occurs even when there is only one stored pattern, as we saw in Section 2.2: the McCulloch-Pitts dynamics may converge to the inverted pattern. You can see this also in Equations (2.47) and (2.55). If $\boldsymbol{s} = \boldsymbol{x}^{(1)}$ is a local minimum of H, then so is $\boldsymbol{s} = -\boldsymbol{x}^{(1)}$.

2.4 Spurious states

Stored patterns may be minima of the energy function (attractors), but they need not be. In addition there can be other local minima (spurious states), different from the stored patterns. For example, we saw in the previous Section that $-\boldsymbol{x}^{(1)}$ is an local minimum if $\boldsymbol{x}^{(1)}$ is a local minimum. This follows from the invariance of H under $\boldsymbol{s} \to -\boldsymbol{s}$.

There are other types of spurious states besides inverted patterns. An example are *mixed states, superpositions* of an odd number $2n+1$ of patterns. For $n = 1$, for example, the bits of a mixed state read:

$$x_i^{(\mathrm{mix})} = \mathrm{sgn}(\pm x_i^{(1)} \pm x_i^{(2)} \pm x_i^{(3)}). \tag{2.56}$$

The number of distinct mixed states increases as n increases. There are $2^{2n+1}\binom{p}{2n+1}$ mixed states that are superpositions of $2n+1$ out of p patterns, for $n = 1, 2, \ldots$ (Exercise 2.4).

It is difficult to determine under which circumstances the network dynamics converges to a certain mixed state. But we can at least check whether a mixed state is recognised by the network (although we do not want this to happen). As an example consider the mixed state

$$x_i^{(\mathrm{mix})} = \mathrm{sgn}(x_i^{(1)} + x_i^{(2)} + x_i^{(3)}). \tag{2.57}$$

To check whether this state is recognised, we must determine whether or not

$$\mathrm{sgn}\left(\frac{1}{N}\sum_{\mu=1}^{p}\sum_{j=1}^{N} x_i^{(\mu)} x_j^{(\mu)} x_j^{(\mathrm{mix})}\right) = x_i^{(\mathrm{mix})}, \tag{2.58}$$

$x_j^{(1)}$	$x_j^{(2)}$	$x_j^{(3)}$	$x_j^{(\mathrm{mix})}$	$s_j^{(1)}$	$s_j^{(2)}$	$s_j^{(3)}$
1	1	1	1	1	1	1
1	1	-1	1	1	1	-1
1	-1	1	1	1	-1	1
1	-1	-1	-1	-1	1	1
-1	1	1	1	-1	1	1
-1	1	-1	-1	1	-1	1
-1	-1	1	-1	1	1	-1
-1	-1	-1	-1	1	1	1

Table 2.1: Mixed states. Possible signs of $s_j^{(\mu)} = x_j^{(\mu)} x_j^{(\mathrm{mix})}$ for a given bit j (see text).

under the update (2.8) using Hebb's rule (2.29). To this end we split the sum in the usual fashion

$$\frac{1}{N}\sum_{\mu=1}^{p}\sum_{j=1}^{N} x_i^{(\mu)} x_j^{(\mu)} x_j^{(\mathrm{mix})} = \sum_{\mu=1}^{3} x_i^{(\mu)} \frac{1}{N}\sum_{j=1}^{N} x_j^{(\mu)} x_j^{(\mathrm{mix})} + \text{cross-talk term}. \tag{2.59}$$

Let us ignore the cross-talk term for the moment and check whether the first term reproduces $x_i^{(\mathrm{mix})}$. To make progress we assume random patterns [Equation (2.33)], and compute the probability that the sum on the r.h.s of Equation (2.59) yields $x_i^{(\mathrm{mix})}$. The sum over j on the r.h.s. of Equation (2.59) is an average of $s_j^{(\mu)} = x_j^{(\mu)} x_j^{(\mathrm{mix})}$. Table 2.1 lists all possible combinations of bits, and the corresponding values of $s_j^{(\mu)}$. We see that on average $\langle s_j^{(\mu)} \rangle = \frac{1}{2}$, so that

$$\frac{1}{N}\sum_{\mu=1}^{p}\sum_{j=1}^{N} x_i^{(\mu)} x_j^{(\mu)} x_j^{(\mathrm{mix})} = \frac{1}{2}\sum_{\mu=1}^{3} x_i^{(\mu)} + \text{cross-talk term}. \tag{2.60}$$

Neglecting the cross-talk term and taking the sgn-function we see that $\boldsymbol{x}^{(\mathrm{mix})}$ is reproduced. So mixed states such as (2.57) are recognised, at least for small α, and it may happen that the network converges to these states.

Finally, for large values of p there are local minima of H that are not correlated with any number of the stored patterns $x_j^{(\mu)}$. Such *spin-glass* states are discussed further in the book by Hertz, Krogh and Palmer [1].

2.5 Summary

We analysed how Hopfield networks recognise (or *retrieve*) patterns. Hopfield nets are networks of McCulloch-Pitts neurons. Their layout is defined by connection

Algorithm 1 pattern recognition with deterministic Hopfield net

1: store patterns $\boldsymbol{x}^{(\mu)}$ using Hebb's rule;
2: feed distorted pattern $\boldsymbol{x}$ into network by assigning $s_i(t=0) \leftarrow x_i$;
3: **for** $t = 1, \ldots, T$ **do**
4: choose a value of i and update $s_i(t) \leftarrow \operatorname{sgn}\big(\sum_j w_{ij} s_j(t-1)\big)$;
5: **end for**
6: read out pattern $\boldsymbol{s}(T)$;
7: end;

strengths (weights) w_{ij}, chosen according to Hebb's rule. The w_{ij} are symmetric, and the network is in general fully connected. Hebb's rule ensures that stored patterns are recognised, at least most of the time if the storage capacity is not too large. A single-step estimate for the error probability was given in Section 2.2. If one iterates several steps, the error probability is generally much larger, but it is difficult to evaluate. For stochastic Hopfield nets the steady-state error probability can be estimated more easily, because the dynamics converges to a definite steady state.

2.6 Exercises

2.1 Modified Hebb's rule. Show that the modified rule Hebb's rule (2.30) satisfies Equation (2.12) if we store only one pattern, for $p = 1$.

2.2 Orthogonal patterns. Show that the cross-talk term vanishes for orthogonal patterns, so that $P_{\text{error}}^{t=1} = 0$. Show that this works for both versions of Hebb's rule, (2.29) as well as (2.30).

2.3 Cross-talk term. Expression (2.37) for the cross-talk term was derived using modified Hebb's rule, Equation (2.30). How does Equation (2.37) change if you use the rule (2.29) instead? Show that the distribution of $C_i^{(\nu)}$ then acquires a non-zero mean, obtain an estimate for this mean value, and compute the one-step error probability. Show that your result approaches (2.43) for small values of α. Explain why your result is different from (2.43) for large α.

2.4 Mixed states. Explain why there are no mixed states that are superpositions of an even number of stored patterns. Show that there are $2^{2n+1}\binom{p}{2n+1}$ mixed states that are superpositions of $2n+1$ out of p patterns, for $n = 1, 2, \ldots$.

2.5 One-step error probability for mixed states. Write a computer program implementing the asynchronous deterministic dynamics of a Hopfield network to

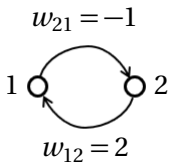


Figure 2.10: Neural net consisting of two neurons with asymmetric connections.

determine the one-step error probability for the mixed state (2.57). Plot how the one-step error probability depends on α for $N = 50$ and $N = 100$. Repeat this exercise for mixed patterns that are superpositions of the bits of 5 and 7 patterns.

2.6 Energy function. Figure 2.10 shows a network with two neurons with asymmetric weights, $w_{12} = 2$ and $w_{21} = -1$. Show that the energy function $H = -\frac{w_{12}+w_{21}}{2} s_1 s_2$ can increase under the deterministic McCulloch-Pitts rule $s_2' = \mathrm{sgn}(w_{21} s_1)$.

2.7 Higher-order Hopfield nets. Determine under which conditions the energy function $H = -\frac{1}{2}\sum_{ij} w_{ij}^{(2)} s_i s_j - \frac{1}{6}\sum_{ijk} w_{ijk}^{(3)} s_i s_j s_k$ is a Lyapunov function for the modified McCulloch-Pitts dynamics $s_m' = \mathrm{sgn}(b_m)$ with $b_m = \partial H/\partial s_m$.

2.8 Hebb's rule and energy function for $0/1$ units. Write down Hebb's rule for $0/1$ units and show that if one stores only one pattern, then this pattern is recognised. Show that $H = -\frac{1}{2}\sum_{ij} w_{ij} n_i n_j + \sum_i \mu_i n_i$ cannot increase under the asynchronous update rule $n_m' = \theta_{\mathrm{H}}(b_m)$ with $b_m = \sum_j w_{mj} n_j - \mu_m$ (it is assumed that the weights are symmetric, and that $w_{jj} \geq 0$). See Ref. [21].

2.9 Energy function and synchronous dynamics. Analyse how the energy function (2.49) changes under the synchronous dynamics (2.10). Assume that the weights are symmetric, and that the diagonal weights are zero. Is the energy function a Lyapunov function?

2.10 Continuous Hopfield net. Hopfield [22] also analyed a version of his model with continuous-time dynamics. Here we use $\tau \frac{\mathrm{d}}{\mathrm{d}t} n_i = -n_i + g(\sum_j w_{ij} n_j - \theta_i)$ with $g(b) = (1+\mathrm{e}^{-b})^{-1}$ (this dynamical equation is slightly different from the one used by Hopfield [22]). Show that the energy function $E = -\frac{1}{2}\sum_{ij} w_{ij} n_i n_j + \sum_i \theta_i n_i + \sum_i \int_0^{n_i} \mathrm{d}n\, g^{-1}(n)$ cannot increase under the network dynamics if the weights are symmetric. It is not necessary to assume that $w_{ii} \geq 0$.

2.11 Hopfield network with four neurons. The pattern shown in Fig. 2.11 is stored in a Hopfield network using Hebb's rule $w_{ij} = \frac{1}{N} x_i^{(1)} x_j^{(1)}$. There are 2^4 four-bit patterns. Apply each of these to the Hopfield network, and perform one synchronous

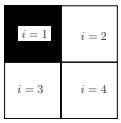


Figure 2.11: The pattern $\boldsymbol{x}^{(1)}$ has $N=4$ bits, $x_1^{(1)}=1$, and $x_i^{(1)}=-1$ for $i=2,3,4$. Exercise 2.11.

update. List the patterns you obtain and discuss your results.

2.12 Recognising letters with a Hopfield network. Five patterns are shown in Figure 2.12, each with $N=32$ bits. Store the patterns $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$ in a Hopfield network using Hebb's rule $w_{ij}=\frac{1}{N}\sum_{\mu=1}^{2}x_i^{(\mu)}x_j^{(\mu)}$. Which of the patterns in Figure 2.12 remain unchanged after one synchronous update with $s_i'=\mathrm{sgn}(\sum_{j=1}^{N}w_{ij}s_j)$? *Hint*: read off $\sum_{j=1}^{N}x_j^{(\mu)}x_j^{(\nu)}$ from the Hamming distances between the patterns, and use this quantity to express the local fields $b_i^{(\mu)}$ as linear combinations of $x_i^{(1)}$ and $x_i^{(2)}$.

2.13 Diluted Hopfield network. In the diluted Hopfield network with N neurons, only a fraction $\frac{K}{N}\ll 1$ of the weights w_{ij} is active: $w_{ij}=\frac{K_{ij}}{K}\sum_{\mu=1}^{p}x_i^{(\mu)}x_j^{(\mu)}$ where K_{ij} are random numbers, equal to $K_{ij}=1$ with probability $\frac{K}{N}$ and equal to zero otherwise. The parameter K determines the average number of connections to neuron i, $\langle\sum_{j=1}^{N}K_{ij}\rangle_{\mathrm{c}}=K$, where $\langle\ldots\rangle_{\mathrm{c}}$ denotes the average over random realisations of K_{ij}. Derive the approximate self-consistent equation [1]

$$m_\nu=\mathrm{erf}\Big(\frac{m_\nu}{\sqrt{2p/K}}\Big) \tag{2.61}$$

for the order parameter $m_\nu=\langle\frac{1}{N}\sum_{i=1}^{N}x_i^{(\nu)}\langle s_i\rangle_{\mathrm{c}}\rangle$ in the deterministic limit, and assuming that $K\gg 1$ and $1\ll p\ll N$. Here the outer average is over random patterns. *Hint*: show that the distribution of the cross-talk term $\langle C_i^{(\nu)}\rangle_{\mathrm{c}}$ over input patterns is Gaussian with mean zero. Determine how the variance of $\langle C_i^{(\nu)}\rangle_{\mathrm{c}}$ depends upon K.

2.14 Mixed states. Consider p random patterns $\boldsymbol{x}^{(\mu)}$ $(\mu=1,\ldots,p)$ with N bits $x_i^{(\mu)}$

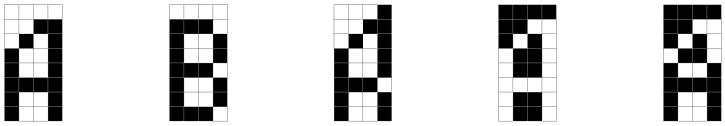


Figure 2.12: Each of the five patterns consists of 32 bits $x_i^{(\mu)}$. A black pixel i in pattern μ corresponds to $x_i^{(\mu)}=1$, a white one to $x_i^{(\mu)}=-1$. Exercise 2.12.

($i = 1, \ldots, N$), equal to 1 or -1 with probability $\frac{1}{2}$. Store the patterns in a deterministic Hopfield network using Hebb's rule $w_{ij} = \frac{1}{N}\sum_{\mu=1}^{p} x_i^{(\mu)} x_j^{(\mu)}$. In the limit of $N \gg 1$, $p \gg 1$, $p \ll N$, show that the network recognises bit $x_i^{(\mathrm{mix})}$ of the *mixed state* $\boldsymbol{x}^{(\mathrm{mix})}$ with bits

$$x_i^{(\mathrm{mix})} = \mathrm{sgn}\big(x_i^{(1)} + x_i^{(2)} + x_i^{(3)}\big), \tag{2.62}$$

after a *single asynchronous* update $s_i \leftarrow \mathrm{sgn}(\sum_{j=1}^{N} w_{ij} s_j)$. Follow the steps outlined below.

(a) Feed the mixed state (2.62) to the network. Use the weights w_{ij} you obtained by applying Hebb's rule and express $\sum_{j=1}^{N} w_{ij} x_j^{(\mathrm{mix})}$ in terms of $\langle s_\mu \rangle$, defined by $\langle s_\mu \rangle = \frac{1}{N}\sum_{j=1}^{N} x_j^{(\mu)} x_j^{(\mathrm{mix})}$, for $\mu = 1 \ldots p$.

(b) Assume that the bits $x_i^{(\mu)}$ are independent random numbers, equal to 1 or -1 with equal probabilities. What is the value of $\langle s_\mu \rangle$ for $\mu = 1$, 2 and 3? What is the value for $\langle s_\mu \rangle$ for $\mu > 3$?

(c) Rewrite the expression you derived in (a) as a sum of two terms. The first term is a sum over $\mu = 1, 2, 3$. The second term is the cross-talk term, a sum over the remaining values of μ. Explain why the cross-talk term can be neglected in the limit stated above.

(d) Combine the results of (a), (b) and (c) to show that the network recognises the mixed state (2.62).

3 Stochastic Hopfield networks

Two related problems became apparent in the previous Chapter. First, the Hopfield dynamics may get stuck in spurious minima. In fact, if there is a local minimum downhill from a given initial state, between this state and the correct attractor, then the dynamics arrests in the local minimum, so that the algorithm fails to converge to the correct attractor. Second, the energy function usually is a strongly varying function over a high-dimensional state space. Therefore it is difficult to predict the long-time dynamics of the network. Which is the first local minimum encountered on the down-hill path that the network takes?

Both problems are solved by introducing a little bit of noise into the dynamics. This is a trick that works for many neural-network algorithms. But in general it is very challenging to analyse the noisy dynamics. For the Hopfield network, by contrast, much is known. The reason is that the stochastic Hopfield network is closely related to systems studied in statistical mechanics, so-called spin glasses. Like these systems – and like many other physical systems – the stochastic Hopfield network exhibits an *order-disorder transition*. This transition becomes sharp in the limit of large N. This has important consequences. It may be that the network produces satisfactory results for a given number of patterns with a certain number of bits. But if one tries to store just one more pattern, the network may fail to recognise anything. The goal of this Chapter is to explain why this occurs, and how it can be avoided.

3.1 Noisy dynamics

The asynchronous update rule (2.8) reads

$$s'_m = \mathrm{sgn}(b_m) \tag{3.1}$$

for $i = m$, the states of all other neurons remain the same, $s'_i = s_i$ for $i \neq m$. As before, $b_m = \sum_j w_{mj} s_j - \theta_m$ is the local field. This rule is called *deterministic*, because a given set of states s_i determines the outcome of the update.

To introduce noise, one replaces the rule (3.1) by the *stochastic* rule

$$s_m = \begin{cases} +1 & \text{with probability} \quad p(b_m), \\ -1 & \text{with probability} \quad 1-p(b_m), \end{cases} \tag{3.2a}$$

where $p(b)$ is given by:

$$p(b) = \frac{1}{1+e^{-2\beta b}}\,. \tag{3.2b}$$

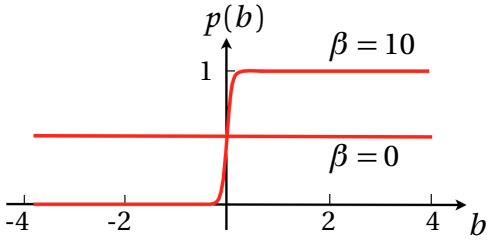


Figure 3.1: Probability function (3.2b) used in the definition of the stochastic rule (3.2), plotted for $\beta = 10$ and $\beta = 0$.

The function $p(b)$ is plotted in Figure 3.1. The parameter β is the noise parameter. When β is large the noise level is small. In particular one obtains the deterministic dynamics (3.1) as β tends to infinity. In this limit, the function $p(b)$ approaches zero if b is negative, and it tends to unity if b is positive. So for $\beta \to \infty$, the stochastic update rule (3.2b) is the same as the deterministic rule (3.1). In the opposite limit, when $\beta = 0$, the function $p(b)$ simply equals $\frac{1}{2}$. In this case s_i is updated to -1 or $+1$ randomly, with equal probability. The dynamics does not depend upon the stored patterns (contained in the local field b_i).

The idea is to run the network for a small but finite noise level, that is at large value of β. Then the dynamics is very similar to the deterministic Hopfield dynamics analysed in the previous Chapter. But the noise allows the system to sometimes also go uphill, making it possible to escape spurious minima. Since the dynamics is noisy, it is necessary to rephrase the convergence criterion (2.4). This is discussed next.

3.2 Order parameters

If we feed one of the stored patterns, $\boldsymbol{x}^{(1)}$ for example, then we want the noisy dynamics to stay in the vicinity of $\boldsymbol{x}^{(1)}$. This can only work if the noise is weak enough, and even then it is not guaranteed. Success is measured by the *order parameter* m_μ

$$m_\mu \equiv \lim_{T\to\infty} m_\mu(T) \quad \text{with} \quad m_\mu(T) = \frac{1}{T}\sum_{t=1}^{T}\left(\frac{1}{N}\sum_{i=1}^{N} s_i(t)x_i^{(\mu)}\right). \tag{3.3}$$

Here $m_\mu(T)$ is the finite-time average of $\frac{1}{N}\sum_{i=1}^{N} s_i(t)x_i^{(\mu)}$ over the noisy dynamics of the network from $t = 1$ to $t = T$, for given bits $x_i^{(\mu)}$. Since we decided to feed pattern $\boldsymbol{x}^{(1)}$ to the network, we have initially $s_i(t=0) = x_i^{(1)}$ and thus

$$\frac{1}{N}\sum_{i=1}^{N} s_i(t=0)\, x_i^{(1)} = 1\,. \tag{3.4}$$

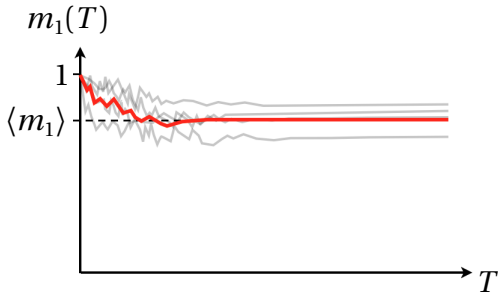


Figure 3.2: Illustrates how the finite-time average $m_1(T)$ depends upon the total iteration time T. The light grey lines show different realisations of $m_1(T)$ for different realisations of random patterns stored in the network, at a large but finite value of N. The thick red line is the average of $m_1(T)$ over the different realisations of random patterns.

After a *transient*, the quantity $\frac{1}{N}\sum_{i=1}^{N} s_i(t)x_i^{(1)}$ settles into a *steady state*, where it fluctuates around a mean value with a definite distribution that becomes independent of the iteration number t. If the network works well, we expect that $s_i(t)$ remain close to $x_i^{(1)}$, so that the finite-time average $m_1(T)$ converges to a value of order unity as $T \to \infty$, to the order parameter m_1. Since there is noise, m_1 is usually smaller than unity. This is illustrated in Figure 3.2.

This Figure shows that there is a subtlety. For finite values of N the order parameter m_1 depends upon the stored patterns. Different realisations $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(p)}$ of random patterns yield different values of m_1. In the limit of $N \to \infty$ this problem does not occur, the order parameter m_1 is independent of the stored patterns. We say that the system is *self averaging* in the limit $N \to \infty$. When N is finite, however, this is not the case. To obtain a definite value for the order parameter, one usually averages m_1 over different realisations of random patterns stored in the network (thick red line in Figure 3.2). The dashed line in Figure 3.2 shows $\langle m_1 \rangle$.

The other order parameters $m_2, \ldots, m_p$ are expected to be small. This is certainly true for random patterns with many bits, where the bits of the patterns $\boldsymbol{x}^{(2)}$ to $\boldsymbol{x}^{(p)}$ are independent from those of $\boldsymbol{x}^{(1)}$. As a consequence the individual terms in the sum over i in Equation (3.3) cancel approximately upon summation if $s_i(t) \approx x_i^{(1)}$. In summary, we expect in the limit of large N

$$m_\mu \approx \begin{cases} 1 & \text{if} \quad \mu = 1, \\ 0 & \text{otherwise.} \end{cases} \tag{3.5}$$

The aim is now to compute how m_1 depends on the values of p, N, and β.

3.3 Mean-field theory

Assume that the dynamics of the stochastic Hopfield network reaches a steady state, as illustrated in Figure 3.2. The order parameter is defined as a time average over the stochastic dynamics of the network, in the limit of $T \to \infty$. In practice we cannot choose T to be infinite, but T must be large enough so that the average can be estimated accurately, and so that any *initial transient* does not matter.

It is a challenging task to compute the average over the dynamics. To simplify the problem we consider first the case of only one neuron, $N = 1$. In this case there are no connections to other neurons, but we can still assume that the network is defined by a local field b_1, corresponding to the threshold θ_1 in Equation (2.12). To compute the order parameter we must evaluate the time average of $s_1(t)$. We denote this time average as follows:

$$\langle s_1 \rangle = \lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} s_1(t). \tag{3.6}$$

Note that this is a different average from the one defined on page 19. The average there is over random bits, here it is over the stochastic network dynamics (3.2). We can evaluate the time average $\langle s_1 \rangle$ using Equation (3.2):

$$\langle s_1 \rangle = \text{Prob}(s_1 = +1) - \text{Prob}(s_1 = -1) = p(b_1) - [1 - p(b_1)]. \tag{3.7}$$

Equation (3.2b) yields:

$$\langle s_1 \rangle = \frac{e^{\beta b_1}}{e^{\beta b_1} + e^{-\beta b_1}} - \frac{e^{-\beta b_1}}{e^{\beta b_1} + e^{-\beta b_1}} = \tanh(\beta b_1). \tag{3.8}$$

Figure 3.3 illustrates how $\langle s_1 \rangle$ depends upon b_1. For weak noise levels (large β), the threshold b_1 acts as a bias. When b_1 is negative then $\langle s_1 \rangle \approx -1$, while $\langle s_1 \rangle \approx 1$ when b_1 is positive. So the state s_1 reflects the bias most of the time. When however the noise is large (small β), then $\langle s_1 \rangle \approx 0$. In this case the state variable s_1 is essentially unaffected by the bias, s_1 is equally likely to be negative as positive so that its average evaluates to zero.

How can we generalise this calculation to a Hopfield model with many neurons? It can be done approximately at least, in the limit of large values of N. Consider neuron number i. The fate of s_i is determined by b_i. But b_i in turn depends on all s_j in the network:

$$b_i(t) = \sum_{j=1}^{N} w_{ij} s_j(t). \tag{3.9}$$

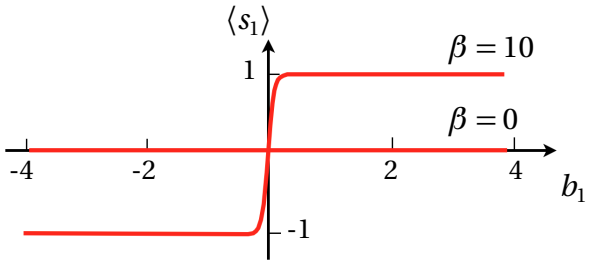


Figure 3.3: Average $\langle s_1 \rangle$ under the dynamics (3.2), as a function of b_1 for different noise levels.

This makes the problem challenging. But when N is large we may assume that $b_i(t)$ remains essentially constant in the steady state, independent of t. The argument is that fluctuations of $s_j(t)$ average out when summing over j, at least when N is large. In this case $b_i(t)$ is approximately given by its time average:

$$b_i(t) = \langle b_i \rangle + \text{small fluctuations in the limit } N \to \infty . \tag{3.10}$$

Here the average local field $\langle b_i \rangle$ is called the *mean field*, and theories that neglect the small corrections in Equation (3.10) are called *mean-field theories*. Let us assume that the fluctuations in Equation (3.10) are negligible and write

$$b_i(t) \approx \langle b_i \rangle , \tag{3.11}$$

so that the problem of evaluating the average of $s_i(t)$ reduces to the case discussed above, $N = 1$. From Equations (2.30) and (3.8) one deduces that

$$\langle s_i \rangle = \tanh(\beta \langle b_i \rangle) \quad \text{with} \quad \langle b_i \rangle = \frac{1}{N} \sum_{\mu} \sum_{j \neq i} x_i^{(\mu)} x_j^{(\mu)} \langle s_j \rangle . \tag{3.12}$$

These mean-field equations are a set of N non-linear equations for $\langle s_i \rangle$, for given fixed patterns $x_i^{(\mu)}$. The aim is to solve these equations to determine the time averages $\langle s_i \rangle$, and then the order parameters from

$$m_\mu = \frac{1}{N} \sum_{j=1}^{N} \langle s_j \rangle x_j^{(\mu)} . \tag{3.13}$$

If we initially feed pattern $\boldsymbol{x}^{(1)}$ we hope that $m_1 \approx 1$ while $m_\mu \approx 0$ for $\mu \neq 1$. To determine under which circumstances this has a chance to work we express the mean field in terms of the order parameters m_μ:

$$\langle b_i \rangle = \frac{1}{N} \sum_{\mu=1}^{p} \sum_{j \neq i} x_i^{(\mu)} x_j^{(\mu)} \langle s_j \rangle \approx \sum_{\mu} x_i^{(\mu)} m_\mu . \tag{3.14}$$

The last equality is only approximate because the j-sum in the definition of m_μ contains the term $j = i$. Whether or not to include this term makes only a small difference to m_μ, in the limit of large N.

Now *assume* that $m_1 \approx m$ with m of order unity, and $m_\mu \approx 0$ for $\mu \neq 1$. Then the first term in the sum over μ dominates, provided that the small terms do not add up to a contribution of order unity. This is the case if

$$\alpha = \frac{p}{N} \tag{3.15}$$

is small enough. Roughly speaking this works if α is of order N^{-1} (a more precise estimate is that α is at most $(\log N)/N$ [23]). Now we use the mean-field equations (3.12) to approximate

$$\langle s_i \rangle = \tanh(\beta \langle b_i \rangle) \approx \tanh(\beta m_1 x_i^{(1)}). \tag{3.16}$$

Applying the definition of the order parameter

$$m_1 = \frac{1}{N} \sum_{i=1}^{N} \langle s_i \rangle x_i^{(1)}$$

we find

$$m_1 = \frac{1}{N} \sum_{i=1}^{N} \tanh\left(\beta m_1 x_i^{(1)}\right) x_i^{(1)}. \tag{3.17}$$

Using that $\tanh(z) = -\tanh(-z)$ as well as the fact that the bits $x_i^{(\mu)}$ can only assume the values ± 1, we get

$$m_1 = \tanh(\beta m_1). \tag{3.18}$$

This is a self-consistent equation for m_1. We assumed that m_1 is of order unity. So the question is: does this equation admit such solutions? For $\beta \to 0$ there is one solution, $m_1 = 0$. This is not the desired one. For $\beta \to \infty$, by contrast, there are three solutions, $m_1 = 0, \pm 1$. Figure 3.4 shows results of the numerical evaluation of Equation (3.18) for intermediate values of β. Below a critical noise level there are three solutions, namely for β larger than

$$\beta_c = 1. \tag{3.19}$$

For $\beta > \beta_c$, the solution $m_1 = 0$ is *unstable* (this can be shown by computing the derivatives of the *free energy* of the Hopfield network [1]). Even if we were to start with an initial condition that corresponds to $m_1 = 0$, the network would not stay there. The other two solutions are *stable*: when the network is initialised close to $\boldsymbol{x}^{(1)}$, then it converges to $m_1 = +m$ (with $m > 0$).

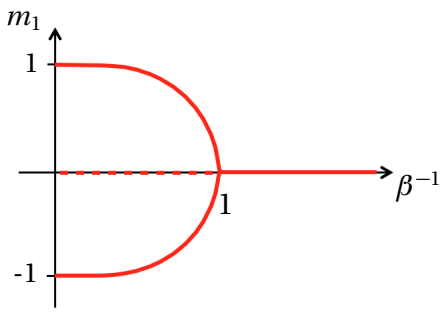


Figure 3.4: Solutions of the mean-field equation (3.18). The critical noise level is $\beta_c = 1$. The dashed line corresponds to an unstable solution.

The symmetry of the problem dictates that there must also be a solution with $m_1 = -m$ at small noise levels. This solution corresponds to the inverted pattern $-\boldsymbol{x}^{(1)}$ (Section 2.4). If we start in the vicinity of $\boldsymbol{x}^{(1)}$, then the network is unlikely to converge to $-\boldsymbol{x}^{(1)}$, provided that N is large enough. The probability of $\boldsymbol{x}^{(1)} \to -\boldsymbol{x}^{(1)}$ vanishes very rapidly as N increases and as the noise level decreases. If this transition were to happen in a simulation, the network would then stay near $-\boldsymbol{x}^{(1)}$ for a very long time. Consider the limit where T tends to ∞ at a finite but possibly large value of N. Then the network would (at a very small rate) jump back and forth between $\boldsymbol{x}^{(1)}$ and $-\boldsymbol{x}^{(1)}$, so that the order parameter would average to zero. This shows that the limits of large N and large T do not commute

$$\lim_{T\to\infty}\lim_{N\to\infty} m_1(T) \neq \lim_{N\to\infty}\lim_{T\to\infty} m_1(T). \tag{3.20}$$

In practice the interesting limit is the left one, that of a large network run for a time T much longer than the initial transient, but not infinite. This is precisely where the mean-field theory applies. It correponds to taking the limit $N \to \infty$ first, at finite but large T. This describes simulations where the transition $\boldsymbol{x}^{(1)} \to -\boldsymbol{x}^{(1)}$ does not occur.

In summary, Equation (3.18) predicts that the order parameter converges to a definite value, m_1, independent of the stored patterns when N is large enough. Figure 3.2 shows that the order parameter converges for large but finite values of N. However, the limiting value does depend on the stored patterns, as mentioned above. The system is not self averaging. When N is finite we should therefore average the result over different realisations of the stored patterns.

The value of $\langle m_1 \rangle$ determines the average number of correctly retrieved bits in the steady state:

$$\langle N_{\text{correct}} \rangle = \left\langle \frac{1}{2}\sum_{i=1}^{N}\left(1 + \langle s_i \rangle x_i^{(1)}\right)\right\rangle = \frac{N}{2}(1 + \langle m_1 \rangle). \tag{3.21}$$

The outer average is over different realisations of random patterns (the inner average is over the network dynamics). Since $m_1 \to 1$ as $\beta \to \infty$ we see that the number of correctly retrieved bits approaches N

$$\langle N_{\text{correct}} \rangle \to N \,. \tag{3.22}$$

This is expected since the stored patterns $\boldsymbol{x}^{(\mu)}$ are recognised for small enough values of α in the deterministic limit, because the cross-talk term is negligible. But it is important to know that the stochastic dynamics *slows down* as the noise level tends to zero. The lower the noise level, the longer the network remains stuck in local minima, so that it takes longer time to reach the steady state, and to sample the steady-state statistics of H. Conversely when the noise is strong, then

$$\langle N_{\text{correct}} \rangle \to \tfrac{N}{2} \,. \tag{3.23}$$

In this limit the stochastic network ceases to function. If one were to assign N bits entirely randomly, then half of them would be correct, on average.

We define the error probability in the steady state as

$$P_{\text{error}}^{t=\infty} = \frac{N - \langle N_{\text{correct}} \rangle}{N} \,. \tag{3.24}$$

From Equation (3.21) we find

$$P_{\text{error}}^{t=\infty} = \tfrac{1}{2}(1 - \langle m_1 \rangle) \,. \tag{3.25}$$

In the deterministic limit the steady-state error probability approaches zero as m_1 tends to one. Let us compare this result with the one-step error probability $P_{\text{error}}^{t=1}$ derived in Chapter 2 in the deterministic limit. We should take the limit $\alpha = p/N \to 0$ in Equation (2.43) because the result (3.25) was derived assuming that α is very small. In this limit we find that the one-step and the steady-state error probabilities agree (they are both equal to zero). Above the critical noise level, for $\beta < \beta_{\text{c}} = 1$, the order parameter vanishes. In this case $P_{\text{error}}^{t=\infty}$ equals $\frac{1}{2}$. So when the noise is too large the network fails.

It is important to note that noise can also help, because mixed states have lower critical noise levels than the stored patterns $x_i^{(\mu)}$. This can be seen as follows [1, 24]. To derive the above mean-field result we assumed that $m_\mu = m\delta_{\mu 1}$. Mixed states correspond to solutions where an odd number of components of $\boldsymbol{m}$ is non-zero, for example:

$$\boldsymbol{m} = \begin{bmatrix} m \\ m \\ m \\ 0 \\ \vdots \end{bmatrix} \,. \tag{3.26}$$

Neglecting the cross-talk term, the mean-field equation reads

$$\langle s_i \rangle = \tanh\Big(\beta \sum_{\mu=1}^{p} m_\mu x_i^{(\mu)}\Big). \tag{3.27}$$

In the limit of $\beta \to \infty$, the $\langle s_i \rangle$ converge to the mixed states (2.57) when $\boldsymbol{m}$ is given by Equation (3.26). Using the definition of m_μ and averaging over the bits of the random patterns one finds:

$$m_\mu = \Big\langle x_i^{(\mu)} \tanh\Big(\beta \sum_{\nu=1}^{p} m_\nu x_i^{(\nu)}\Big)\Big\rangle. \tag{3.28}$$

The numerical solution of Equation (3.26) shows that there is a non-zero solution for $\beta > \beta_c = 1$. Yet this solution is unstable close to the critical noise level, more precisely for $1 < \beta < 2.17$ [24]. In other words, the mixed states have a lower critical noise level, $1/2.17$.

3.4 Storage capacity

The preceding analysis replaced the sum (3.14) by its first term, $x_i^{(1)} m_1$. This corresponds to neglecting the cross-talk term. We expect that this can only work if p/N is small enough. The influence of the cross-talk term was studied in Section 2.2, where the storage capacity

$$\alpha = \frac{p}{N}$$

was defined. When we computed $P_{\text{error}}^{t=1}$ in Section 2.2, only the first *initial* update step was considered, because it was too difficult to analyse the long-time limit of the deterministic dynamics. It is expected that the error probability increases as t increases, at least when α is large enough so that the cross-talk term matters.

The stochastic dynamics is simpler to analyse in the long-time limit because it approaches as steady state. The remainder of this Section describes the mean-field analysis of the steady state for larger values of α. We store p patterns in the network using Hebb's rule (2.30) and feed pattern $\boldsymbol{x}^{(1)}$ to the network. The aim is to determine the order parameter m_1 and the corresponding error probability in the steady state for $p \sim N$, so that α remains finite as $N \to \infty$. In this case we can no longer approximate the sum in Equation (3.14) just by its first term, because the other terms for $\mu > 1$ may sum up to a contribution that is of the same order as m_1. Instead we must evaluate all m_μ to compute the mean field $\langle b_i \rangle$.

The relevant calculation is summarised in Chapter 4 of Geszti [25]. It is also outlined in Section 2.5 of Hertz, Krogh and Palmer [1]. The remainder of this Section

follows this outline quite closely. One starts by rewriting the mean-field equations (3.12) in terms of the order parameters m_μ. Using

$$\langle s_i \rangle = \tanh(\beta \sum_\mu x_i^{(\mu)} m_\mu) \tag{3.29}$$

we find

$$m_\nu = \frac{1}{N} \sum_i x_i^{(\nu)} \langle s_i \rangle = \frac{1}{N} \sum_i x_i^{(\nu)} \tanh\left(\beta \sum_\mu x_i^{(\mu)} m_\mu \right). \tag{3.30}$$

This coupled set of p non-linear equations is equivalent to the mean-field equation (3.12).

Now feed pattern $\boldsymbol{x}^{(1)}$ to the network. The strategy of solving Equation (3.30) is to assume that the network stays close to the pattern $\boldsymbol{x}^{(1)}$ in the steady state, so that m_1 remains of order unity. Since we cannot simply approximate the sum over μ on the r.h.s. of Equation (3.30) by its first term, we need to estimate the other order parameters

$$m_\mu = \frac{1}{N} \sum_j x_j^{(\mu)} \langle s_j \rangle \quad \text{for} \quad \mu \neq 1$$

as well. The trick is to estimate these order parameters assuming random patterns, so that the m_μ, $\mu = 2, \ldots, p$, become random numbers that fluctuate around zero with variance $\langle m_\mu^2 \rangle$ (this average is over random patterns). We use Equation (3.30) to compute the variance approximately. In the μ-sum on the r.h.s of Equation (3.30) we must treat the term $\mu = \nu$ separately (because the index ν appears also on the l.h.s. of this equation). Also the term $\mu = 1$ must be treated separately, as before, because $\mu = 1$ is the index of the pattern that was fed to the network.

As a consequence the calculations of m_1 and m_μ for $\mu \neq 1$ proceed slightly differently. We start with the second case and write

$$\begin{aligned} m_\nu &= \frac{1}{N} \sum_i x_i^{(\nu)} \tanh\left(\beta\, x_i^{(1)} m_1 + \beta\, x_i^{(\nu)} m_\nu + \beta \sum_{\substack{\mu \neq 1 \\ \mu \neq \nu}} x_i^{(\mu)} m_\mu \right) \\ &= \frac{1}{N} \sum_i x_i^{(\nu)} x_i^{(1)} \tanh\Bigg(\underbrace{\beta\, m_1}_{①} + \underbrace{\beta\, x_i^{(1)} x_i^{(\nu)} m_\nu}_{②} + \underbrace{\beta \sum_{\substack{\mu \neq 1 \\ \mu \neq \nu}} x_i^{(\mu)} x_i^{(1)} m_\mu}_{③} \Bigg). \end{aligned} \tag{3.31}$$

Now consider the three terms in the argument of tanh(...). The term ① is of order unity, it is independent of N. The term ③ may be of the same order, because the sum over μ contains $\sim pN$ terms. The term ②, by contrast, is small for large

values of N. Therefore it is a good approximation to Taylor-expand the argument of $\tanh(\ldots)$:

$$\tanh\Big(①+②+③\Big) \approx \tanh\Big(①+③\Big) + ② \frac{\mathrm{d}}{\mathrm{d}x} \tanh\Big|_{①+③} + \ldots . \tag{3.32}$$

Using $\frac{\mathrm{d}}{\mathrm{d}x}\tanh(x) = 1-\tanh^2(x)$ one gets

$$\begin{aligned} m_\nu = \frac{1}{N}\sum x_i^{(\nu)} x_i^{(1)} \tanh\Bigg(\underbrace{\beta m_1}_{①} + \underbrace{\beta \sum_{\substack{\mu\neq 1\\ \mu\neq\nu}} x_i^{(\mu)} x_i^{(1)} m_\mu}_{③} \Bigg) \\ + \frac{1}{N}\sum_i x_i^{(\nu)} x_i^{(1)} \underbrace{\beta\, x_i^{(1)} x_i^{(\nu)} m_\nu}_{②} \Bigg[1 - \tanh^2\Bigg(\beta m_1 + \beta \sum_{\substack{\mu\neq 1\\ \mu\neq\nu}} x_i^{(\mu)} x_i^{(1)} m_\mu \Bigg)\Bigg]. \end{aligned} \tag{3.33}$$

Using the fact that $x^{(\mu)} = \pm 1$ and thus $[x_i^{(\mu)}]^2 = 1$, this expression simplifies:

$$\begin{aligned} m_\nu = \frac{1}{N}\sum_i x_i^{(\nu)} x_i^{(1)} \tanh\Bigg(\beta m_1 + \beta \sum_{\substack{\mu\neq 1\\ \mu\neq\nu}} x_i^{(1)} m_\mu \Bigg) + \\ + \beta\, m_\nu \frac{1}{N}\sum_i \Bigg[1 - \tanh^2\Bigg(\beta m_1 + \beta \sum_{\substack{\mu\neq 1\\ \mu\neq\nu}} x_i^{(\mu)} x_i^{(1)} m_\mu \Bigg)\Bigg]. \end{aligned} \tag{3.34}$$

The next steps are similar to the analysis of the cross-talk term in Section 2.2. We assume that the patterns are random, that their bits $x_i^{(\mu)} = \pm 1$ are independently randomly distributed. Since the sums in Equation (3.34) contain many terms, we can estimate the sums using the central-limit theorem. The variable

$$z \equiv \sum_{\substack{\mu\neq 1\\ \mu\neq\nu}} x_i^{(\mu)} x_i^{(1)} m_\mu \tag{3.35}$$

is then approximately Gaussian distributed, with mean zero. The variance of z is given by an average over a double sum. Since bits $x_i^{(\mu)}$ and $x_j^{(\mu')}$ are independent when $\mu \neq \mu'$, only the diagonal in this double sum contributes:

$$\sigma_z^2 = \sum_{\substack{\mu\neq 1\\ \mu\neq\nu}} \langle m_\mu^2 \rangle \approx p \langle m_\mu^2 \rangle \quad \text{for any} \quad \mu \neq 1, \nu. \tag{3.36}$$

Here we assumed that p is large so that $p-2 \approx p$. Now return to Equation (3.34). The sum $\frac{1}{N}\sum_i$ in the second line can be approximated as an average over the Gaussian distributed variable z:

$$\beta m_\nu \int_{-\infty}^{\infty} \mathrm{d}z \frac{1}{\sqrt{2\pi}\sigma_z} \mathrm{e}^{-\frac{z^2}{2\sigma_z^2}} \left[1-\tanh^2\left(\beta m_1 + \beta z\right)\right]. \tag{3.37}$$

We write the expression (3.37) as

$$\beta m_\nu \left[1 - \int_{-\infty}^{\infty} \mathrm{d}z \frac{1}{\sqrt{2\pi}\sigma_z} \mathrm{e}^{-\frac{z^2}{2\sigma_z^2}} \tanh^2\left(\beta m_1 + \beta z\right)\right] \equiv \beta m_\nu (1-q), \tag{3.38}$$

using the following definition of the parameter q:

$$q = \int_{-\infty}^{\infty} \mathrm{d}z \frac{1}{\sqrt{2\pi}\sigma_z} \mathrm{e}^{-\frac{z^2}{2\sigma_z^2}} \tanh^2\left(\beta m_1 + \beta z\right). \tag{3.39}$$

Returning to Equation (3.34) we see that it takes the form

$$m_\nu = \frac{1}{N}\sum_i x^{(\nu)} x_i^{(1)} \tanh\left(\beta m_1 + \beta \sum_{\substack{\mu\neq 1 \\ \mu\neq\nu}} x_i^{(\mu)} x_i^{(1)} m_\mu\right) + (1-q)\beta m_\nu. \tag{3.40}$$

Solving for m_ν we find:

$$m_\nu = \frac{\frac{1}{N}\sum_i x^{(\nu)} x_i^{(1)} \tanh\left(\beta m_1 + \beta \sum_{\substack{\mu\neq 1 \\ \mu\neq\nu}} x_i^{(\mu)} x_i^{(1)} m_\mu\right)}{1-\beta(1-q)}, \tag{3.41}$$

for $\nu \neq 1$. This expression allows us to compute the variance σ_z, defined by Equation (3.36). Equation (3.41) shows that the average $\langle m_\nu^2 \rangle$ contains a double sum over the bit index, i. Since the bits are independent, only the diagonal terms contribute, so that

$$\langle m_\nu^2 \rangle \approx \frac{\frac{1}{N^2}\sum_i \tanh^2\left(\beta m_1 + \beta \sum_{\substack{\mu\neq 1 \\ \mu\neq\nu}} x_i^{(\mu)} x_i^{(1)} m_\mu\right)}{[1-\beta(1-q)]^2}, \tag{3.42}$$

independent of ν. The numerator is just q/N, from Equation (3.39). So the variance evaluates to

$$\sigma_z^2 = \frac{\alpha q}{[1-\beta(1-q)]^2}. \tag{3.43}$$

Up to now it was assumed that $\nu \neq 1$. One can derive an Equation for m_1 by repeating almost the same steps as above, but now for $\nu = 1$. The result is:

$$m_1 = \int \frac{\mathrm{d}z}{\sqrt{2\pi\sigma_z^2}} e^{-\frac{z^2}{2\sigma_z^2}} \tanh(\beta m_1 + \beta z). \tag{3.44}$$

This is a self-consistent equation for m_1. In summary there are three coupled equations, for m_1, q, and σ_z. Equations (3.38), (3.43), and (3.44). They must be solved together to determine how m_1 depends on β and α.

To compare with the results described in Section 2.2 we must take the deterministic limit, $\beta \to \infty$. In this limit q approaches unity, yet $\beta(1-q)$ remains finite. Setting $q = 1$ in Equation (3.43) but retaining $\beta(1-q)$ we find [1]:

$$\sigma_z^2 = \frac{\alpha}{[1-\beta(1-q)]^2}. \tag{3.45a}$$

The deterministic limits of Equations (3.38) and (3.44) become:

$$\beta(1-q) = \sqrt{\frac{2}{\pi\sigma_z^2}}\, e^{-\frac{m_1^2}{2\sigma_z^2}}, \tag{3.45b}$$

$$m_1 = \operatorname{erf}\left(\frac{m_1}{\sqrt{2\sigma_z^2}}\right). \tag{3.45c}$$

Recall the definition (3.24) of the steady-state error probability. Inserting Equation (3.45c) for m_1 into this expression we find in the deterministic limit:

$$P_{\text{error}}^{t=\infty} = \frac{1}{2}\left[1 - \operatorname{erf}\left(\frac{m_1}{\sqrt{2\sigma_z^2}}\right)\right]. \tag{3.46}$$

Compare this with Equation (2.43) for the one-step error probability in the deterministic limit. That equation was derived for only one step in the dynamics of the network, while Equation (3.46) describes the long-time limit. Yet it turns out that Equation (3.46) reduces to (2.43) in the limit of $\alpha \to 0$. To see this one solves the set of Equations (3.45) by introducing the variable $y = m_1/\sqrt{2\sigma_z^2}$ [1]. One obtains the following one-dimensional equation for y [26]:

$$y(\sqrt{2\alpha} + (2/\sqrt{\pi})\, e^{-y^2}) = \operatorname{erf}(y). \tag{3.47}$$

The physical solutions are those satisfying $0 \le \operatorname{erf}(y) \le 1$, because the order parameter is restricted to this range (transitions to $-m_1$ do not occur in the limit $N \to \infty$). Figure 3.5 shows the steady-state error probability obtained from Equations (3.46) and (3.47). Also shown is the one-step error probability

$$P_{\text{error}}^{t=1} = \frac{1}{2}\left[1 - \operatorname{erf}\left(\frac{1}{\sqrt{2\alpha}}\right)\right]$$

derived in Section 2.2. You see that $P_{\text{error}}^{t=\infty}$ approaches $P_{\text{error}}^{t=1}$ for small α. This means that the error probability does not increase significantly as one iterates the network,

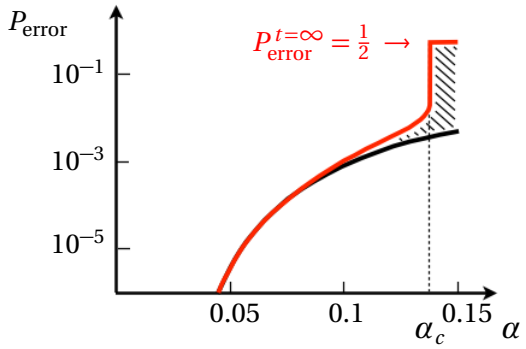


Figure 3.5: Error probability as a function of the storage capacity α in the deterministic limit. The one-step error probability $P_{\text{error}}^{t=1}$ [Equation (2.43)] is shown as a black line, the steady-state error probability $P_{\text{error}}^{t=\infty}$ [Equation (3.24)] is shown as a red line. In the hashed region error avalanches increase the error probability. Similar to Figure 1 in Ref. [26].

at least for small α. In this case errors in earlier iterations have little effect on the probability that later errors occur. But the situation is different at larger values of α. In that case $P_{\text{error}}^{t=1}$ significantly underestimates the steady-state error probability. In the hashed region, errors in the dynamics increase the probability of errors in subsequent steps, giving rise to *error avalanches*.

Figure 3.5 illustrates that there is a critical value α_c where the steady-state error probability tends to $\frac{1}{2}$. Solution of the mean-field Equations gives

$$\alpha_c \approx 0.1379\,. \tag{3.48}$$

When $\alpha > \alpha_c$ the steady-state error probability equals $\frac{1}{2}$, in this region the network produces just noise. When α is small, the error probability is small, here the network works well. Figure 3.5 shows that the steady-state error probability changes very abruptly near α_c. Assume you store 137 patterns with 1000 bits in a Hopfield network. Figure 3.5 demonstrates that the network can reliably retrieve the patterns. However, if you try to store one or two more patterns, the network fails to produce any output meaningfully related to the stored patterns. This rapid change is an example of a *phase transition*. In many physical systems one observes similar transitions between ordered and disordered phases.

What happens at higher noise levels? The numerical solution of Equations (3.38), (3.44), and (3.43) shows that the critical storage capacity α_c decreases as the noise level increases (smaller values of β). This is shown schematically in Figure 3.6. Below the red line the error probability is smaller than $\frac{1}{2}$ so that the network operates reliably (although less so as one approaches the phase-transition boundary). Outside this region the the error probability equals $\frac{1}{2}$. In this region the network fails.

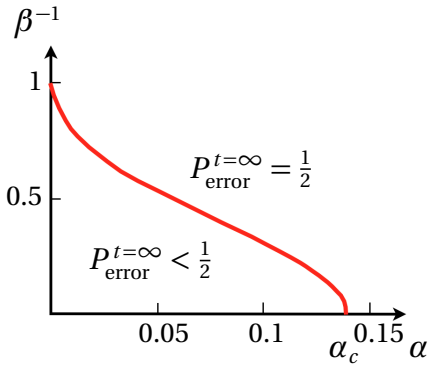


Figure 3.6: Phase diagram of the Hopfield network in the limit of large N (schematic). The region with $P_{\text{error}}^{t=\infty} < \frac{1}{2}$ is the ordered phase, the region with $P_{\text{error}}^{t=\infty} = \frac{1}{2}$ is the disordered phase. After Figure 2 in Ref. [26].

In the limit of small α the critical noise level is $\beta_c = 1$. In this limit the network is described by the theory explained in Section 3.3, Equation (3.18).

Often these two different phases of the Hopfield network are characterised in terms of the order parameter m_1. We see that $m_1 > 0$ in the hashed region, while $m_1 = 0$ outside.

3.5 Beyond mean-field theory

The theory summarised in this Chapter rests on a mean-field approximation for the local field, Equation (3.11). The main result is the phase diagram shown in Figure 3.6. It is important to note that it was derived in the limit $N \to \infty$. For smaller values of N one expects the transition to be less sharp, so that m_1 is non-zero for values of α larger than α_c.

But even for large values of N the question remains how accurate the mean-field theory really is. To answer this question, one must take into account fluctuations. The corresponding calculation is more difficult than the ones outlined earlier in this Chapter, and it requires several steps. One starts from the steady-state distribution of $\boldsymbol{s}$ for fixed patterns $\boldsymbol{x}^{(\mu)}$. In Chapter 4 we will see that the steady-state distribution for the McCulloch-Pitts dynamics is the *Boltzmann distribution*

$$P_{\text{B}}(\boldsymbol{s}) = Z^{-1} \mathrm{e}^{-\beta H(\boldsymbol{s})} \tag{3.49}$$

(the proof in Chapter 4 assumes that the diagonal weights are set to zero). The normalisation factor Z is called the *partition function*

$$Z = \sum_{\boldsymbol{s}} \mathrm{e}^{-\beta H(\boldsymbol{s})}. \tag{3.50}$$

One can compute the order parameter by adding a threshold term to the energy function (2.49)

$$H = -\frac{1}{2}\sum_{ij} w_{ij} s_i s_j + \sum_{\mu} \lambda_\mu \sum_i x_i^{(\mu)} s_i \,. \tag{3.51}$$

Then the order parameter m_μ is obtained by taking a derivative w.r.t λ_μ:

$$m_\mu = \Big\langle \frac{1}{N}\sum_i x_i^{(\mu)} \langle n_i \rangle \Big\rangle = -\frac{1}{N\beta}\frac{\partial}{\partial \lambda_\mu}\Big\langle \log Z \Big\rangle . \tag{3.52}$$

The outer average is over different realisations of random patterns. Since the logarithm of Z is difficult to average, one resorts to the *replica* trick. The idea is to represent the average of the logarithm as

$$\langle \log Z \rangle = \lim_{n\to 0} \tfrac{1}{n}(\langle Z^n \rangle - 1), \tag{3.53}$$

The function Z^n looks like the partition function of n copies of the system, thus the name *replica* trick. It is still debated when the replica trick works and when not [27]. Nevertheless, the most accurate theoretical result for the critical storage capacity is obtained in this way [28]

$$\alpha_c = 0.138187 \,. \tag{3.54}$$

The mean-field result (3.48) is different from (3.54), but it is very close. Most of the time, mean-field theories do not give such good results. Usually they yield at most a qualitative understanding of phase transitions, but not quantitatively accurate results as here. In the Hopfield model the mean-field theory works so well because the connections are global: every neuron is connected with every other neuron. This helps to average out the fluctuations in Equation (3.11).

The most precise Monte-Carlo simulations (Section 4.4) for finite values of N [29] yield upon extrapolation to $N = \infty$

$$\alpha_c = 0.143 \pm 0.002 \,. \tag{3.55}$$

This is close to, yet significantly different from the best theoretical estimate, Equation (3.54), and also different from the mean-field result (3.48).

3.6 Correlated and non-random patterns

In the above Sections we assumed random patterns with independently identically distributed bits. This allowed us to calcluate the storage capacity of the Hopfield network using the central-limit theorem. The hope is that the result describes what

happens for typical, non-random patterns, or for random patterns with correlated bits.

Correlations affect the distribution of the cross-talk term, and thus the storage capacity of the Hopfield net. It has been argued that the storage capacity increases when the patterns are more strongly correlated, while others have claimed that the capacity decreases in this limit (see Ref. [30] for a discussion).

When we must deal with a definite set of patterns (no randomness to average over), the situation seems to be even more challenging. Yet there is a way of modifying Hebb's rule to deal with this problem, at least when the patterns are linearly independent (this requires $p \leq N$). The recipe is explained by Hertz, Krogh, and Palmer [1]. One simply incorporates the overlaps

$$Q_{\mu\nu} = \frac{1}{N} \boldsymbol{x}^{(\mu)} \cdot \boldsymbol{x}^{(\nu)}. \tag{3.56}$$

into Hebb's rule. To this end one defines the $p \times p$ *overlap matrix* $\mathbb{Q}$ with elements $Q_{\mu\nu}$ and writes:

$$w_{ij} = \frac{1}{N} \sum_{\mu\nu} x_i^{(\mu)} \left(\mathbb{Q}^{-1}\right)_{\mu\nu} x_j^{(\nu)}. \tag{3.57}$$

For orthogonal patterns ($Q_{\mu\nu} = \delta_{\mu\nu}$) this modified Hebb's rule is identical to Equation (2.29). For non-orthogonal patterns, the rule (3.57) ensures that all patterns are recognised.

The rule (2.29) rests on the assumption that the matrix $\mathbb{Q}$ is invertible. This means that its columns must be linearly independent (and this implies that the rows are linearly independent too). This restricts the number of patterns one can store with the rule (2.29), because $p > N$ implies linear dependence.

For linearly independent patterns one can find the weights w_{ij} iteratively, by successive improvement from an arbitrary starting point. We can say that the network learns the task through a sequence of weight changes. This is the idea used to solve classification tasks with perceptrons (Part II).

3.7 Summary

In this Chapter we analysed the dynamics of Hopfield nets. We asked under which circumstances the network dynamics can reliably retrieve stored patterns. For random patterns we explained how the the performance of the Hopfield net depends on its parameters: the number of stored patterns, the number of bits per pattern, and the noise level.

Hopfield networks share many properties with the networks discussed later on in these lectures. The most important point is perhaps that introducing noise in the

dynamics allows to study the convergence and performance of the network: in the presence of noise there is a well-defined steady state that can be analysed. Without noise, in the deterministic limit, the network dynamics arrests in local minima of the energy function, and may not reach the stored patterns. Naturally the noise must be small enough for the network to function reliably. Apart from the noise level there is a second significant parameter, the storage capacity α, equal to the ratio of the number of patterns to the number of bits per pattern. When α is small then the network is reliable. A mean-field analysis of the $N \to \infty$-limit shows that there is a phase transition in the parameter plane (*phase diagram*) of the Hopfield network, Figure 3.6.

The building blocks of Hopfield networks are McCulloch-Pitts neurons and Hebb's rule for the weights. These elements are fundamental to the networks discussed in the coming Chapters.

3.8 Further reading

The statistical mechanics of Hopfield networks is explained in the book by Hertz, Krogh, and Palmer [1]. Starting from the Boltzmann distribution, Chapter 10 in this book explains how to compute the order parameters, and how to evaluate the stability of the corresponding solutions. For more details on the replica trick, see to Ref. [23].

3.9 Exercises

3.1 Mixed states. Write a computer program that implements the stochastic dynamics of a Hopfield model. Compute how the order parameter for mixed states that are superpositions of the bits of three stored patterns depends on the noise level for $0.5 \leq \beta \leq 2.5$. Compare your numerical results with the predictions in Section 3.3. Repeat the exercise for mixed states that consist of superpositions of the bits of five stored patterns. To this end, first derive the mean-field equation for the order parameter and solve this equation numerically. Second, perform your computer simulations and compare.

3.2 Order parameter. Derive the self-consistent Equation (3.44) for the order parameter m_1.

3.3 Deterministic limit. Derive the deterministic limit (3.45) of the three coupled Equations (3.38), (3.43), and (3.44) for m_1, q, and σ_z.

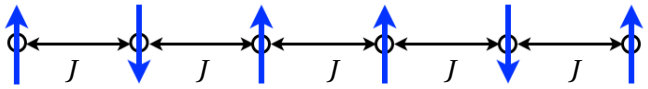


Figure 3.7: The Ising model is a model for ferromagnetism. It describes N spins that can either point up ($\uparrow$) or down ($\downarrow$), arranged on an integer lattice (here shown in one spatial dimension), interacting with their nearest neighbours with interaction strength J, and subject to an external magnetic field h. The state of spin i is described by the variable s_i, with $s_i = 1$ for $\uparrow$ and $s_i = -1$ for $\downarrow$.

3.4 Phase diagram of the Hopfield network. Derive Equation (3.47) from Equation (3.45). Numerically solve (3.47) to find the critical storage capacity α_c in the deterministic limit. Quote your result with three-digit accuracy. To determine how the critical storate capacity depends on the noise level, numerically solve the three coupled Equations (3.44), (3.39), and (3.43). Compare your result with the schematic Figure 3.6.

3.5 Non-orthogonal patterns. Show that the rule (3.57) ensures that all patterns are recognised, for any set of non-orthogonal patterns that gives rise to an invertible matrix $\mathbb{Q}$. Demonstrate this by showing that the cross-talk term evaluates to zero, assuming that $\mathbb{Q}^{-1}$ exists.

3.6 Ising model. The Ising model is a model for ferromagnetism, N spins $s_i = \pm 1$ are arranged on a d-dimensional integer lattice as shown in Figure 3.7. The energy function for the Ising model is $H = -J \sum_{i,j=\mathrm{nn}(i)} s_i s_j - h \sum_i s_i$. Here J is the ferromagnetic coupling between nearest-neighbour spins, h is an external magnetic field, and $\mathrm{nn}(i)$ denotes the nearest-neighbours of site i on the lattice. In equilibrium at temperature T, the states are distributed according to the Boltzmann distribution with $\beta = 1/(k_B T)$. Derive a mean-field approximation for the magnetisation of the system, $m = \langle \frac{1}{N} \sum_i s_i \rangle$, assuming that N is large enough that the contribution of the boundary spins can be neglected. Derive an expression for the critical temperature below which mean-field theory predicts that the system becomes ferromagnetic, $m \neq 0$. Discuss how the critical temperature depends on the dimension d. Note: mean-field theory fails for the one-dimensional Ising model. Its predictions become more accurate as d increases.

4 Stochastic optimisation

Hopfield networks were introduced as models that can recognise patterns. In Section 2.3 it was shown that this corresponds to minimising an energy function H. In this Chapter we see how this can be used to solve combinatorial optimisation problems using neural networks. Such problems admit 2^k or $k!$ configurations - too many to list and check in a serial approach when k is large.

The idea is to write down an energy function that is at most quadratic in the state variables, like Equation (2.49). Then one can use the Hopfield dynamics to minimize H. The problem is of course that the deterministic network dynamics arrests in first local minimum encountered, usually not the desired optimum (Figure 4.1). Therefore it is important to introduce a little bit of noise. As discussed in Chapter 3, the noise helps the network to escape local minima.

A common strategy is to lower the noise level on the fly. In the beginning of the simulation the noise level is high, so that the network explores the rough features of the energy landscape. When the noise level is lowered, the network can see finer and finer features, and the hope is that it ends up in the global minimum. This method is called *simulated annealing* [31].

4.1 Combinatorial optimisation problems

A well-known combinatorial optimisation problem is the *travelling-salesman problem*. Given the coordinates of k cities, the goal is to determine the shortest journey visiting each city exactly once before returning to the starting point. The coordinates of seven cities A,...,F are given in Figure 4.2 (this Figure illustrates the problem for $k = 7$). The Figure shows two different solutions. We see that the path in panel (**a**) is the shorter one. Denoting the distance between city A and B by d_{AB} and so forth,

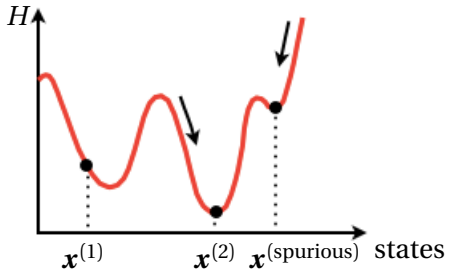


Figure 4.1: Deterministic gradient descent may not reach the desired minimum ($\boldsymbol{x}^{(2)}$ for example) because it arrests in a local minimum corresponding to spurious state $\boldsymbol{x}^{(\text{spurious})}$.

city	coordinates
A	(0.1,0.15)
B	(0.4 0.2)
C	(0.5 0.7)
D	(0.2 0.1)
E	(0.1 0.8)
F	(0.8 0.9)
G	(0.9 0.3)

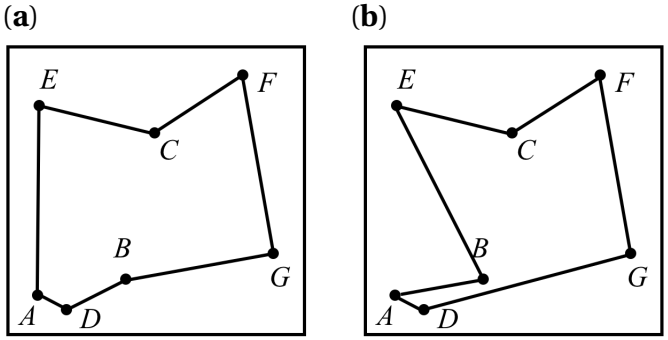


Figure 4.2: Traveling-salesman problem for $k = 7$. Given are the coordinates of k cities as points in the unit square. The problem is to find the shortest connected path that joins all cities, visits each city exactly once, and returns to the starting point. (**a**) optimal solution corresponding to the shortest path, (**b**) second solution with a longer path.

the length of the path in panel (**a**) is

$$L = d_{AD} + d_{DB} + d_{BG} + d_{GF} + d_{FC} + d_{CE} + d_{EA}. \tag{4.1}$$

Paths are represented in terms of $k \times k$ matrices as follows. Each row of the matrix corresponds to a city, and the j-th element in this row has the entry 1 if the city is the j-th stop along the path. The other entries are zero. Path (**a**), for example, is represented by the matrix

$$\begin{array}{c} \text{(stop)} \\ \\ \mathbb{M} = \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \\ \text{E} \\ \text{F} \\ \text{G} \end{array} \quad \text{(city)}. \tag{4.2}$$

Since each city is visited only once, there can be only one 1 in each row. Since each visit corresponds to exactly one city, there can be only one 1 in each column. Any permutation of the elements that satisfies these constraints is an allowed path. There are $k!$ such permutations. They are $2k$-fold degenerate (there are k paths of the same length that differ by which city is visited first, and each path can be traveled clockwise or anti-clockwise). Therefore there are $k!/(2k)$ possible paths to consider in trying to determine the shortest one. This makes the problem hard. Note that *integer linear-programming* methods for solving the travelling-salesman problem usually use a different representation of the paths [32].

The *k-queens problem* is derived from the game of chess. The question is how to arrange k queens on a $k \times k$ chess board so that they cannot take each other out.

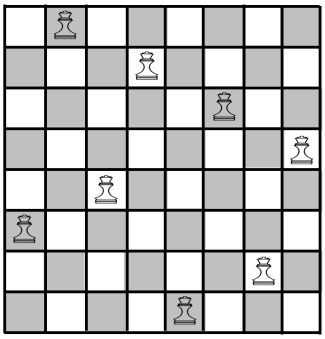


Figure 4.3: One solution of the k-queens problem for $k = 8$.

This means that each row and column as well as each diagonal can have only one queen. The problem is illustrated in Figure 4.3. This Figure shows one solution for $k = 8$. The task is to find all solutions. Each arrangement of queens can be represented as a matrix $\mathbb{M}$, where one sets $M_{ij} = 1$ if there is a queen on site (i, j). All other elements are set to zero. To represent valid solutions, $\mathbb{M}$ must satisfy the following constraints [33]

$$\sum_{i=1}^{k}\sum_{j=1}^{k} M_{ij} = k\,, \tag{4.3}$$

$$\text{If} \quad M_{ij} = M_{pq} = 1 \quad \text{for} \quad (i,j) \neq (p,q) \quad \text{then}$$
$$i \neq p \quad \text{and} \quad j \neq q \quad \text{and} \quad i - j \neq p - q \quad \text{and} \quad i + j \neq p + q\,.$$

The *double-digest problem.* The Human Genome sequence was first assembled by piecing together overlapping DNA segments in the right order by making sure that overlapping segments share the same DNA sequence. To this end it is necessary to uniquely identify the DNA segments. The actual DNA sequence of a segment is a unique identifier. But it is sufficient and more efficient to identify a DNA segment by a *fingerprint*, for example the sequence of *restriction sites*. These are short subsequences (four or six base pairs long) that are recognised by enzymes that cut (*digest*) the DNA strand precisely at these sites. A DNA segment is identified by the types and locations of restriction sites that it contains, the so-called *restriction map*.

When a DNA segment is cut by two different enzymes one can experimentally determine the lengths of the resulting fragments. Is it possible to determine how the cuts were ordered in the DNA sequence of the segment from the fragment lengths? This is the double-digest problem [34]. The order of cut sites is precisely the restriction map. In a double-digest experiment, a given DNA sequence is first digested by one enzyme (A say). Assume that this results in n fragments with lengths a_i $(i = 1, \ldots, n)$. Second, the DNA sequence is digested by another enzyme, B. In this

case m fragments are found, with lengths $b_1, b_2, \ldots, b_m$. Third, the DNA sequence is digested with both enzymes A and B, yielding l fragments with lengths $c_1, \ldots, c_l$. The question is now to determine all possible ordering of the a- and b-cuts that result in l fragments with lengths $c_1, c_2, \ldots, c_l$?

4.2 Energy functions

The first two problems introduced in the previous Section are similar in that possible solutions can be represented in terms of $k \times k$ matrices $\mathbb{M}$ with 0/1 entries and certain constraints. It turns out that one can represent these problems in terms of a Hopfield network with $N = k^2$ neurons n_{ij} that take the values 0/1 [35].

As an example, consider the travelling-salesman problem. We label the cities A to G by integers $m = 1, \ldots, k$, and denote their distances by d_{mn}. Then the path length to be minimised can be written as [35]

$$L = \tfrac{1}{2} \sum_{mnj} d_{mn} M_{mj} (M_{nj-1} + M_{nj+1}) \tag{4.4}$$

(Exercise 4.1). Here periodic boundary conditions in j were assumed ($j+1 = 1$ for $j = k$). The column- and row-constraints upon the matrix $\mathbb{M}$

$$\sum_j M_{mj} = 1 \quad \text{row constraint}, \tag{4.5a}$$

$$\sum_m M_{mj} = 1 \quad \text{column constraint} \tag{4.5b}$$

are incorporated using *Lagrange multipliers* λ_1 and λ_2 (both positive), so that the function to be minimised becomes

$$H = L + \tfrac{\lambda_1}{2} \sum_m \Big(1 - \sum_j M_{mj}\Big)^2 + \tfrac{\lambda_2}{2} \sum_j \Big(1 - \sum_m M_{mj}\Big)^2 . \tag{4.6}$$

When the constraints (4.5) are satisfied, their contributions to H vanish, otherwise they are positive. We conclude that H has a global minimum at the desired solution. If we use a stochastic method to minimise H, it is not guaranteed that the algorithm finds the global minimum, either because the constraints are not exactly satisfied, or because the path found is not the shortest one. The magnitudes of the Lagrange multipliers λ_1 and λ_2 determine how strongly the constraints are enforced, during the search and in sub-optimal solutions.

The expression (4.6) is a quadratic function of M_{mj}. This suggests that we can write H as the energy function of a Hopfield model with 0/1 neurons n_{ij} (Exercise 2.8):

$$H = -\frac{1}{2} \sum_{ijkl} w_{ijkl} n_{ij} n_{kl} + \sum_{ij} \mu_{ij} n_{ij} + \text{constant}. \tag{4.7}$$

The weights w_{ijkl} and thresholds μ_{ij} are determined by comparing Equations (4.4), (4.6), and (4.7). Note that the neurons carry two indices (not one as in Section 1.2). It turns out that the weights are symmetric, $w_{ijkl} = w_{klij}$ and that the diagonal weights are zero (Exercise 4.2). Since $n_{ij}^2 = n_{ij}$ we can always assume that the diagonal weights vanish, because they can be re-interpreted as thresholds. The first term in Equation (4.7) is familiar [Equation (2.49)]. The second term is a threshold term, as in Equation (2.54). The constant term can be dropped, it does not affect the minimisation.

4.3 Simulated annealing

Since the weights w_{ijkl} are symmetric and the diagonal weights w_{ijij} are zero, only updates satisfying $H(n'_{ij}) - H(n_{ij}) \leq 0$ occur under the deterministic dynamics (1.2),

$$n'_{ij} = \theta_{\mathrm{H}}(b_{ij}) \tag{4.8}$$

with $b_{ij} = \sum_{kl} w_{ijkl} n_{kl} - \mu_{ij}$. As a consequence, the network dynamics stops in local minima. To avoid this, one introduces noise, as explained in Section 3.1. The stochastic rule for 0/1 units takes almost the same for as the one for $+1/-1$ units [Equation (3.2)]:

$$n'_{ij} = \begin{cases} 1 & \text{with probability} \quad p(b_{ij}), \\ 0 & \text{with probability} \quad 1 - p(b_{ij}), \end{cases} \tag{4.9}$$

The only difference is that now $p(b) = \frac{1}{1+e^{-\beta b}}$, which differs from Equation (3.2b) by a factor of 2 in the argument of the exponential function. For $\beta \to \infty$ the stochastic rule reduces to the deterministic dynamics where $H(n'_{ij}) - H(n_{ij}) \leq 0$. At large but finite values of β the energy function may increase in an update, yet down-moves are much more likely than up-moves. This becomes particularly clear in an alternative yet equivalent formulation of the network dynamics:

1. Choose at random a unit ij.

2. *Change* n_{ij} to $n'_{ij} \neq n_{ij}$ with probability

$$\text{Prob}(n_{ij} \to n'_{ij}) = \frac{1}{1+e^{\beta \Delta H_{ij}}}, \tag{4.10a}$$

where

$$\Delta H_{ij} = H(\ldots, n'_{ij}, \ldots) - H(\ldots, n_{ij}, \ldots). \tag{4.10b}$$

Equation (4.10) shows that moves with $\Delta H_{ij} > 0$ are less likely when β is large. This means that the energy tends to decrease. For the Hopfield net, Equation (4.10) is equivalent to Equation (4.9). To demonstrate this, consider

$$H(\ldots, n_{ij}=1, \ldots) - H(\ldots, n_{ij}=0, \ldots) = -\tfrac{1}{2}\sum_{kl}(w_{ijkl} + w_{klij})n_{kl} + \mu_{ij} = -b_{ij}, \tag{4.11a}$$

and

$$H(\ldots, n_{ij}=0, \ldots) - H(\ldots, n_{ij}=1, \ldots) = \tfrac{1}{2}\sum_{kl}(w_{ijkl} + w_{klij})n_{kl} - \mu_{ij} = b_{ij}. \tag{4.11b}$$

This means that $\Delta H_{ij} = -b_{ij}(n'_{ij} - n_{ij})$. Here we used that the weights are symmetric, and that the diagonal weights vanish. Next, we break the prescription (4.10) up into different cases. *Changes* occur with the following probabilities:

$$\text{if } n_{ij}=0 \quad \text{obtain } n'_{ij}=1 \text{ with prob.} \quad \frac{1}{1+e^{-\beta b_{ij}}} = p(b_{ij}), \tag{4.12a}$$

$$\text{if } n_{ij}=1 \quad \text{obtain } n'_{ij}=0 \text{ with prob.} \quad \frac{1}{1+e^{\beta b_{ij}}} = 1 - p(b_{ij}). \tag{4.12b}$$

In the second row we used that $1 - p(b) = 1 - \frac{1}{1+e^{-\beta b}} = \frac{1+e^{-\beta b}-1}{1+e^{-\beta b}} = \frac{1}{1+e^{\beta b}}$. The state remains unchanged with the complimentary probabilities:

$$\text{if } n_{ij}=0 \quad \text{obtain } n_{ij}=0 \text{ with prob.} \quad \frac{1}{1+e^{-\beta b_{ij}}} = 1 - p(b_{ij}), \tag{4.12c}$$

$$\text{if } n_{ij}=1 \quad \text{obtain } n_{ij}=1 \text{ with prob.} \quad \frac{1}{1+e^{\beta b_{ij}}} = p(b_{ij}). \tag{4.12d}$$

Comparing with Equation (4.9) we see that the two schemes (4.9) and (4.10) are equivalent. But Equation (4.10) is more general than the stochastic Hopfield dynamics. Equation (4.10) does not require the energy function to be of the form (4.7), and in particular it is neither needed that the weights are symmetric, nor that the

diagonal weights are non-negative. But note that Equations (4.9) and (4.10) are not equivalent if these conditions are not satisfied. See also Exercise 4.3.

How does the network find a solution of the optimisation problem? We let it run with stochastic dynamics and compute $\langle n_{ij} \rangle$. If $\langle n_{ij} \rangle \simeq 1$, we set $M_{ij} = 1$, otherwise 0. When the noise is weak, we expect that all $\langle n_{ij} \rangle$ are either be close to zero or to one. One strategy is to change the noise level as the simulation proceeds. One starts with larger noise, so that the network explores first the rough features of the energy landscape. As the simulation proceeds, one reduces the noise level, so that the network can learn finer features of the landscape. This is called *simulated annealing*. See Section 10.9 in *Numerical Recipes* [36].

4.4 Monte-Carlo simulation

The algorithms described in the previous Section are equivalent to the *Markov-chain Monte-Carlo* algorithm, which in turn is closely related to the *Metropolis* algorithm. This method is widely used in Statistical Physics and in Mathematical Statistics. It is therefore important to understand the connections between the different formulations.

The Markov-chain Monte-Carlo algorithm is a method to sample from a distribution that is too expensive to compute. This is not a contradiction in terms! An important example is the so-called Boltzmann distribution. It was mentioned in Chapter 3 that the state vector $\boldsymbol{s}(t)$ is distributed in this way [Equation (3.49)] in the limit of $t \to \infty$. This follows from the form of the stochastic update rule (3.2b), as we shall see shortly.

More generally, the Boltzmann distribution describes the probabilities of observing configurations of a large class of physical systems in their steady states. The statistical mechanics of systems with energy function (Hamiltonian) H shows that their configurations are distributed according to the Boltzmann distribution in thermodynamic equilibrium at a given temperature (in this context $\beta^{-1} = k_{\mathrm{B}} T$ where k_{B} is the Boltzmann constant), and free from any other constraints. If we denote the configuration of a system by the vector $\boldsymbol{n}$, then the Boltzmann distribution takes the form

$$P_{\mathrm{B}}(\boldsymbol{n}) = Z^{-1} \mathrm{e}^{-\beta H(\boldsymbol{n})} \tag{4.13}$$

Here $Z = \sum_{\boldsymbol{n}} \mathrm{e}^{-\beta H(\boldsymbol{n})}$ is a normalisation factor, called *partition function*. For systems with a large number of interacting degrees of freedom, the function $P_{\mathrm{B}}(\boldsymbol{n})$ can be very expensive to compute, because then the sum over $\boldsymbol{n}$ in the normalisation factor Z contains many terms. Therefore, instead of computing the distribution directly

one constructs a *Markov chain* of states

$$\boldsymbol{n}_1 \to \boldsymbol{n}_2 \to \boldsymbol{n}_3 \to \ldots \tag{4.14}$$

that are distributed according to $P_{\rm B}(\boldsymbol{n})$, possibly after an initial transient. A Markov chain is a *memoryless* random sequence of states defined by *transition probabilities* $p_{l\to k}$ from state $\boldsymbol{n}_l$ to $\boldsymbol{n}_k$ [37]. The transition probability $p_{l\to k}$ connects arbitrary states, allowing for local moves (as in the previous Section where only one element of $\boldsymbol{n}$ was changed) or global moves – where many neurons are allowed to change in a single step.

One step of the Monte-Carlo algorithm consists of two parts. First, assume that you are at state $\boldsymbol{n}_l$. A new state $\boldsymbol{n}_k$ is suggested with probability $p^{(\rm s)}_{l\to k}$. Second, the new state $\boldsymbol{n}_k$ is accepted with probability $p^{(\rm a)}_{l\to k}$. As result, the transition probability is given by a product of two factors, $p_{l\to k} = p^{(\rm s)}_{l\to k} p^{(\rm a)}_{l\to k}$. These steps are repeated many times, creating a sequence of states. If the process satisfies the *detailed-balance condition*

$$P_{\rm B}(\boldsymbol{n}_l) p_{l\to k} = P_{\rm B}(\boldsymbol{n}_k) p_{k\to l}\,, \tag{4.15}$$

then the Markov chain of states $\boldsymbol{n}_l$ has the steady-state distribution $P_{\rm B}(\boldsymbol{n})$. Usually this means that the distribution of states generated in this way converges to $P_{\rm B}(\boldsymbol{n})$ (see Ref. [37] for details).

Now consider the formulation of the network dynamics on page 56. The dynamics is local: a single neuron is picked randomly so that $p^{(\rm s)}$ is a constant, independent of l or k, and then accepted with the *acceptance probability*

$$p^{(\rm a)}_{l\to k} = \frac{1}{1+{\rm e}^{\beta\Delta H}}\,, \tag{4.16}$$

where $\Delta H = H(\boldsymbol{n}_k) - H(\boldsymbol{n}_l)$. Equation (4.16) shows that downhill steps are more frequently accepted than uphill steps. To prove that condition (4.15) holds, we use that $p^{(\rm s)}$ is symmetric (because it is independent of l or k), and we use Equations (4.13) and (4.16):

$$\frac{p^{(\rm s)}{\rm e}^{-\beta H(\boldsymbol{n}_l)}}{1+{\rm e}^{\beta[H(\boldsymbol{n}_k)-H(\boldsymbol{n}_l)]}} = \frac{p^{(\rm s)}}{{\rm e}^{\beta H(\boldsymbol{n}_l)}+{\rm e}^{\beta H(\boldsymbol{n}_k)}} = \frac{p^{(\rm s)}{\rm e}^{-\beta H(\boldsymbol{n}_k)}}{1+{\rm e}^{\beta[H(\boldsymbol{n}_l)-H(\boldsymbol{n}_k)]}}\,. \tag{4.17}$$

This demonstrates that the Boltzmann distribution is a steady state of the Markov chain. If the simulation converges to the steady state (as it usually does), then states visited by the Markov chain are distributed according to the Boltzmann distribution.

Returning for a moment to Chapter 3, the above reasoning implies that the steady-state distribution for the Hopfield model is the Boltzmann distribution, as stated in Section 3.5. But note that the algorithm applies to energy functions of arbitrary form, not just to the Hopfield network.

In summary, while the Boltzmann distribution may be difficult to evaluate (since Z involves a sum over many states), ΔH is cheap to compute for local moves. But note also that the states in the sequence (4.14) are correlated, in particular when the moves are local, because then subsequent configurations are similar. Generating many quite strongly correlated samples from a distribution is not a very efficient way of sampling this distribution. Sometimes it may therefore be more efficient to suggest global moves instead, to avoid that subsequent states in the Markov chain are similar. But it is not guaranteed that global moves lead to weaker correlations. For global moves, ΔH may be more likely to assume large positive values, so that fewer suggested moves are accepted. As a consequence the Markov chain may stay at certain states, increasing correlations in the sequence. Usually a compromise is most efficient, moves that are neither local nor global.

It is important to stress that the detailed-balance condition must hold for the transition probability $p_{l\to k} = p^{(\mathrm{s})}_{l\to k} p^{(\mathrm{a})}_{l\to k}$, not just for the acceptance probability $p^{(\mathrm{a})}_{l\to k}$. For the local moves discussed above and in the previous Chapters, $p^{(\mathrm{s})}$ is a constant, so that $p_{l\to k} \propto p^{(\mathrm{a})}_{l\to k}$. In this case it is sufficient to check the detailed-balance condition for the accpetance probability. In general, and in particular for global moves, it is necessary to include $p^{(\mathrm{s})}_{l\to k}$ in the detailed-balance check [38].

In practice one uses a slightly different form of the transition probabilities (*Metropolis algorithm*). Assuming that $p^{(\mathrm{s})}$ is constant one takes:

$$p_{l\to k} = p^{(\mathrm{s})} \begin{cases} \mathrm{e}^{-\beta\Delta H} & \text{when} \quad \Delta H > 0\,, \\ 1 & \text{when} \quad \Delta H \le 0\,, \end{cases} \tag{4.18}$$

with $\Delta H = H(\boldsymbol{n}_k) - H(\boldsymbol{n}_l)$. Equation (4.18) has the advantage that the transition probabilities are higher than in (4.16) so that moves are more frequently accepted. That the Metropolis rates obey the detailed-balance condition (4.15) can be seen using Equations (4.13) and (4.18):

$$\begin{aligned} P_{\mathrm{B}}(\boldsymbol{n}_l) p_{l\to k} &= Z^{-1} p^{(\mathrm{s})} \mathrm{e}^{-\beta H(\boldsymbol{n}_l)} \begin{cases} \mathrm{e}^{-\beta[H(\boldsymbol{n}_k)-H(\boldsymbol{n}_l)]} & \text{if} \quad H(\boldsymbol{n}_k) > H(\boldsymbol{n}_l) \\ 1 & \text{otherwise} \end{cases} \\ &= Z^{-1} p^{(\mathrm{s})} \mathrm{e}^{-\beta \max\{H(\boldsymbol{n}_k), H(\boldsymbol{n}_l)\}} \\ &= Z^{-1} p^{(\mathrm{s})} \mathrm{e}^{-\beta H(\boldsymbol{n}_k)} \begin{cases} \mathrm{e}^{-\beta[H(\boldsymbol{n}_l)-H(\boldsymbol{n}_k)]} & \text{if} \quad H(\boldsymbol{n}_l) > H(\boldsymbol{n}_k) \\ 1 & \text{otherwise} \end{cases} \\ &= P_{\mathrm{B}}(\boldsymbol{n}_k) p_{k\to l}\,. \end{aligned} \tag{4.19}$$

The fact that the algorithm produces states distributed according to Equation (4.13) offers a different perspective upon the idea of simulated annealing. Slowly lowering the temperature through the simulation mimics slow cooling of a physical system. It passes through a sequence of quasi-equilibrium Boltzmann distributions with

lower and lower temperatures, until the system finds the global minimum $H_{\min}$ of the energy function at zero temperature, where $P_{\mathrm{B}}(\boldsymbol{n}) = 0$ when $H(\boldsymbol{n}) > H_{\min}$, but $P_{\mathrm{B}}(\boldsymbol{n}) > 0$ when $H(\boldsymbol{n}) = H_{\min}$.

4.5 Summary

In this Chapter it was shown how Hopfield networks can perform optimisation tasks, exploring the energy function with stochastic Hopfield dynamics. This approach is equivalent to the Markov-chain Monte-Carlo algorithm. In simulated annealing one gradually reduces the noise level as the simulation proceeds. This mimics the slow cooling of a physical system, an efficient way of bringing the system into its global optimum.

4.6 Further reading

An older but still good reference for Monte-Carlo methods in Statistical Physics is the book *Monte Carlo methods in Statistical Physics* edited by Binder [39]. A more recent source is the book by Newman and Barkema [40]. For simulated annealing, you can refer to Ref. [41].

4.7 Exercises

4.1 Travelling-salesman problem. Derive Equation (4.4) for the path length in the travelling-salesman problem.

4.2 Weights and thresholds for travelling-salesman problem. Derive expressions for the weights and the thresholds for the energy function (4.7) for the travelling-salesman problem. See Ref. [32].

4.3 Asymmetric weights. Show that Equations (4.9) and (4.10) are not equivalent for the network shown in Figure 2.10. The reason is that the weights are not symmetric.

4.4 Simulated annealing with $S = \pm 1$ units. Show that Equation (4.10) is equivalent to the stochastic rule (3.2), for $H = -\frac{1}{2}\sum_{ij} w_{ij} s_i s_j + \sum_i \theta_i s_i$, if $w_{ij} = w_{ji}$ and $w_{ii} = 0$. Demonstrate that Equations (4.10) and (3.2) are *not* equivalent when $w_{ii} \neq 0$, even if the weights are symmetric and w_{ii} is positive.

$L = 10000$

$a =$ [5976, 1543, 1319, 1120, 42]

$b =$ [4513, 2823, 2057, 607]

$c =$ [4513, 1543, 1319, 1120, 607, 514, 342, 42]

$L = 20000$

$a =$ [8479, 4868, 3696, 2646, 169, 142]

$b =$ [11968, 5026, 1081, 1050, 691, 184]

$c =$ [8479, 4167, 2646, 1081, 881, 859, 701, 691, 184, 169, 142]

$L = 40000$

$a =$ [9979, 9348, 8022, 4020, 2693, 1892, 1714, 1371, 510, 451]

$b =$ [9492, 8453, 7749, 7365, 2292, 2180, 1023, 959, 278, 124, 85]

$c =$ [7042, 5608, 5464, 4371, 3884, 3121, 1901, 1768, 1590, 959, 899, 707, 702, 510, 451, 412, 278, 124, 124, 85]

Table 4.1: Example configurations for the double-digest problem for three different chromosome lengths L. For each example, three ordered fragment sets are given, corresponding to the result of digestion with A, with B, and with both A and B.

4.5 Metropolis algorithm. Use the Metropolis algorithm to generate a Markov chain that samples the exponential distribution $P(x) = \exp(-x)$.

4.6 Double-digest problem. Implement the Metropolis algorithm for the double-digest problem. Denote the ordered set of fragment lengths produced by digesting with enzyme A by $a = \{a_1, \ldots, a_n\}$, where $a_1 \geq a_2 \geq \ldots \geq a_n$. Similarly $b = \{b_1, \ldots, b_m\}$ ($b_1 \geq b_2 \geq \ldots \geq b_m$) for fragment lengths produced by digesting with enzyme B, and $c = \{c_1, \ldots, c_l\}$ ($c_1 \geq c_2 \geq \ldots \geq c_l$) for fragment lengths produced by digesting first with A and then with B. Given permutations σ and μ of the sets a and b correspond to a set of c-fragments we denote it by $\hat{c}(\sigma, \mu)$. Use the energy function $H(\sigma, \mu) = \sum_j c_j^{-1}[c_j - \hat{c}_j(\sigma, \mu)]^2$. Configuration space is the space of all permutation pairs (σ, μ). Local moves correspond to inversions of short subsequence of σ and/or μ. Check that the scheme of suggesting new states is symmetric. This is necessary for the algorithm to converge. The solutions of the double-digest problem are degenerate. Determine the degeneracy of the solutions for the fragment sets shown in Table 4.1.

PART II
SUPERVISED LEARNING

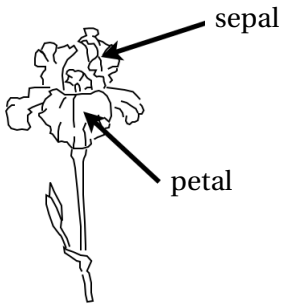
The Hopfield nets described in Part I recognise (or retrieve) patterns. The neurons of the network act as inputs and outputs. In the pattern-recognition problem, a distorted pattern is fed into the network, the recursive network dynamics is run until a steady state is reached. The aim is that the steady-state values of the neurons converge to those of the correct pattern *associated* with the distorted one.

A related type of problem that is very common are *classification* tasks. The *machine-learning repository* [42] at the University of California Irvine contains a large number of such problems. A well-known example is the *iris data set*. It lists attributes of 150 iris plants. The data set was described by the geneticist R. A. Fisher [43]. For each plant four attributes are given (Figure 5.1): its sepal length, sepal width, petal length, and petal width. Also, each plant is classified into one of three classes: *iris setosa*, *iris versicolor*, or *iris virginica*. The task is to program a neural network that determines the class of a plant from its attributes. To each input (attributes of an iris plant) the network should associate the correct output, the class of the plant. The correct output is referred to as the *target*.

In *supervised learning* one uses a *training* data set of correct input/output pairs. One feeds an input from the training data into the input terminals of the network and compares the states of the output neurons to the target values. The weights and thresholds are changed to minimise the differences between network outputs and targets for all input patterns in the training set. In this way the network learns to associate input patterns in the training set with the correct target values. A crucial question is whether the trained network can *generalise*: does it find the correct targets for input patterns that were not in the training set?

The networks used for supervised learning are called *perceptrons* [12, 13]. They consist of layers of McCulloch-Pitts neurons: an input layer, a number of *hidden* layers, and an output layer. The layers are usually arranged from the left (input) to the right (output). All connections are one-way, from neurons in one layer to neurons in the layer immediately to the right. There are no connections between neurons in a given layer, or back to layers on the left. This arrangement ensures convergence of the training algorithm (*stochastic gradient descent*). During training with this algorithm the weights are updated iteratively. In each step, an input is applied and the weights of the network are updated to reduce the error in the output. In a sense each step corresponds to adding a little bit of Hebb's rule to the weights. This is repeated until the network classifies the training set correctly.

Stochastic gradient descent for multi-layer perceptrons has received much attention recently, after it was realised that networks with many hidden layers can be trained to reliably recognise and classify image data, for self-driving cars for instance but also for other applications (*deep learning*).



sepal length	sepal width	petal length	petal width	classification
6.3	2.5	5.0	1.9	virginica
5.1	3.5	1.4	0.2	setosa
5.5	2.6	4.4	1.2	versicolor
4.9	3.0	1.4	0.2	setosa
6.1	3.0	4.6	1.4	versicolor
6.5	3.0	5.2	2.0	virginica

Figure 5.1: Left: petals and sepals of the iris flower. Right: six entries of the iris data set [42]. All lengths in cm. The whole data set contains 150 entries.

5 Perceptrons

Perceptrons [12, 13] are layered feed-forward networks (Figure 5.2). The leftmost layer contains input terminals (black in Figure 5.2). To the right follows a number of layers of McCulloch-Pitts neurons. The right-most layer of neurons is the output layer where the output of the network is read out. The other neuron layers are called hidden layers, because they feed into other neurons, their states are not read out. All connections w_{ij} are one-way: every neuron (or input terminal) feeds only to neurons in the layer immediately to the right. There are no connections within layers, or back connections, or connections that jump over a layer.

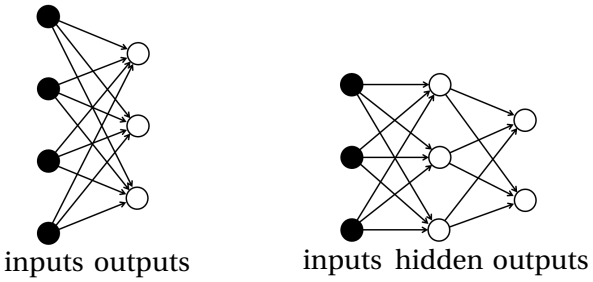


Figure 5.2: Feed-forward network without hidden layer (left), and with one hidden layer (right). The input terminals are coloured black.

There are N input terminals. As in Part I we denote the input patterns by

$$\boldsymbol{x}^{(\mu)} = \begin{bmatrix} x_1^{(\mu)} \\ x_2^{(\mu)} \\ \vdots \\ x_N^{(\mu)} \end{bmatrix} . \tag{5.1}$$

The index μ labels the different input patterns in the training set. It ranges from 1 to p. All neurons are McCulloch-Pitts neurons. The output neurons in the network on the left of Figure 5.2, for example, perform the computation:

$$O_i = g(B_i) \quad \text{with} \quad B_i = \sum_j W_{ij} x_j - \Theta_i \tag{5.2}$$

The index i labels the output neurons, it ranges from 1 to M. Each output neuron has a threshold, Θ_i. In the literature on *deep learning* the thresholds are sometimes referred to as *biases*, defined as $-\Theta_i$. The function g is an activation function as described in Section (1.2). Now consider the network on the right of Figure 5.2. The states of the neurons in the hidden layer are denoted by V_j, with thresholds θ_j and weights w_{jk}. In summary:

$$V_j = g\left(b_j\right) \quad \text{with} \quad b_j = \sum_k w_{jk} x_k - \theta_j , \tag{5.3a}$$

$$O_i = g(B_i) \quad \text{with} \quad B_i = \sum_j W_{ij} V_j - \Theta_i . \tag{5.3b}$$

A classification problem is given by a training set of input patterns $\boldsymbol{x}^{(\mu)}$ and corresponding *target values*

$$\boldsymbol{t}^{(\mu)} = \begin{bmatrix} t_1^{(\mu)} \\ t_2^{(\mu)} \\ \vdots \\ t_M^{(\mu)} \end{bmatrix} . \tag{5.4}$$

The perceptron is trained by choosing its weights and thresholds so that the network produces the desired output.

$$O_i^{(\mu)} = t_i^{(\mu)} \quad \text{for all} \quad i \quad \text{and} \quad \mu . \tag{5.5}$$

If we take $t_i^{(\mu)} = x_i^{(\mu)}$ for $i = 1, \ldots, N$ then the task is pattern recognition, as discussed in Part I. In the Hopfield networks described in Part I, the weights were assigned using Hebb's rule (2.30). Perceptrons, by contrast, are trained by iteratively updating

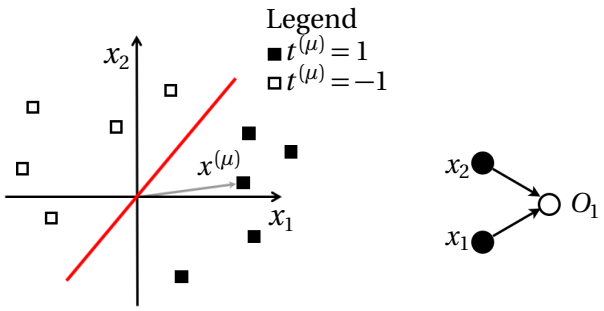


Figure 5.3: Left: classification problem with two-dimensional real-valued inputs and targets equal to ±1. The red line is the decision boundary (see text). Right: corresponding perceptron.

their weights and thresholds until Equation (5.5) is satisfied. This is achieved by iteratively adding small multiples of Hebb's rule to the weights (Section 5.2). An equivalent approach is to define an energy function, a function of the weights of the network that has a global minimum when Equation (5.5) is satisfied. The network is trained by taking small steps in weight space that reduce the energy function (gradient descent, Section 5.3).

5.1 A classification problem

To illustrate how perceptrons can solve classification problems, we consider a very simple example (Figure 5.3). There are ten patterns, each has two real-valued components:

$$\boldsymbol{x}^{(\mu)} = \begin{bmatrix} x_1^{(\mu)} \\ x_2^{(\mu)} \end{bmatrix}. \tag{5.6}$$

In Figure 5.3 the patterns are drawn as points in the x_1-x_2 plane, the *input plane*. There are two classes of patterns, with targets ±1

$$t^{(\mu)} = 1 \quad \text{for} \quad \blacksquare \quad \text{and} \quad t^{(\mu)} = -1 \quad \text{for} \quad \square. \tag{5.7}$$

The activation function consistent with the possible target values is the signum function, $g(b) = \text{sgn}(b)$. The perceptron has two input terminals connected to a single output neuron. Since there is only one neuron, we can arrange the weights into a weight vector

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}. \tag{5.8}$$

The network performs the computation

$$O = \text{sgn}(w_1 x_1 + w_2 x_2 - \theta) = \text{sgn}(\boldsymbol{w} \cdot \boldsymbol{x} - \theta). \tag{5.9}$$

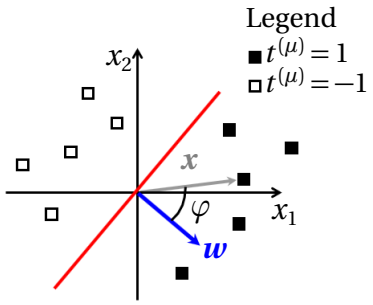


Figure 5.4: The perceptron classifies the patterns correctly for the weight vector $\boldsymbol{w}$ shown, orthogonal to the decision boundary.

Here $\boldsymbol{w}\cdot\boldsymbol{x}=w_1x_1+w_2x_2$ is the scalar product between the vectors $\boldsymbol{w}$ and $\boldsymbol{x}$. This allows us to find a geometrical interpretation of the classification problem. We see in Figure 5.3 that the patterns fall into two clusters: □ to the right and ■ to the left. We can classify the patterns by drawing a line that separates the two clusters, so that everything on the right of the line has $t=1$, while the patterns on the left of the line have $t=-1$. This line is called the *decision boundary*. To find the geometrical significance of Equation (5.9), let us ignore the threshold for a moment, so that

$$O=\mathrm{sgn}(\boldsymbol{w}\cdot\boldsymbol{x}). \tag{5.10}$$

The classification problem takes the form

$$\mathrm{sgn}\left(\boldsymbol{w}\cdot\boldsymbol{x}^{(\mu)}\right)=t^{(\mu)}. \tag{5.11}$$

To evaluate the scalar product we write the vectors as

$$\boldsymbol{w}=|\boldsymbol{w}|\begin{pmatrix}\cos\beta\\ \sin\beta\end{pmatrix}\quad\text{and}\quad\boldsymbol{x}=|\boldsymbol{x}|\begin{pmatrix}\cos\alpha\\ \sin\alpha\end{pmatrix}. \tag{5.12}$$

Here $|\boldsymbol{w}|=\sqrt{w_1^2+w_2^2}$ denotes the norm of the vector $\boldsymbol{w}$, and α and β are the angles of the vectors with the x_1-axis. Then $\boldsymbol{w}\cdot x=|\boldsymbol{w}||\boldsymbol{x}|\cos(\alpha-\beta)=|\boldsymbol{w}||\boldsymbol{x}|\cos\varphi$, where φ is the angle between the two vectors. When φ is between $-\pi/2$ and $\pi/2$, the scalar product is positive, otherwise negative. As a consequence, the network classifies the patterns in Figure 5.3 correctly if the weight vector is orthogonal to the decision boundary drawn in Figure 5.4.

What is the role of the threshold θ? Equation (5.9) shows that the decision boundary is parameterised by $\boldsymbol{w}\cdot\boldsymbol{x}=\theta$, or

$$x_2=-(w_1/w_2)\,x_1+\theta/w_2. \tag{5.13}$$

Therefore the threshold determines the intersection of the decision boundary with the x_2-axis (equal to θ/w_2). This is illustrated in Figure 5.5.

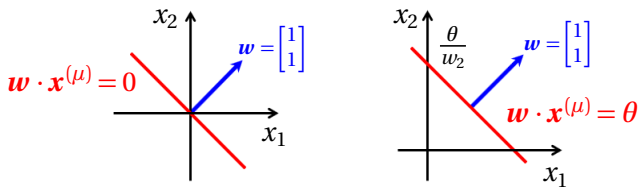


Figure 5.5: Decision boundaries without and with threshold.

The decision boundary – the straight line orthogonal to $\boldsymbol{w}$ – should divide inputs with positive and negative targets. If no such line can be found, then the problem cannot be solved with a single neuron. Conversely, if such a line exists, the problem can be solved (and it is called *linearly separable*). Otherwise the problem is not linearly separable. This can occur only when $p > N$. Examples of problems that are linearly separable and not linearly separable are shown in Figure 5.6.

Other examples are *Boolean functions*. A Boolean function takes N binary inputs and has one binary output. The Boolean AND function (two inputs) is illustrated in Figure 5.7. The value table of the function is shown on the left. The graphical representation is shown in the centre of the Figure (□ corresponds to $t = -1$ and ■ to $t = +1$). Also shown is the decision boundary, the weight vector $\boldsymbol{w}$, and the network layout with the corresponding values of the weights and the threshold. It is important to note that the decision boundary is not unique, neither are the weight and threshold values that solve the problem. The norm of the weight vector, in particular, is arbitrary. Neither is its direction uniquely specified.

Figure 5.8 shows that the Boolean XOR function is not linearly separable [44]. There are 16 different Boolean functions of two variables. Only two are not linearly separable, the XOR and the NOT XOR function.

Up to now we discussed only one output unit. If the classification problem requires several output units, each has its own weight vector $\boldsymbol{w}_i$ and threshold θ_i. We can group the weight vectors into a weight matrix as in Part I, so that the $\boldsymbol{w}_i$ are the rows of $\mathbb{W}$.

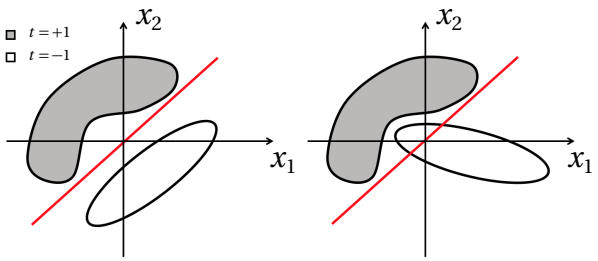
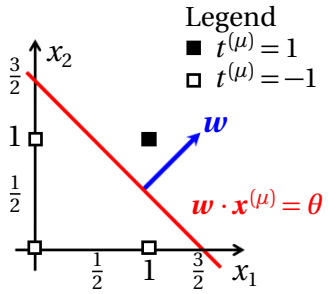


Figure 5.6: Linearly separable and non-separable data.

x_1	x_2	t
0	0	-1
0	1	-1
1	0	-1
1	1	+1



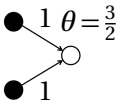


Figure 5.7: Boolean AND function: value table, geometrical representation, and network layout. The weight values are written next to the connections.

5.2 Iterative learning algorithm

In the previous Section we determined the weights and threshold for the Boolean AND function by inspection. Now we discuss an algorithm that allows a computer to find the weights iteratively. How this works is illustrated in Figure 5.9. In panel (**a**), the pattern $\boldsymbol{x}^{(8)}$ ($t^{(8)}=1$) is on the wrong side of the decision boundary. To turn the decision boundary anti-clockwise one *adds* a small multiple of the pattern vector $\boldsymbol{x}^{(8)}$ to the weight vector

$$\boldsymbol{w}'=\boldsymbol{w}+\delta\boldsymbol{w} \quad \text{with} \quad \delta\boldsymbol{w}=\eta\boldsymbol{x}^{(8)}. \tag{5.14}$$

The parameter $\eta>0$ is called the *learning rate*. It must be small, so that the decision boundary is not rotated too far. The result is shown in panel (**b**). Panel (**c**) shows another case, where pattern $\boldsymbol{x}^{(4)}$ ($t^{(4)}=-1$) is on the wrong side of the decision boundary. In order to turn the decision boundary in the right way, anti-clockwise, one *subtracts* a small multiple of $\boldsymbol{x}^{(4)}$:

$$\boldsymbol{w}'=\boldsymbol{w}+\delta\boldsymbol{w} \quad \text{with} \quad \delta\boldsymbol{w}=-\eta\boldsymbol{x}^{(4)}. \tag{5.15}$$

Note the minus sign. These two learning rules combine to

$$\boldsymbol{w}'=\boldsymbol{w}+\delta\boldsymbol{w}^{(\mu)} \quad \text{with} \quad \delta\boldsymbol{w}^{(\mu)}=\eta\, t^{(\mu)}\boldsymbol{x}^{(\mu)}. \tag{5.16}$$

x_1	x_2	t
0	0	-1
0	1	+1
1	0	+1
1	1	-1

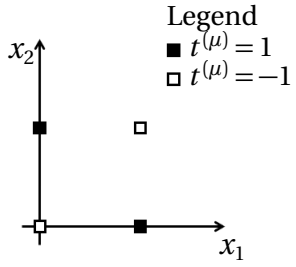


Figure 5.8: The Boolean XOR function is not linearly separable.

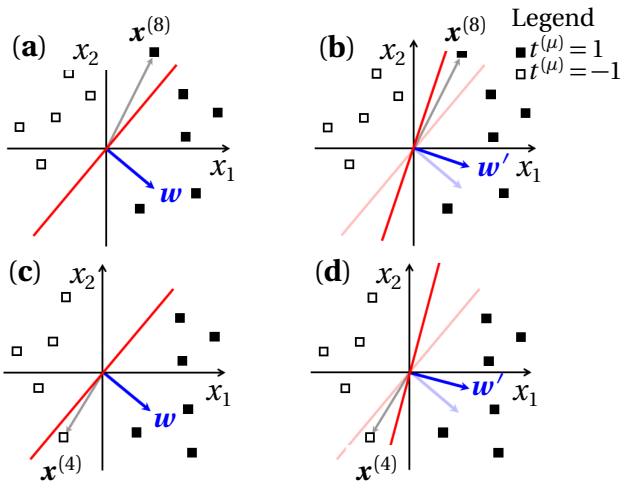


Figure 5.9: Illustration of the learning algorithm. In panel (**a**) the $t=1$ pattern $x^{(8)}$ is on the wrong side of the decision boundary. To correct the error the weight must be rotated anti-clockwise [panel (**b**)]. In panel (**c**) the $t=-1$ pattern $x^{(4)}$ is on the wrong side of the decision boundary. To correct the error the weight must be rotated anti-clockwise [panel (**d**)].

For more than one output unit the rule reads

$$w'_{ij} = w_{ij} + \delta w_{ij}^{(\mu)} \quad \text{with} \quad \delta w_{ij}^{(\mu)} = \eta t_i^{(\mu)} x_j^{(\mu)}. \tag{5.17}$$

This rule is reminiscent of Hebb's rule (2.13), except that here inputs and outputs are associated with distinct units. Therefore we have $t_i^{(\mu)} x_j^{(\mu)}$ instead of $x_i^{(\mu)} x_j^{(\mu)}$. One applies (5.17) iteratively for a sequence of randomly chosen patterns μ, until the problem is solved. This corresponds to adding a little bit of Hebb's rule in each iteration. To ensure that the algorithm stops when the problem is solved, one can use

$$\delta w_{ij}^{(\mu)} = \eta(t_i^{(\mu)} - O_i^{(\mu)}) x_j^{(\mu)}. \tag{5.18}$$

5.3 Gradient-descent learning

In this Section the learning algorithm (5.18) is derived in a different way, by minimising an energy function using gradient descent. This requires differentiation, therefore we must choose a differentiable activation function. The simplest choice is $g(b)=b$. We set $\theta=0$, so that the network computes:

$$O_i^{(\mu)} = \sum_k w_{ik} x_k^{(\mu)} \tag{5.19}$$

(*linear unit*). The outputs $O_i^{(\mu)}$ assume continuous values, but not necessarily the targets $t_i^{(\mu)}$. For linear units, the classification problem

$$O_i^{(\mu)} = t_i^{(\mu)} \quad \text{for} \quad i = 1, \ldots, N \quad \text{and} \quad \mu = 1, \ldots, p \tag{5.20}$$

has the formal solution

$$w_{ik} = \frac{1}{N} \sum_{\mu\nu} t_i^{(\mu)} \left(\mathbb{Q}^{-1} \right)_{\mu\nu} x_k^{(\nu)}, \tag{5.21}$$

as you can verify by inserting Equation (5.21) into (5.19). Here $\mathbb{Q}$ is the overlap matrix with elements

$$Q_{\mu\nu} = \tfrac{1}{N} \boldsymbol{x}^{(\mu)} \cdot \boldsymbol{x}^{(\nu)} \tag{5.22}$$

(page 48). For the solution (5.21) to exist, the matrix $\mathbb{Q}$ must be invertible. As explained in Section 3.6, this requires that $p \leq N$, because otherwise the pattern vectors are *linearly dependent*, and thus also the columns (or rows) of $\mathbb{Q}$. If the matrix $\mathbb{Q}$ has linearly dependent columns or rows it cannot be inverted.

Let us assume that the patterns are linearly independent, so that the solution (5.21) exists. In this case we can find the solution iteratively. To this end one defines the energy function

$$H(\{w_{ij}\}) = \frac{1}{2} \sum_{i\mu} \left(t_i^{(\mu)} - O_i^{(\mu)} \right)^2 = \frac{1}{2} \sum_{i\mu} \left(t_i^{(\mu)} - \sum_j w_{ij} x_j^{(\mu)} \right)^2. \tag{5.23}$$

Here H is regarded as a function of the weights w_{ij}, unlike the energy function in Part I which is a function of the state-variables of the neurons. The energy function (5.23) is non-negative, and it vanishes for the optimal w_{ij} if the pattern vectors $\boldsymbol{x}^{(\mu)}$ are linearly independent. This solution of the classification problem corresponds to the global minimum of H.

To find the global minimum of H one uses *gradient descent*: one repeatedly updates the weights by adding increments

$$w'_{mn} = w_{mn} + \delta w_{mn} \quad \text{with} \quad \delta w_{mn} = -\eta \frac{\partial H}{\partial w_{mn}}. \tag{5.24}$$

The small parameter $\eta > 0$ is the learning rate. The negative gradient points in the direction of steepest descent of H. The idea is to take many downhill steps until one hopefully (but not necessarily) reaches the global minimum.

To evaluate the derivatives one uses the chain rule together with

$$\frac{\partial w_{ij}}{\partial w_{mn}} = \delta_{im} \delta_{jn}. \tag{5.25}$$

Here δ_{kl} is the Kronecker delta, $\delta_{kl}=1$ if $k=l$ and zero otherwise. So $\delta_{im}\delta_{jn}=1$ only if $i=m$ and $j=n$. Otherwise the product of Kronecker deltas equals zero. Illustration. *The linear function, x, and the constant function are going for a walk. When they suddenly see the derivative approaching, the constant function gets worried. "I'm not worried" says the function x confidently, "I'm not put to zero by the derivative." When the derivative comes closer, it says "Hi! I'm $\partial/\partial y$. How are you?"* Moral: when $i\neq m$ or $j\neq n$ then w_{ij} and w_{mn} are independent variables, so that the derivative (5.25) vanishes.

Equation (5.25) gives

$$\delta w_{mn}=\eta\sum_{\mu}\left(t_m^{(\mu)}-O_m^{(\mu)}\right)x_n^{(\mu)}. \tag{5.26}$$

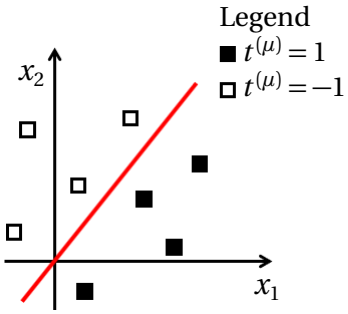
This learning rule is very similar to Equation (5.18). The difference is that Equation (5.26) contains a sum over all patterns (*batch training* or *batch mode*). An advantage of the rule (5.26) is that it is derived from an energy function. This allows to analyse the convergence of the algorithm.

Linear units [Equation (5.19)] are special. You cannot solve the Boolean AND problem (Figure 5.7) with a linear unit – although the problem is linearly separable – because the pattern vectors $\boldsymbol{x}^{(\mu)}$ are linearly dependent. This means that the solution (5.21) does not exist. Shifting the patterns or introducing a threshold does not change this fact. Linear separability does not imply linear independence (but the converse is true).

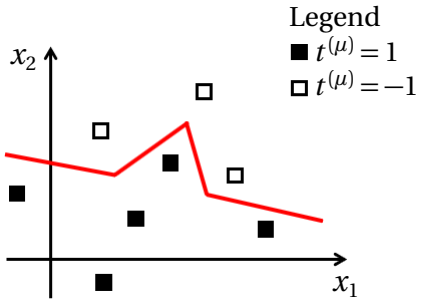
Therefore we usually use *non-linear units*, McCulloch-Pitts neurons with non-linear activation functions $g(b)$. There are four important points to keep in mind. First, for non-linear units it matters less whether or not the patterns are linearly dependent, but it is important whether the problem is linearly separable or not. Second, if the problem is linearly separable then we can use gradient descent to determine suitable weights (and thresholds). Third, for gradient descent we must require that the activation function $g(b)$ is differentiable, or at least piecewise differentiable. Fourth, we calculate the gradients using the chain rule, resulting in factors of derivatives $g'(b)=\frac{\mathrm{d}}{\mathrm{d}b}g(b)$. This is the origin of the *vanishing-gradient problem* (Chapter 7).

5.4 Multi-layer perceptrons

In Sections 5.1 and 5.2 we discussed how to solve linearly separable problems [Figure 5.10(**a**)]. The aim of this Section is to show that non-separable problems like the one in Figure 5.10(**b**) can be solved by a perceptron with one hidden layer. A network that does the trick for the classification problem in Figure 5.10(**b**) is depicted in



(a) Linearly separable problem.



(b) Problem not linearly separable.

Figure 5.10: Problems that are not linearly separable can be solved by a piecewise linear decision boundary.

Figure 5.11. Here the hidden neurons are 0/1 units, but the output neuron gives ±1, as in the previous Section. The network computes with the following rules:

$$\begin{aligned} V_j^{(\mu)} &= \theta_{\mathrm{H}}\left(b_j^{(\mu)}\right) \quad \text{with} \quad b_j^{(\mu)} = \sum_k w_{jk} x_k^{(\mu)} - \theta_j \,, \\ O_1^{(\mu)} &= \mathrm{sgn}\left(B_1^{(\mu)}\right) \quad \text{with} \quad B_1^{(\mu)} = \sum_j W_{1j} V_j^{(\mu)} - \Theta_1 \,. \end{aligned} \tag{5.27}$$

Here $\theta_{\mathrm{H}}(b)$ is the Heaviside function (Figure 1.5). Each of the three neurons in the hidden layer has its own decision boundary. The idea is to choose weights and thresholds in such a way that the three decision boundaries partition the input plane into distinct regions, so that each region contains either only $t=0$ patterns or $t=1$ patterns. We shall see that the values of the hidden neurons encode the different regions. Finally, the output neuron associates the correct target value with each region.

How this construction works is shown in Figure 5.12. The left part of the Figure shows the three decision boundaries. The indices of the corresponding hidden

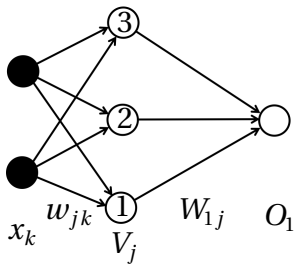
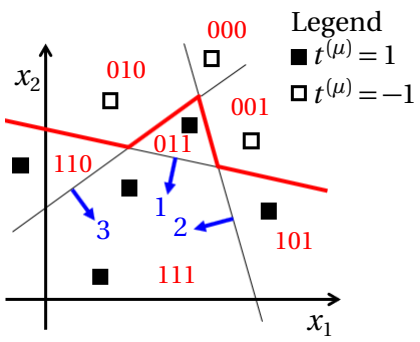


Figure 5.11: Hidden-layer perceptron to solve the problem shown in Figure 5.10 (**b**). The three hidden neurons are 0/1 neurons, the output neuron produces ±1.



V_1	V_2	V_3	target
0	0	0	-1
1	0	0	-
0	1	0	-1
0	0	1	-1
1	1	0	+1
1	0	1	+1
0	1	1	+1
1	1	1	+1

Figure 5.12: Left: decision boundaries and regions. Right: encoding of the regions and corresponding targets. The region 100 does not exist.

neurons are drawn in blue. Also shown are the weight vectors. The regions are encoded with a three-digit binary code. The value of the j-th digit is the value of the j-th hidden neuron: $V_j = 1$ if the pattern is on the weight-vector side of the decision boundary, and $V_j = 0$ on the other side. The Table shows the targets associated with each region, together with the code of the region.

A graphical representation of the output problem is shown in Figure 5.13. The problem is linearly separable. The following function computes the correct output for each region:

$$O_1^{(\mu)} = \mathrm{sgn}\left(V_1^{(\mu)} + V_2^{(\mu)} + V_3^{(\mu)} - \tfrac{3}{2}\right). \tag{5.28}$$

This completes the construction of a solution. It is not unique.

In summary, one can solve non-linearly separable problems by adding a hidden layer. The neurons in the hidden layer define segments of a piecewise linear decision boundary. More neurons are needed if the decision boundary is very wiggly.

Figure 5.14 shows another example, how to solve the Boolean XOR problem with a perceptron that has two 0/1 neurons in a hidden layer, with thresholds $\frac{1}{2}$ and

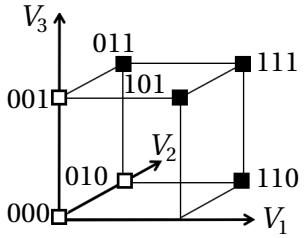
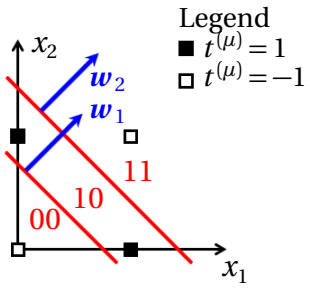
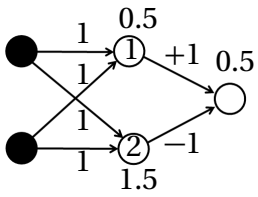


Figure 5.13: Graphical representation of the output problem for the classification problem shown in Figure 5.12.





V_1	V_2	t
0	0	-1
1	0	+1
0	1	+1
1	1	-1

Figure 5.14: Boolean XOR function: geometrical representation, network layout, and value table for the output neuron. The two hidden neurons as 0/1 neurons, the output produces ±1.

$\frac{3}{2}$, and all weights equal to unity. The output neuron has weights +1 and −1 and threshold $\frac{1}{2}$:

$$O_1 = \text{sgn}(V_1 - V_2 - \tfrac{1}{2}). \tag{5.29}$$

Minsky and Papert [44] proved in 1969 that all Boolean functions can be represented by multilayer perceptrons, but that at least one hidden neuron must be connected to *all* input terminals. This means that not all neurons in the network are *locally* connected (have only a few incoming weights). Since fully connected networks are much harder to train, Minsky and Papert offered a somewhat pessimistic view of learning with perceptrons, resulting in a controversy [45]. Now, almost 50 years later, the perspective has changed. Convolutional networks (Chapter 7) have only local connections to the inputs and can be trained to recognise objects in images with high accuracy.

In summary, perceptrons are trained on a training set $(\boldsymbol{x}^{(\mu)}, \boldsymbol{t}^{(\mu)})$ with $\mu = 1, \ldots, p$ by moving the decision boundaries into the correct positions. This is achieved by repeatedly applying Hebb's rule to adjust all weights. This corresponds to using gradient-descent learning on the energy function (5.23). We have not discussed how to update the *thresholds* yet, but it is clear that they can be updated with gradient-descent learning (Section 5.3).

Once all decision boundaries are in the right place we must ask: what happens if we apply the trained network to a new dataset? Does it classify the new inputs correctly? In other words, can the network *generalise*? An example is shown in Figure 5.15. Panel (**a**) shows the result of training the network on a training set. The decision boundary separates $t = -1$ patterns from $t = 1$ patterns, so that the network classifies all patterns in the training set correctly. In panel (**b**) the trained network is applied to patterns in a *validation set*. We see that most patterns are correctly classified, save for one error. This means that the energy function (5.23) is not exactly zero for the validation set. Nevertheless, the network does quite a good job. Usually it is not a good idea to try to precisely classify all patterns near the

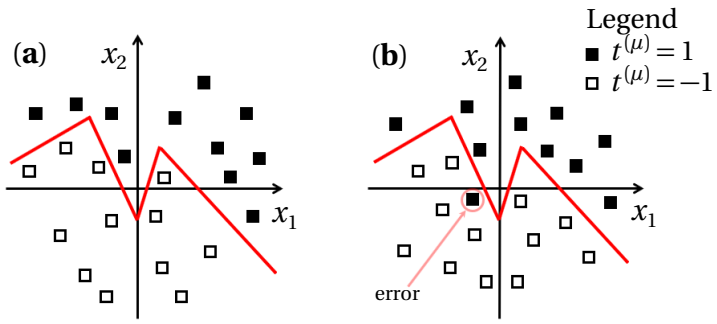


Figure 5.15: (**a**) Result of training the network on a training set. (**b**) Validation by feeding the patterns of a validation set.

decision boundary, because real-world data sets are subject to noise. It is a futile effort to try to learn and predict noise.

5.5 Summary

Perceptrons are layered feed-forward networks that can learn to classify data in a training set $(\boldsymbol{x}^{(\mu)}, \boldsymbol{t}^{(\mu)})$. For each input pattern $\boldsymbol{x}^{(\mu)}$ the network finds the correct targets $\boldsymbol{t}^{(\mu)}$. We discussed the learning algorithm for a simple example: real-valued patterns with just two components, and one binary target. This allowed us to represent the classification problem graphically. There are three different ways of understanding how the perceptron learns. First, geometrically, to learn means to move the decision boundaries into the right places. Second, this can be achieved by repeatedly adding a little bit of Hebb's rule. Third, this algorithm corresponds to gradient descent on the energy function (5.23).

5.6 Further reading

A short account of the history of perceptron research is the review by Kanal [45]. He discusses the work of Rosenblatt [12, 13], McCulloch and Pitts [11], as well as the early controversy around the book by Minsky and Papert [44].

5.7 Exercises

5.1 Boolean AND problem. Show that the Boolean AND problem (Figure 5.7) cannot be solved by the rule (5.21).

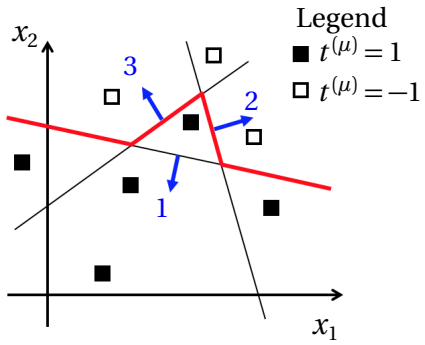


Figure 5.16: Alternative solution of the classification problem shown in Figure 5.12. Exercise 5.4.

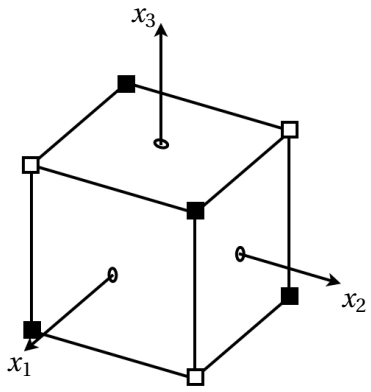


Figure 5.17: Three-dimensional parity problem, with targets $t^{(\mu)} = 1$ (■), $t^{(\mu)} = -1$ (□). Exercise 5.5.

5.2 Boolean functions. How many Boolean functions with three-dimensional inputs are there? How many of them are linearly separable?

5.3 Output problem for binary classification. The binary classification problem shown in Figure 5.12 can be solved with a network with one hidden layer and one output neuron. Figure 5.13 shows the problem that the output neuron has to solve. Show that such output problems are linear separable if the decision boundaries corresponding to the hidden units allow to partition the input plane into distinct regions that contain either only $t = 1$ or only $t = -1$ patterns.

5.4 Piecewise linear decision boundary. Find an alternative solution for the classification problem shown in Figure 5.12, where the weight vectors are chosen as depicted in Figure 5.16.

5.5 Boolean functions. Any N-dimensional Boolean function can be represented using a perceptron with one hidden layer consisting of 2^N neurons. Here we consider

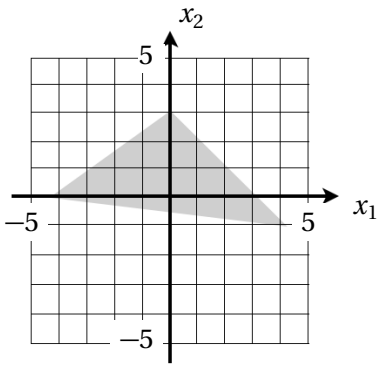


Figure 5.18: Classification problem. Exercise 5.6.

$N = 3$ and $N = 2$. The three-dimensional parity problem is specified in Figure 5.17. The input bits $x_k^{(\mu)}$ for $k = 1, 2, 3$ are either $+1$ or -1. The output $O^{(\mu)}$ of the network is $+1$ if there is an odd number of positive bits in $\boldsymbol{x}^{(\mu)}$, and -1 if the number of positive bits are even. In one solution, the state $V_j^{(\mu)}$ of neuron $j = 1, \ldots, 2^N$ in the hidden layer is given by:

$$V_j^{(\mu)} = \begin{cases} 1 & \text{if} \quad -\theta_j + \sum_k w_{jk} x_k^{(\mu)} > 0\,, \\ 0 & \text{if} \quad -\theta_j + \sum_k w_{jk} x_k^{(\mu)} \leq 0\,, \end{cases} \tag{5.30}$$

where the weights and thresholds are given by $w_{jk} = x_k^{(j)}$ and $\theta_j = 2$. The network output is computed as $O^{(\mu)} = \sum_j W_j V_j^{(\mu)}$. Determine the weights W_j. Draw the decision boundaries of the hidden neurons for the special case $N = 2$ (XOR problem).

5.6 Linearly inseparable problem. A classification problem is specified in Figure 5.18. If an input $\boldsymbol{x}^{(\mu)}$ lies inside the gray triangle its target is $t^{(\mu)} = 1$, if $\boldsymbol{x}^{(\mu)}$ lies outside the triangle, then the target is $t^{(\mu)} = 0$. How patterns on the triangle boundary are classified is not important. The problem can be solved by a perceptron with one hidden layer with three neurons $V_j^{(\mu)} = \theta_{\mathrm{H}}\big(-\theta_j + \sum_{k=1}^{2} w_{jk} x_k^{(\mu)}\big)$, for $j = 1, 2, 3$. The network output is computed as $O^{(\mu)} = \theta_{\mathrm{H}}(-\Theta + \sum_{j=1}^{3} W_j V_j^{(\mu)})$. Find weights w_{jk}, W_j and thresholds θ_j, Θ that solve the classification problem.

5.7 Perceptron with one hidden layer. A perceptron has one input layer, one layer of hidden neurons, and one output unit. It receives two-dimensional input patterns $\boldsymbol{x}^{(\mu)} = [x_1^{(\mu)}, x_2^{(\mu)}]^{\mathsf{T}}$. They are mapped to four hidden neurons $V_i^{(\mu)}$ as

$$V_j^{(\mu)} = \begin{cases} 0 & \text{if} \quad -\theta_j + \sum_k w_{jk} x_k^{(\mu)} \leq 0\,, \\ 1 & \text{if} \quad -\theta_j + \sum_k w_{jk} x_k^{(\mu)} > 0\,, \end{cases} \tag{5.31}$$

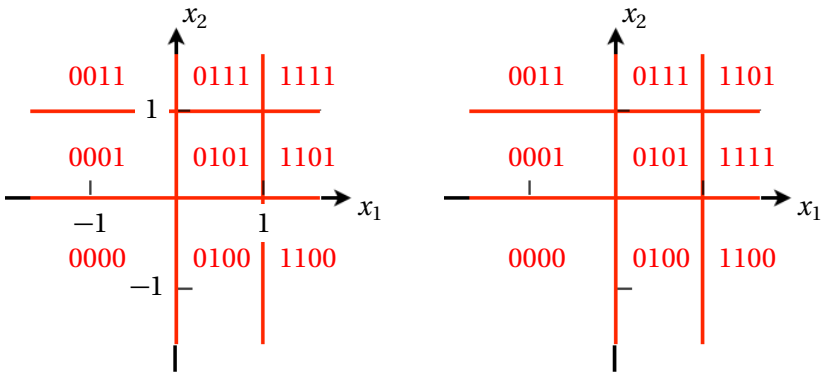


Figure 5.19: Left: input plane with decision boundaries of hidden neurons V_j (red lines). The boundaries partition input space into nine regions labeld by the binary code $V_1 V_2 V_3 V_4$. Right: same, but for a different labeling. Exercise 5.7.

where w_{jk} and θ_j are weights and thresholds of the hidden neurons. The network output is computed as

$$O^{(\mu)} = \begin{cases} 0 & \text{if} \quad -\Theta + \sum_j W_j V_j^{(\mu)} \leq 0\,, \\ 1 & \text{if} \quad -\Theta + \sum_j W_j V_j^{(\mu)} > 0\,, \end{cases} \tag{5.32}$$

with $W_1 = W_3 = W_4 = 1$, $W_2 = -1$, and $\Theta = \frac{1}{2}$. Figure 5.19(left) shows how input space is mapped to the the hidden neurons. Draw the decision boundary of the network. Give values of w_{ij} and θ_i that yield the pattern in Figure 5.19(left). Show that one cannot map the input space to the space of hidden neurons as in Figure 5.19(right).

5.8 Multilayer perceptron. A classification problem is shown in Figure 5.20. It can be solved by a multilayer perceptron with two inputs, three hidden neurons $V_j^{(\mu)} = \theta_H\Big(\sum_{i=1}^{2} w_{jk} x_k^{(\mu)} - \theta_j\Big)$, and one output $O^{(\mu)} = \theta_H\Big(\sum_{j=1}^{3} W_j V_j^{(\mu)} - \Theta\Big)$. A possible solution is illustrated in Fig. 5.20. Compute weights w_{jk} and thresholds θ_j of the hidden neurons that determine the three decision boundaries (red lines). Draw a representation of the problem in the space with axes V_1, V_2, and V_3. Find output weights W_j and threshold Θ that solve the problem.

μ	$x_1^{(\mu)}$	$x_2^{(\mu)}$	$t^{(\mu)}$
1	0.1	0.95	0
2	0.2	0.85	0
3	0.2	0.9	0
4	0.3	0.75	1
5	0.4	0.65	1
6	0.4	0.75	1
7	0.6	0.45	0
8	0.8	0.25	0
9	0.1	0.65	1
10	0.2	0.75	1
11	0.7	0.2	1

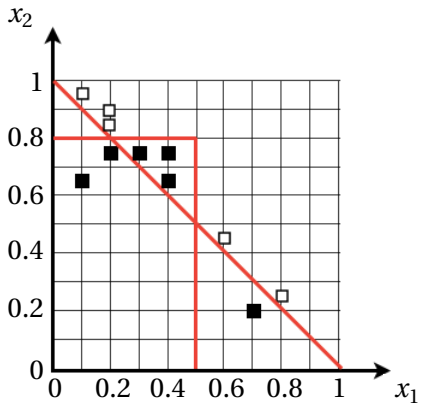


Figure 5.20: Inputs and targets for a classification problem. The targets are either $t^{(\mu)} = 0$ (□) or $t^{(\mu)} = 1$ (■). The three decision boundaries (red lines) illustrate a solution to the problem by a multilayer perceptron. Exercise 5.8.

6 Stochastic gradient descent

In Chapter 5 we discussed how a hidden layer helps to classify problems that are not linearly separable. We explained how the decision boundary in Figure 5.12 is represented in terms of the weights and thresholds of the hidden neurons, and introduced a training algorithm based on gradient descent. In this Section, the training algorithm is discussed in more detail. The solution is explained how it is implemented, why it converges, and how its performance can be improved. Figure 6.1 shows the layout of the network to be trained. There are p input patterns $\boldsymbol{x}^{(\mu)}$ with N components each, as before. The output of the network has M components:

$$\boldsymbol{O}^{(\mu)} = \begin{bmatrix} O_1^{(\mu)} \\ O_2^{(\mu)} \\ \vdots \\ O_M^{(\mu)} \end{bmatrix}, \tag{6.1}$$

to be matched to the targets $\boldsymbol{t}^{(\mu)}$. The activation functions must be differentiable (or at least piecewise differentiable), but apart from that there is no need to specify them further at this point. The network shown in Figure 6.1 performs the computation

$$\begin{aligned} V_j^{(\mu)} &= g\left(b_j^{(\mu)}\right) \quad \text{with} \quad b_j^{(\mu)} = \sum_k w_{jk} x_k^{(\mu)} - \theta_j\,, \\ O_i^{(\mu)} &= g\left(B_i^{(\mu)}\right) \quad \text{with} \quad B_i^{(\mu)} = \sum_j W_{ij} V_j^{(\mu)} - \Theta_i\,. \end{aligned} \tag{6.2}$$

So the outputs are computed in terms of nested activation functions:

$$O_i^{(\mu)} = g\Bigg(\sum_j W_{ij} \underbrace{g\Bigg(\sum_k w_{jk} x_k^{(\mu)} - \theta_j\Bigg)}_{V_j^{(\mu)}} - \Theta_i\Bigg). \tag{6.3}$$

This is a consequence of the network layout of the perceptron: all incoming connections to a given neuron are from the layer immediately to the left, all outgoing connections to the layer immediately to the right. The more hidden layers a network has, the deeper is the nesting of the activation functions.

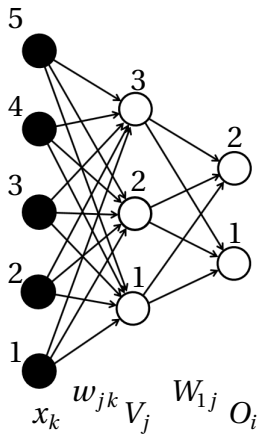


Figure 6.1: Neural network with one hidden layer. Illustrates the notation used in Section 6.1.

6.1 Chain rule and error backpropagation

The network is trained by gradient-descent learning on the energy function (5.23), in the same way as in Section 5.3:

$$H = \frac{1}{2}\sum_{\mu i}\left(t_i^{(\mu)} - O_i^{(\mu)}\right)^2 . \tag{6.4}$$

The weights are updated using the increments

$$\delta W_{mn} = -\eta \frac{\partial H}{\partial W_{mn}} \quad \text{and} \quad \delta w_{mn} = -\eta \frac{\partial H}{\partial w_{mn}} . \tag{6.5}$$

As in Section 5.3, the small parameter $\eta > 0$ is the learning rate. The derivatives of the energy function are evaluated with the *chain rule*. For the weights connecting to the output layer we apply the chain rule once

$$\frac{\partial H}{\partial W_{mn}} = -\sum_{\mu i}\left(t_i^{(\mu)} - O_i^{(\mu)}\right)\frac{\partial O_i^{(\mu)}}{\partial W_{mn}} , \tag{6.6a}$$

and then once more:

$$\frac{\partial O_i^{(\mu)}}{\partial W_{mn}} = \frac{\partial}{\partial W_{mn}} g\Big(\sum_j W_{ij} V_j^{(\mu)} - \Theta_i\Big) = g'(B_i^{(\mu)})\delta_{im} V_n^{(\mu)} . \tag{6.6b}$$

Here $g'(B) = \mathrm{d}g/\mathrm{d}B$ is the derivative of the activation function with respect to the local field B. An important point here is that the values V_j of the neurons in

the hidden layer do not depend on W_{mn}. The neurons V_j do not have incoming connections with these weights, a consequence of the *feed-forward layout* of the network. In summary we obtain for the increments of the weights connecting to the output layer:

$$\delta W_{mn} = -\eta \frac{\partial H}{\partial W_{mn}} = \eta \sum_{\mu=1}^{p} \underbrace{\left(t_m^{(\mu)} - O_m^{(\mu)}\right) g'\left(B_m^{(\mu)}\right)}_{\equiv \Delta_m^{(\mu)}} V_n^{(\mu)} \tag{6.7}$$

The quantity $\Delta_m^{(\mu)}$ is a weighted *error*: it vanishes when $O_m^{(\mu)} = t_m^{(\mu)}$.

The weights connecting to the hidden layer are updated in a similar fashion, by applying the chain rule three times:

$$\frac{\partial H}{\partial w_{mn}} = -\sum_{\mu i} \left(t_i^{(\mu)} - O_i^{(\mu)}\right) \frac{\partial O_i^{(\mu)}}{\partial w_{mn}}, \tag{6.8a}$$

$$\frac{\partial O_i^{(\mu)}}{\partial w_{mn}} = \frac{\partial}{\partial w_{mn}} g\Big(\sum_j W_{ij} V_j^{(\mu)} - \Theta_i\Big) = g'(B_i^{(\mu)}) \sum_j W_{ij} \frac{\partial V_j^{(\mu)}}{\partial w_{mn}} \tag{6.8b}$$

$$\frac{\partial V_j^{(\mu)}}{\partial w_{mn}} = \frac{\partial}{\partial w_{mn}} g\Big(\sum_k w_{jk} x_k^{(\mu)} - \theta_j\Big) = g'(b_j^{(\mu)}) \delta_{jm} x_n^{(\mu)}. \tag{6.8c}$$

Here δ_{jm} is the Kronecker delta (δ_{jm}=1 if $j = m$ and zero otherwise). Taking these results together, one has

$$\delta w_{mn} = \eta \sum_{\mu} \underbrace{\sum_i \Delta_i^{(\mu)} W_{im} g'\left(b_m^{(\mu)}\right)}_{=\delta_m^{(\mu)}} x_n^{(\mu)} \tag{6.9}$$

The quantities $\delta_m^{(\mu)}$ are errors associated with the hidden layer (they vanish when the output errors $\Delta_i^{(\mu)}$ are zero). Equation (6.9) shows that the errors are determined recursively, in terms of the errors in the layer to the right:

$$\delta_m^{(\mu)} = \sum_i \Delta_i^{(\mu)} W_{im} g'\left(b_m^{(\mu)}\right). \tag{6.10}$$

In other words, the error $\delta_m^{(\mu)}$ for the hidden layer is computed in terms of the output errors $\Delta_i^{(\mu)}$. Equations (6.7) and (6.9) show that the weight increments have the same form:

$$\delta W_{mn} = \eta \sum_{\mu=1}^{p} \Delta_m^{(\mu)} V_n^{(\mu)} \quad \text{and} \quad \delta w_{mn} = \eta \sum_{\mu=1}^{p} \delta_m^{(\mu)} x_n^{(\mu)}. \tag{6.11}$$

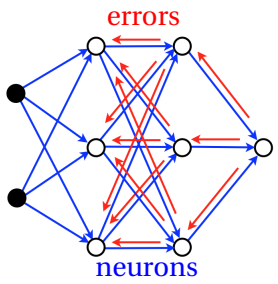


Figure 6.2: Backpropagation algorithm: the states of the neurons are updated forward (from left to right) while errors are updated backward (right to left).

The rule (6.11) is sometimes referred to as the δ-rule. It is *local*: the increments of the weights feeding into a certain layer are determined by the errors associated with that layer, and by the states of the neurons in the layer immediately to the left.

If the network has more hidden layers, then their errors are computed recursively using Equation (6.10), and the formula for the weight increments has the same form as Equation (6.11) (Algorithm 2). Figure 6.2 illustrates the different ways in which neurons and errors are updated. The feed-forward structure of the layered network means that the neurons are updated from left to right (blue arrows). Equation (6.10), by contrast, implies that the errors are updated from the right to the left (red arrows), from the output layer to the hidden layer. The term *backpropagation* refers to this difference: the neurons are updated forward, the errors are updated backward.

The thresholds are updated in a similar way:

$$\delta\Theta_m = -\eta\frac{\partial H}{\partial \Theta_m} = \eta\sum_\mu \left(t_m^{(\mu)} - O_m^{(\mu)}\right)\left[-g'\left(B_m^{(\mu)}\right)\right] = -\eta\sum_\mu \Delta_m^{(\mu)}, \tag{6.12a}$$

$$\delta\theta_m = -\eta\frac{\partial H}{\partial \theta_m} = \eta\sum_\mu\sum_i \Delta_i^{(\mu)} W_{im}\left[-g'\left(b_m^{(\mu)}\right)\right] = -\eta\sum_\mu \delta_m^{(\mu)}. \tag{6.12b}$$

The general form for the threshold increments looks like Equation (6.11)

$$\delta\Theta_m = -\eta\sum_\mu \Delta_m^{(\mu)} \quad \text{and} \quad \delta\theta_m = -\eta\sum_\mu \delta_m^{(\mu)}, \tag{6.13}$$

but without the state variables of the neurons (or the inputs), as expected. A way to remember the difference between Equations (6.11) and (6.13) is to note that the formula for the threshold increments looks like the one for the weight increments if one sets the values of the neurons to -1.

Stochastic gradient descent

The backpropagation rules (6.7), (6.9), and (6.12) contain sums over patterns. This corresponds to feeding all patterns at the same time to compute the increments of weights and thresholds (*batch training* or *batch mode*). Alternatively one may choose a single pattern, update the weights by backpropagation, and then continue to iterate these training steps many times. This is called *sequential training*. One *iteration* corresponds to feeding a single pattern, p iterations are called one *epoch* (in batch training, one iteration corresponds to one epoch). If one chooses the patterns randomly, then sequential training results in *stochastic gradient descent*. Since the sum over pattern is absent, the steps do not necessarily decrease the energy function. Their directions fluctuate, but the average weight increment (averaged over all patterns) points downhill. The result is a *stochastic path* through weight and threshold space, less prone to getting stuck in local minima (Chapters 3 and 7).

The stochastic gradient-descent algorithm is summarised in Section 6.2. It applies to networks with *feed-forward* layout, where neurons in a given layer take input only from the neurons in the layer immediately to the left.

Mini batches

In practice, the stochastic gradient-descent dynamics may be too noisy. It is often better to average over a small number of randomly chosen patterns. Such a set is called *mini batch*, of size m_B say. In *stochastic gradient descent with mini batches* one replaces Equations (6.11) and (6.13) by

$$\delta W_{mn} = \eta \sum_{\mu=1}^{m_B} \Delta_m^{(\mu)} V_n^{(\mu)} \quad \text{and} \quad \delta\Theta_m = -\eta \sum_{\mu=1}^{m_B} \Delta_m^{(\mu)}, \tag{6.14}$$

$$\delta w_{mn} = \eta \sum_{\mu=1}^{m_B} \delta_m^{(\mu)} x_n^{(\mu)} \quad \text{and} \quad \delta\theta_m = -\eta \sum_{\mu=1}^{m_B} \delta_m^{(\mu)}.$$

Sometimes the mini-batch rule is quoted with prefactors of m_B^{-1} before the sums. This does not make any fundamental difference, the factors m_B^{-1} can just be absorbed in the learning rate. But when you compare learning rates for different implementations, it is important to check whether or not there are factors of m_B^{-1} in front of the sums in Equation (6.14). How does one assign inputs to mini batches? This is discussed in Section 6.3.1: at the beginning of each epoch, one should randomly *shuffle* the sequence of the input patterns in the training set. Then the first mini batch contains patterns $\mu = 1, \ldots, m_B$, and so forth.

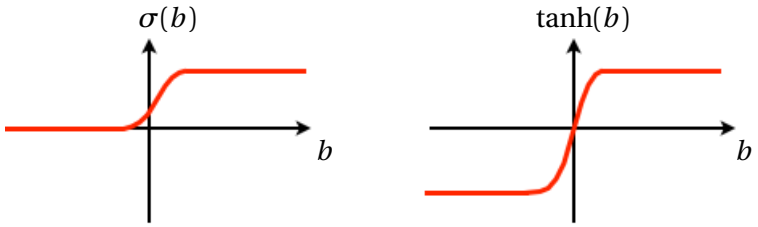


Figure 6.3: Saturation of the activation functions (6.15): the derivative $g'(b)$ tends to zero for large values of $|b|$.

Activation functions

Common choices for $g(b)$ are the *sigmoid* function or tanh:

$$g(b) = \frac{1}{1+e^{-b}} \equiv \sigma(b), \tag{6.15a}$$

$$g(b) = \tanh(b)\,. \tag{6.15b}$$

Their derivatives can be expressed in terms of the function itself:

$$\tfrac{\mathrm{d}}{\mathrm{d}b}\sigma(b) = \sigma(b)[1-\sigma(b)], \quad \tfrac{\mathrm{d}}{\mathrm{d}b}\tanh(b) = \left[1-\tanh^2(b)\right]. \tag{6.16}$$

Other activation functions are discussed in Chapter 7. As illustrated in Figure 6.3, the activation functions (6.15) saturate at large values of $|b|$, so that the derivative $g'(b)$ tends to zero. Since the backpropagation rules (6.7), (6.9), and (6.12) contain factors of $g'(b)$, this implies that the algorithm slows down. It is a good idea to monitor the values of the local fields during training, to check that they do not become too large.

Initialisation of weights and thresholds

The initial weights and thresholds must be chosen so that the local fields are not too large (but not too small either). A standard procedure is to take all weights to be initially randomly distributed, for example Gaussian with mean zero and a suitable variance. The performance of networks with many hidden layers (*deep* networks) can be sensitive to the initialisation of the weights (Section 7.2.4). The initial values of the thresholds are not so critical, they are often learned more rapidly than the weights, at least initially. The thresholds are usually initialised to zero.

Training

The training continues until the global minimum of H has been reached, or until H is deemed sufficiently small. The resulting weights and thresholds are not unique. In Figure 5.14 all weights for the Boolean XOR function are equal to ± 1. But the training algorithm (6.7), (6.9), and (6.12) corresponds to repeatedly adding weight increments. This may cause the weights to grow.

6.2 Stochastic gradient-descent algorithm

The algorithm described in the previous Section applies to networks with any number of layers. It is summarised below (Algorithm 2). We label the layers by the index ℓ. The layer of input terminals has label $\ell = 0$, while the $\ell = L$ denotes the layer of output neurons. The state variables for the neurons in layer ℓ are $V_j^{(\ell)}$, the weights connecting into these neurons from the left are $w_{jk}^{(\ell)}$, the errors associated with layer l are denoted by $\delta_k^{(\ell)}$. In this notation (illustrated in Figure 6.4), Equations (6.2) read:

$$V_j^{(\ell)} = g\Big(\sum_k w_{jk}^{(\ell)} V_k^{(\ell-1)} - \theta_j^{(\ell)}\Big), \tag{6.17}$$

where $b_j^{(\ell)} = \sum_k w_{jk}^{(\ell)} V_k^{(\ell-1)} - \theta_j^{(\ell)}$ is the local field for $V_j^{(\ell)}$. It involves the matrix-vector product between the weight matrix $\mathbb{W}^{(\ell)}$ and the vector $\boldsymbol{V}^{(\ell-1)}$. The errors are propagated backwards as determined by Equation (6.10):

$$\delta_j^{(\ell-1)} = \sum_i \delta_i^{(\ell)} w_{ij}^{(\ell)} g'(b_j^{(\ell-1)}). \tag{6.18}$$

The result, a vector with components $\delta_j^{(\ell-1)}$, is obtained by component-wise multiplication of $[\mathbb{W}^{(\ell)\mathsf{T}} \boldsymbol{\delta}^{(\ell)}]_j$ with $g'(b_j^{(\ell-1)})$. Component-wise multiplication of vectors is somtimes called Schur or Hadamard product [46], denoted by $\boldsymbol{a} \odot \boldsymbol{b} = [a_1 b_1, \ldots, a_N b_N]^{\mathsf{T}}$. It does not have a geometric meaning like the scalar or the cross product of vectors, and therefore there is little point in using it. It is more important to note that the vector $\boldsymbol{\delta}^{(\ell)}$ of errors is multiplied by the transpose of the weight matrix, $\mathbb{W}^{(\ell)\mathsf{T}}$, rather than by the weight matrix itself.

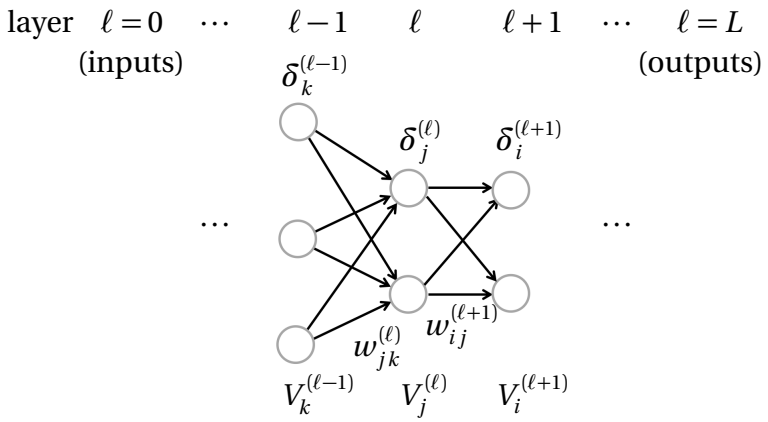


Figure 6.4: Illustrates the notation used in Algorithm 2.

Algorithm 2 stochastic gradient descent

initialise weights $w_{mn}^{(\ell)}$ to random numbers, thresholds to zero, $\theta_m^{(\ell)} = 0$;
for $t = 1, \ldots, T$ **do**
  choose a value of μ and apply pattern $\boldsymbol{x}^{(\mu)}$ to input layer, $\boldsymbol{V}^{(0)} \leftarrow \boldsymbol{x}^{(\mu)}$;
  for $\ell = 1, \ldots, L$ **do**
    propagate forward: $V_j^{(\ell)} \leftarrow g\Big(\sum_k w_{jk}^{(\ell)} V_k^{(\ell-1)} - \theta_j^{(\ell)}\Big)$;
  end for
  compute errors for output layer: $\delta_i^{(L)} \leftarrow g'(b_i^{(L)})(t_i - V_i^{(L)})$;
  for $\ell = L, \ldots, 2$ **do**
    propagate backward: $\delta_j^{(\ell-1)} \leftarrow \sum_i \delta_i^{(\ell)} w_{ij}^{(\ell)} g'(b_j^{(\ell-1)})$;
  end for
  for $\ell = 1, \ldots, L$ **do**
    update: $w_{mn}^{(\ell)} \leftarrow w_{mn}^{(\ell)} + \eta \delta_m^{(\ell)} V_n^{(\ell-1)}$ and $\theta_m^{(\ell)} \leftarrow \theta_m^{(\ell)} - \eta \delta_m^{(\ell)}$;
  end for
end for
end;

6.3 Recipes for improving the performance

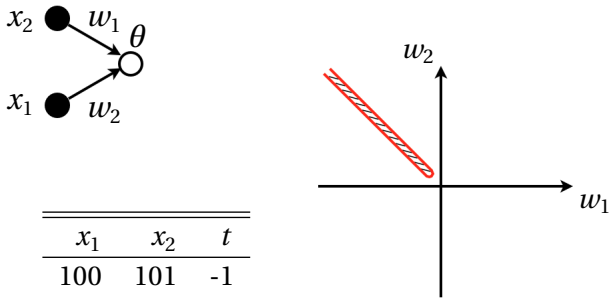
6.3.1 Transformation of input data

It can be useful to preprocess the input data, although any preprocessing may remove information from the data. Nevertheless, it is usually advisable to rigidly shift the data so that its mean vanishes

$$\langle x_k \rangle = \frac{1}{p} \sum_{\mu=1}^{p} x_k^{(\mu)} = 0\,. \tag{6.19}$$

There are several reasons for this. The first one is illustrated in Figure 6.5. The Figure shows how the energy function for a single output with tanh activation and two input terminals depends on the two weights w_1 and w_2. The classification problem is given in the Table. The input data has large mean values in both components, x_1 and x_2. Since it is difficult to visualise the dependence of H on both weights and threshold, the graph on the right shows how the energy function H depends on the weights for zero threshold. The large mean values of the inputs cause steep cliffs in the energy function that are difficult to maneuver with gradient descent.

Different input-data variances in different directions have a similar effect. Therefore one usually *scales* the inputs so that the input-data distribution has the same



x_1	x_2	t
100	101	-1
100	99	1

Figure 6.5: Illustrates the effect of non-zero input mean upon the energy function for one output neuron with tanh activation function and two input terminals. The graph plots the contours of H for $\theta = 0$ for the training set on the left. The plot illustrates that H is close to zero only at the bottom of a very narrow trough (hashed region) with steep sides.

variance in all directions (Figure 6.6), equal to unity for instance:

$$\sigma_k^2 = \frac{1}{p}\sum_{\mu=1}^{p}\left(x_k^{(\mu)} - \langle x_k \rangle\right)^2 = 1 \tag{6.20}$$

Second, to avoid saturation of the neurons connected to the inputs, their local fields must not be too large. If one initialises the weights in the above example to Gaussian random numbers with mean zero and unit variance, large activations are quite likely.

Third, enforcing zero input mean by shifting the input data avoids that the weights of the neurons in the first hidden layer must decrease or increase together [47]. Equation (6.14) shows that the increments $\delta \boldsymbol{w}_m$ into neuron m are likely to have the same signs if the inputs have large mean values. This means that the weight increments have the same signs. This makes it difficult for the network to learn to differentiate.

In summary, one usually shifts and scales the input-data distribution so that it has mean zero and unit variance. This is illustrated in Figure 6.6. The same transformation (using the mean values and scaling factors determined for the training set) should be applied to any new data set that the network is supposed to classify after it has been trained on the training set.

Figure 6.7 shows a distribution of inputs that falls into two distinct clusters. The difference between the clusters is sometimes called *covariate shift*, here *covariate* is just another term for input. Imagine feeding first just inputs from one of the clusters to the network. It will learn local properties of the decision boundary, instead of

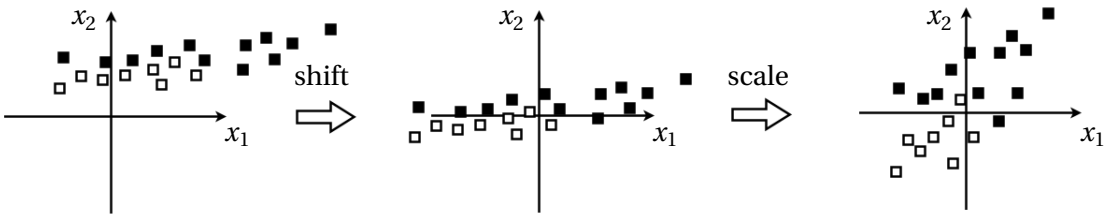


Figure 6.6: Shift and scale the input data to achieve zero mean and unit variance.

its global features. Such global properties are efficiently learned if the network is more frequently confronted with unfamiliar data. For sequential training (stochastic gradient descent) this is not a problem, because the sequence of input patterns presented to the network is random. However, if one trains with mini batches, the mini batches should contain randomly chosen patterns in order to avoid covariate shifts. To this end one randomly *shuffles* the sequence of the input patterns in the training set, at the beginning of each epoch.

It is sometimes recommended [47] to observe the output errors during training. If the errors are similar for a number of subsequent learning steps, the corresponding inputs appear familiar to the network. Larger errors correspond to unfamiliar inputs, and Ref. [47] suggests to feed such inputs more often.

Dimensionality reduction

Often the input data is very high dimensional, requiring many input terminals. This usually means that one should use many neurons in the hidden layers. This can be problematic because it increases the risk of overfitting the input data. To avoid this as far as possible, one can reduce the dimensionality of the input data by *principal-component analysis*. This method allows to project high-dimensional data to a lower dimensional subspace (Figure 6.8).

The data in Figure 6.8(left) falls approximately onto a straight line, the *principal direction*) $\boldsymbol{v}_1$. The coordinate orthogonal to the principal direction is not useful in

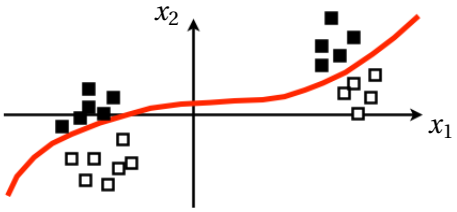


Figure 6.7: When the input data falls into clusters as shown in this Figure, one should randomly pick data from either cluster, to avoid that patterns become too familiar. The decision boundary is shown in red.

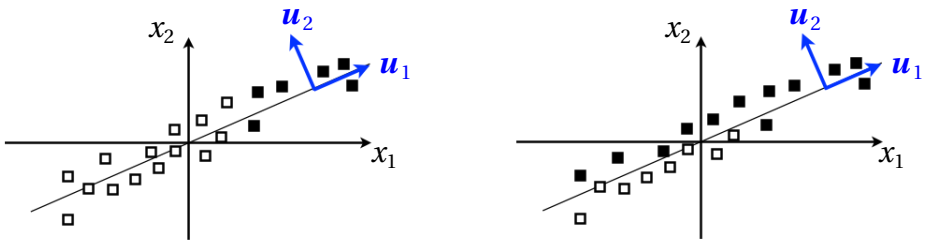


Figure 6.8: Principal-component analysis (schematic). The data set on the left can be classified keeping only the principal component $\boldsymbol{u}_1$ of the data. This is not true for the data set on the right.

classifying the data. Consequently this coordinate can be disregarded, reducing the dimensionality of the data set. But note that the data set shown on the right of Figure 6.8 is much harder to classify if we use only the principal component alone. This illustrates a potential problem: we may lose important information by projecting the data on its principal component.

The idea of principal component analysis is to rotate the basis in input space so that the variance of the data along the first axis of the new coordinate system is maximised, $\boldsymbol{v}_1$ in the example above. The data variance along a direction $\boldsymbol{v}$ reads

$$\sigma_v^2 = \langle (\boldsymbol{x}\cdot\boldsymbol{v})^2\rangle - \langle \boldsymbol{x}\cdot\boldsymbol{v}\rangle^2 = \boldsymbol{v}\cdot\mathbb{C}\boldsymbol{v}\,. \tag{6.21}$$

Here

$$\mathbb{C} = \langle \delta\boldsymbol{x}\,\delta\boldsymbol{x}^\mathsf{T}\rangle \quad \text{with} \quad \delta\boldsymbol{x} = \boldsymbol{x} - \langle \boldsymbol{x}\rangle \tag{6.22}$$

is the data *covariance matrix*. The variance σ_v^2 is maximal when $\boldsymbol{v}$ points in the direction of the leading eigenvector of the covariance matrix $\mathbb{C}$. This can be seen as follows. The covariance matrix is symmetric, therefore its eigenvectors $\boldsymbol{u}_1,\ldots,\boldsymbol{u}_N$ form an orthonormal basis of input space. This allows us to express the matrix $\mathbb{C}$ as

$$\mathbb{C} = \sum_{\alpha=1}^{N} \lambda_\alpha \boldsymbol{u}_\alpha \boldsymbol{u}_\alpha^\mathsf{T}\,. \tag{6.23}$$

The eigenvalues λ_α are non-negative. This follows from Equation (6.22) and the eigenvalue equation $\mathbb{C}\boldsymbol{u}_\alpha = \lambda_\alpha \boldsymbol{u}_\alpha$. We arrange the eigenvalues by magnitude, $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_N \geq 0$. Using Equation (6.23) we can write for the variance

$$\sigma_v^2 = \sum_{\alpha=1}^{N} \lambda_\alpha v_\alpha^2 \tag{6.24}$$

with $v_\alpha = \boldsymbol{v}\cdot\boldsymbol{u}_\alpha$. We want to show that σ_v^2 is maximal for $\boldsymbol{v} = \pm\boldsymbol{u}_1$ subject to the constraint that $\sum_\alpha v_\alpha^2 = 1$. To ensure that the constraint is satisfied one introduces a

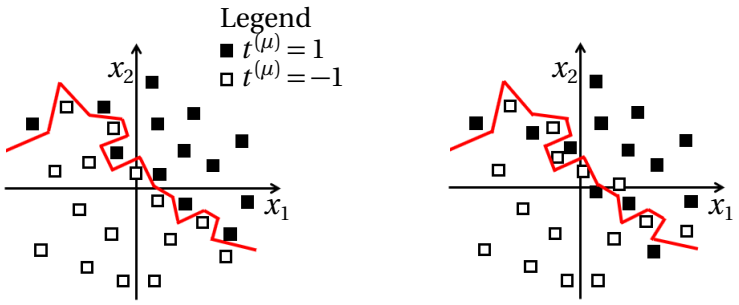


Figure 6.9: Overfitting. Left: accurate representation of the decision boundary in the training set, for a network with 15 neurons in the hidden layer. Right: this new data set differs from the first one just by a little bit of noise. The points in the vicinity of the decision boundary are not correctly classified.

Lagrange multiplier λ as in Chapter 4. The function to maximise reads

$$\mathscr{L} = \sum_\alpha \lambda_\alpha v_\alpha^2 - \lambda\big(1 - \sum_\alpha v_\alpha^2\big). \tag{6.25}$$

Variation of $\mathscr{L}$ yields that $v_\beta(\lambda_\beta + \lambda) = 0$. This means that all components v_α must vanish, except one. The maximum is obtained by $\lambda = -\lambda_1$, where λ_1 is the maximal eigenvalue of $\mathbb{C}$ with eigenvector $\boldsymbol{u}_1$. This shows that the variance σ_v^2 is maximised by the principal direction.

In more than two dimensions there is usually more than one direction along which the data varies significantly. These k principal directions correspond to the k eigenvectors of $\mathbb{C}$ with the largest eigenvalues. This can be shown recursively. One projects the data to the subspace orthogonal to $\boldsymbol{u}_1$ by applying the projection matrix $\mathbb{P}_1 = \mathbb{1} - \boldsymbol{u}_1 \boldsymbol{u}_1^\mathsf{T}$. Then one repeats the procedure outlined above, and finds that the data varies maximally along $\boldsymbol{u}_2$. Iterating one obtains the k principal directions $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k$. Often there is a gap between the k largest eigenvalues and the small ones (all close to zero). Then one can safely project the data onto the subspace spanned by the k principal directions. If there is no gap then it is less clear what to do.

6.3.2 Overfitting

The goal of supervised learning is to generalise from a training set to new data. Only general properties of the training set can be generalised, not specific ones that are particular to the training set and that could be very different in new data. A network with more neurons may classify the training data better, because it accurately represents all specific features of the data. But those specific properties could look quite different in new data (Figure 6.9). As a consequence, we must look for a compromise: between accurate classification of the training set and the ability of the network to

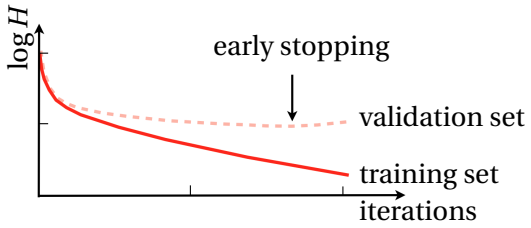


Figure 6.10: Progress of training and validation errors. The plot is schematic, and the data is smoothed. Shown is the natural logarithm of the energy functions for the training set (solid line) and the validation set (dashed line) as a function of the number of training iterations. The training is stopped when the validation energy begins to increase. In Section 7.4 a precise criterion for this *early stopping* is discussed, *validation patience*, a criterion that works for fluctuating data.

generalise. The problem illustrated in Figure 6.9 is also referred to as *overfitting*: the network fits too fine details (for instance noise in the training set) that have no general meaning. The tendency to overfit is larger for networks with more neurons.

One way of avoiding overfitting is to use *cross validation* and *early stopping*. One splits the training data into two sets: a *training set* and a *validation set*. The idea is that these sets share the general features to be learnt. But although training and validation data are drawn from the same distribution, they differ in details that are not of interest. The network is trained on the training set. During training one monitors not only the energy function for the training set, but also the energy function evaluated on the validation data. As long as the network learns general features of the input distribution, both training and validation energies decrease. But when the network starts to learn specific features of the training set, then the validation energy saturates, or may start to increase. At this point the training should be stopped. This scheme is illustrated in Figure 6.10.

Often the possible values of the output neurons are continuous while the targets assume only discrete values. Then it is important to also monitor the *classification error* of the validation set. The definition of the classification error depends on the type of the classification problem. Assume first that there is one output unit, and that the targets take the values $t = 0/1$. Then the classification error can be defined as

$$C = \frac{1}{p}\sum_{\mu=1}^{p}\left|t^{(\mu)} - \theta_{\mathrm{H}}(O^{(\mu)} - \tfrac{1}{2})\right|. \tag{6.26a}$$

If, by contrast, the targets take the values $t = \pm 1$, then the classification error reads:

$$C = \frac{1}{2p}\sum_{\mu=1}^{p}\left|t^{(\mu)} - \mathrm{sgn}(O^{(\mu)})\right|. \tag{6.26b}$$

Now consider a classification problem where inputs must be classified into M

output			targets				correct?
0.4	0.4	0.55	0	0	1	setosa	yes
0.4	0.55	0.4	0	1	0	versicolor	yes
0.1	0.2	0.8	1	0	0	virginica	no

output			targets				correct?
0.1	0.2	0.8	0	0	1	setosa	yes
0.1	0.8	0.2	0	1	0	versicolor	yes
0.4	0.4	0.55	1	0	0	virginica	no

Table 6.1: Illustrates the difference between energy function and classification error. Each table shows network outputs for three different inputs from the iris data set, as well as the correct classfications.

mutually exclusive classes. An example is the MNIST data set of hand-written digits (Section 7.4) where $M = 10$. Another example is given in Table 6.1, with $M = 3$. In both examples one of the targets $t_i^{(\mu)} = 1$ while the others equal zero, for a given input $\boldsymbol{x}^{(\mu)}$. As a consequence, $\sum_i^M t_i^{(\mu)} = 1$. Assume that the network has sigmoid outputs, $O_i^{(\mu)} = \sigma(b_i^{(\mu)})$. To classify input $\boldsymbol{x}^{(\mu)}$ from the network outputs $O_i^{(\mu)}$ we compute for the given value of μ:

$$y_i^{(\mu)} = \begin{cases} 1 & \text{if } O_i^{(\mu)} \text{ is the largest of all outputs } i = 1, \ldots, M, \\ 0 & \text{otherwise.} \end{cases} \tag{6.27a}$$

In this case the classification error can be computed from

$$C = \frac{1}{2p} \sum_{\mu=1}^{p} \sum_{i=1}^{M} \left| t_i^{(\mu)} - y_i^{(\mu)} \right|. \tag{6.27b}$$

In all cases, the *classification accuracy* is defined as $(1-C)100\%$, it is usually quoted in percent.

While the classification error is designed to show the fraction of inputs that are classified wrongly, it contains less information than the energy function (which is in fact a mean-squared error of the outputs). This is illustrated by the two problems in Table 6.1. Both problems have the same classification error, but the energy function is much lower for the second problem, reflecting the better quality of its solution. Yet another measure of classification success is the *cross-entropy error*. It is discussed in Chapter 7.

6.3.3 Weight decay and regularisation

Figure 5.14 shows a solution of the classification problem defined by the Boolean XOR function. All weights are of unit modulus, and also the thresholds are of order unity. If one uses the backpropagation algorithm to find a solution to this problem, one may find that the weights continue to grow during training. This can be problematic, if it means that the local fields become too large, so that the algorithm reaches the plateau of the activation function. Then training slows down, as explained in Section 6.1.

One solution to this problem is to reduce the weights by some factor during training, either at each iteration or in regular intervals, $w_{ij} \to (1-\varepsilon)w_{ij}$ for $0 < \varepsilon < 1$, or

$$\delta w_{mn} = -\varepsilon\, w_{mn} \quad \text{for} \quad 0 < \varepsilon < 1\,. \tag{6.28}$$

This is achieved by adding a term to the energy function

$$H = \underbrace{\frac{1}{2}\sum_{i\mu}\left(t_i^{(\mu)} - O_i^{(\mu)}\right)^2}_{\equiv H_0} + \frac{\gamma}{2}\sum_{ij} w_{ij}^2\,. \tag{6.29}$$

Gradient descent on H gives:

$$\delta w_{mn} = -\eta\frac{\partial H_0}{\partial w_{mn}} - \varepsilon\, w_{mn} \tag{6.30}$$

with $\varepsilon = \eta\gamma$. One can add a corresponding term for the thresholds, but this is usually not necessary. The scheme summarised here is sometimes called L_2*-regularisation*. An alternative scheme is L_1*-regularisation*. It amounts to

$$H = \frac{1}{2}\sum_{i\mu}\left(t_i^{(\mu)} - O_i^{(\mu)}\right)^2 + \frac{\gamma}{2}\sum_{ij}|w_{ij}|\,. \tag{6.31}$$

This gives the update rule

$$\delta w_{mn} = -\eta\frac{\partial H_0}{\partial w_{mn}} - \varepsilon\,\mathrm{sgn}(w_{mn})\,. \tag{6.32}$$

The discontinuity of the update rule at $w_{mn} = 0$ is cured by defining $\mathrm{sgn}(0) = 0$. Comparing Equations (6.30) and (6.32) we see that L_1-regularisation reduces small weights much more than L_2-regularisation. We expect therefore that the L_1-scheme puts more weights to zero, compared with the L_2-scheme.

These two weight-decay schemes are referred to as *regularisation* schemes because they tend to help against overfitting. How does this work? Weight decay

adds a constraint to the problem of minimising the energy function. The result is a compromise, depending upon the value γ, between a small value of H and small weight values. The idea is that a network with smaller weights is more robust to the effect of noise. When the weights are small, then small changes in some of the patterns do not give a substantially different training result. When the network has large weights, by contrast, it may happen that small changes in the input give significant differences in the training result that are difficult to *generalise* (Figure 6.9). Other regularisation schemes are discussed in Chapter 7.

6.3.4 Adaptation of the learning rate

It is tempting to choose larger learning rates, because they enable the network to escape more efficiently from shallow minima. But clearly this causes problems when the energy function varies rapidly. As a result the training may fail because the training dynamics starts to oscillate. This can be avoided by changing the learning rule

$$\delta w_{mn}^{(t)} = -\eta \frac{\partial H}{\partial w_{mn}}\bigg|_{\{w_{ij}\}=\{w_{ij}^{(t)}\}} + \alpha \delta w_{mn}^{(t-1)}. \tag{6.33}$$

Here $t = 1,2,\ldots,T$ labels the iteration number, and You see that the increment at step t depends not only on the instantaneous gradient, but also on the weight increment $\delta w_{mn}^{(t-1)}$ of the previous iteration. We say that the dynamics becomes *inertial*, the weights gain *momentum*. The parameter $\alpha \geq 0$ is called *momentum constant*. It determines how strong the inertial effect is. Obviously $\alpha = 0$ corresponds to the usual backpropagation rule. When α is positive, then how does inertia change the learning rule? Iterating Equation (6.33) yields

$$\delta w_{mn}^{(T)} = -\eta \sum_{t=0}^{T} \alpha^{T-t} \frac{\partial H}{\partial w_{mn}^{(t)}}. \tag{6.34}$$

Here and in the following we use the short-hand notation

$$\frac{\partial H}{\partial w_{mn}^{(t)}} \equiv \frac{\partial H}{\partial w_{mn}}\bigg|_{\{w_{ij}\}=\{w_{ij}^{(t)}\}}.$$

Equation (6.34) shows that $\delta w_{mn}^{(T)}$ is a weighted average of the gradients encountered during training. Now assume that the training is stuck in a shallow minimum. Then the gradient $\partial H/\partial w_{mn}^{(t)}$ remains roughly constant through many time steps. To illustrate what happens, let us assume that $\partial H/\partial w_{mn}^{(t)} = \partial H/\partial w_{mn}^{(0)}$ for $t = 1,\ldots,T$. In this case we can write

$$\delta w_{mn}^{(T)} \approx -\eta \frac{\partial H}{\partial w_{mn}^{(0)}} \sum_{t=0}^{T} \alpha^{T-t} = -\eta \frac{\alpha^{T+1}-1}{\alpha-1} \frac{\partial H}{\partial w_{mn}^{(0)}}. \tag{6.35}$$

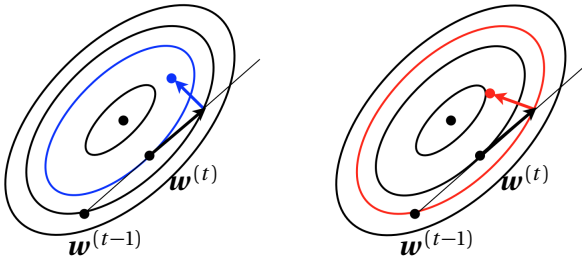


Figure 6.11: Left: Momentum method (6.33). The blue arrow represents the increment $-\eta(\partial H/\partial w_{mn})|_{\{w_{ij}^{(t)}\}}$. Right: Nesterov's accelerated gradient method (6.37). The red arrow represents $-\eta(\partial H/\partial w_{mn})|_{\{w_{ij}^{(t)}+\alpha_{t-1}\delta w_{ij}^{(t-1)}\}}$. The location of $\boldsymbol{w}^{(t+1)}$ (red point) is closer to the minimum (black point) than in the Figure on the left.

In this situation, convergence is accelerated when α is close to unity. We also see that it is necessary that $\alpha < 1$ for the sum in Equation (6.35) to converge. The other limit to consider is that the gradient changes rapidly from iteration to iteration. How is the learning rule modified in this case? As an example, let us assume that the gradient remains of the same magnitude, but that its sign oscillates, $\partial H/\partial w_{mn}^{(t)} = (-1)^t \partial H/\partial w_{mn}^{(0)}$ for $t = 1,\ldots,T$. Inserting this into Equation (6.34), we obtain:

$$\delta w_{mn}^{(T)} \approx -\eta \left|\frac{\partial H}{\partial w_{mn}^{(0)}}\right| \sum_{t=0}^{T} (-1)^t \alpha^{T-t} = -\eta \frac{\alpha^{T+1}+(-1)^T}{\alpha+1} \left|\frac{\partial H}{\partial w_{mn}^{(0)}}\right|, \tag{6.36}$$

so that the increments are much smaller compared with those in Equation (6.35). This shows that introducing inertia can substantially accelerate convergence without sacrificing accuracy. The disadvantage is, of course, that there is yet another parameter to choose, namely the momentum constant α.

Nesterov's *accelerated gradient* method [48] is another way of implementing momentum. The algorithm was developed for smooth optimisation problems, but it is often used in stochastic gradient descent when training deep neural networks. The algorithm can be summarised as follows [49]:

$$\delta w_{mn}^{(t)} = -\eta \frac{\partial H}{\partial w_{mn}}\bigg|_{\{w_{ij}^{(t)}+\alpha_{t-1}\delta w_{ij}^{(t-1)}\}} + \alpha_{t-1}\delta w_{mn}^{(t-1)}. \tag{6.37}$$

A suitable sequence of coefficients α_t is defined by recursion [49]. The coefficients α_t approach unity from below as t increases.

Nesterov's accelerated-gradient method is more efficient than the simple momentum method, because the accelerated-gradient method evaluates the gradient at an extrapolated point, not at the initial point. Figure 6.11 illustrates a situation where Nesterov's method converges more rapidly. Since Nesterov's method often

works better than the simple momentum scheme and because it is not much more difficult to implement, it is used quite frequently. There are other ways of adapting the learning rate during training, see Section 4.10 in Haykin's book [2]. Finally, the learning rate need not be the same for all neurons. If the weights of neurons in different layers change at very different speeds (Section 7.2.1), it may be advantageous to define a layer-dependent learning rate η_ℓ that is larger for neurons with smaller gradients.

6.3.5 Pruning

The term *pruning* refers to removing unnecessary weights or neurons from the network, to improve its efficiency. The simplest approach is *weight elimination* by *weight decay* [50]. Weights that tend to remain very close to zero during training are removed by setting them to zero and not updating them anymore. Neurons that have zero weights for all incoming connections are effectively removed (*pruned*). It has been shown that this method can help the network to generalise [51].

An efficient pruning algorithm is based on the idea to remove weights in such a way that the effect upon the energy function is as small as possible [52]. The idea is to find the optimal weight, to remove it, and to change the other weights in such a way that the energy function increases as little as possible. The algorithm works as follows. Assume that the net was trained, so that it reached a (local) minimum of the energy function H. One expands the energy function around this minimum: The expansion of H reads:

$$H = H_{\min} + \tfrac{1}{2}\delta\boldsymbol{w}\cdot\mathbb{M}\delta\boldsymbol{w} \quad + \quad \text{higher orders in} \quad \delta\boldsymbol{w}\,. \tag{6.38}$$

The term linear in $\delta\boldsymbol{w}$ vanishes because we expand around a local minimum. The matrix $\mathbb{M}$ is the *Hessian*, the matrix of second derivatives of the energy function.

For the next step it is convenient to adopt the following notation [52]. One groups all weights in the network into a long weight vector $\boldsymbol{w}$ (as opposed to grouping them into a weight matrix $\mathbb{W}$ as we did in Chapter 2). A particular component w_q is extracted from the vector $\boldsymbol{w}$ as follows:

$$w_q = \hat{\mathbf{e}}_q\cdot\boldsymbol{w} \quad \text{where} \quad \hat{\mathbf{e}}_q = \begin{bmatrix}\vdots\\ 1 \leftarrow q\\ \vdots\end{bmatrix}. \tag{6.39}$$

Here $\hat{\mathbf{e}}_q$ is the Cartesian unit vector in the direction q, with components $[\hat{\mathbf{e}}_{qj}]_j = \delta_{qj}$. In this notation, the elements of $\mathbb{M}$ are $M_{pq} = \partial^2 H/\partial w_p \partial w_q$.

Now, eliminating the weight w_q amounts to setting

$$\delta w_q = -w_q\,. \tag{6.40}$$

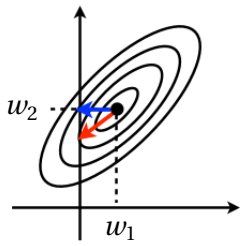


Figure 6.12: Pruning algorithm (schematic). The minimum of H is located at (w_1, w_2). The contours of the quadratic approximation to H are represented as solid black lines. The weight change $\delta \boldsymbol{w} = [-w_1, 0]^\mathsf{T}$ (blue) leads to a smaller increase in H than $\delta \boldsymbol{w} = [0, -w_2]^\mathsf{T}$. The red arrow represents the optimal $\delta \boldsymbol{w}_q^*$ which leads to an even smaller increase in H.

To minimise the damage to the network one wants to eliminate the weight that has least effect upon H, changing the other weights at the same time so that H as little as possible (Figure 6.12). This is achieved by minimising

$$\min_q \min_{\delta \boldsymbol{w}} \{\tfrac{1}{2} \delta \boldsymbol{w} \cdot \mathbb{M} \delta \boldsymbol{w}\} \quad \text{subject to the constraint} \quad \hat{\mathbf{e}}_q \cdot \delta \boldsymbol{w} + w_q = 0\,. \tag{6.41}$$

The constant term $H_{\min}$ was dropped because it does not matter. Now we first minimise H w.r.t. $\delta \boldsymbol{w}$, for a given value of q. The linear constraint is incorporated using a *Lagrange multiplier* as in Section 6.3.1 and in Chapter 4, to form the *Lagrangian*

$$\mathcal{L} = \tfrac{1}{2} \delta \boldsymbol{w} \cdot \mathbb{M} \delta \boldsymbol{w} + \lambda (\hat{\mathbf{e}}_q \cdot \delta \boldsymbol{w} + w_q)\,. \tag{6.42}$$

A necessary condition for a minimum $(\delta \boldsymbol{w}, \lambda)$ satisfying the constraint is

$$\frac{\partial \mathcal{L}}{\partial\, \delta \boldsymbol{w}} = \mathbb{M} \delta \boldsymbol{w} + \lambda \hat{\mathbf{e}}_q = 0 \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial \lambda} = \hat{\mathbf{e}}_q \cdot \delta \boldsymbol{w} + w_q = 0\,. \tag{6.43}$$

We denote the solution of these Equations by $\delta \boldsymbol{w}^*$ and λ^*. It is obtained by solving the linear system

$$\begin{bmatrix} \mathbb{M} & \hat{\mathbf{e}}_q \\ \hat{\mathbf{e}}_q^\mathsf{T} & 0 \end{bmatrix} \begin{bmatrix} \delta \boldsymbol{w}^* \\ \lambda^* \end{bmatrix} = \begin{bmatrix} 0 \\ -w_q \end{bmatrix}. \tag{6.44}$$

If $\mathbb{M}$ is invertible, then the top rows of Eq. (6.44) give

$$\delta \boldsymbol{w}^* = -\mathbb{M}^{-1} \hat{\mathbf{e}}_q \lambda^*\,. \tag{6.45}$$

Inserting this result into $\hat{\mathbf{e}}_q^\mathsf{T} \delta \boldsymbol{w}^* + w_q = 0$ we find

$$\delta \boldsymbol{w}^* = -\mathbb{M}^{-1} \hat{\mathbf{e}}_q \, w_q (\hat{\mathbf{e}}_q^\mathsf{T} \mathbb{M}^{-1} \hat{\mathbf{e}}_q)^{-1} \text{and} \quad \lambda^* = w_q (\hat{\mathbf{e}}_q^\mathsf{T} \mathbb{M}^{-1} \hat{\mathbf{e}}_q)^{-1}\,. \tag{6.46}$$

You see that $\hat{\mathbf{e}}_q \cdot \delta \boldsymbol{w}^* = -w_q$, so that the weight w_q is eliminated. The other weights are also changed (red arrow in Figure 6.12.

The final step is to find the optimal q by minimising

$$\mathscr{L}(\delta \boldsymbol{w}^*, \lambda^*; q) = \frac{1}{2} w_q^2 (\hat{\mathbf{e}}_q^\mathsf{T} \mathbb{M}^{-1} \hat{\mathbf{e}}_q)^{-1} . \tag{6.47}$$

The Hessian of the energy function is expensive to evaluate, and so is the inverse of this matrix. Usually one resorts to an approximate expression for $\mathbb{M}^{-1}$ [52]. One possibility is to set the off-diagonal elements of $\mathbb{M}$ to zero [53]. But in this case the other weights are not adjusted, because $\hat{\mathbf{e}}_{q'} \cdot \delta \boldsymbol{w}_q^* = 0$ for $q' \neq q$, if $\mathbb{M}$ is diagonal. In this case it is necessary to retrain the network after weight elimination.

The algorithm is summarised in Algorithm 3. It succeeds better than elimination by weight decay in removing the unnecessary weights in the network [52]. Weight decay eliminates the smallest weights. One obtains weight elimination of the smallest weights by substituting $\mathbb{M} = \mathbb{I}$ in the algorithm described above [Equation (6.47)].But small weights are often needed to achieve a small training error. So this is usually not a good approximation.

Algorithm 3 pruning least important weight

1: train the network to reach $H_{\min}$;
2: compute $\mathbb{M}^{-1}$ approximately;
3: determine q^* as the value of q for which $\mathscr{L}(\delta \boldsymbol{w}^*, \lambda^*; q)$ is minimal;
4: **if** $\mathscr{L}(\delta \boldsymbol{w}^*, \lambda^*; q^*) \ll H_{\min}$ **then**
5: update all weights using $\delta \boldsymbol{w} = -w_{q^*} \mathbb{M}^{-1} \hat{\mathbf{e}}_{q^*} (\hat{\mathbf{e}}_{q^*}^\mathsf{T} \mathbb{M}^{-1} \hat{\mathbf{e}}_{q^*})^{-1}$;
6: goto 2;
7: **else**
8: end;
9: **end if**

6.4 Summary

Backpropagation is an efficient algorithm for stochastic gradient-descent on the energy function in weight space, because it refers only to quantities that are local to the weight to be updated. It is sometimes argued that biological neural networks are local in this way [2].

6.5 Further reading

The backpropagation algorithm is explained in Section 6.1. of Hertz, Krogh and Palmer [1], and in Chapter 4 of Haykin's book [2]. The paper [47] by LeCun *et al.* predates deep learning, but it is still a very nice collection of recipes for making backpropagation more efficient. Historical note: one of the first papers on error backpropagation is the one by Rumelhart *et al.* [54]. Have a look! The paper gives an excellent explanation and summary of the backpropagation algorithm. The authors also describe results of different numerical experiments, one of them introduces convolutional nets (Section 7.3) to learn to tell the difference between the letters T and C (Figure 6.13).

Figure 6.13: Patterns detected by the convolutional net of Ref. [54]. After Fig. 13 in Ref. [54].

6.6 Exercises

6.1 Covariance matrix. Show that the eigenvalues of the data covariance matrix $\mathbb{C}$ defined in Equation (6.22) are real and non-negative.

6.2 Principal-component analysis. Compute the data covariance matrix $\mathbb{C}$ for the example shown in Figure 6.14 and determine the principal direction. Determine the principal direction for the data shown in Figure 6.15.

6.3 Nesterov's accelerated-gradient method. The version (6.37) of Nesterov's algorithm is slightly different from the original formulation [48]. This point is discussed in [49]. Show that both versions are equivalent.

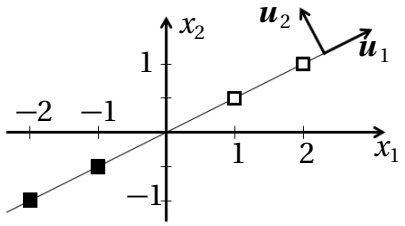


Figure 6.14: The principal direction of this data set is $\boldsymbol{u}_1$.

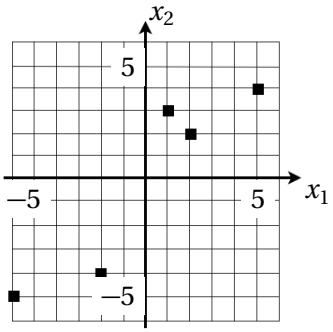


Figure 6.15: Calculate the principal component of this data set. Exercise 6.2.

6.4 Pruning. Show that the expression (6.46) for the weight increment $\delta \boldsymbol{w}^*$ minimises the Lagrangian (6.42) subject to the constraint (6.40).

6.5 Skipping layers. Show how the backpropagation algorithm can be generalised for feed-forward networks that allow for connections from the two nearest layers to the left, not only from the nearest layer to the left (Section 7.6).

6.6 Momentum. Section 6.3.4 describes how to speed up gradient descent by introducing momentum. To explain how this works it was assumed that the gradient is constant throughout, for all time steps $t = 0, \ldots, T$. A more realistic assumption is that the gradient approaches a constant from a certain time step $t_{\min} > 0$ onwards, not from $t = 0$. Rephrase the arguments in Section 6.3.4 assuming that the gradient approaches a constant and then remains constant for $t \geq t_{\min}$.

6.7 Backpropagation. Explain how to train a multi-layer perceptron by backpropagation. Draw a flow-chart of the algorithm. In your discussion, refer to and explain the following terms: *forward propagation, backward propagation, hidden layer, energy function, gradient descent, local energy minima, batch training, training set, validation set, classification error,* and *overfitting.* Your answer must not be longer than one A4 page.

6.8 Stochastic gradient descent. To train a multi-layer perceptron using stochastic gradient descent one needs update formulae for the weights and thresholds in the network. Derive these update formulae for *sequential training* using backpropagation for the network shown in Fig. 6.16. The weights for the first and second hidden layer, and for the output layer are denoted by $w_{jk}^{(1)}$, $w_{mj}^{(2)}$, and W_{1m}. The corresponding thresholds are denoted by $\theta_j^{(1)}$, $\theta_m^{(2)}$, and Θ_1, and the activation function by $g(\cdots)$. The target value for input pattern $\boldsymbol{x}^{(\mu)}$ is $t_1^{(\mu)}$, and the pattern index μ ranges from 1 to p. The energy function is $H = \frac{1}{2}\sum_{\mu=1}^{p}(t_1^{(\mu)} - O_1^{(\mu)})^2$.

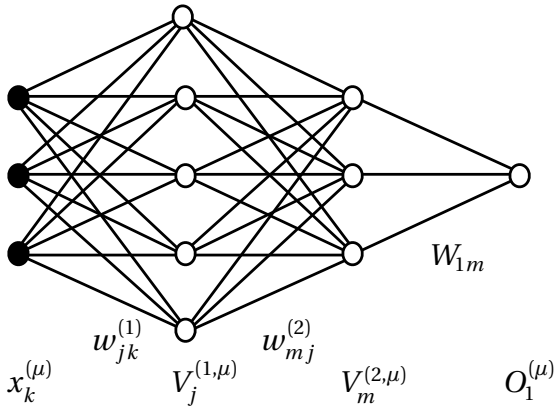


Figure 6.16: Multi-layer perceptron with three input terminals, two hidden layers, and one output unit. Exercise 6.8.

6.9 Multi-layer perceptron. A perceptron has hidden layers $\ell = 1,\ldots,L-1$ and output layer $l = L$. Neuron $V_j^{(\ell,\mu)}$ in layer ℓ computes $V_j^{(\ell,\mu)} = g(b_j^{(\ell,\mu)})$ with $b_j^{(\ell,\mu)} = -\theta_j^{(\ell)} + \sum_k w_{jk}^{(\ell)} V_k^{(\ell-1,\mu)}$, where $\boldsymbol{V}^{(0,\mu)} = \boldsymbol{x}^{(\mu)}$, $w_{jk}^{(\ell)}$ are weights, $\theta_j^{(\ell)}$ are thresholds, $g(b)$ is the activation function, and $V_i^{(L)} = O_i^{(\mu)} = g(b_i^{(L,\mu)})$. Draw this network. Indicate where the elements $x_k^{(\mu)}$, $b_j^{(\ell,\mu)}$, $V_j^{(\ell,\mu)}$, $O_i^{(\mu)}$, $w_{jk}^{(\ell)}$ and $\theta_j^{(\ell)}$ for $\ell = 1,\ldots,L$ belong. Determine how the derivatives $\partial V_i^{(\ell,\mu)}/\partial w_{mn}^{(p)}$ depend upon the derivatives $\partial V_j^{(\ell-1,\mu)}/\partial w_{mn}^{(p)}$ for $p < \ell$. Evaluate the derivative $\partial V_j^{(\ell,\mu)}/\partial w_{mn}^{(p)}$ for $p = \ell$. Using gradient descent on the energy function $H = \frac{1}{2}\sum_{i\mu}\left(t_i^{(\mu)} - O_i^{(\mu)}\right)^2$, find the update rule for the weight $w_{mn}^{(L-2)}$ with learning rate η.

Figure 7.1: Images of iris flowers. From left to right: iris setosa (copyright T. Monto), iris versicolor (copyright R. A. Nonemacher), and iris virginica (copyright A. Westermoreland). All images are copyrighted under the creative commons license.

7 Deep learning

7.1 How many hidden layers?

In Chapter 5 we saw why it is sometimes necessary to have a hidden layer: this make it possible to solve problems that are not linearly separable. Under which circumstances is one hidden layer sufficient? Are there problems that require more than one hidden layer? Even if not necessary, may additional hidden layers improve the performance of the network?

The second question is more difficult to answer than the first, so we start with the first question. To understand how many hidden layers are necessary it is useful to view the classification problem as an *approximation problem* [55]. Consider the classification problem $(\boldsymbol{x}^{(\mu)}, t^{(\mu)})$ for $\mu = 1, \ldots, p$. This problem defines a *target function* $t(\boldsymbol{x})$. Training a network to solve this task corresponds to approximating the target function $t(\boldsymbol{x})$ by the output function $O(\boldsymbol{x})$ of the network, from N-dimensional input space to one-dimensional output space.

How many hidden layers are necessary or sufficient to approximate a given set of

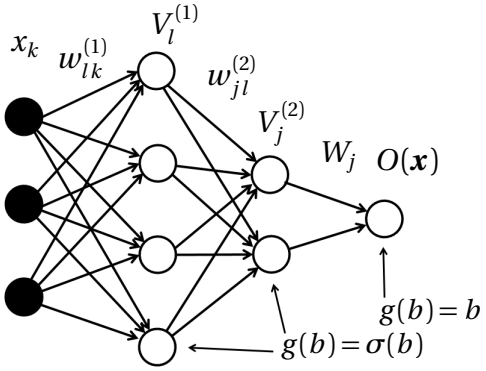


Figure 7.2: Multi-layer perceptron for function approximation.

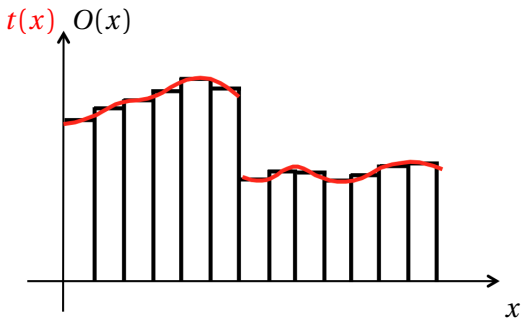


Figure 7.3: The neural-network output $O(x)$ approximates the target function $t(x)$.

functions to a certain accuracy, by choosing weights and thresholds? The answer depends on the nature of the set of functions. Are they real-valued or do they assume only discrete values? If the functions are real-valued, are they continuous or not?

We start with real-valued inputs and output. Consider the network drawn in Figure 7.2. The neurons in the hidden layers have sigmoid activation functions $\sigma(b) = (1+\mathrm{e}^{-b})^{-1}$. The output is continuous, with activation function $g(b) = b$. With two hidden layers the task is to approximate the function $t(\boldsymbol{x})$ by

$$O(\boldsymbol{x}) = \sum_j W_j g\Big(\sum_l w^{(2)}_{jl}\, g\Big(\sum_k w^{(1)}_{lk}\, x_k - \theta^{(1)}_l\Big) - \theta^{(2)}_j\Big) - \Theta\,. \tag{7.1}$$

In the simplest case the inputs are one-dimensional (Figure 7.3). The training set consists of pairs $(x^{(\mu)}, t^{(\mu)})$. The task is then to approximate the corresponding target function $t(x)$ by the network output $O(x)$:

$$O(x) \approx t(x)\,. \tag{7.2}$$

We approximate the real-valued function $t(x)$ by linear combinations of the basis functions $b(x)$ shown in Figure 7.4(**a**). Any reasonable real-valued function $t(x)$ can be approximated by a sums of such basis functions, each suitably shifted and scaled. Furthermore, these basis functions can be expressed as differences of activation

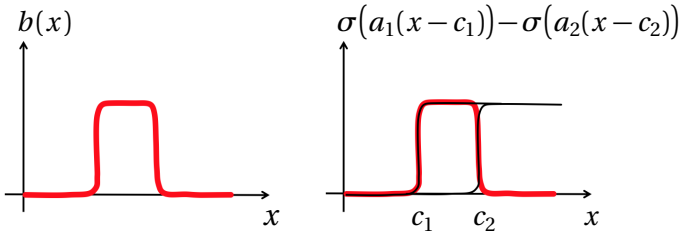


Figure 7.4: (**a**) basis function. (**b**) linear combination of two sigmoid functions for a large value of $a_1 = a_2$.

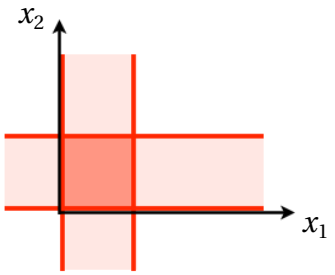


Figure 7.5: To make a localised basis function with two inputs out of neurons with sigmoid activation functions, one needs two hidden layers. One layer determines the lightly shaded cross in terms of a linear combination of four sigmoid outputs. The second layer localises the final output to the darker square [Equation (7.4)].

functions [Figure 7.4(**b**)]

$$b(x) = \sigma(w_1 x - \theta_1) - \sigma(w_2 x - \theta_2). \tag{7.3}$$

Comparing with Equation (7.1) shows that one hidden layer is sufficient to construct the function $O(x)$ in this way. Now consider two-dimensional inputs. In this case, suitable basis functions are (Figure 7.5):

$$\sigma\Big\{w^{(2)}\Big[\sigma(w_{11}^{(1)}x_1) - \sigma(w_{21}^{(1)}x_1 - \theta_2^{(1)}) + \sigma(w_{32}^{(1)}x_2) - \sigma(w_{42}^{(1)}x_2 - \theta_4^{(1)})\Big] - \theta^{(2)}\Big\}. \tag{7.4}$$

So for two input dimensions two hidden layers are sufficient. For each basis function we require four neurons in the first hidden layer and one neuron in the second hidden layer. In general, for N inputs, two hidden layers are sufficient, with $2N$ neurons in the first and one neuron in second layer, for each basis function.

Yet it is not always necessary to use two layers for real-valued functions. For continuous functions, one hidden layer is sufficient. This is ensured by the *universal approximation theorem* [2]. This theorem says any continuous function can be approximated to arbitrary accuracy by a network with a single hidden layer, for sufficiently many neurons in the hidden layer.

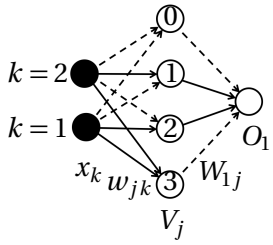
In Chapter 5 we considered discrete Boolean functions. It turns out that any Boolean function with N-dimensional inputs can be represented by a network with one hidden layer, using 2^N neurons in the hidden layer. An example for such a network is discussed in Ref. [1]:

$$\begin{array}{ll} x_k \in \{+1,-1\} & k = 1,\ldots,N \quad \text{inputs} \\ V_j & j = 0,\ldots,2^N-1 \quad \text{hidden neurons} \\ g(b) = \tanh(b) & \text{activation function of hidden neurons} \\ g(b) = \operatorname{sgn}(b) & \text{activation function of output unit} \end{array} \tag{7.5}$$

(**a**)

x_1	x_2	t
-1	-1	-1
+1	-1	+1
-1	+1	+1
+1	+1	-1

(**b**)



(**c**)

j	digit 1	digit 2
0	0	0
1	0	1
2	1	0
3	1	1

Figure 7.6: Boolean XOR function. (**a**) value table, (**b**) network layout. For the weights feeding into the hidden layer, dashed lines correspond to $w_{jk} = -\delta$, solid lines to $w_{jk} = \delta$. For the weights feeding into the output neuron, dashed lines correspond to $W_{1j} = -\gamma$, and solid lines to $W_{1j} = \gamma$ (**c**) construction principle for the weights of the hidden layer.

A difference compared with the Boolean networks in Section 5.4 is that here the inputs take the values ± 1. The reason is that this simplifies the proof, which is by construction [1]. For each hidden neuron one assigns the weights as follows

$$w_{jk} = \begin{cases} \delta & \text{if the } k^{\text{th}}\text{digit of binary representation of } j \text{ is } 1\,, \\ -\delta & \text{otherwise,} \end{cases} \tag{7.6}$$

with $\delta > 1$ (see below). The thresholds θ_j of all hidden neurons are the same, equal to $N(\delta - 1)$. The idea is that each input pattern turns on exactly one neuron in the hidden layer (called the *winning unit*). This requires that δ is large enough, as we shall see. The weights feeding into the output neuron are assigned as follows. If the output for the pattern represented by neuron V_j is +1, let $W_{1j} = \gamma > 0$, otherwise $W_{1j} = -\gamma$. The threshold is $\Theta = \sum_j W_{1j}$.

To show how this construction works, consider the Boolean XOR function as an example. First, for each pattern only the corresponding winning neuron gives a positive signal. For pattern $\boldsymbol{x}^{(1)} = [-1,-1]^{\mathsf{T}}$, for example, this is the first neuron in the hidden layer ($j = 0$). To see this, compute the local fields for this input pattern:

$$\begin{aligned} b_0^{(1)} &= 2\delta - 2(\delta - 1) = 2\,, \\ b_1^{(1)} &= -2(\delta - 1) = 2 - 2\delta\,, \\ b_2^{(1)} &= -2(\delta - 1) = 2 - 2\delta\,, \\ b_3^{(1)} &= -2\delta - 2(\delta - 1) = 2 - 4\delta\,. \end{aligned} \tag{7.7}$$

If we choose $\delta > 1$ then the output of the first hidden neuron gives a positive output ($V_0 > 0$), the other neurons produce negative outputs, $V_j < 0$ for $j = 1,2,3$. Now

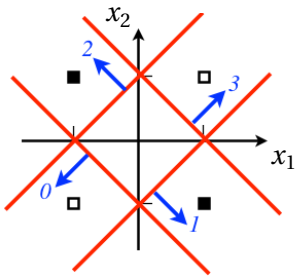


Figure 7.7: Shows how the XOR network depicted in Figure 7.6 partitions the input plane. Target values are encoded as in Figure 5.8: □ corresponds to $t=-1$ and ■ to $t=+1$).

consider $\boldsymbol{x}^{(3)}=[-1,+1]^{\mathsf{T}}$. In this case

$$\begin{aligned} b_0^{(3)} &= -2(\delta-1)=2-2\delta \\ b_1^{(3)} &= -2\delta-2(\delta-1)=2-4\delta \\ b_2^{(3)} &= 2\delta-2(\delta-1)=2 \\ b_3^{(3)} &= -2(\delta-1)=2-2\delta \end{aligned} \tag{7.8}$$

So in this case the third hidden neuron gives a positive output, while the others yield negative outputs. It works in the same way for the other two patterns, $\boldsymbol{x}^{(2)}$ and $\boldsymbol{x}^{(4)}$. This confirms that there is a unique winning neuron for each pattern. That pattern $\mu=k$ gives the winning neuron $j=k-1$ is of no importance, it is just a consequence of how the patterns are ordered in the value table in 7.6. How do the decision boundaries corresponding to V_j for $j=0,\ldots,3$ partition the input plane? This is shown in Figure 7.7.

Second, the output neuron computes

$$O_1=\mathrm{sgn}(-\gamma V_1+\gamma V_2+\gamma V_3-\gamma V_4) \tag{7.9}$$

with $\gamma>0$, and $\Theta=\sum_j W_{1j}=0$. For $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(4)}$ we find the correct result $O_1=-1$. The same is true for $\boldsymbol{x}^{(2)}$ and $\boldsymbol{x}^{(3)}$, we obtain $O_1=1$. In summary, this example illustrates how an N-dimensional Boolean function is represented by a network with one hidden layer, with 2^N neurons. The problem is of course that this network is expensive to train for large N because the number of hidden neurons is very large.

There are more efficient layouts if one uses more than one hidden layer. As an example, consider the *parity function* for N binary inputs equal to 0 or 1. The function measures the parity of the input sequence. It gives 1 if there is an odd number of ones in the input, otherwise 0. A construction similar to the above yields a network layout with 2^N neurons in the hidden layer. If one instead wires together the XOR networks shown in Figure 5.14, one can solve the parity problem with $O(N)$

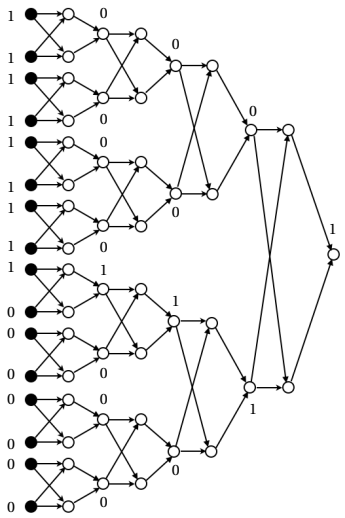


Figure 7.8: Solution of the parity problem for N-dimensional inputs. The network is built from XOR units (Figure 5.14). Each XOR unit has a hidden layer with two neurons. Above only the states of the inputs and outputs of the XOR units are shown, not those of the hidden neurons. In total, the whole network has $O(N)$ neurons.

neurons, as Figure 7.8 demonstrates. When N is a power of two then this network has $3(N-1)$ neurons. To see this, set the number of inputs to $N = 2^k$. Figure 7.8 shows that the number $\mathcal{N}_k$ of neurons satisfies the recursion $\mathcal{N}_{k+1} = 2\mathcal{N}_k + 3$ with $\mathcal{N}_1 = 3$. The solution of this recursion is $\mathcal{N}_k = 3(2^k - 1)$.

This example also illustrates a second reason why it may be useful to have more than one hidden layer. To design a network for a certain task it is often convenient to build the network from building blocks. One wires them together, often in a hierarchical fashion. In Figure 7.8 there is only one building block, the XOR network from Figure 5.14.

Another example are convolutional networks for image analysis (Section 7.3).

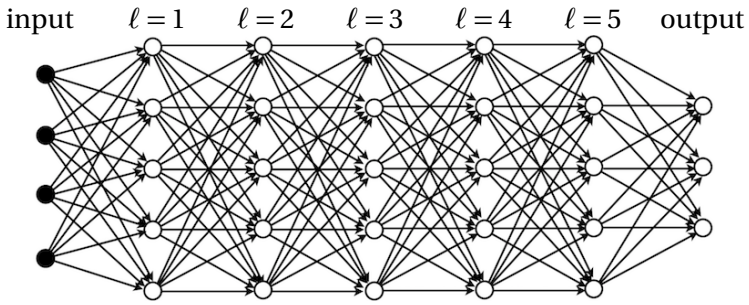


Figure 7.9: Fully connected deep network with five hidden layers. How deep is deep? Usually one says: deep networks have two or more hidden layers.

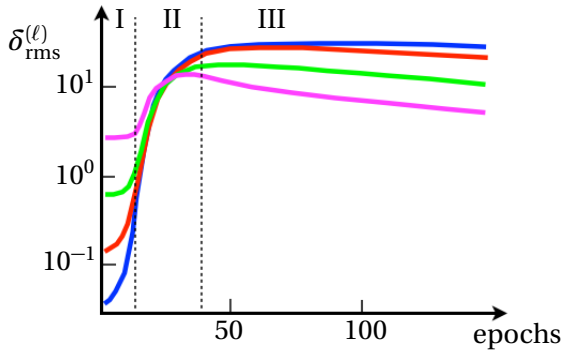


Figure 7.10: Vanishing-gradient problem for a network with four fully connected hidden layers. The Figure illustrates schematically how the r.m.s. error $\delta^{(\ell)}_{\rm rms} = \sqrt{\langle N_\ell^{-1} \sum_{j=1}^{N_\ell} [\delta_j^{(\ell)}]^2 \rangle}$ in layer ℓ depends on the number of training epochs, for $\ell = 1$ (blue), $\ell = 2$ (red), $\ell = 3$ (green), and $\ell = 4$ (magenta). During phase I the vanishing-gradient problem is severe, during phase II the network starts to learn, phase III is the convergence phase where the errors decline.

Here the fundamental building blocks are *feature maps*, they recognise different geometrical features in the image, such as edges or corners.

7.2 Training deep networks

It was believed for a long time that networks with many hidden layers (*deep* networks) are so difficult to train that it is not practical to use many hidden layers. But the past few years have witnessed a paradigm shift regarding this question. It has been demonstrated that fully connected deep networks can in fact be trained efficiently with backpropagation (Section 6.2), and that these networks can solve complex classification tasks with very small error rates.

Browsing through the recent literature of the subject one may get the impression that training deep networks such as the one shown in Figure 7.9 is more an art than a science, some read like manuals, or like collections of engineering recipes. But there are several fundamental facts about how deep networks learn. I summarise them in this Section. A comprehensive summary of these (and more) principles is given in the book *Deep learning* [4].

7.2.1 Vanishing gradients

In Chapter 6 we discussed that the learning slows down when the gradients, the factors of $g'(b)$ contained in the errors, become small. When the network has many

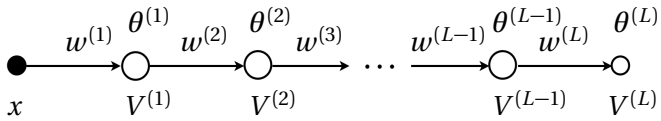


Figure 7.11: 'Network' illustrating the vanishing-gradient problem, with neurons $V^{(\ell)}$, weights $w^{(\ell)}$, and thresholds $\theta^{(\ell)}$.

hidden layers, this problem is more severe. One finds that the neurons in hidden layers close to the input layer (small values of ℓ in Figure 7.9) change only by small amounts, the smaller the more hidden layers the network has.

Figure 7.10 demonstrates that the r.m.s. errors averaged over different realisations of random initial weights, $\left(\langle N_\ell^{-1}\sum_{j=1}^{N_\ell}[\delta_j^{(\ell)}]^2\rangle\right)^{1/2}$, tend to be very small for the first 20 training epochs. In this regime the gradient (and thus the speed of training) vanishes exponentially as $\ell \to 1$. This *slowing down* is the result of the diminished effect of the neurons in layer ℓ upon the output, when ℓ is small. This is the *vanishing-gradient problem*.

To explain this phenomenon, consider the very simple case shown in Figure 7.11: a deep network with only one neuron per layer. Algorithm 2 shows that the errors are given recursively by $\delta^{(\ell)} = \delta^{(\ell+1)} w^{(\ell+1)} g'(b^{(\ell)})$. Using $\delta^{(L)} = [t - V^{(L)}(x)]g'(b^{(L)})$ we have

$$\delta^{(\ell)} = [t - V^{(L)}(x)]g'(b^{(L)}) \prod_{k=L}^{\ell+1} [w^{(k)} g'(b^{(k-1)})]. \tag{7.10}$$

Consider first the early stages of training. If one initialises the weights as described in Chapter 6 to Gaussian random variables with mean zero and variance σ_w^2, and the thresholds to zero, then the factors $w^{(k)}g'(b^{(k-1)})$ are usually smaller than unity (for the activation functions (6.15), the maximum of $g'(b)$ is $\frac{1}{2}$ and 1, respectively). The product of these factors vanishes quickly as ℓ decreases. So the slowing down is a consequence of multiplying many small numbers to get something really small (*vanishing-gradient problem*). This is phase I in Figure 7.10.

What happens at later times? One might argue that the weights may grow during training, as a function of ℓ. If that happened, the problem might become worse still, because $g'(b)$ tends to zero exponentially as $|b|$ grows. This indicates that the first layers may continue to learn slowly. Figure 7.10 shows that the effect persists for about 20 epochs. But then even the first layers begin to learn faster (phase II). This does not contradict the above discussion, because it assumed random weights. As the network learns, the weights are no longer independent random numbers. But there is to date no mathematical theory describing how this transition occurs. Much later in training, the errors decay during the convergence phase (phase III in

Figure 7.10).

In summary, Equation (7.10) demonstrates that different layers of the network learn at different speeds initially, when the weights are still random. There is a second, equivalent, point of view: the learning is slow in a layer far from the output because the output is not very sensitive to the state of these neurons. To measure the effect of a given neuron on the output, we calculate how the output of the network changes when changing the *state* of a neuron in a particular layer. For the example shown in Figure 7.11, the output $V^{(L)}$ is given by nested activation functions

$$V^{(L)}=g\Big(w^{(L)}g\Big(w^{(L-1)}\cdots g\big(w^{(2)}g(w^{(1)}x-\theta^{(1)})-\theta^{(2)}\big)\ldots-\theta^{(L-1)}\Big)-\theta^{(L)}\Big). \tag{7.11}$$

The effects of the neurons in Figure 7.11 are computed using the chain rule:

$$\begin{aligned}\frac{\partial V^{(L)}}{\partial V^{(L-1)}} &= g'(b^{(L)})w^{(L)}\\ \frac{\partial V^{(L)}}{\partial V^{(L-2)}} &= \frac{\partial V^{(L)}}{\partial V^{(L-1)}}\frac{\partial V^{(L-1)}}{\partial V^{(L-2)}}=g'(b^{(L)})w^{(L)}g'(b^{(L-1)})w^{(L-1)}\\ &\vdots\end{aligned} \tag{7.12}$$

where $b^{(k)}=w^{(k)}V^{(k-1)}-\theta^{(k)}$ is the local field for neuron k. This yields the following expression for $\partial V^{(L)}/\partial V^{(\ell)}$:

$$\frac{\partial V^{(L)}}{\partial V^{(\ell)}}=\prod_{k=L}^{\ell+1}[g'(b^{(k)})w^{(k)}]. \tag{7.13}$$

Again it is a product of $g'(b)$ that determines how the effect of $V^{(\ell)}$ on the output decreases as ℓ decreases. Equations (7.13) and (7.10) are in fact equivalent, since $\delta^{(\ell)}=[t-V^{(L)}]\partial V^{(L)}/\partial(-\theta^{(\ell)})=[t-V^{(L)}]\partial V^{(L)}/\partial V^{(\ell)}g'(b^{(\ell)})$.

More generally, note that the product in Equation (7.13) is unlikely to remain of order unity when L is large. To see this, assume that the weights are independently distributed random numbers. Taking the logarithm and using the *central-limit theorem* shows that the distribution of the product is log normal. This means that the learning speed can be substantially different in different layers. This is also referred to the problem of *unstable gradients*. The example shown in Figure 7.11 illustrates the origin of this problem: it is due to the fact that multiplying many small numbers together produces a result that is very small. Multiplying many numbers that are larger than unity, by contrast, yields a large result.

In networks like the one shown in Figure 7.9 the principle is the same, but instead of multiplying numbers one multiplies matrices. The product (7.13) of random numbers becomes of product of random matrices. Assume that all layers $\ell=1,\ldots,L$

have N neurons. We denote the matrix with elements $J_{ij}^{(\ell,L)} = \partial V_i^{(L)}/\partial V_j^{(\ell)}$ by $\mathbb{J}_{\ell,L}$. Using the chain rule we find:

$$\frac{\partial V_i^{(L)}}{\partial V_j^{(\ell)}} = \sum_{lm\cdots n} \frac{\partial V_i^{(L)}}{\partial V_l^{(L-1)}} \frac{\partial V_l^{(L-1)}}{\partial V_m^{(L-2)}} \cdots \frac{\partial V_n^{(\ell+1)}}{\partial V_j^{(\ell)}}. \tag{7.14}$$

Using the update rule

$$V_i^{(k)} = g\Big(\sum_j w_{ij}^{(k)} V_j^{(k-1)} - \theta_i^{(k)}\Big) \tag{7.15}$$

we can evaluate each factor:

$$\frac{\partial V_p^{(k)}}{\partial V_l^{(k-1)}} = g'(b_p^{(k)}) w_{pl}^{(k)}. \tag{7.16}$$

In summary, this yields the following expression for $\mathbb{J}_{\ell,L}$:

$$\mathbb{J}_{\ell,L} = \mathbb{D}^{(L)}\mathbb{W}^{(L)}\mathbb{D}^{(L-1)}\mathbb{W}^{(L-1)}\cdots\mathbb{D}^{(\ell+1)}\mathbb{W}^{(\ell+1)}, \tag{7.17}$$

where $\mathbb{W}^{(k)}$ is the matrix of weights feeding into layer k, and

$$\mathbb{D}^{(k)} = \begin{bmatrix} g'(b_1^{(k)}) & & \\ & \ddots & \\ & & g'(b_N^{(k)}) \end{bmatrix}. \tag{7.18}$$

This expression is analogous to Equation (7.13).

The eigenvalues of the matrix $\mathbb{J}_{0,k}$ describe how small changes $\delta \boldsymbol{V}^{(0)}$ to the inputs $\boldsymbol{V}^{(0)}$ (or small differences between the inputs) grow as they propagate through the layers. If the maximal eigenvalue is larger than unity, then $|\delta \boldsymbol{V}^{(0)}|$ grows exponentially as a function of layer index k. This is quantified by the maximal *Lyapunov exponent* [56]

$$\lambda = \lim_{k\to\infty} \frac{1}{2k} \Big\langle \log \mathrm{tr}\big(\mathbb{J}_{0,k}^{\mathsf{T}} \mathbb{J}_{0,k}\big) \Big\rangle \tag{7.19}$$

where the average is over realisations of weights and thresholds. The matrix $\mathbb{J}_{0,k}^{\mathsf{T}}\mathbb{J}_{0,k}$ is called the *right Cauchy-Green* matrix, and tr denotes the *trace* of this matrix, the sum of its diagonal elements. The right Cauchy-Green matrix is symmetric, and it is positive definite. The eigenvectors of $\mathbb{J}_{0,k}^{\mathsf{T}}\mathbb{J}_{0,k}$ are called *forward Lyapunov vectors*. They describe how small corrections to the inputs rotate, shrink, or stretch as they propagate through the network.

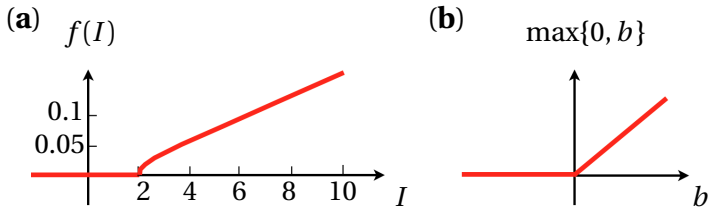


Figure 7.12: (**a**) Firing rate of a *leaky integrate-and-fire* neuron as a function of the electrical current I through the cell membrane, Equation (7.21) for $\tau = 25$ and $U_c/R = 2$ (see text). (**b**) Rectified linear unit, $g(b) = \max\{0, b\}$.

If we multiply the matrix $\mathbb{J}_{k,L}$ from the left with the transpose of the vector $\boldsymbol{\delta}^{(L)}$ of output errors, we see how the errors change as they propagate backwards from layer k to the leftmost hidden layer, how this vector rotates, shrinks, or stretches.

There are a number of different tricks that help to suppress vanishing gradients, to some extent at least. First, it is usually argued that it helps to use an activation function that does not saturate at large b, such as the ReLU function introduced in Section 7.2.2. But the results of Ref. [57] show that the effect is perhaps not as strong as originally thought. Second, batch normalisation (Section 7.2.6) may help against the unstable gradient problem. Third, introducing connections that skip layers (*residual network*) can also reduce the unstable-gradient problem. This is discussed in Section 7.6.

7.2.2 Rectified linear units

Glorot *et al.* [58] suggested to use a different activation function. Their choice is motivated by the response curve of *leaky integrate-and-fire* neurons. This is a model for the relation between the electrical current I through the cell membrane into the neuron cell, and the membrane potential U. The simplest models for the dynamics of the membrane potential represent the neuron as a capacitor. In the leaky integrate-and-fire neuron, leakage is added by a resistor R in parallel with the capacitor C, so that

$$I = \frac{U}{R} + C\frac{\mathrm{d}U}{\mathrm{d}t}\,. \tag{7.20}$$

For a constant current, the membrane potential grows from zero as a function of time, $U(t) = RI[1-\exp(-t/\tau)]$, where $\tau = RC$ is the time constant of the model. One says that the neuron produces a *spike* when the membrane potential exceeds a critical value, U_c. Immediately after, the membrane potential is set to zero (and begins to grow again). In this model, the *firing rate* $f(I)$ is thus given by t_c^{-1}, where t_c is the solution of $U(t) = U_c$. It follows that the firing rate exhibits a threshold

behaviour (the system works like a rectifier):

$$f(I)=\begin{cases} 0 & \text{for} \quad I \le U_c/R\,, \\ \left[\tau \log\left(\frac{RI}{RI-U_c}\right)\right]^{-1} & \text{for} \quad I > U_c/R\,. \end{cases} \tag{7.21}$$

This response curve is illustrated in Figure 7.12 (**a**). The main message is that there is a threshold below which the response is strictly zero (this is not the case for the activation function shown in Figure 1.6). The response function looks qualitatively like the *ReLU* function

$$g(b)=\mathrm{ReLU}(b)\equiv \max\{0,b\}\,, \tag{7.22}$$

shown in panel (**b**). Neurons with this activation function are called *rectified linear units*. The derivative of the ReLU function is discontinuous at $b=0$. A common convention is to set the derivative to zero at $b=0$.

What is the point of using rectified linear units? When training a deep network with ReLU functions it turns out that many of the hidden neurons (as many as 50%) produce outputs strictly equal to zero. This means that the network of active neurons (non-zero output) is *sparsely* connected. It is thought that sparse networks have desirable properties, and sparse representations of a classification problem are more likely to be linearly separable (as shown in Section 10.1). Figure 7.13 illustrates that for a given input pattern only a certain fraction of hidden neurons is active. For these neurons the computation is linear, yet different input patterns give different sets of active neurons. The product in Equation (7.17) acquires a particularly simple structure: the matrices $\mathbb{D}^{(k)}$ are diagonal with 0/1 entries. But while the weight matrices are independent, the $\mathbb{D}^{(k)}$-matrices are correlated: which elements vanish depends on the states of the neurons in the corresponding layer, which in turn depend on the weights to the right of $\mathbb{D}^{(k)}$ in the matrix product.

A hidden layer with only one or very few active neurons might act as a *bottleneck* preventing efficient backpropagation of output errors which could in principle slow down training. For the examples given in Ref. [58] this does not occur.

The ReLU function is unbounded for large positive local fields. Therefore, the vanishing-gradient problem (Section 7.2.1) is thought to be less severe in networks made of rectified linear units, but see Ref. [57]. Since the ReLU function does not saturate, the weights tend to increase. Glorot *et al.* [58] suggested to use L_1-weight decay (Section 6.3.3) to make sure that the weights do not grow.

Finally, using ReLU functions instead of sigmoid functions speeds up the training, because the ReLU function has piecewise constant derivatives. Such function calls are faster to evaluate than sigmoid functions, for example.

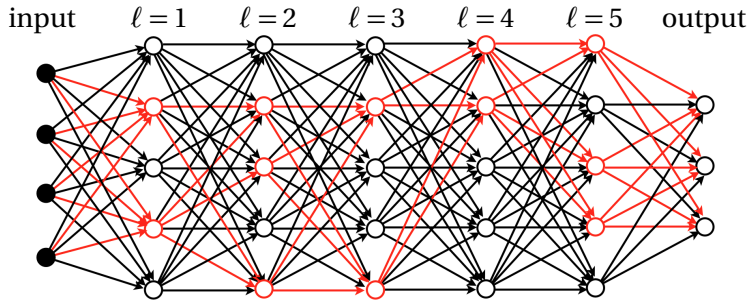


Figure 7.13: Sparse network of active neurons with ReLU activation functions. The red paths correspond to *active* neurons with positive local fields.

7.2.3 Outputs and energy functions

Up to now we discussed networks that have the same activation functions for all neurons in all layers, either sigmoid or tanh activation functions (Equation 6.15), or ReLU functions (Section 7.2.2). These networks are trained by stochastic gradient descent on the quadratic energy function (5.23). It has been shown that it may be advantageous to employ a different energy function, and to use slightly different activation functions for the neurons in the output layer, so-called *softmax* outputs, defined as

$$O_i = \frac{e^{\alpha b_i^{(L)}}}{\sum_{k=1}^{M} e^{\alpha b_k^{(L)}}} . \tag{7.23}$$

Here $b_i^{(L)} = \sum_j w_{ij}^{(L)} V_j^{(L-1)} - \theta_i^{(L)}$ are the local fields in the output layer. Usually the constant α is taken to be unity. In the limit $\alpha \to \infty$, you see that $O_i = \delta_{i i_0}$ where i_0 is the index of the *winning* output unit, the one with the largest value $b_i^{(L)}$ (Chapter 9). Usually one takes $\alpha = 1$, then Equation (7.23) is a *soft* version of this maximum criterion, thus the name *softmax*. Three important properties of softmax outputs are, first, that $0 \leq O_i \leq 1$. Second, the values of the outputs sum to one

$$\sum_{i=1}^{M} O_i = 1 . \tag{7.24}$$

This means that the outputs of softmax units can be interpreted as probabilities. Third, when $b_i^{(L)}$ increases then O_i increases but the values O_k of the other output neurons $k \neq i$ decrease.

Softmax output units can simplify interpreting the network output for classification problems where the inputs must be assigned to one of M classes. In this problem, the output $O_i^{(\mu)}$ of softmax unit i represents the probability that the input $\boldsymbol{x}^{(\mu)}$ is in class i (in terms of the targets: $t_i^{(\mu)} = 1$ while $t_k^{(\mu)} = 0$ for $k \neq i$). Softmax units are often used in conjunction with a different energy function. It is defined in

terms of *negative log likelihoods*

$$H = -\sum_{i\mu} t_i^{(\mu)} \log O_i^{(\mu)}. \tag{7.25}$$

Here and in the following log stands for the natural logarithm. The function (7.25) is minimal when $O_i^{(\mu)} = t_i^{(\mu)}$. Since the function (7.25) is different from the energy function used in Chapter 6, the details of the backpropagation algorithm are slightly different. To find the correct formula for backpropagation, we need to evaluate

$$\frac{\partial H}{\partial w_{mn}} = -\sum_{i\mu} \frac{t_i^{(\mu)}}{O_i^{(\mu)}} \frac{\partial O_i^{(\mu)}}{\partial w_{mn}}. \tag{7.26}$$

Here I did not write out the labels L that denote the output layer, and in the following equations I also drop the index μ that refers to the input pattern. Using the identities

$$\frac{\partial O_i}{\partial b_l} = O_i(\delta_{il} - O_l) \quad \text{and} \quad \frac{\partial b_l}{\partial w_{mn}} = \delta_{lm} V_n, \tag{7.27}$$

one obtains

$$\frac{\partial O_i}{\partial w_{mn}} = \sum_l \frac{\partial O_i}{\partial b_l} \frac{\partial b_l}{\partial w_{mn}} = O_i(\delta_{im} - O_m) V_n. \tag{7.28}$$

So

$$\delta w_{mn} = -\eta \frac{\partial H}{\partial w_{mn}} = \eta \sum_{i\mu} t_i^{(\mu)} (\delta_{im} - O_m^{(\mu)}) V_n^{(\mu)} = \eta \sum_{\mu} (t_m^{(\mu)} - O_m^{(\mu)}) V_n^{(\mu)}, \tag{7.29}$$

since $\sum_{i=1}^{M} t_i^{(\mu)} = 1$ for the type of classification problem where each input belongs to precisely one class. The corresponding expression for the threshold updates reads

$$\delta \theta_m = -\eta \frac{\partial H}{\partial \theta_m} = -\eta \sum_{\mu} (t_m^{(\mu)} - O_m^{(\mu)}). \tag{7.30}$$

Equations (7.29) and (7.30) highlight a further advantage of softmax output neurons (apart from the fact that they allow the output to be interpreted in terms of probabilities). The weight and threshold increments for the output layer derived in Section 6 [Equations (6.7) and (6.12a)] contain factors of derivatives $g'(B_m^{(\mu)})$. As noted earlier, these derivatives tend to zero when the activation function saturates, slowing down the learning. But Equations (7.29) and (7.30) do not contain such factors! Here the rate at which the neuron learns is simply proportional to the error, $(t_m^{(\mu)} - O_m^{(\mu)})$, no small factors reduce this rate.

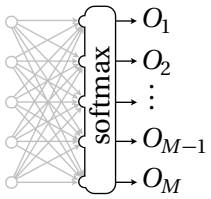


Figure 7.14: The symbol for a softmax layer indicates that the neurons in this layer are not independent.

Softmax units are normally only used in the output layer. First, the derivation shows that the learning speedup mentioned above is coupled to the use of the log likelihood function (7.25). Second, one usually tries to avoid dependence between the neurons in a given hidden layer, but Equation (7.23) shows that the output of neuron i depends on all local fields in the hidden layer (Figure 7.14). A better alternative is usually the ReLU activation function discussed in Section 7.2.2.

There is an alternative way of choosing the energy function that is very similar to the above, but works with sigmoid units:

$$H = -\sum_{i\mu} t_i^{(\mu)} \log O_i^{(\mu)} + (1 - t_i^{(\mu)}) \log(1 - O_i^{(\mu)}), \tag{7.31}$$

with $O_i = \sigma(b_i)$ where σ is the sigmoid function (6.15a). The function (7.31) is called *cross-entropy* function. To compute the weight increments, we apply the chain rule:

$$\frac{\partial H}{\partial w_{mn}} = \sum_{i\mu} \left(\frac{t_i^{(\mu)}}{O_i^{(\mu)}} - \frac{1 - t_i^{(\mu)}}{1 - O_i^{(\mu)}} \right) \frac{\partial O_l}{\partial w_{mn}} = \sum_{i\mu} \frac{t_i^{(\mu)} - O_i^{(\mu)}}{O_i^{(\mu)}(1 - O_i^{(\mu)})} \frac{\partial O_l}{\partial w_{mn}}. \tag{7.32}$$

Using Equation (6.16) we obtain

$$\delta w_{mn} = \eta \sum_{\mu} (t_m^{(\mu)} - O_m^{(\mu)}) V_n^{(\mu)}, \tag{7.33}$$

identical to Equation (7.29). The threshold increments are also updated in the same way, Equation (7.30). Yet the interpretation of the outputs is slightly different, since the values of the softmax units in the output layers sum to unity, while those of the sigmoid units do not. In either case you can use the definition (6.27) for the classification error.

7.2.4 Weight initialisation

The results of Section 7.2.1 point to the importance of initialising the weights in the right way, to avoid that the learning slows down. This is significant because it is

often found that the initial transient learning phase poses a substantial bottleneck to learning [59]. For this initial transient, correct weight initialisation can give a substantial improvement.

Moreover, when training deep networks with sigmoid activation functions in the hidden layers, it was observed that the values of the output neurons remain very close to zero for many training iterations (Figure 2 in Ref. [60]), slowing down the training substantially. It is argued that this is a consequence of the way the weights are initialised, in combination with the particular shape of the sigmoid activation function. It is sometimes argued that tanh activation functions work better than sigmoids, although the reasons are not quite clear.

So, how should the weights be initialised? The standard choice is to initialise the weights to independent Gaussian random numbers with mean zero and unit variance and the thresholds to zero (Section 6.1). But in networks that have large hidden layers with many neurons, this scheme may fail. Consider a neuron i in the first hidden layer with N incoming connections. Its local field

$$b_i = \sum_{j=1}^{N} w_{ij} x_j \tag{7.34}$$

is a sum of many independently identically distributed random numbers. Assume that the input patterns have independent random bits, equal to 0 or 1 with probability $\frac{1}{2}$. From the central-limit theorem we find that the local field is Gaussian distributed in the limit of large N, with mean zero and variance

$$\sigma_b^2 = \sigma_w^2 N/2\,. \tag{7.35}$$

This means that the local field is typically quite large, of order $\sqrt{N}$, and this implies that the units of the first hidden layer saturate – slowing down the learning. This conclusion rests on our particular assumption concerning the input patterns, but it is in general much better to initialise the weights uniformly or Gaussian with mean zero and with variance

$$\sigma_w^2 \propto N^{-1}\,, \tag{7.36}$$

to cancel the factor of N in Equation (7.35). The thresholds can be initialised to zero, as described in Section 6.1.

The normalisation (7.36) makes sure that the weights are not too large initially, but it does not circumvent the vanishing-gradient problem discussed in Section 7.2.1. There the problem was illustrated for $N = 1$, so unit variance for the initial weight distribution corresponds to Equation (7.36).

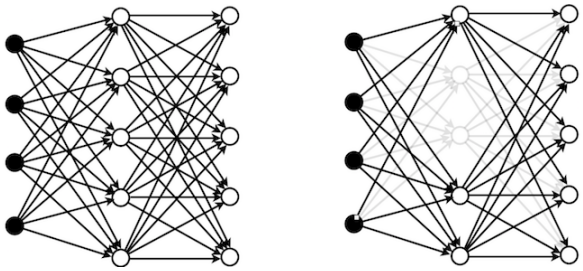


Figure 7.15: Illustrates regularisation by drop out.

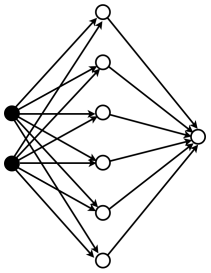
7.2.5 Regularisation

Deeper networks have more neurons, so the problem of overfitting (Figure 6.9) tends to be more severe for deeper networks. Therefore *regularisation* schemes that limit the tendency to overfit are more important for deeper networks. In Section 6.3.3 several regularisation schemes were described, for example L_1- and L_2-regularisation. In training deep networks, a number of other regularisation schemes have proved useful: *drop out*, pruning (Section 6.3.5), and expanding the training set.

Drop out

In this scheme some neurons are ignored during training [61]. Usually this regularisation technique is applied to hidden neurons. The procedure is illustrated in Figure 7.15. In each step of the training algorithm (for each mini batch, or for each individual pattern) one ignores at random a fraction p of neurons from each hidden layer, and updates the weights in the remaining, diluted network in the usual fashion. The weights coming into the dropped neurons are not updated, and as a consequence neither are their outputs. For the next step in the training algorithm, the removed neurons at put back, and another set of hidden neurons is removed. Once the training is completed, all hidden neurons are activated, but their outputs are multiplied by p.

Srivastava *et al.* [61] motivate this method by remarking that the performance of machine-learning algorithms is usually improved by combining the results of several learning attempts. In our case this corresponds to separately training several networks with different layouts on different inputs, and then to average over their outputs. However, for deep networks this is computationally very expensive. Drop out is an attempt to achieve the same goal more efficiently. The idea is that dropout corresponds to effectively training a large number of different networks. If there are k hidden neurons, then there are 2^k different combinations of neurons that are turned on or off. The hope is that the network learns more robust features of the input data in this way, and that this reduces overfitting. In practice the



n	10	8	6	4	2
Training success					
without pruning	98.5	96.8	92.5	78.3	49.1
pruned network	-	-	-	97.9	83.3

Figure 7.16: Boolean XOR problem. The network has one hidden layer with n ReLU neurons. The output neuron has a sigmoid activation function. The network is trained with stochastic gradient descent for 10 000 iterations. The initial weights were Gaussian random numbers with mean zero, standard deviation 0.1, and max-norm regularisation $|w_{ij}| < 2$. The thresholds were initially zero. Training success was measured in an ensemble of 1000 independent training realisations. Data from Ref. [62].

method is usually applied together with another regularisation scheme, *max-norm regularisation*. This means that weights are not allowed to grow larger than a given constant: $|w_{ij}| \leq c$.

Pruning

Pruning (Section 6.3.5) is also a regularisation method: by removing unnecessary weights one reduces the risk of overfitting. As opposed to drop out, where hidden neurons are only temporarily ignored, pruning refers to permanently removing hidden neurons. The idea is to train a large network, and then to prune a large fraction of neurons to obtain a much smaller network. It is usually found that such pruned nets generalise much better than small nets that were trained without pruning. Up to 90% of the hidden neurons can be removed. This method is applied with success to deep networks, but here I want to discuss a simple example: Frankle & Carbin [62] used the Boolean XOR function to illustrate the effectiveness of pruning.

Figure 5.14 shows that the XOR function can be represented by a hidden layer with two neurons. Suitable weights and thresholds are given in this Figure. Frankle & Carbin [62] point out that backpropagation takes a long time to find a valid solution, for random initial weights. They observe that a network with many more neurons in the hidden layer usually learns better. Figure 7.16 lists the fraction of successful trainings for networks with different numbers of neurons in the hidden layer. With two hidden neurons, only 49.1% of the networks learned the task in 10 000 training steps of stochastic gradient descent. Networks with more neurons in the hidden layer ensure better training success. The Figure also shows the training success of pruned networks, that were initially trained with $n = 10$ neurons. Then networks

were pruned iteratively during training, removing the neurons with the largest average magnitude. After training, the weights and threshold were reset to their initial values, the values before training began. One can draw three conclusions from this data (from Ref. [62]). First, iterative pruning during training singles out neurons in the hidden layer that had initial weights and thresholds resulting in the correct decision boundaries. Second, the pruned network with two hidden neurons has much better training success than the network that was trained with only two hidden neurons. Third, despite pruning more than 50% of the hidden neurons, the network with $n = 4$ hidden neurons performs almost as well as the one with $n = 10$ hidden neurons. When training deep networks it is common to start with many neurons in the hidden layers, and to prune up to 90% of them. This results in small trained networks that can efficiently and reliably classify.

Expanding the training set

If one trains a network with a fixed number of hidden neurons on larger training sets, one observes that the network generalises with higher accuracy (better classification success). The reason is that overfitting is reduced when the training set is larger. Thus, a way of avoiding overfitting is to *expand* or *augment* the training set. It is sometimes argued that the recent success of deep neural networks in image recognition and object recognition is in large part due to larger training sets. One example is ImageNet, a database of more than 10^7 hand-classified images, into more than 20 000 categories [63]. Naturally it is expensive to improve training sets in this way. Instead, one can expand a training set *artificially*. For digit recognition (Figure 2.1), for example, one can expand the training set by randomly shifting, rotating, and shearing the digits.

7.2.6 Batch normalisation

Batch normalisation [64] can significantly speed up the training of deep networks with backpropagation. The idea is to shift and normalise the input data for each hidden layer, not only the input patterns (Section 6.3.1). This is done separately for each mini batch (Section 6.2), and for each component of the inputs $V_j^{(\mu)}$, $j = 1, \ldots$ (Algorithm 4). One calculates the average and variance over each mini batch

$$\overline{V}_j = \frac{1}{m_B} \sum_{\mu=1}^{m_B} V_j^{(\mu)} \quad \text{and} \quad \sigma_B^2 = \frac{1}{m_B} \sum_{\mu=1}^{m_B} (V_j^{(\mu)} - \overline{V}_j)^2 , \tag{7.37}$$

subtracts the mean from the $V_j^{(\mu)}$, and divides by $\sqrt{\sigma_B^2 + \epsilon}$. The parameter $\epsilon > 0$ is added to the denominator to avoid division by zero. There are two additional

parameters in Algorithm 4, namely γ_j and β_j. If one were to set $\beta_j = \overline{V}_j$ and $\gamma_j = \sqrt{\sigma_B^2 + \epsilon}$ (algorithm 4), then batch normalisation would leave the $V_j^{(\mu)}$ unchanged. But instead these two parameters are learnt by backpropagation, just like the weights and thresholds. In general the new parameters are allowed to differ from layer to layer, $\gamma_j^{(\ell)}$ and $\beta_j^{(\ell)}$.

Batch normalisation was originally motivated by arguing that it reduces possible covariate shifts faced by hidden neurons in layer ℓ: as the parameters of the neurons in the preceding layer $\ell - 1$ change, their outputs shift thus forcing the neurons in layer ℓ to adapt. However in Ref. [65] it was argued that batch normalisation does not reduce the internal covariate shift. It speeds up the training by effectively smoothing the energy landscape.

Batch normalisation helps to combat the *vanishing-gradient problem* because it prevents local fields of hidden neurons to grow. This makes it possible to use sigmoid functions in deep networks, because the distribution of inputs remains normalised. It is sometimes argued that batch normalisation has a regularising effect, and it has been suggested [64] that batch normalisation can replace drop out (Section 7.2.5). It is also argued that batch normalisation can help the network to generalise better, in particular if each mini batch contains randomly picked inputs. Then batch normalisation corresponds to randomly transforming the inputs to each hidden neuron (by the randomly changing means and variances). This may help to make the learning more robust. There is no theory that proves either of these claims, but it is an empirical fact that batch normalisation often speeds up the training.

Algorithm 4 batch normalisation

1: **for** $j = 1, \ldots$ **do**
2: calculate mean $\overline{V}_j \leftarrow \frac{1}{m_B} \sum_{\mu=1}^{m_B} V_j^{(\mu)}$
3: calculate variance $\sigma_B^2 \leftarrow \frac{1}{m_B} \sum_{\mu=1}^{m_B} (V_j^{(\mu)} - \overline{V}_j)^2$
4: normalise $\hat{V}_j^{(\mu)} \leftarrow (V_j^{(\mu)} - \overline{V}_j)/\sqrt{\sigma_B^2 + \epsilon}$
5: calculate outputs as: $g(\gamma_j \hat{V}_j^{(\mu)} + \beta_j)$
6: **end for**
7: end;

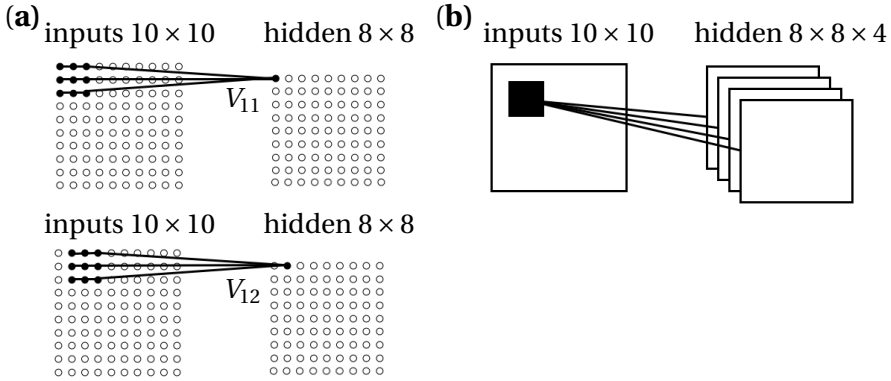


Figure 7.17: (**a**) Feature map, kernel, and receptive field (schematic). A feature map (the 8×8 array of hidden neurons) is obtained by translating a kernel (filter), here with a 3×3 receptive field, over the input image, here a 10×10 array of pixels. (**b**) A convolution layer consists of a number of feature maps, each corresponding to a given kernel that detects a certain feature in parts of the input image.

7.3 Convolutional networks

Convolutional networks have been around since the 1980's. They became widely used after Krizhevsky *et al.* [66] won the ImageNet challenge (Section 7.5) with a convolutional net. One reason for the recent success of convolutional networks is that they have fewer neurons. This has two advantages. Firstly, such networks are obviously cheaper to train. Secondly, as pointed out above, reducing the number of neurons regularises the network, it reduces the risk of overfitting.

Convolutional neural networks are designed for object recognition and pattern detection. They take images as inputs (Figure 7.1), not just a list of attributes (Figure 5.1). Convolutional networks have important properties in common with networks of neurons in the visual cortex of the Human brain [4]. First, there is a spatial array of input terminals. For image analysis this is the two-dimensional array of bits shown in Figure 7.17(**a**). Second, neurons are designed to detect local features of the image (such as edges or corners for instance). The maps learned by such neurons, from inputs to output, are referred to as *feature maps*. Since these features occur in different parts of the image, one uses the same *kernel* (or *filter*) for different parts of the image, always with the same weights and thresholds for different parts of the image. Since these kernels are local, and since they act in a translational-invariant way, the number of neurons from the two-dimensional input array is greatly reduced, compared with fully connected networks. Feature maps are obtained by convolution of the kernel with the input image. Therefore, layers consisting of a number of feature maps corresponding to different kernels are also referred to as *convolution layers*,

Figure 7.17(**b**).

Convolutional networks can have hierarchies of convolution layers. The idea is that the additional layers can learn more abstract features. Apart from feature maps, convolutional networks contain other types of layers. *Pooling layers* connect directly to the convolution layer(s), their task is to simplify the output of the convolution layers. Connected to the pooling layers, convolutional networks may also contain several fully connected layers.

7.3.1 Convolution layers

Figure 7.17(**a**) illustrates how a feature map is obtained by convolution of the input image with a kernel which reads a 3×3 part of the input image. In analogy with the terminology used in neuroscience, this area is called the *local receptive field* of the kernel. The outputs of the kernel from different parts of the input image make up the feature map, here an 8×8 array of hidden neurons: neuron V_{11} connects to the 3×3 area in the upper left-hand corner of the input image. Neuron V_{12} connects to a shifted area, as illustrated in Figure Figure 7.17(**a**), and so forth. Since the input has 10×10 pixels, the dimension of the feature map is 8×8 in this example. The important point is that the neurons V_{11} and V_{12}, and all other neurons in this convolution layer, share their weights and the threshold. In the example shown in Figure 7.17(**a**) there are thus only 9 independent weights, and one threshold. Since the different hidden neurons share weights and thresholds, their computation rule takes the form of a discrete *convolution*:

$$V_{ij} = g\Big(\sum_{p=1}^{3}\sum_{q=1}^{3} w_{pq}\, x_{p+i-1,q+j-1} - \theta\Big). \tag{7.38}$$

The activation function g can be the sigmoid function.

In Figure 7.17(**a**) the local receptive field is shifted by one pixel at a time. Sometimes it is useful to use a different *stride* (s_1, s_2), to shift the receptive field by s_1 pixels horizontally and by s_2 pixels vertically. Also, the local receptive regions need not have size 3×3. If we assume that their size is $Q \times P$, and that $s_1 = s_2 = s$, the rule (7.38) takes the form

$$V_{ij} = g\Big(\sum_{p=1}^{P}\sum_{q=1}^{Q} w_{pq}\, x_{p+s(i-1),q+s(j-1)} - \theta\Big). \tag{7.39}$$

Figure 7.17(**a**) depicts a two-dimensional input array. For colour images there are usually three colour *channels*, in this case the input array is three-dimensional, and the input bits are labeled by three indices: two for position and the last one for colour, x_{pqr}. Usually one connects several feature maps (corresponding to different

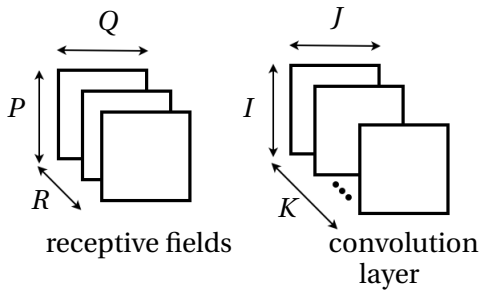


Figure 7.18: Illustration for Equation (7.40). Each feature map as a receptive field of dimension $P \times Q \times R$. There are K feature maps, each of dimension $I \times J$.

kernels) to the input layer, as shown in Figure 7.17(**b**). The different kernels detect different features of the input image, one detects edges for example, and another one detects corners, and so forth. To account for these extra dimensions, one groups weights (and thresholds) into higher-dimensional arrays (*tensors*). The convolution takes the form:

$$V_{ijk} = g\Big(\sum_{p=1}^{P}\sum_{q=1}^{Q}\sum_{r=1}^{R} w_{pqrk}\, x_{p+s(i-1),q+s(j-1),r} - \theta_k\Big) \tag{7.40}$$

(see Figure 7.18). All neurons in a given convolution layer have the same threshold. The software package *TensorFlow* [67] is designed to efficiently perform tensor operations as in Equation (7.40).

If one couples several convolution layers together, the number of neurons in these layers decreases rapidly as one moves to the right. In this case one can *pad* the image (and the convolution layers) by adding rows and columns of bits set to zero. In Figure 7.17(**a**), for example, one obtains a convolution layer of the same dimension as the original image by adding adding one column each on the left-hand and right-hand sides of the image, as well as two rows, one at the bottom and one at the top. In general, the numbers of rows and columns need not be equal, so the amount of padding is specified by four numbers, (p_1, p_2, p_3, p_4).

Convolution layers are trained with backpropagation. Consider the simplest case, Equation (7.38). As usual, we use the chain rule to evaluate the gradients:

$$\frac{\partial V_{ij}}{\partial w_{mn}} = g'(b_{ij})\frac{\partial b_{ij}}{\partial w_{mn}} \tag{7.41}$$

with local field $b_{ij} = \sum_{pq} w_{pq}\, x_{p+i-1,q+j-1} - \theta$. The derivative of b_{ij} is evaluated by applying rule (5.25):

$$\frac{\partial b_{ij}}{\partial w_{mn}} = \sum_{pq} \delta_{mp}\delta_{nq}\, x_{p+i-1,q+j-1} \tag{7.42}$$

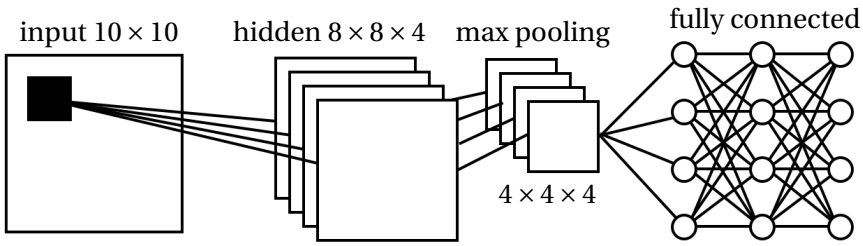


Figure 7.19: Layout of a convolutional neural network for object recognition and image classification. The inputs are stored in a 10×10 array. They feed into a convolution layer with four different feature maps, with 3×3 kernels, stride $(1,1)$, and zero padding. Each convolution layer feeds into its own max-pooling layer, with stride $(2,2)$ and zero padding. Between these and the output layer are a couple of fully connected hidden layers.

In this way one can train several stacked convolution layers too. It is important to keep track of the summation boundaries. To that end it helps to pad out the image and the convolution layers, so that the upper bounds remain the same in different layers.

Details aside, the fundamental principle of feature maps is that the map is applied in the same form to different parts of the image (*translational invariance*). In this way the learning of parameters is shared between pixels, each weight in a given feature map is trained on different parts of the image. This effectively increases the training set for the feature map and combats overfitting.

7.3.2 Pooling layers

Pooling layers process the output of convolution layers. A neuron in a pooling layer takes the outputs of several neighbouring feature maps and summarises their outputs into a single number. *Max-pooling units*, for example, summarise the outputs of nearby feature maps (in a 2×2 square for instance) by taking the maximum over the feature-map outputs. Instead, one may compute the root-mean square of the map values (L_2-*pooling*). There are no weights or thresholds associated with the pooling layers, they compute the output from the inputs using a pre-defined prescription. Other ways of pooling are discussed in Ref. [4]. Just as for convolution layers, we need to specify stride and padding for pooling layers.

Usually several feature maps are connected to the input. Pooling is performed separately on each of them. The network layout looks like the one shown schematically in Figure 7.19. In this Figure, the pooling layers feed into a number of fully connected hidden layers that connect to the output neurons. There are as many output neurons as there are classes to be recognised. This layout is qualitatively similar to the layout used by Krizhesvky *et al.* [66] in the ImageNet challenge (see

Figure 7.20: Examples of digits from the MNIST data set of handwritten digits [68]. The images were produced using MATLAB. But note that by default MATLAB displays the digits white one black background. Copyright for the data set: Y. LeCun and C. Cortes.

Section 7.5 below).

7.4 Learning to read handwritten digits

Figure 7.20 shows patterns from the MNIST data set of handwritten digits [68]. The data set derives from a data set compiled by the National Institute of Standards and Technology (NIST), of digits handwritten by high-school students and employees of the United States Census Bureau. The data contains a data set of 60 000 images of digits with 28×28 pixels, and a *test set* of 10 000 digits. The images are grayscale with 8-bit resolution, so each pixel contains a value ranging from 0 to 255. The images in the database were preprocessed. The procedure is described on the MNIST home page. Each original binary image from the National Institute of Standards and Technology was represented as a 20×20 gray-scale image, preserving the aspect ratio of the digit. The resulting image was placed in a 28×28 image so that the centre-of-mass of the image coincided with its geometrical centre. These steps can make a crucial difference (Section 7.4.3).

We divide the data set into a *training set* with 50 000 digits and a *validation set* with 10 000 digits. The latter is used for cross-validation and early stopping. The test data is used for measuring the classification error after training. For this purpose one should use a data set that was not involved in the training.

The goal of this Section is to show how the principles described in Chapters 6 and 7 allow to learn the MNIST data with low classification error, as outlined in Ref. [5]. You can follow the steps described below with your own computer program, using MATLAB 2017b which is available at StuDAT. But if you prefer you can also use other software packages such as Keras [69], an interface for TensorFlow [67], Theano [70], or PyTorch [71]. The networks described below use ReLU units (Section 7.2.2) in the hidden layers and a softmax output layer (Section 7.2.3) with ten output units O_i and energy function (7.25), so that output O_i is the probability that the pattern fed to the network falls into category i.

Algorithm 5 network layout and training options: no hidden layers, softmax output layer with 10 units. Here `net` is the network object containing the training data set, the network layout, and the training options.

```
layers =    [imageInputLayer([28 28 1])
             fullyConnectedLayer(10)
             softMaxLayer
             classificationLayer];
options =   trainingOptions(
             'sgdm',...
             'MiniBatchSize', 8192,...
             'ValidationData', {xValid, tValid},...
             'ValidationFrequency', 30,...
             'MaxEpochs',200,...
             'Plots', 'Training-Progress',...
             'L2Regularization', 0, ...
             'Momentum', 0.9, ...
             'ValidationPatience', 5, ...
             'Shuffle', 'every-epoch', ...
             'InitialLearnRate', 0.001);
net =       trainNetwork(xTrain, tTrain, layers, options);
```

7.4.1 Fully connected layout

The simplest network has no hidden layers at all, just one softmax output layer with 10 neurons. The representation of this network and its training algorithm in MATLAB is summarised in Algorithm 5. There are three parts. The layout of the network is defined in the array `layers=[...]`. Here `inputLayer([28 28 1])` reads in the 28×28 inputs, and preprocesses the inputs by subtracting the mean image averaged over the whole training set from each input image [Equation (6.19)]. The three array elements `fullyConnectedLayer`, `softMaxLayer` and `classificationLayer` define a softmax layer with 10 output units. First, `fullyConnectedLayer(10)` computes $b_j = \sum_k w_{jk} V_j + \theta_k$ for $j = 1, \ldots, 10$. Note that the sign of the threshold differs from the convention we use (Algorithm 2). Second, `softMaxLayer` computes the softmax function (7.23), where $O_i^{(\mu)}$ is the probability that input pattern $\boldsymbol{x}^{(\mu)}$ belongs to class i. Finally, `classificationLayer` computes the negative log likelihoods (7.25).

The `options` variable defines the training options for the backpropagation algorithm. This includes choices for the learning parameters, such as mini-batch size m_B, learning rate η, momentum constant α, and so forth. To find appropriate

parameter values and network layouts is one of the main difficulties when training a neural network, and it usually requires a fair deal of experimenting. There are recipes for finding certain parameters [72], but the general approach is still trial and error [5].

In `options`, '`sgdm`' means stochastic gradient descent with momentum, Equation (6.33). The momentum constant `Momentum` is set to $\alpha = 0.9$. The mini-batch size [Equation (6.14)] is set to 8912. The validation set is specified by `{xValid, tValid}`. The algorithm computes the validation error during training, in intervals specified by `ValidationFrequency`. This allows for cross validation and early stopping. Roughly speaking training stops when the validation error begins to increase. During training, the algorithm keeps track of the smallest validation error observed so far. Training stops when the validation error was larger than the minimum for a specified number of times, `ValidationPatience`. This variable is set to 5 in Algorithm 5. The variable `Shuffle` determines at which intervals the sequence of patterns in the training set is randomised. The parameter `InitialLearnRate` defines the initial learning rate, $\eta = 0.001$. By default it does not change as a function of time. One `Epoch` corresponds to applying p patterns or p/m_B mini batches (Section 6.1).

The resulting classification accuracy is about 90%. It can be improved by adding a hidden layer with 30 ReLU units (Algorithm 6), giving a classification accuracy of about 96%. The accuracy can be further improved by increasing the number of neurons in the hidden layer. For 100 hidden ReLU units the accuracy becomes about 97.2% after training for 200 epochs (early stopping occurred after 135 epochs). Figure 7.21 shows how the training and the validation energies decrease during training, for both networks. You see that the energies are a little lower for the network with 100 hidden neurons. But we observe overfitting in both cases, because after many training steps the validation energy is much higher than the training energy. As mentioned above, *early stopping* caused the training of the larger network to abort after 135 epochs, this corresponds to 824 iterations.

Now let us add more hidden layers. Experimenting shows that it is better to use a slightly higher learning rate, $\eta = 0.01$. For two hidden layers we obtain classification

Algorithm 6 one fully connected hidden layer, softmax outputs.

```
layers = [imageInputLayer([28 28 1])
          fullyConnectedLayer(30)
          reLuLayer
          fullyConnectedLayer(10)
          softMaxLayer
          classificationLayer];
```

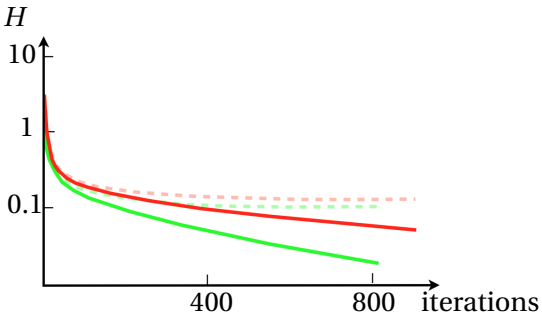


Figure 7.21: Energy functions for the MNIST training set (solid lines) and for the validation set (dashed lines) for Algorithm 6 (red lines) and for a similar algorithm, but with 100 neurons in the hidden layer, green lines. The data was smoothed and the plot is schematic. The x-axis shows iterations. One iteration corresponds to feeding one minibatch of patterns. One epoch consists of $50000/8192 \approx 6$ iterations.

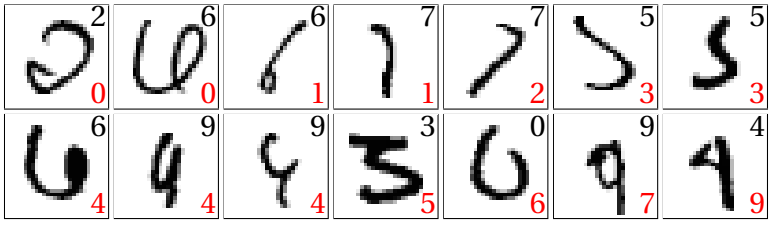


Figure 7.22: Some hand-written digits from the MNIST test set, misclassified by a convolutional net that achieved an overall classification accuracy of 98%. Correct classification (top right), misclassification (bottom right). Data from Oleksandr Balabanov.

accuracies that are only slightly higher, 97.3%. Adding a third hidden layer does not help much either. Try adding even more neurons, and/or more layers. You will see that it is difficult to increase the classification accuracy further. Adding more fully connected hidden layers does not necessarily improve the classification accuracy, even if you train for more epochs. One possible reason is that the network overfits the data (Section 6.3.2). This problem becomes more acute as you add more hidden neurons. The tendency of the network to overfit is reduced by regularisation (Section 7.2.5). For the network with one hidden layer with 100 ReLU units, L_2-regularisation improves the classification accuracy to almost 98%. Here `L2Regularization` was set to 0.03 and the learning rate to $\eta = 0.03$.

7.4.2 Convolutional networks

Deep convolutional networks can yield still higher classification accuracies than those obtained in the previous Section. The layout of Algorithm 7 corresponds to a network with one convolution layer with 20 feature maps, a max-pooling layer, and

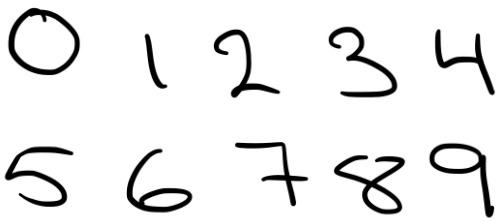


Figure 7.23: Examples of digits drawn on an Ipad. Data from Oleksandr Balabanov.

a fully connected hidden layer with 100 ReLU units, similar to the network shown in Figure 7.19. The classification accuracy obtained after 60 epochs is slightly above 98%. It can be further improved by including a second convolution layer, and batch normalisation (Section 7.2.6). See Algorithm 8, a slightly modified version of an example from MathWorks. The classification accuracy is 98.99% after 30 epochs.

The accuracy can be improved further by tuning parameters and network layout, and by using ensembles of convolutional neural networks [68]. The best classification accuracy found in this way is 99.77% [73]. Several of the MNIST digits are difficult to classify for Humans too (Figure 7.22), so we conclude that convolutional nets really work very well. Yet the above examples show also that it takes much experimenting to find the right parameters and network layout as well as long training times to reach the best classification accuracies. It could be argued that one reaches a stage of *diminishing returns* as the classification error falls below a few percent.

7.4.3 Reading your own hand-written digits

In this Section I outline how you can test the convolutional networks described above on your own data set of hand-written digits, and which conclusions you can draw from such an experiment. Make your own data set by drawing the digits on an Ipad with GoodNotes or a similar program. Draw as many digits as possible, save them in a PNG file, and extract the individual digits using an image-processing program such as `Paint`. Figure 7.23 shows digits obtained in this way. Now try out one of your convolutional networks (Algorithm 7 or 8) trained on the MNIST data set. You will see that the network has great difficulties recognising your digits.

One way of solving this problem is to add digits like the ones from Figure 7.23 to the training set. A second possibility is to try to preprocess the digits from Figure 7.23 in the same way as the MNIST data was preprocessed. The result is shown in Figure 7.24. Using a MNIST-trained convolutional net on these digits yields a classification accuracy of about 90%. So the algorithm does not work very well at all. What is going on?

Compare Figures 7.20 and 7.24. The digits in Figure 7.24 have a much more slen-

Algorithm 7 convolutional network, one convolution layer.

```
layers =    [imageInputLayer([28 28 1])
             convolution2dLayer (5, 20, 'Padding',1)
             reLULayer
             maxPooling2DLayer(2, 'Stride',2)
             fullyConnectedLayer(100)
             reLuLayer
             fullyConnectedLayer(10)
             softMaxLayer
             classificationLayer];
options =   trainingOptions('sgdm',...
             'MiniBatchSize', 8192,...
             'ValidationData', {xValid, tValid},...
             'MaxEpochs',60,...
             'Plots', 'Training-Progress',...
             'L2Regularization', 0, ...
             'InitialLearnRate', 0.001);
net =       trainNetwork(xTrain, tTrain, layers, options);
```

Algorithm 8 several convolution layers, batch normalisation. After MathWorks.

```
layers =  [imageInputLayer([28 28 1])
           convolution2dLayer (3, 20, 'Padding',1)
           batchNormalizationLayer
           reLULayer
           maxPooling2DLayer(2, 'Stride',2)
           convolution2dLayer (3, 30, 'Padding',1)
           batchNormalizationLayer
           reLULayer
           maxPooling2DLayer(2, 'Stride',2)
           convolution2dLayer (3, 50, 'Padding',1)
           batchNormalizationLayer
           reLULayer
           fullyConnectedLayer(10)
           softMaxLayer
           classificationLayer];
```

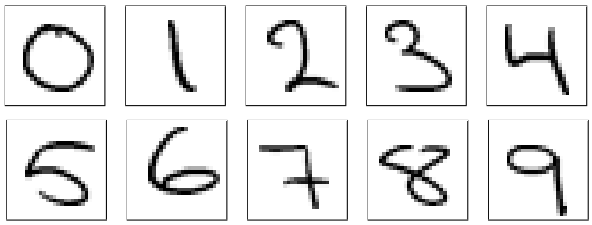


Figure 7.24: Same digits as in Figure 7.23, but preprocessed like the MNIST digits. Data from Oleksandr Balabanov.

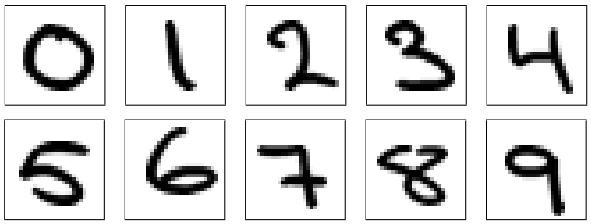


Figure 7.25: Same digits as in Figure 7.24. The difference is that the thickness of the stroke was normalised (see text). Data from Oleksandr Balabanov.

der stroke. It was suggested in Ref. [74] that this may be the reason, since it is known that the line thickness of hand-written text can make a difference for algorithms that read hand-written text [75]. There are different methods for normalising the line thickness of hand-written text. Applying the method proposed in Ref. [75] to our digits results in Figure 7.25. The algorithm of Ref. [75] has a free parameter, T, that specifies the resulting line thickness. In Figure 7.25 it was taken to be $T = 10$, close to the line thickness of the MNIST digits, we measured the latter to $T \approx 9.7$ using the method described in Ref. [75].

If we run a MNIST-trained convolutional net (Algorithm 8) on a data set of 60 digits with normalised line thickness, it fails on only two digits. This corresponds to a classification accuracy of roughly 97%, not so bad – but not as good as the best results in Section 7.4.2. Note that we can only make a rough comparison. In order to to obtain a better estimate of the classification accuracy we need to test many more than 60 digits. A question is of course whether there are perhaps other differences between our own hand-written digits and those in the MNIST data. It would also be of interest to try digits that were drawn using *Paint*, or a similar program. How does do MNIST-trained convolutional nets perform on computer-drawn digits?

At any rate, the results of this Section show that the way the input data are processed can make a big difference. This raises a point of fundamental importance. We have seen that convolutional nets can be trained to represent a distribution of input patterns with very high accuracy. But if you test the network on a data set that

Figure 7.26: Object recognition using a deep convolutional network. Shown is a frame from a movie recorded on a telephone. The network was trained on the Pascal VOC data set [76] using YOLO [77]. Details on how to obtain the weights and how to install the software are given on the YOLO website.

has a slightly different distribution, perhaps because it was preprocessed differently, the network may not work as well.

7.5 Deep learning for object recognition

Deep learning has become so popular in the last few years because deep convolutional networks are good at recognising objects in images. Figure 7.26 shows a frame from a movie taken from a car with my mobile telephone. A deep convolutional network trained on the Pascal VOC training set [76] recognises objects in the movie by putting bounding boxes around the objects and classifying them. The Pascal VOC data set is a training set for object-class recognition in images. It contains circa 20 000 images, each annotated with one of 20 classes. The people behind this data set ran image classification challenges from 2005 to 2012. A more recent challenge is the *ImageNet large-scale visual recognition challenge* (ILSVRC) [78], a competition for image classification and object recognition using the ImageNet database [63]. The challenge is based on a subset of ImageNet. The training set contains more than 10^6 images manually classified into one of 1000 classes. There are approximately 1000 images for each class. The validation set contains 50 000 images.

The ILSVRC challenge consists of several tasks. One task is *image classification*, to list the object classes found in the image. A common measure for accuracy is the so-called *top-5 error* for this classification task. The algorithm lists the five object classes it identified with highest probabilities. The result is considered correct if the annotated class is among these five. The error equals the fraction of incorrectly

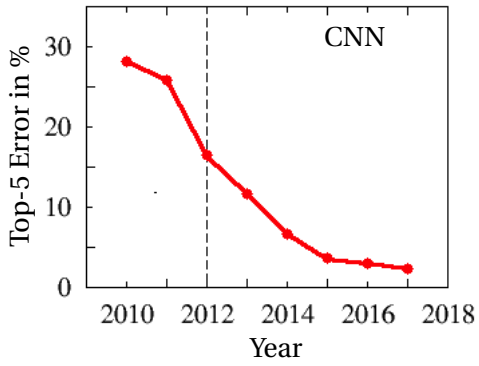


Figure 7.27: Smallest classification error for the ImageNet challenge [78]. The data up to 2014 comes from Ref. [78]. The data for 2015 comes from Ref. [79], for 2016 from Ref. [80], and for 2017 from Ref. [81]. From 2012 onwards the smallest error was achieved by convolutional neural networks (CNN). After Goodfellow *et al.* [4].

classified images. Why does one not simply judge whether the most probable class is the correct one? The reason is that the images in the ImageNet database are annotated by a single-class identifier. Often this is not unique. The image in Figure 7.20, for example, shows not only a car but also trees, yet the image is annotated with the class label *car*. This is ambiguous. The ambiguity is significantly smaller if one considers the top five classes the algorithm gives, and checks whether the annotated class is among them.

The tasks in the ILSVRC challenge are significantly more difficult than the digit recognition described in Section 7.4, and also more difficult than the VOC challenges. One reason is that the ImageNet classes are organised into a deep hierarchy of subclasses. This results in highly specific sub classes that can be very difficult to distinguish. The algorithm must be very sensitive to small differences between similar sub classes. We say that the algorithm must have high *inter-class variability* [82]. Different images in the same sub class, on the other hand, may look quite different. The algorithm should nevertheless recognise them as similar, belonging to the same class, the algorithm should have small *intra-class variability* [82].

Since 2012, algorithms based on deep convolutional networks won the ILSVRC challenge. Figure 7.27 shows that the error has significantly decreased until 2017, the last year of the challenge in the form described above. We saw in previous Sections that deep networks are difficult to train. So how can these algorithms work so well? It is generally argued that the recent success of deep convolutional networks is mainly due to three factors.

First, there are now much larger and better annotated training sets available. ImageNet is an example. Excellent training data is now recognised as one of the most important factors, and companies developing software for self-driving cars

Figure 7.28: Reproduced from xkcd.com/1897 under the creative commons attribution-noncommercial 2.5 license.

and systems that help to avoid accidents recognise that good training sets is one of the most important factors, and difficult to achieve: to obtain reliable training data one must *manually* collect and annotate the data (Figure 7.28). This is costly, but at the same time it is important to have as large data sets as possible, to reduce overfitting. In addition one must aim for a large variability in the collected data.

Second, the hardware is much better today. Deep networks are nowadays implemented on single or multiple GPUs. There are also dedicated chips, such as the tensor processing unit [83].

Third, improved regularisation techniques (Section 7.2.5) and weight sharing in convolution layers help to fight overfitting, and ReLU units (Section 7.2.2) render the networks less susceptible to the vanishing-gradient problem (Section 7.2.1).

The winning algorithm for 2012 was based on a network with five convolution layers and three fully connected layers, using drop out, ReLU units, and data-set augmentation [66]. The algorithm was implemented on GPU processors. The 2013 ILSVRC challenge was also won by a convolutional network [84], with 22 layers. Nevertheless, the network has substantially fewer free parameters (weights and thresholds) than the 2012 network: 4×10^6 instead of 60×10^6. In 2015, the winning algorithm [79] had 152 layers. One significant new element in the layout were connections that skip layers (*residual networks*, Section 7.6). The 2016 [85] and 2017 [81] winning algorithms used ensembles of convolutional networks.

7.6 Residual networks

The network that won the 2015 ILSVRC challenge had connections that skip layers [79]. Empirical evidence shows that skipping layers makes deep networks easier to train. This is usually motivated by saying that skipping layers reduces the vanishing-gradient problem.

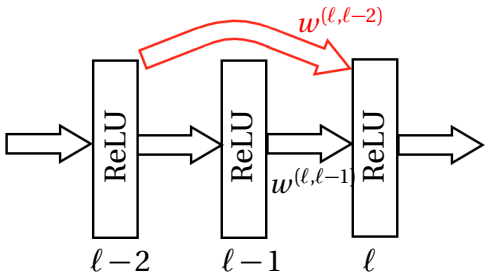


Figure 7.29: Schematic illustration of a network with skipping connections.

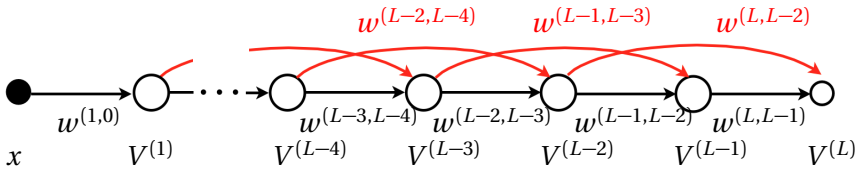


Figure 7.30: 'Network' with connections that skip layers.

The layout is illustrated schematically in Figure 7.29. Black arrows stand for usual feed-forward connections. The notation differs somewhat from that of Algorithm 2. Here the weights from layer $\ell-1$ to ℓ are denoted by $w_{jk}^{(\ell,\ell-1)}$, and those from layer $\ell-2$ to ℓ by $w_{ij}^{(\ell,\ell-2)}$ (red arrow in Figure 7.29). Note that the superscripts are ordered in the same way as the subscripts: the *right* index refers to the layer on the *left*. Neuron j in layer ℓ computes

$$V_j^{(\ell)} = g\Big(\sum_k w_{jk}^{(\ell,\ell-1)} V_k^{(\ell-1)} - \theta_j^{(\ell)} + \sum_n w_{jn}^{(\ell,\ell-2)} V_n^{(\ell-2)}\Big). \tag{7.43}$$

The weights of connections that skip layers are trained in the usual fashion, by stochastic gradient descent. To illustrate the structure of the resulting formulae consider a 'network' with just one neuron per layer (Figure 7.30). To begin with we calculate the increments of the weights $w^{(\ell,\ell-1)}$. To update $w^{(L,L-1)}$ we require

$$\frac{\partial V^{(L)}}{\partial w^{(L,L-1)}} = g'(b^{(L)}) V^{(L-1)}. \tag{7.44}$$

This gives

$$\delta w^{(L,L-1)} = \eta \delta^{(L)} V^{(L-1)} \quad \text{with} \quad \delta^{(L)} = (t - V^{(L)}) g'(b^{(L)}), \tag{7.45}$$

as in Algorithm 2. The factor $(t - V^{(L)})$ comes from the outer derivative of the energy function (6.4). The outputs are $O = V^{(L)}$. As in Algorithm 2, I have omitted the sum over μ (stochastic gradient descent, page 86).

Also the update for $w^{(L-1,L-2)}$ is the same as in Algorithm 2:

$$\delta w^{(L-1,L-2)} = \eta \delta^{(L-1)} V^{(L-2)} \quad \text{with} \quad \delta^{(L-1)} = \delta^{(L)} w^{(L,L-1)} g'(b^{(L-1)}). \tag{7.46}$$

But the update for $w^{(L-2,L-3)}$ is different because the short cuts come into play. The extra connection from layer $L-2$ to L gives rise to an extra term:

$$\frac{\partial V^{(L)}}{\partial w^{(L-2,L-3)}} = \Big(\frac{\partial V^{(L)}}{\partial V^{(L-1)}} \frac{\partial V^{(L-1)}}{\partial V^{(L-2)}} + \frac{\partial V^{(L)}}{\partial V^{(L-2)}} \Big) \frac{\partial V^{(L-2)}}{\partial w^{(L-2,L-3)}}. \tag{7.47}$$

Evaluating the partial derivatives we find

$$= \Big(g'(b^{(L)}) w^{(L,L-1)} g'(b^{(L-1)}) w^{(L-1,L-2)} + g'(b^{(L)}) w^{(L,L-2)} \Big) g'(b^{(L-2} V^{(L-2)}.$$

This implies

$$\delta^{(L-2)} = \delta^{(L-1)} w^{(L-1,L-2)} g'(b^{(L-2}) + \delta^{(L)} w^{(L,L-2)} g'(b^{(L-2)}). \tag{7.48}$$

In general, the error-backpropagation rule reads

$$\delta^{(\ell-1)} = \delta^{(\ell)} w^{(\ell,\ell-1)} g'(b^{(\ell-1)}) + \delta^{(\ell+1)} w^{(\ell+1,\ell-1)} g'(b^{(\ell-1)}) \tag{7.49}$$

for $\ell = L-1, L-2, \ldots$. The first term is the same as in step 9 of Algorithm 2. The second term is due to the skipping connections.

The update formula for $w^{(\ell,\ell-1)}$ is

$$\delta w^{(\ell,\ell-1)} = \eta \delta^{(\ell)} V^{(\ell-1)}. \tag{7.50}$$

The updates of the weights $w^{(\ell+1,\ell-1)}$ are given by

$$\delta w^{(\ell+1,\ell-1)} = \eta \delta^{(\ell+1)} V^{(\ell-1)}, \tag{7.51}$$

with the same errors as in Equation (7.50).

Skipping connections reduce the vanishing-gradient problem. To see this, note that we can write the error $\delta^{(\ell)}$ as

$$\delta^{(\ell)} = \delta^{(L)} \sum_{\ell_1, \ell_2, \ldots, \ell_n} w^{(L,\ell_n)} g'(b^{(\ell_n)}) \cdots w^{(\ell_2,\ell_1)} g'(b^{(\ell_1)}) w^{(\ell_1,\ell)} g'(b^{(\ell)}) \tag{7.52}$$

where the sum is over all paths $L > \ell_n > \ell_{n-1} > \cdots > \ell_1 > \ell$ back through the network. The smallest gradients are dominated by the product corresponding to the path with the smallest number of steps (factors), resulting in a smaller probability to get small gradients. Introducing connections that skip more than one layer tends to increase the small gradients, as Equation (7.52) shows. Recently it has been suggested to

randomise the layout by randomly short-circuiting the network. Equation (7.52) remains valid for this case too.

The network described in Ref. [79] used unit weights for the skipping connections,

$$V_j^{(\ell)} = g\Big(\sum_k w_{jk}^{(\ell,\ell-1)} V_k^{(\ell-1)} - \theta_j^{(\ell)} + V_j^{(\ell-2)}\Big), \tag{7.53}$$

so that the hidden layer $V_k^{(\ell-1)}$ learns the difference between the input $V_j^{(\ell-2)}$ and the output $V_j^{(\ell)}$. Therefore such networks are called *residual networks*.

7.7 Summary

Networks with many hidden layers are called deep networks. It has recently been shown that such networks can be trained to recognise objects in images with high accuracy. It is sometimes stated that convolutional networks are now *better than Humans*, in that they recognise objects with lower classification errors than Humans [86]. This statement is problematic for several reasons. To start with, the article refers to the 2015 ILSVRC competition, and the company mentioned in the Guardian article was later caught out cheating. At any rate, this and similar statements refer to an experiment showing that the Human classification error in recognising objects in the ImageNet database is about 5.1% [87], worse than the most recent convolutional neural-network algorithms (Figure 7.27).

Yet it is clear that these algorithms learn in quite a different way from Humans. They can detect local features, but since these convolutional networks rely on translational invariance, they do not easily understand global features, and can mistake a leopard-patterned sofa for a leopard [88]. It may help to include more sofas in the training data set, but the essential difficulty remains: translational invariance imposes constraints on what convolutional networks can learn [88].

More fundamentally one may argue that Humans learn differently, by abstraction instead of going through vast training sets. Just try it out for yourself, this website [89] allows you to learn like a convolutional network. Nevertheless, the examples described in this Chapter illustrate the tremendous success of deep convolutional networks.

We have also seen that training deep networks suffers from a number of fundamental problems. First, networks with many hidden neurons have many free parameters (their weights and thresholds). This increases the risk of overfitting. Overfitting reduces the power of the network to generalise. The tendency of deep networks to overfit can be reduced by cross-validation (Section 6.3.2) and by regularisation (weight decay, drop out, pruning, and data set augmentation, Section 7.2.5). In this regard convolutional nets have an advantage because they have fewer

weights, and the weights of a given feature map are trained on different parts of the input images, effectively increasing the training set.

Second, the examples described in Section 7.4 show that convolutional nets are sensitive to differences in how the input data are preprocessed. You may run into problems if you train a network on given training and validation sets, but apply it to a test set that was preprocessed in a different way – so that the test set corresponds to a different input distribution. Convolutional nets excel at learning the properties of a given input distribution, but they may have difficulties in recognising patterns sampled from a slightly different distribution, even if the two distributions appear very similar to the Human eye. Note also that this problem cannot be solved by cross-validation, because training and validation sets are drawn from the same input distribution, but here we are concerned with what happens when the network is applied to a input distribution different from the one that was trained on. Here is another example illustrating this point: the authors of Ref. [90] trained a convolutional network on perturbed grayscale images from the ImageNet data base, adding a little bit of noise independently to each pixel (*white noise*) before training. This network failed to recognise images that were weakly perturbed in a different way, by setting a small number of pixels to white or black. When we look at the images we have no difficulties seeing through the noise.

Third, error backpropagation in deep networks suffers from the vanishing-gradient problem. This is more difficult to combat. It can be reduced by using ReLU units, by initialising the weights in certain ways, and by networks with connections that skip layers. Yet vanishing or exploding gradients remain a fundamental difficulty, slowing learning down in the initial phase of training. Brute force (computer power) helps to alleviate the problem. As a consequence, convolutional neural networks have become immensely successful in object recognition, outperforming other algorithms significantly.

Fourth, Refs. [91, 92] illustrate intriguing failures of convolutional networks. Szegedy *et al.* [91] show that the way convolutional nets partition input space can lead to surprising results. The authors took an image that the network classifies correctly with high confidence, and it perturbed slightly. The difference between the original and perturbed images (*adversarial images*) is undetectable to the Human eye, yet the network misclassifies the perturbed image with high confidence [91]. This indicates that decision boundaries are always close in input space, not intuitive but possible in high dimensions. Figure 1 in Ref. [92] shows images that are completely unrecognisable to the Human eye. Yet a convolutional network classifies these images with high confidence. This illustrates that there is no telling what a network may do if the input is far away from the training distribution. Unfortunately the network can sometimes be highly confident yet wrong.

To conclude, convolutional networks are very good at recognising objects in

images. But we should not imagine that they *understand* what they see in the same way as Humans. The theory of deep learning has somewhat lagged behind the performance in practice. But some progress has been made in recent years, and there are many interesting open questions.

7.8 Further reading

What do the hidden layers in a convolutional layer actually compute? Feature maps that are directly coupled to the inputs detect local features, such as edges or corners. Yet it is unclear precisely how hidden convolutional layers help the network to learn. Therefore it is interesting to visualise the activity of deep layers by asking: which input patterns maximise the outputs of the neurons in a certain layer [93]?

Another question concerns the structure of the energy landscape. It seems that local minima are perhaps less important for deep networks, because their energy functions tend to have more saddle points than minima [94].

Deep networks suffer from *catastrophic forgetting*: when you train a network on a new input distribution that is quite different from the one the network was originally trained on, then the network tends to forget what it learned initially. Recently there has been much interest in this question. A good starting point is Ref. [95].

The stochastic-gradient descent algorithm (with or without minibatches) samples the input-data distribution uniformly randomly. As mentioned in Section 6.3.1, it may be advantageous to sample those inputs more frequently that initially cause larger output errors. More generally, the algorithm may use other criteria to choose certain input data more often, with the goal to speed up learning. It may even suggest how to augment a given training set most efficiently, by asking to specifically label certain types of input data (*active learning*) [96].

7.9 Exercises

7.1 Decision boundaries for XOR problem. Figure 7.6 shows the layout of a network that solves the Boolean XOR problem. Draw the decision boundaries for the four hidden neurons in the input plane, and label the boundaries and the regions as in Figure 5.12.

7.2 Vanishing-gradient problem. Train the network shown in Figure 7.9 on the iris data set, available from the Machine learning repository of the University of California Irvine. Measure the effects upon of the neurons in the different layers, by calculating the derivative of the energy function H w.r.t. the thresholds of the neurons in question.

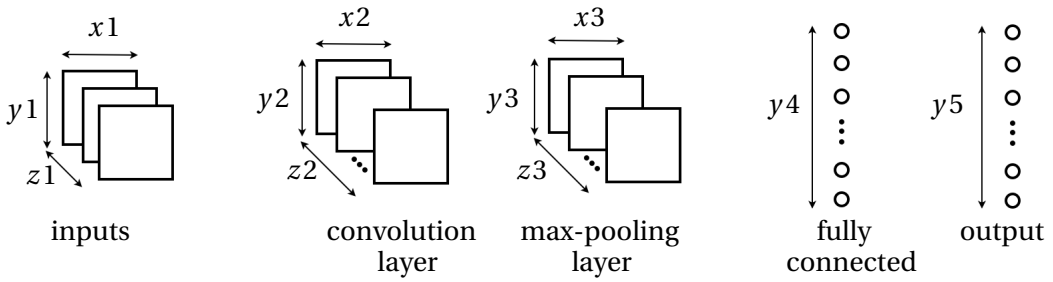


Figure 7.31: Layout of convolutional net for Exercise 7.3.

7.3 Number of parameters of a convolutional net. A convolutional net has the following layout (Figure 7.31): an input layer of size $21 \times 21 \times 3$, a convolutional layer with ReLU activations with 16 kernels with local receptive fields of size 2×2, stride $(1,1)$, and padding $(0,0,0,0)$, a max-pooling layer with local receptive field of size 2×2, stride $= (2,2)$, padding $= (0,0,0,0)$, a fully connected layer with 20 neurons with sigmoid activations, and a fully connected output layer with 10 neurons. In one or two sentences, explain the function of each of the layers. Enter the values of the parameters $x1, y1, z1, x2, \ldots, y5$ into Figure 7.31 and determine the number of trainable parameters (weights and thresholds) for the connections into each layer of the network.

7.4 Residual network. Derive (7.52) for the error $\delta^{(\ell)}$ in layer ℓ of the residual 'network' shown in Figure 7.30.

7.5 Parity function. The parity function outputs 1 if and only if the input sequence of N binary numbers has an odd number of ones, and zero otherwise. The parity function for $N = 2$ is the XOR function. The XOR function can be represented using a multilayer perceptron with two inputs, a fully connected hidden layer with two hidden neurons, and one output unit. Determine suitable weights w_{jk} and thresholds θ_j for the two hidden units ($j = 1,2$), as well as the weights W_j and the threshold Θ for the output unit. Draw the corresponding decision boundaries in the input plane. Also draw the problem in the V_1-V_2 plane, with the decision boundary corresponding to the output neuron. Describe how to combine several such XOR units to represent the parity function for $N > 2$. Explain how the total number of neurons in the network grows with the input dimension N, for $N = 2^k$.

7.6 Cross-entropy function. The cross-entropy function (7.31) is an energy function for sigmoid output neurons. Write down a cross-entropy function for tanh-output units, and show that it has a global minimum at $\boldsymbol{O}^{(\mu)} = \boldsymbol{t}^{(\mu)}$, where the cross-entropy function takes the value zero.

7.7 Softmax outputs. Consider a perceptron with L layers and softmax output units. For pattern μ, the state of the output neuron i is given by

$$O_i^{(\mu)} = \frac{e^{b_i^{(L,\mu)}}}{\sum_j e^{b_j^{(L,\mu)}}}, \tag{7.54}$$

where $b_j^{(L,\mu)}$ denotes the *local field* $b_j^{(L,\mu)} = -\theta_j^{(L)} + \sum_k w_{jk}^{(L)} V_k^{(L-1,\mu)}$. Here $\theta_j^{(L)}$ and $w_{jk}^{(L)}$ are thresholds and weights, and $V_k^{(L-1,\mu)}$ is the state of the k^{th} neuron in layer $L-1$, evaluated for pattern μ. Compute the derivative of output $O_i^{(\mu)}$ with respect to the local field $b_m^{(L,\mu)}$ of output neuron m.
The network is trained by gradient descent on the negative log-likelihood function,

$$H = -\sum_{i\mu} t_i^{(\mu)} \log\bigl(O_i^{(\mu)}\bigr). \tag{7.55}$$

The targets $t_i^{(\mu)}$ satisfy the constraint $\sum_i t_i^{(\mu)} = 1$, for all patterns μ. When updating, the increment of a weight $w_{mn}^{(\ell)}$ in layer ℓ is given by

$$\delta w_{mn}^{(\ell)} = -\eta \frac{\partial H}{\partial w_{mn}^{(\ell)}}, \tag{7.56}$$

where η denotes the learning rate. Derive the increment for weight $w_{mn}^{(L)}$ in layer L.

7.8 Convolutional neural net. Figure 7.19 shows a schematic layout of a convolutional net. Explain how a *convolution layer* works. In your discussion, refer to the terms *convolution*, *colour channel*, *receptive field*, *feature map*, *stride*, and explain the meaning of the parameters in the computation rule

$$V_{ij} = g\Bigl(\sum_{p=1}^{P}\sum_{q=1}^{Q} w_{pq}\, x_{p+s(i-1),q+s(j-1)} - \theta\Bigr). \tag{7.57}$$

Explain how a *pooling layer* works, and why it is useful.

7.9 Feature map. The two patterns shown in Figure 7.32(a) are processed by a very simple convolutional network that has one convolution layer with one single 3×3 kernel with ReLU units, zero threshold, weights as given in Figure 7.32(b), and stride (1,1). The resulting feature map is fed into a 3×3 max-pooling layer with stride (1,1). Finally there is a fully connected classification layer with two output units with Heaviside activation functions.
For both patterns determine the resulting feature map and the output of the max-pooling layer. Determine weights and thresholds of the classification layer that allow to classify the two patterns into different classes.

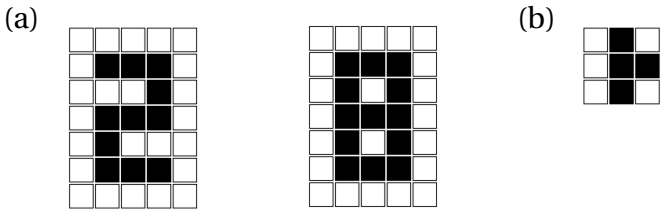


Figure 7.32: (a) Input patterns with 0/1 bits (□ corresponds to $x_i = 0$ and ■ to $x_i = 1$). (b) 3×3 kernel of a feature map. ReLU units, zero threshold, weights either 0 or 1 (□ corresponds to $w = 0$ and ■ to $w = 1$). (Exercise 7.9).

8 Supervised recurrent networks

The layout of the perceptrons analysed in the previous Chapters is special. All connections are one way, and only to the layer immediately to the right, so that the update rule for the i-th neuron in layer ℓ becomes

$$V_i^{(\ell)} = g\Big(\sum_j w_{ij}^{(\ell)} V_j^{(\ell-1)} - \theta_i^{(\ell)}\Big). \tag{8.1}$$

The backpropagation algorithm relies on this *feed-forward* layout. It means that the derivatives $\partial V_j^{(\ell-1)}/\partial w_{mn}^{(\ell)}$ vanish. This ensures that the outputs are nested functions of the inputs, which in turn implies the simple iterative structure of the backpropagation algorithm on page 89.

In some cases it is necessary or convenient to use networks that do not have this simple layout. The Hopfield networks discussed in part I are examples where all connections are symmetric. More general networks may have a feed-forward layout with *feedbacks*, as shown in Figure 8.1. Such networks are called *recurrent networks*. There are many different ways in which the feedbacks can act: from the output layer to hidden neurons for example (Figure 8.1), or there could be connections between the neurons in a given layer. Neurons 3 and 4 in Figure 8.1 are output units, they are associated with targets just as in Chapters 5 to 7. The layout of recurrent networks is very general, but because of the feedback links we must consider how such networks can be trained.

Unlike the multi-layer perceptrons, recurrent networks are commonly used as *dynamical networks*, just as in Chapters 1 and 2 [c.f. Equation (1.4)]. The dynamics can either be discrete

$$V_i(t) = g\Big(\sum_j w_{ij}^{(vv)} V_j(t-1) + \sum_k w_{ik}^{(vx)} x_k - \theta_i^{(v)}\Big) \quad \text{for} \quad t = 1, 2, \ldots, \tag{8.2}$$

or continuous (Exercise 2.10)

$$\tau \frac{\mathrm{d}V_i}{\mathrm{d}t} = -V_i + g\Big(\sum_j w_{ij}^{(vv)} V_j(t) + \sum_k w_{ik}^{(vx)} x_k - \theta_i^{(v)}\Big), \tag{8.3}$$

with time constant τ. The parameters $\theta_i^{(v)}$ are thresholds. We shall see in a moment why it can be advantageous to use a dynamical network.

Recurrent networks can learn in different ways. One possibility is to use a training set of pairs $(\boldsymbol{x}^{(\mu)}, \boldsymbol{y}^{(\mu)})$ with $\mu = 1, \ldots, p$. To avoid confusion with the iteration index t, the targets are denoted by y in this Chapter. One feeds a pattern from this set and runs the dynamics (8.2) or (8.3) for the given $\boldsymbol{x}^{(\mu)}$ until the dynamics reaches a

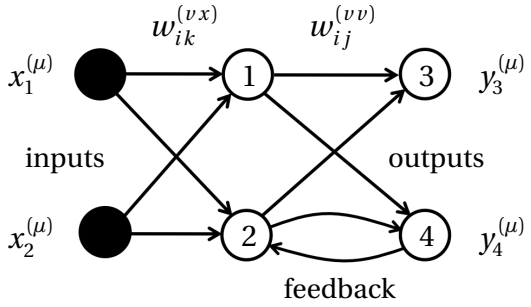


Figure 8.1: Network with a feedback connection. Neurons 1 and 2 are hidden neurons. The weights from the input x_k to the neurons V_i are denoted by $w_{ik}^{(vx)}$, the weight from neuron V_j to neuron V_i is $w_{ij}^{(vv)}$. Neurons 3 and 4 are output neurons, with prescribed target values y_i. To avoid confusion with the iteration index t, the targets are denoted by y in this Chapter.

steady state $\boldsymbol{V}^*$ (if this does not happen the training fails). Then one updates the weights by gradient descent using the energy function

$$H = \frac{1}{2}\sum_k E_k^2 \quad \text{where } E_k = \begin{cases} y_k - V_k & \text{if } V_k \text{ is an output unit,} \\ 0 & \text{otherwise,} \end{cases} \tag{8.4}$$

evaluated at $\boldsymbol{V} = \boldsymbol{V}^*$, that is $H^* = \frac{1}{2}\sum_k (E_k^*)^2$ with $E_k^* = y_k - V_k^*$. Instead of defining the energy function in terms of the mean-squared output errors, one could also use the negative log-likelihood function (7.31). These steps are repeated until the steady-state outputs yield the correct targets for all input patterns. This is reminiscent of the algorithms discussed in Chapters 5 to 7, and we shall see that the backpropagation algorithm can be modified (*recurrent backpropagation*) to make the networks learn as described earlier.

Another possibility is that inputs and targets change as functions of time t while the network dynamics runs. In this way the network can solve *temporal association tasks* where it learns to output certain targets in response to the sequence $\boldsymbol{x}(t)$ of input patterns, and targets $\boldsymbol{y}(t)$. In this way recurrent networks can translate written text or recognise speech. Such networks can be trained by unfolding their dynamics in time as explained in Section 8.2 (*backpropagation in time*), although this algorithm suffers from the vanishing-gradient problem discussed in Chapter 7.

8.1 Recurrent backpropagation

accelerating convergence. IEEE Transactions on Neural Networks, 11(3):697709, 2000. Recall Figure 8.1. We want to train a network with N real-valued units V_i with sigmoid activation functions, and weights w_{ij} from V_j to V_i. Several of the units

may be connected to inputs $x_k^{(\mu)}$. Other units are output units with associated target values $y_i^{(\mu)}$. We take the dynamics to be continuous in time, Equation (8.3), and assume that the dynamics runs into a steady state

$$\boldsymbol{V}(t) \to \boldsymbol{V}^* \quad \text{so that} \quad \frac{\mathrm{d}V_i^*}{\mathrm{d}t} = 0\,. \tag{8.5}$$

From Equation (8.3) we deduce

$$V_i^* = g\Big(\sum_j w_{ij}^{(vv)} V_j^* + \sum_k w_{ik}^{(vx)} x_k - \theta_i^{(v)}\Big). \tag{8.6}$$

In other words we assume that the dynamics (8.3) has a *stable* steady state, so that small perturbations δV_i away from V_i^* decay with time. Equation (8.6) is a nonlinear self-consistent Equation for V_i^*, in general difficult to solve. However, if the fixed points V_i^* are stable then we can use the dynamics (8.3) to automatically pick out the steady-state solution $\boldsymbol{V}^*$. This solution depends on the pattern $\boldsymbol{x}^{(\mu)}$, but in Equations (8.5) and (8.6) and also in the following I have left out the superscript (μ).

The goal is to find weights so that the outputs give the correct target values in the steady state, those associated with $\boldsymbol{x}^{(\mu)}$. To this end we use stochastic gradient descent on the energy function (8.4). Consider first how to update the weights $w_{ij}^{(vv)}$. We must evaluate

$$\delta w_{mn}^{(vv)} = -\eta \frac{\partial H}{\partial w_{mn}^{(vv)}} = \eta \sum_k E_k^* \frac{\partial V_k^*}{\partial w_{mn}^{(vv)}}\,. \tag{8.7}$$

To calculate the gradients of $\boldsymbol{V}^*$ we use Equation (8.6):

$$\frac{\partial V_i^*}{\partial w_{mn}^{(vv)}} = g'(b_i^*)\frac{\partial b_i^*}{\partial w_{mn}^{(vv)}} = g'(b_i^*)\Big(\delta_{im} V_n^* + \sum_j w_{ij}^{(vv)} \frac{\partial V_j^*}{\partial w_{mn}^{(vv)}}\Big), \tag{8.8}$$

where $b_i^* = \sum_j w_{ij}^{(vv)} V_j^* + \sum_k w_{ik}^{(vx)} x_k - \theta_i^{(v)}$. Equation (8.8) is a self-consistent equation for the gradient, as opposed to the explicit equations we found in Chapters 5 to 7. The reason for the difference is that the recurrent network has feedbacks. Since Equation (8.8) is linear in the gradients, we can solve it by matrix inversion. To this end, define the matrix $\mathbb{L}$ with elements $L_{ij} = \delta_{ij} - g'(b_i^*) w_{ij}^{(vv)}$. Equation (8.8) can be written as

$$\sum_j L_{ij} \frac{\partial V_j^*}{\partial w_{mn}^{(vv)}} = \delta_{im} g'(b_i^*) V_n^*\,. \tag{8.9}$$

Applying $\sum_k \big(\mathbb{L}^{-1}\big)_{ki}$ to both sides we find

$$\frac{\partial V_j^*}{\partial w_{mn}^{(vv)}} = \big(\mathbb{L}^{-1}\big)_{jm} g'(b_m^*) V_n^*\,. \tag{8.10}$$

Inserting this result into (8.7) we finally obtain for the weight increments:

$$\delta w_{mn}^{(vv)} = \eta \sum_k E_k^* \left(\mathbb{L}^{-1}\right)_{km} g'(b_m^*) V_n^* . \tag{8.11}$$

This learning rule can be written in the form of the backpropagation rule by introducing the error

$$\Delta_m^* = g'(b_m^*) \sum_k E_k^* \left(\mathbb{L}^{-1}\right)_{km} . \tag{8.12}$$

Then the learning rule (8.11) takes the form

$$\delta w_{mn}^{(vv)} = \eta \Delta_m^* V_n^* , \tag{8.13}$$

compare Equation (6.11). A problem is that a matrix inversion is required to compute the errors Δ_m^*, an expensive operation. But as outlined in Chapter 5, we can try find the inverse iteratively. We can find a linear differential equation for

$$\Delta_i^* = g'(b_i^*) \sum_k E_k^* \left(\mathbb{L}^{-1}\right)_{ki} . \tag{8.14}$$

that does not involve the inverse of $\mathbb{L}$. I used the index i here because it makes the following calculation a bit easier to follow. The first step is to multiply both sides of Equation (8.14) with $L_{ij}/g'(b_i^*)$ and to sum over i: This gives

$$\sum_i \Delta_i^* L_{ij}/g'(b_i^*) = \sum_{ki} E_k^* \left(\mathbb{L}^{-1}\right)_{ki} L_{ij} = E_j^* . \tag{8.15}$$

Using $L_{ij}/g'(b_i^*) = \delta_{ij}/g'(b_j^*) - w_{ij}^{(vv)}$ we find

$$\sum_i \Delta_i^* \left(\delta_{ij} - w_{ij}^{(vv)} g'(b_j^*)\right) = g'(b_j^*) E_j^* . \tag{8.16}$$

The trick is now to write down a dynamical equation for Δ_i that has a steady state at the solution of Equation (8.16):

$$\tau \tfrac{\mathrm{d}}{\mathrm{d}t} \Delta_j = -\Delta_j + \sum_i \Delta_i w_{ij}^{(vv)} g'(b_j^*) + g'(b_j^*) E_j^* . \tag{8.17}$$

Compare this with the dynamical rule (8.3). Equations (8.3) and (8.17) exhibit the same *duality* as Algorithm 2, between forward propagation of states of neurons (step 5) and backpropagation of errors (step 9). The sum in Equation (8.17) has the same form as the recursion for the errors in Algorithm 2 (step 9), except that there are no layer indices ℓ here.

The solution of (8.16) is a fixed point of this Equation. But is it stable? To decide this we linearise the dynamical equations (8.3) and (8.17). To this end we write

$$V_i(t) = V_i^* + \delta V_i(t) \quad \text{and} \quad \Delta_i(t) = \Delta_i^* + \delta\Delta_i(t), \tag{8.18}$$

and insert this *ansatz* into (8.3) and (8.17). To leading order we find:

$$\tau \tfrac{\mathrm{d}}{\mathrm{d}t}\delta V_i = -\delta V_i + g'(b_i^*)\sum_j w_{ij}^{(vv)}\delta V_j = -\sum_j L_{ij}\delta V_j\,, \tag{8.19}$$

$$\tau \tfrac{\mathrm{d}}{\mathrm{d}t}\delta\Delta_j = -\delta\Delta_j + \sum_i \delta\Delta_i\, w_{ij}^{(vv)} g'(b_j^*) = -\sum_i \delta\Delta_i g'(b_i^*)L_{ij}/g'(b_j^*). \tag{8.20}$$

Since the matrices with elements L_{ij} and $g'(b_i^*)L_{ij}/g'(b_j^*)$ have the same eigenvalues, Δ_i^* is a stable fixed point of (8.17) if V_n^* is a stable fixed point of (8.3). This was assumed in the beginning, Equation (8.5). If this assumption does not hold, the algorithm does not converge.

Now consider the update formula for the weights $w_{mn}^{(vx)}$ from the inputs:

$$\delta w_{mn}^{(vx)} = -\eta\frac{\partial H}{\partial w_{mn}^{(vx)}} = \eta\sum_k E_k^*\frac{\partial V_k^*}{\partial w_{mn}^{(vx)}}\,, \tag{8.21a}$$

$$\frac{\partial V_i^*}{\partial w_{mn}^{(vx)}} = g'(b_i^*)\Big(\delta_{ij}x_n + \sum_j w_{ij}^{(vv)}\frac{\partial V_j^*}{\partial w_{mn}^{(vx)}}\Big). \tag{8.21b}$$

Equation (8.21) is analogous to Equation (8.8). Consequently

$$\delta w_{mn}^{(vx)} = \eta\Delta_m^* x_n\,. \tag{8.22}$$

The algorithm for recurrent backpropagation is summarised in Algorithm 9.

Algorithm 9 recurrent backpropagation

1: initialise all weights;
2: **for** $t = 1,\ldots,T$ **do**
3: choose a value of μ and apply $\boldsymbol{x}^{(\mu)}$ to the inputs;
4: find V_n^* by relaxing $\tau\frac{\mathrm{d}V_n}{\mathrm{d}t} = -V_n + g\big(\sum_j w_{nj}^{(vv)}V_j + \sum_k w_{nk}^{(vx)}x_k - \theta_n^{(v)}\big)$;
5: compute $E_k^* = y_k - V_k^*$ for all output units;
6: find Δ_m^* by relaxing $\tau\frac{\mathrm{d}\Delta_m}{\mathrm{d}t} = -\Delta_m + \sum_j \Delta_j w_{jm}g'(b_m^*) + g'(b_m^*)E_m^*$;
7: update all weights: $w_{mn}^{(vv)} \leftarrow w_{mn} + \delta w_{mn}^{(vv)}$ with $\delta w_{mn}^{(vv)} = \eta\Delta_m^* V_n^*$ and $w_{mn}^{(vx)} \leftarrow w_{mn}^{(vx)} + \delta w_{mn}^{(vx)}$ with $\delta w_{mn}^{(vx)} = \eta\Delta_m^* x_n$;
8: **end for**
9: end;

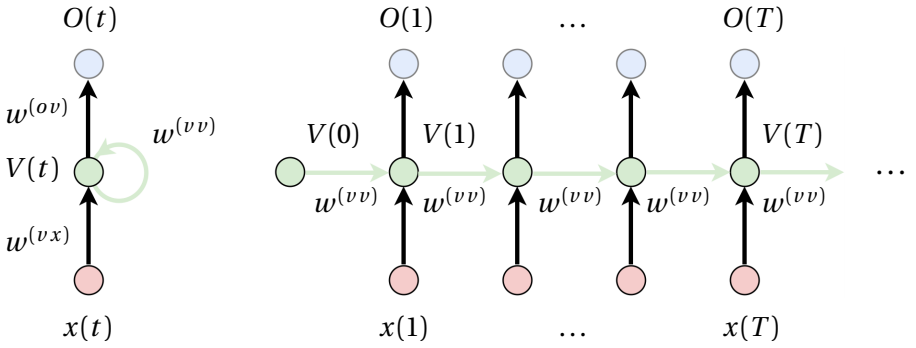


Figure 8.2: Left: recurrent network with one hidden neuron (green) and one output neuron (blue). The input terminal is drawn red. Right: same network but unfolded in time. The weights $w^{(vv)}$ remain unchanged as drawn, also the weights $w^{(vx)}$ and $w^{(ov)}$ remain unchanged (not drawn).

8.2 Backpropagation through time

Recurrent networks can be used to learn sequential inputs, as in speech recognition and machine translation. The training set is a time sequence of inputs and targets $[\boldsymbol{x}(t), \boldsymbol{y}(t)]$. The network is trained on the sequence and learns to predict the targets. In this context the layout is changed a little bit compared with the one described in the previous Section. There are two main differences. Firstly, the inputs and targets depend on t and one uses a discrete-time update rule. Secondly, separate output units $O_i(t)$ are added to the layout. The update rule takes the form

$$V_i(t) = g\Big(\sum_j w_{ij}^{(vv)} V_j(t-1) + \sum_k w_{ik}^{(vx)} x_k(t) - \theta_i^{(v)}\Big), \tag{8.23a}$$

$$O_i(t) = g\Big(\sum_j w_{ij}^{(ov)} V_j(t) - \theta_i^{(o)}\Big). \tag{8.23b}$$

The activation function of the outputs O_i can be different from that of the hidden neurons V_j. Often the softmax function is used for the outputs [97, 98].

To train recurrent networks with time-dependent inputs and targets and with the dynamics (8.23) one uses *backpropagation through time*. The idea is to unfold the network in time to get rid of the feedbacks, at the expense of as many copies of the original neurons as there are time steps. This is illustrated in Figure 8.2 for a recurrent network with one hidden neuron, one input, and one output. The unfolded network has T inputs and outputs. It can be trained in the usual way with *stochastic gradient descent*. The errors are calculated using backpropagation as in Algorithm 2, but here the error is propagated back in time, not from layer to layer.

The energy function is the squared error summed over all time steps

$$H = \frac{1}{2}\sum_{t=1}^{T} E_t^2 \quad \text{with} \quad E_t = y_t - O_t\,. \tag{8.24}$$

Since there is only one hidden neuron, and since the inputs and outputs are also one-dimensional, it is simpler to write the time argument as a subscript, O_t instead of $O(t)$, and so forth. Note also that one could use the negative log-likelihood function (7.25) instead of Equation (8.24).

Consider first how to update the weight $w^{(vv)}$. The gradient-descent rule (5.24) gives a result that is of the same form as Equation (8.7):

$$\delta w^{(vv)} = \eta \sum_{t=1}^{T} E_t \frac{\partial O_t}{\partial w^{(vv)}} = \eta \sum_{t=1}^{T} \Delta_t w^{(ov)} \frac{\partial V_t}{\partial w^{(vv)}}\,. \tag{8.25}$$

Here $\Delta_t = E_t g'(B_t)$ is an output error, $B_t = w^{(ov)} V_t - \theta^{(o)}$ is the local field of the output neuron at time t [Equation (8.23)], and $\partial V_t / \partial w^{(vv)}$ is evaluated with the chain rule, as usual. Equation (8.23a) yields the recursion

$$\frac{\partial V_t}{\partial w^{(vv)}} = g'(b_t)\Big(V_{t-1} + w^{(vv)} \frac{\partial V_{t-1}}{\partial w^{(vv)}}\Big) \tag{8.26}$$

for $t \geq 1$. Since $\partial V_0 / \partial w^{(vv)} = 0$ we have:

$$\begin{aligned}
\frac{\partial V_1}{\partial w^{(vv)}} &= g'(b_1) V_0\,, \\
\frac{\partial V_2}{\partial w^{(vv)}} &= g'(b_2) V_1 + g'(b_2) w^{(vv)} g'(b_1) V_0\,, \\
\frac{\partial V_3}{\partial w^{(vv)}} &= g'(b_3) V_2 + g'(b_3) w^{(vv)} g'(b_2) V_1 + g'(b_3) w^{(vv)} g'(b_2) w^{(vv)} g'(b_1) V_0 \\
&\vdots \\
\frac{\partial V_{T-1}}{\partial w^{(vv)}} &= g'(b_{T-1}) V_{T-2} + g'(b_{T-1}) w^{(vv)} g'(b_{T-2}) V_{T-3} + \ldots \\
\frac{\partial V_T}{\partial w^{(vv)}} &= g'(b_T) V_{T-1} + g'(b_T) w^{(vv)} g'(b_{T-1}) V_{T-2} + \ldots
\end{aligned}$$

Equation (8.25) says that we must sum over t. Regrouping the terms in this sum

yields:

$$
\begin{aligned}
&\Delta_1 \frac{\partial V_1}{\partial w^{(vv)}} + \Delta_2 \frac{\partial V_2}{\partial w^{(vv)}} + \Delta_3 \frac{\partial V_3}{\partial w^{(vv)}} + \ldots \\
&= [\Delta_1 g'(b_1) + \Delta_2 g'(b_2) w^{(vv)} g'(b_1) + \Delta_3 g'(b_3) w^{(vv)} g'(b_2) w^{(vv)} g'(b_1) + \ldots] V_0 \\
&+ [\Delta_2 g'(b_2) + \Delta_3 g'(b_3) w^{(vv)} g'(b_2) + \Delta_4 g'(b_4) w^{(vv)} g'(b_3) w^{(vv)} g'(b_2) + \ldots] V_1 \\
&+ [\Delta_3 g'(b_3) + \Delta_4 g'(b_4) w^{(vv)} g'(b_3) + \Delta_5 g'(b_5) w^{(vv)} g'(b_4) w^{(vv)} g'(b_3) + \ldots] V_2 \\
&\vdots \\
&+ [\Delta_{T-1} g'(b_{T-1}) + \Delta_T g'(b_T) w^{(vv)} g'(b_{T-1})] V_{T-2} \\
&+ [\Delta_T g'(b_T)] V_{T-1} .
\end{aligned}
$$

To write the learning rule in the usual form, we define *errors* δ_t recursively:

$$
\delta_t = \begin{cases} \Delta_T w^{(ov)} g'(b_T) & \text{for } t = T, \\ \Delta_t w^{(ov)} g'(b_t) + \delta_{t+1} w^{(vv)} g'(b_t) & \text{for } 0 < t < T. \end{cases} \tag{8.27}
$$

Then the learning rule takes the form

$$
\delta w^{(vv)} = \eta \sum_{t=1}^{T} \delta_t V_{t-1} , \tag{8.28}
$$

just like Equation (6.10), or like the recursion in step 9 of Algorithm 2.

The factor $w^{(vv)} g'(b_{t-1})$ in the recursion (8.27) gives rise to a product of many such factors in δ_t when T is large, exactly as described in Section 7.2.1 for multilayer perceptrons. This means that the training of recurrent nets suffers from *unstable gradients*, as backpropagation of multilayer perceptrons does (Section 7.2.1). If the factors $|w^{(vv)} g'(b_p)|$ are smaller than unity then the errors δ_t become very small when t becomes small (*vanishing-gradient problem*). This means that the early states of the hidden neuron no longer contribute to the learning, causing the network to forget what it has learned about early inputs. When $|w^{(vv)} g'(b_p)| > 1$, on the other hand, exploding gradients make learning impossible. In summary, the *unstable gradients* in recurrent neural networks occurs much in the same way as in multilayer perceptrons (Section 7.2.1). The resulting difficulties for training recurrent neural networks are discussed in more detail in Ref. [99].

A slight variation of the above algorithm (*truncated backpropagation through time*) suffers less from the exploding-gradient problem. The idea is that the exploding gradients are tamed by truncating the memory. This is achieved by limiting the error propagation backwards in time, errors are computed back to $T - \tau$ and not further, where τ is the truncation time [2]. Naturally this implies that long-time correlations cannot be learnt.

Finally, the update formulae for the weights $w^{(vx)}$ are obtained in a similar fashion. Equation (8.23a) yields the recursion

$$\frac{\partial V_t}{\partial w^{(vx)}} = g'(b_t)\Big(x_t + w^{(vv)}\frac{\partial V_{t-1}}{\partial w^{(vx)}}\Big). \tag{8.29}$$

This looks just like Equation (8.26), except that V_{t-1} is replaced by x_t. As a consequence we have

$$\delta w^{(vx)} = \eta \sum_{t=1}^{T} \delta_t x_t \,. \tag{8.30}$$

The update formula for $w^{(ov)}$ is simpler to derive. From Equation (8.23b) we find by differentiation w.r.t. $w^{(ov)}$:

$$\delta w^{(ov)} = \eta \sum_{t=1}^{T} E_t g'(B_t) V_t \,. \tag{8.31}$$

How are the thresholds updated? Going through the above derivation we see that we must replace V_{t-1} and x_t in Equations (8.28) and (8.30) by -1. It works in the same way for the output threshold.

In order to keep the formulae simple, I only described the algorithm for a single hidden and a single output neuron, so that I could leave out the indices referring to different hidden neurons and/or different output components. You can add those indices yourself, the structure of the Equations remains exactly the same, save for a number of extra sums over those indices:

$$\delta w_{mn}^{(vv)} = \eta \sum_{t=1}^{T} \delta_m^{(t)} V_n^{(t-1)} \tag{8.32}$$

$$\delta_j^{(t)} = \begin{cases} \sum_i \Delta_i^{(t)} w_{ij}^{(ov)} g'(b_j^{(t)}) & \text{for } t = T, \\ \sum_i \Delta_i^{(t)} w_{ij}^{(ov)} g'(b_j^{(t)}) + \sum_i \delta_i^{(t+1)} w_{ij}^{(vv)} g'(b_j^{(t)}) & \text{for } 0 < t < T. \end{cases}$$

The second term in the recursion for $\delta_j^{(t)}$ is analogous to the recursion in step 9 of Algorithm 2. The time index t here plays the role of the layer index ℓ in Algorithm 2. A difference is that the weights in Equation (8.32) are the same for all time steps.

In summary you see that backpropagation through time for recurrent networks is similar to backpropagation for multilayer perceptrons. After the recurrent network is unfolded to get rid of the feedback connections it can be trained by backpropagation. The time index t takes the role of the layer index ℓ. Backpropagation through time is the standard approach for training recurrent nets, despite the fact that it suffers from the vanishing-gradient problem. The next Section describes how improvements to the layout make it possible to efficiently train recurrent networks.

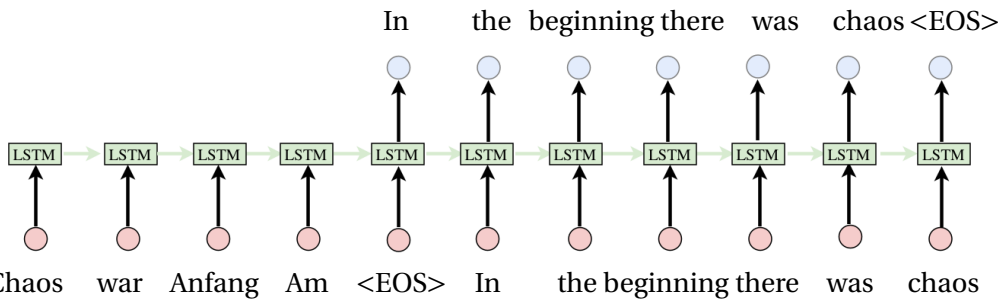


Figure 8.3: Schematic illustration of unfolded recurrent network for machine translation, after Refs. [97, 98]. The green rectangular boxes represent the hidden states in the form of long short-term memory (LSTM) units, see Section 8.3. Otherwise the network layout is like the one shown in Figure 8.2. Sutskever *et al.* [97] found that the network translates much better if the sentence is read in reverse order, from the end. The tag <EOS> denotes the end-of-sentence tag. Here it denotes the beginning of the sentence.

8.3 Long short-term memory

Hochreiter and Schmidhuber [100] suggested to replace the hidden neurons of the recurrent network with computation units that are specially designed to eliminate the vanishing-gradient problem. The method is referred to as *long short-term memory* (LSTM). The basic ingredient is the same as in *residual networks* (Section 7.6): short cuts reduce the vanishing-gradient problem. For our purposes we can think of LSTMs as units that replace the hidden neurons.

8.4 Recurrent networks for machine translation

Recurrent networks are used for machine translation [98]. How does this work? The networks are trained using backpropagation through time. The vanishing-gradient problem is dealt with by using LSTMs (Section 8.3).

How are the network inputs and outputs represented? For machine translation one must represent words in the dictionary in terms of a code. The simplest code is a binary code where 100... represents the first word in the dictionary, 010... the second word, and so forth. Each input is a vector with as many components as there are words in the dictionary. A sentence corresponds to a sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T$. Each sentence ends with an end-of-sentence tag, <EOS>. Softmax outputs give the probability $p(\boldsymbol{O}_1, \ldots, \boldsymbol{O}_{T'} | \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ of an output sequence conditional on the input sequence. The translated sentence is the one with the highest probability (it also contains the end-of-sentence tag <EOS>). So both inputs and outputs are represented by high-dimensional vectors $\boldsymbol{x}_t$ and $\boldsymbol{O}_t$. Other encoding schemes are

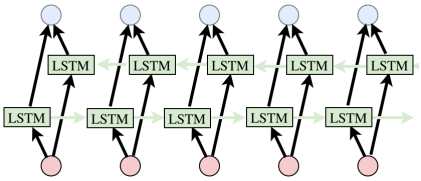


Figure 8.4: Schematic illustration of a bidirectional recurrent network . The net consists of two hidden states that are unfolded in different ways. The hidden states are represented by LSTMs.

described in Ref. [98].

What is the role of the hidden states, represented in terms of an LSTM? The network encodes the input sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T$ in these states. Upon encountering the <EOS> tag in the input sequence, the network outputs the first word of the translated sentence using the information about the input sequence stored in $\boldsymbol{V}_T$ as shown in Figure 8.3. The first output is fed into the next input, and the network continues to translate until it produces an <EOS> tag for the output sequence. In short, the network calculates the probabilities

$$p(\boldsymbol{O}_1, \ldots, \boldsymbol{O}_{T'} | \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T) = \prod_{t=1}^{T'} p(\boldsymbol{O}_t | \boldsymbol{O}_1, \ldots, \boldsymbol{O}_{t-1}; \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T), \tag{8.33}$$

where $p(O_t|O_1, \ldots, O_{t-1}; \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ is the probability of the next word in the output sequence give the inputs and the output sequence up to O_{t-1} [7].

There is a large number of recent papers on machine translation with recurrent neural nets. Most studies are based on the training algorithm described in Section 8.2, backpropagation through time. The different approaches mainly differ in their network layouts. Google's machine translation system uses a deep network with layers of LSTMs [7]. Different hidden states are unfolded forward as well as backwards in time, as illustrated in Figure 8.4. In this Figure the hidden states are represented by LSTMs. In the simplest case the hidden states are just encoded in hidden neurons, as in Figure 8.2 and Equation (8.23). If we represent the hidden states by neurons, as in Section 8.2, then the corresponding bidirectional network has the dynamics

$$\begin{aligned}
V_i(t) &= g\Big(\sum_j w_{ij}^{(vv)} V_j(t-1) + \sum_k w_{ik}^{(vx)} x_k(t) - \theta_i^{(v)}\Big), \\
U_i(t) &= g\Big(\sum_j w_{ij}^{(uu)} U_j(t+1) + \sum_k w_{ik}^{(ux)} x_k(t) - \theta_i^{(u)}\Big), \\
O_i(t) &= g\Big(\sum_j w_{ij}^{(ov)} V_j(t) + \sum_j w_{ij}^{(ou)} U_j(t) - \theta_i^{(o)}\Big).
\end{aligned} \tag{8.34}$$

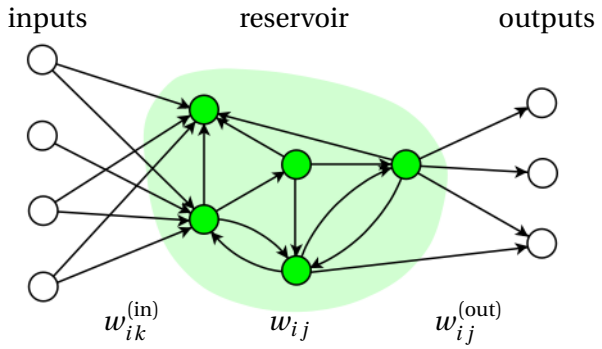


Figure 8.5: Reservoir computing (schematic). Not all connections are drawn. There can be connections from all inputs to all neurons in the reservoir (green), and from all reservoir neurons to all output neurons.

It is natural to use bidirectional nets for machine translation because correlations go either way in a sentence, forward and backwards. In German, for example, the finite verb form is usually at the end of the sentence.

Different schemes for scoring the accuracy of a translation are described by Lipton *et al.* [98]. One difficulty is that there are often several different valid translations of a given sentence, and the score must compare the machine translation with all of them. Recent papers on machine translation usually use the so-called BLEU score to evaluate the translation accuracy. The acronym stands for *bilingual evaluation understudy*. The scheme was proposed by Papieni *et al.* [101], and it is commonly judged to score not too differently from how Humans would score.

8.5 Reservoir computing

Three sets of weights $w_{ij}^{(\mathrm{in})}$, $w_{ij}^{(\mathrm{out})}$, and internal weights w_{ij}. Inputs $\boldsymbol{x}(t)$, output neurons $\boldsymbol{y}(t)$. Reservoir neurons $\boldsymbol{r}(t)$. Standard formulation similar to the update rule (8.23) for recurrent nets:

$$r_i(t) = g\Big(\sum_j w_{ij} r_j(t-1) + \sum_k w_{ik}^{(\mathrm{in})} x_k(t)\Big), \tag{8.35}$$

$$y_i(t) = \sum_j w_{ij}^{(\mathrm{out})} r_j(t). \tag{8.36}$$

Reservoir computing is different from the training algorithms for recurrent neural nets described in the previous Sections. Train only the output weights $w_{ij}^{(\mathrm{out})}$. What is the purpose of the reservoir? The idea is that the reservoir represents higher-order correlations in the input data [102], just like the hidden neurons of a Boltzmann machine (Chapter **??**). Reviews [102–104]

Lyapunov exponent (Echo-state networks).

Prediction of spatio-temporally chaotic dynamics [105]. Predicting rare events arxiv:1908.03771.

8.6 Summary

It is sometimes said that recurrent networks learn *dynamical systems* while multilayer perceptrons learn *input-output maps*. This notion refers to backpropagation in time. I would emphasise, by contrast, that both networks are trained in similar ways, by backpropagation. Neither is it given that the tasks must differ: recurrent networks are also used to learn time-independent data. It is true though that tools from *dynamical-systems theory* have been used with success to analyse the dynamics of recurrent networks [99, 106].

Recurrent neural networks are trained by stochastic gradient descent after unfolding the network in time to get rid of feedback connections. This algorithm suffers from the vanishing-gradient problem. To overcome this difficulty, the hidden states in the recurrent network are usually represented by LSTMs. Recent layouts for machine translation use deep bidirectional networks with layers of LSTMs.

8.7 Further reading

The training of recurrent networks is discussed in Chapter 15 of Ref. [2]. Recurrent backpropagation is described by Hertz, Krogh and Palmer [1], for a slightly different network layout. For a recent review of recurrent neural nets, see Ref. [98]. This webpage [107] gives a very enthusiastic overview about what recurrent nets can do. A more pessimistic view is expressed in this blog.

8.8 Exercises

8.1 Recurrent backpropagation. Show that recurrent backpropagation is a special case of the backpropagation algorithm for layered feed-forward networks.

8.2 Learning rules for backpropagation through time. Derive the learning rules (8.30) and (8.31) from Equation (8.23).

8.3 Recurrent network. Figure 8.6 shows a simple recurrent network with one hidden neuron $V(t)$, one input $x(t)$ and one output $O(t)$. The network learns a time series of input-output pairs $[x(t), y(t)]$ for $t = 1, 2, 3, \ldots, T$. Here t is a discrete time

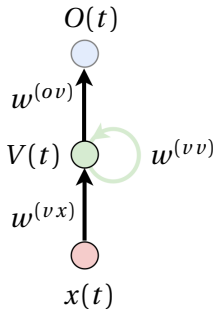


Figure 8.6: Recurrent network with one input unit $x(t)$ (red), one hidden neuron $V(t)$ (green) and one output neuron $O(t)$ (blue). Exercise 8.3.

index and $y(t)$ is the target value at time t (the targets are denoted by y to avoid confusion with the time index t). The hidden unit is initialised to a value $V(0)$ at $t=0$. This network can be trained by backpropgation by *unfolding it in time*.
(a) Draw the unfolded network, label the connections using the labels shown in Figure 8.6, and discuss the layout (max half an A4 page).
(b) Write down the dynamical rules for this network, the rules that determine $V(t)$ in terms of $V(t-1)$ and $x(t)$, and $O(t)$ | in terms of $V(t)$. Assume that both $V(t)$ and $O(t)$ have the same activation function $g(b)$.
(c) Derive the update rule for $w^{(ov)}$ for gradient descent on the energy function

$$H=\frac{1}{2}\sum_{t=1}^{T}E(t)^2 \quad \text{where } E(t)=y(t)-O(t). \tag{8.37}$$

Denote the learning rate by η. *Hint:* the update rule for $w^{(ov)}$ is much simpler to derive than those for $w^{(vx)}$ and $w^{(vv)}$. (**1**p).

(d) Explain how recurrent networks are used for machine translation. Draw the layout, describe how the inputs are encoded. How is the *unstable-gradient problem* overcome? (Max one A4 page).

8.4 Backpropagation through time. A recurrent network with two hidden neurons is shown in Figure 8.7. Write down the dynamical rules for this network. Assume that all neurons have the same activation function $g(b)$. Draw the unfolded network. Derive the update rules for the weights.

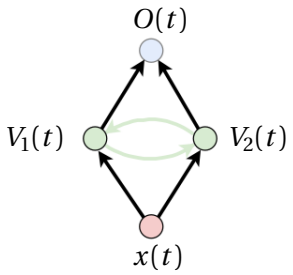


Figure 8.7: . Recurrent network used in Exercise 8.4.

PART III
UNSUPERVISED LEARNING

Figure 9.1: Supervised learning finds decision boundaries (left). Unsupervised learning can find clusters in the input data (right).

Chapters 5, 6, 7, and 8 describe supervised learning where the networks are trained to produce the correct outputs. The remaining Chapters discuss *unsupervised learning*. In this case there is no feedback telling the network whether it has learnt correctly or not. Instead the network organises the input data in relevant ways. This requires *redundancy* in the input data. Possible tasks are to determine the *familiarity* of input patterns, or to find *clusters* (Figure 9.1) in high-dimensional input data. A further application is to determine spatial maps of spatially distributed inputs, so that nearby inputs activate nearby output units. Such learning algorithms are explained in Chapter 9. It is sometimes said that unsupervised learning corresponds to learning without a teacher, which could be taken to mean that the network itself discovers suitable ways of organising the input data. This is not entirely accurate, as we shall see. Unsupervised nets operate with a pre-determined learning rule, instead of adapting the weights to minimise the difference between actual and target outputs. Chapter 10 introduces radial-basis function networks, they learn using a hybrid algorithm with supervised and unsupervised learning. A different hybrid algorithm is discussed in Chapter 11, *reinforcement learning*. Here the idea is that the network receives only partial feedback on its performance, it cannot access the full set of target values. For instance, the feedback may just be +1 (good solution) or −1 (not so good). In this case the network can learn by building up its own training set from its outputs with +1 feedback.

9 Unsupervised Hebbian learning

The primary source for the material in this Chapter is the book by Hertz, Krogh, and Palmer [1], Chapters 8 and 9. The simplest example for unsupervised learning is given by a distribution $P_{\text{data}}(\boldsymbol{x})$ of input patterns

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \tag{9.1}$$

with continuous-valued components x_i. Patterns are drawn from this distribution and fed one after another to the network shown in Figure 9.2. It has one linear

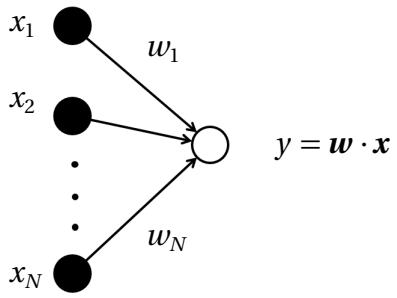


Figure 9.2: Network for unsupervised Hebbian learning, with a single linear output unit that has weight vector $\boldsymbol{w}$. The network output is denoted by y in this Chapter.

output unit $y = \boldsymbol{w} \cdot \boldsymbol{x}$ with weight vector

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_N \end{bmatrix} . \tag{9.2}$$

In this Chapter we follow a common convention and denote the output of unsupervised-learning algorithms by y.

The network can detect how *familiar* certain input patterns are. The idea is that the output is the larger the more frequently the input pattern occurs in $P_{\text{data}}(\boldsymbol{x})$. This learning goal is achieved by Hebb's rule:

$$\boldsymbol{w}' = \boldsymbol{w} + \delta\boldsymbol{w} \quad \text{with} \quad \delta\boldsymbol{w} = \eta\, y\, \boldsymbol{x}\,, \tag{9.3}$$

where $y = \boldsymbol{w} \cdot \boldsymbol{x}$ is the output. The rule (9.3) is also called *Hebbian unsupervised learning*. As usual, $\eta > 0$ is small learning rate. How does this learning rule work? Since we keep adding multiples of the pattern vectors $\boldsymbol{x}$ to the weights, the magnitude of the output $|y|$ becomes the larger the more often the input pattern occurs in the distribution $P_{\text{data}}(\boldsymbol{x})$. So the most familiar pattern produces the largest output.

A problem is that the weight vector continues to grow as we keep on adding increments. This means that the simple Hebbian learning rule (9.3) does not converge to a steady state. To achieve definite learning outcomes we require the network to approach a steady state. Therefore the learning rule (9.3) must be modified. One possibility is to introduce weight decay as described in Section 6.3.3. This is discussed in the next Section.

Algorithm 10 Oja's rule

1: initialise weights randomly;
2: **for** $t = 1, \ldots, T$ **do**
3: draw an input pattern $\boldsymbol{x}$ from $P_{\text{data}}(\boldsymbol{x})$ and apply it to the network;
4: update all weights using $\delta \boldsymbol{w} = \eta\, y(\boldsymbol{x} - y\,\boldsymbol{w})$;
5: **end for**
6: end;

9.1 Oja's rule

Adding a weight-decay term with coefficient proportional to y^2 to Equation (9.3) ensures that the weights remain normalised. The update rule with weight decay reads:

$$\delta \boldsymbol{w} = \eta\, y(\boldsymbol{x} - y\,\boldsymbol{w}). \tag{9.4}$$

Using that the output is $y = \boldsymbol{w} \cdot \boldsymbol{x} = \boldsymbol{w}^{\mathsf{T}} \boldsymbol{x} = \boldsymbol{x}^{\mathsf{T}} \boldsymbol{w}$, we can write Eq. (9.4) as

$$\delta \boldsymbol{w} = \eta \left\{ \boldsymbol{x}\boldsymbol{x}^{\mathsf{T}} \boldsymbol{w} - [\boldsymbol{w} \cdot (\boldsymbol{x}\boldsymbol{x}^{\mathsf{T}})\boldsymbol{w}]\boldsymbol{w} \right\}. \tag{9.5}$$

To see why Equation (9.5) does the trick, consider an analogy: a vector $\boldsymbol{q}$ that obeys the differential equation

$$\tfrac{\mathrm{d}}{\mathrm{d}t} \boldsymbol{q} = \mathbb{A}(t)\boldsymbol{q}. \tag{9.6}$$

For a general matrix $\mathbb{A}(t)$, the norm $|\boldsymbol{q}|$ may increase or decrease. We can ensure that $\boldsymbol{q}$ remains normalised by adding a term to Equation (9.6):

$$\tfrac{\mathrm{d}}{\mathrm{d}t} \boldsymbol{w} = \mathbb{A}(t)\boldsymbol{w} - [\boldsymbol{w} \cdot \mathbb{A}(t)\boldsymbol{w}]\boldsymbol{w}\,. \tag{9.7}$$

The vector $\boldsymbol{w}$ turns in the same way as $\boldsymbol{q}$, and if we set $|\boldsymbol{w}| = 1$ initially, then $\boldsymbol{w}$ remains normalised ($\boldsymbol{w} = \boldsymbol{q}/|\boldsymbol{q}|$), see Exercise 9.1. Equation (9.7) describes the dynamics of the normalised orientation vector of a small rod in turbulence [108], where $\mathbb{A}(t)$ is the matrix of fluid-velocity gradients.

But let us return to Equation (9.4). It is called *Oja's rule* [109]. Oja's learning algorithm is summarised in Algorithm 10. One draws a pattern $\boldsymbol{x}$ from the distribution $P_{\text{data}}(\boldsymbol{x})$ of input patterns, applies it to the network, and updates the weights as prescribed in Equation (9.4). This is repeated many times. In the following we denote the average over T input patterns as $\langle \cdots \rangle = \frac{1}{T} \sum_{t=1}^{T} \cdots$.

The idea behind Algorithm 10 is that the algorithm converges to a steady state $\boldsymbol{w}^*$. For zero-mean input data, $\boldsymbol{w}^*$ corresponds to the principal component of the input data, as illustrated in Figure 9.3. To show how this works we start from the steady-state condition

$$0 = \langle \delta \boldsymbol{w} \rangle_{\boldsymbol{w}^*}. \tag{9.8}$$

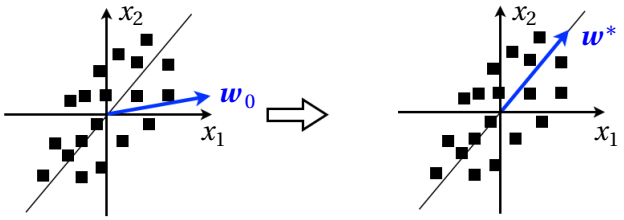


Figure 9.3: Oja's rule finds the principal component of zero-mean data (schematic). The initial weight vector is $\boldsymbol{w}_0$.

Here $\langle\cdots\rangle_{\boldsymbol{w}^*}$ is an average at fixed $\boldsymbol{w}^*$ (the presumed steady state). So $\boldsymbol{w}^*$ is not averaged over, just $\boldsymbol{x}$. Equation (9.8) says that the increments $\delta\boldsymbol{w}$ must average to zero in the steady state, because the weights would otherwise either grow or decrease. Equation (9.8) is a condition upon $\boldsymbol{w}^*$. Using the learning rule (9.4), Equation (9.8) implies

$$0 = \mathbb{C}'\boldsymbol{w}^* - (\boldsymbol{w}^* \cdot \mathbb{C}'\boldsymbol{w}^*)\boldsymbol{w}^* \quad \text{with} \quad \mathbb{C}' = \langle \boldsymbol{x}\boldsymbol{x}^\mathsf{T} \rangle . \tag{9.9}$$

For zero-mean input data, $\mathbb{C}'$ equals the data-covariance matrix, Equation (6.22), but not otherwise.

It follows from Equation (9.9) that $\boldsymbol{w}^*$ must obey $\mathbb{C}'\boldsymbol{w}^* = \lambda\boldsymbol{w}^*$. In other words, $\boldsymbol{w}^*$ must be an eigenvector of the matrix $\mathbb{C}'$. We denote the eigenvalues and eigenvectors of $\mathbb{C}'$ by λ_α and $\boldsymbol{u}_\alpha$. Since $\mathbb{C}'$ is symmetric, its eigenvalues are real and the eigenvectors can be chosen orthonormal, $\boldsymbol{u}_\alpha \cdot \boldsymbol{u}_\beta = \delta_{\alpha\beta}$, and they form a basis. Moreover, $\mathbb{C}'$ is positive semidefinite. This means that the eigenvalues cannot be negative. It also follows from Equation (9.9) that $|\boldsymbol{w}^*| = 1$: since $\boldsymbol{w}^*$ is an eigenvector of $\mathbb{C}'$, the first term on the r.h.s. of Equation (9.9) evaluates to $\lambda\boldsymbol{w}^*$, whereas the second term gives $-\lambda|\boldsymbol{w}^*|^2\boldsymbol{w}^*$. For these terms to cancel, we must have $|\boldsymbol{w}^*|^2$. In conclusion, $\boldsymbol{w}^*$ is a normalised eigenvector of $\mathbb{C}'$. But which one?

Only the eigenvector corresponding to the maximal eigenvalue λ_1 represents a stable steady state. To demonstrate this, we investigate the linear stability of $\boldsymbol{w}^*$ (in the same way as in Section 8.1). We use the *ansatz*:

$$\boldsymbol{w} = \boldsymbol{w}^* + \boldsymbol{\epsilon} \tag{9.10}$$

where $\boldsymbol{\epsilon}$ is a small initial displacement from $\boldsymbol{w}^*$. Now we determine the average change of $\boldsymbol{\epsilon}$ after one iteration step:

$$\langle \delta\boldsymbol{\epsilon} \rangle = \langle \delta\boldsymbol{w} \rangle_{\boldsymbol{w}^*+\boldsymbol{\epsilon}} . \tag{9.11}$$

To this end we expand Equation (9.11) in $\boldsymbol{\epsilon}$ to leading order:

$$\langle \delta\boldsymbol{\epsilon} \rangle \approx \eta\Big[\mathbb{C}'\boldsymbol{\epsilon} - 2(\boldsymbol{\epsilon} \cdot \mathbb{C}'\boldsymbol{w}^*)\boldsymbol{w}^* - (\boldsymbol{w}^* \cdot \mathbb{C}\boldsymbol{w}^*)\boldsymbol{\epsilon}\Big]. \tag{9.12}$$

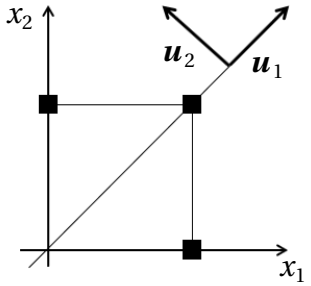


Figure 9.4: Input data with non-zero mean. Algorithm 10 converges to $\boldsymbol{u}_1$, but the principal direction is $\boldsymbol{u}_2$.

We know that $\boldsymbol{w}^*$ must be an eigenvector of $\mathbb{C}'$. So we choose a value of α, put $\boldsymbol{w}^* = \boldsymbol{u}_\alpha$, and multiply with $\boldsymbol{u}_\beta$ from the left, to determine the β-th component of $\langle \delta\boldsymbol{\epsilon} \rangle$:

$$\boldsymbol{u}_\beta \cdot \langle \delta\boldsymbol{\epsilon} \rangle \approx \eta[(\lambda_\beta - \lambda_\alpha) - 2\lambda_\alpha \delta_{\alpha\beta}](\boldsymbol{u}_\beta \cdot \boldsymbol{\epsilon}). \tag{9.13}$$

Here we used that $\delta_{\alpha\beta}(\boldsymbol{u}_\alpha \cdot \boldsymbol{\epsilon}) = \delta_{\alpha\beta}(\boldsymbol{u}_\beta \cdot \boldsymbol{\epsilon})$. We conclude: if $\lambda_\beta > \lambda_\alpha$ the component of the displacement along $\boldsymbol{u}_\beta$ must grow, on average. So if λ_α is not the largest eigenvalue, the corresponding eigenvector $\boldsymbol{u}_\alpha$ cannot be a steady-state direction. The eigenvector corresponding to the maximal eigenvalue, on the other hand, represents a steady state. So only $\lambda_\alpha = \lambda_1$ leads to a steady state. In summary, Algorithm 10 finds the principal component of zero-mean data. This also implies that Algorithm 10 maximises $\langle y^2 \rangle$ over all $\boldsymbol{w}$ with $|\boldsymbol{w}| = 1$, as shown in Section 6.3.1. Note that $\langle y \rangle = 0$ for zero-mean input data.

Finally consider inputs with non-zero mean. In this case Algorithm 10 still finds the maximal eigenvalue direction of $\mathbb{C}'$. But for inputs with non-zero means, this direction is different from the maximal principal direction (Section 6.3.1). Figure 9.4 illustrates this difference. The Figure shows three data points in a two-dimensional input plane. The elements of $\mathbb{C}' = \langle \boldsymbol{x}\boldsymbol{x}^\mathsf{T} \rangle$ are

$$\mathbb{C}' = \frac{1}{3}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \tag{9.14}$$

with eigenvalues and eigenvectors

$$\lambda_1 = 1\,, \quad \boldsymbol{u}_1 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \text{and} \quad \lambda_2 = \tfrac{1}{3}\,, \quad \boldsymbol{u}_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} -1 \\ 1 \end{bmatrix}. \tag{9.15}$$

It turns out that the maximal eigenvalue direction is $\boldsymbol{u}_1$. To compute the principal direction of the data we must determine the data-covariance matrix $\mathbb{C}$ (6.22). The maximal-eigenvalue direction of $\mathbb{C}$ is $\boldsymbol{u}_2$. This is the maximal principal component of the data shown in Figure 9.4.

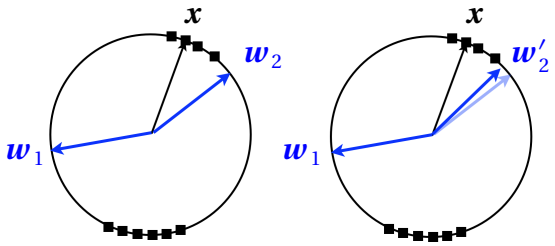


Figure 9.5: Detection of clusters by unsupervised learning.

Oja's rule can be generalised to determine M principal components of zero-mean input data using M output neurons that compute $y_i = \boldsymbol{w}_i \cdot \boldsymbol{x}$ for $i = 1, \ldots, M$:

$$\delta w_{ij} = \eta y_i \Big(x_j - \sum_{k=1}^{M} y_k w_{kj} \Big) \quad \text{Oja's } M\text{-rule}. \tag{9.16}$$

For $M = 1$ this reduces to Oja's rule.

9.2 Competitive learning

In Equations (**??**) and (9.16) several outputs can be active (non-zero) at the same time. An alternative approach is *competitive learning* where only one output is active at a time, as in Section 7.1.

Such algorithms can categorise or cluster input data: similar inputs are classified to belong to the same category, and activate the same output unit. Figure 9.5 shows input patterns on the unit circle that cluster into two distinct clusters. The idea is to find weight vectors $\boldsymbol{w}_i$ that point into the direction of the clusters. To this end we take M linear output units i with weight vectors $\boldsymbol{w}_i$, $i = 1, \ldots, M$. We feed a pattern $\boldsymbol{x}$ from the distribution $P_{\text{data}}(\boldsymbol{x})$ into the units and *define* the *winning unit* i_0 as the one that has minimal angle between its weight and the pattern vector $\boldsymbol{x}$. This is illustrated in Figure 9.5, where $i_0 = 2$. Then only this weight vector is updated by adding a little bit of the difference $\boldsymbol{x} - \boldsymbol{w}_{i_0}$ between the pattern vector and the weight of the winning unit. The other weights remain unchanged:

$$\delta \boldsymbol{w}_i = \begin{cases} \eta(\boldsymbol{x} - \boldsymbol{w}_i) & \text{for} \quad i = i_0(\boldsymbol{x}, \boldsymbol{w}_1 \ldots \boldsymbol{w}_M), \\ 0 & \text{otherwise}. \end{cases} \tag{9.17}$$

In other words, only the winning unit is updated, $\boldsymbol{w}'_{i_0} = \boldsymbol{w}_{i_0} + \delta \boldsymbol{w}_{i_0}$. Equation (9.17) is called *competitive-learning* rule.

The learning rule (9.17) has the following geometrical interpretation: the weight of the winning unit is drawn towards the pattern $\boldsymbol{x}$. Upon iterating (9.17), the weight vectors are drawn to *clusters* of inputs. So if the patterns are normalised as in Figure

9.5, the weights end up normalised on average, even though $|\boldsymbol{w}_{i_0}| = 1$ does not imply that $|\boldsymbol{w}_{i_0} + \delta \boldsymbol{w}_{i_0}| = 1$, in general. The algorithm for competitive learning is summarised in Algorithm 11.

When weight and input vectors are normalised, then the winning unit i_0 is the one with the largest scalar product $\boldsymbol{w}_i \cdot \boldsymbol{x}$. For linear output units $y_i = \boldsymbol{w}_i \cdot \boldsymbol{x}$ (Figure 9.2) this is simply the unit with the largest output. Equivalently, the winning unit is the one with the smallest distance $|\boldsymbol{w}_i - \boldsymbol{x}|$. Output units with $\boldsymbol{w}_i$ that are very far away from any pattern may never be updated (*dead units*). There are several strategies to avoid this problem [1]. One possibility is to initialise the weights to directions found in the inputs.

The solution of the clustering problem is not uniquely defined. One possibility is to monitor progress by an energy function

$$H = \frac{1}{2T} \sum_{ijt} M_{it} \left(x_j^{(t)} - w_{ij} \right)^2 . \tag{9.18}$$

Here $\boldsymbol{x}^{(t)}$ is the pattern fed in iteration number t, T is the total number of iterations, and

$$M_{it} = \begin{cases} 1 & \text{for} \quad i = i_0(\boldsymbol{x}^{(t)}, \boldsymbol{w}_1, \ldots, \boldsymbol{w}_M), \\ 0 & \text{otherwise}. \end{cases} \tag{9.19}$$

Note that i_0 is a function of the patterns $\boldsymbol{x}^{(\mu)}$ and of the weights $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_M$. For given patterns, the indicator $M_{i\mu}$ is a piecewise constant function of the weights. Gradient descent on the energy function (9.18) gives

$$\langle \delta w_{mn} \rangle = -\eta \frac{\partial H}{\partial w_{mn}} = \frac{\eta}{T} \sum_t M_{mt} (x_n^{(t)} - w_{mn}). \tag{9.20}$$

Apart from the sum over patterns this is the same as the competitive learning rule (9.17). The angular brackets on the l.h.s. of Equation (9.20) indicate that the weight increments are summed over patterns.

Algorithm 11 competitive learning (Figure 9.5)

1: initialise weights to vectors with random angles and norm $|\boldsymbol{w}_i| = 1$;
2: **for** $t = 1, \ldots, T$ **do**
3: draw a pattern $\boldsymbol{x}$ from $P_{\text{data}}(\boldsymbol{x})$ and feed it to the network:
4: find the winning unit i_0 (smallest angle between $\boldsymbol{w}_{i_0}$ and $\boldsymbol{x}$);
5: update only the winning unit $\delta \boldsymbol{w}_{i_0} = \eta(\boldsymbol{x} - \boldsymbol{w}_{i_0})$;
6: **end for**
7: end;

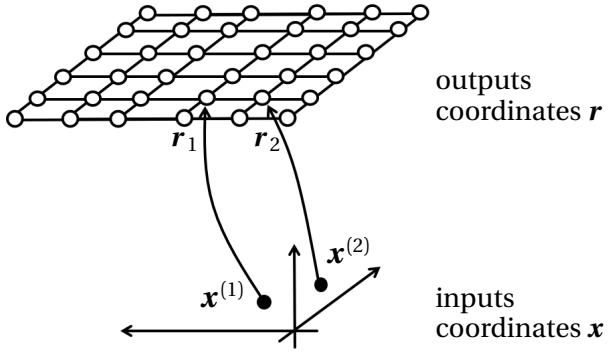


Figure 9.6: Spatial map. If patterns $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$ are close in input space, then the two patterns activate neighbouring outputs, $\boldsymbol{r}_1 \approx \boldsymbol{r}_2$.

If we define

$$y_i = \delta_{i i_0} = \begin{cases} 1 & \text{for } i = i_0 \\ 0 & \text{otherwise} \end{cases} \tag{9.21}$$

then the rule (9.17) can be written in the form of Oja's M-rule:

$$\delta w_{ij} = \eta y_i \Big(x_j - \sum_{k=1}^{M} y_k w_{kj} \Big). \tag{9.22}$$

This is again Hebb's rule with weight decay.

9.3 Kohonen's algorithm

Kohonen's algorithm can learn spatial maps. To this end one arranges the output units geometrically. The idea is that close inputs activate nearby outputs (Figure 9.6). Not that input and output space need not have the same dimension. The spatial map is learned with a competitive learning rule (9.22) , similar to the previous Section. But an important difference is that the rule must be modified to incorporate spatial information. In Kohonen's rule this is done by updating not only the winning unit, but also its neighbours in the output array.

$$\delta w_{ij} = \eta \Lambda(i, i_0)(x_j - w_{ij}). \tag{9.23}$$

Here $\eta > 0$ is the learning rate, as before. The function $\Lambda(i, i_0)$ is called the *neighbourhood function*. A common choice is

$$\Lambda(i, i_0) = \exp\Big(-\frac{\big|\boldsymbol{r}_i - \boldsymbol{r}_{i_0}\big|^2}{2\sigma^2} \Big). \tag{9.24}$$

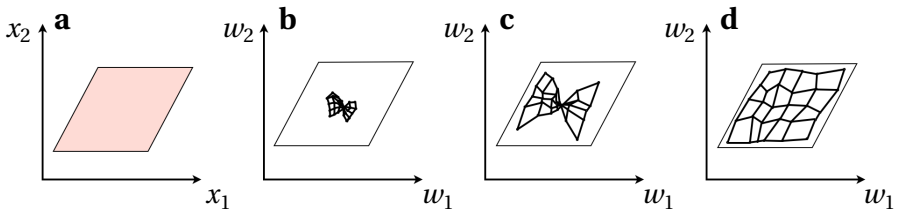


Figure 9.7: Learning a shape with Kohonen's algorithm. (**a**) Input-pattern distribution. $P_{\text{data}}(\boldsymbol{x})$ is unity within a parallelogram with unit area, and zero outside. (**b**) to (**d**) Illustration of the dynamics in terms of an *elastic net*. (**b**) Initial condition. (**c**) Intermediate stage (note the *kink*). (**d**) In the steady-state the elastic net resembles the shape defined by the input-pattern distribution.

As a result, nearby output units respond to inputs that are close in input space. Kohonen's rule drags the winning weight vector $\boldsymbol{w}_{i_0}$ towards $\boldsymbol{x}$, just as the competitive learning rule (9.17), but it also drags the neighbouring weight vectors along.

Figure 9.7 illustrates a geometrical interpretation of Kohonen's rule. We can think of the weight vectors as pointing to the nodes of an *elastic net* that has the same layout as the output array. As one feeds patterns from the input distribution, the weights are updated, causing the nodes of the network to move. This changes the shape of the elastic net. In the steady state, this shape resembles the shape defined by the distribution of input patterns.

Kohonen's rule has two parameters: the learning rate η, and the width σ of the neighbourhood function. Usually one adjusts these parameters as the learning proceeds. Typically one begins with large values for η and σ (*ordering phase*), and then reduces these parameters as the elastic net evolves (*convergence phase*): quickly at first and then in smaller steps, until the algorithm converges. Details are given by Hertz, Krogh and Palmer [1]. As for competitive learning one can monitor progress of the learning with an energy function

$$H = \frac{1}{2T}\sum_{it}\Lambda(i,i_0)\left(\boldsymbol{x}^{(t)}-\boldsymbol{w}_i\right)^2 . \tag{9.25}$$

Gradient descent yields

$$\langle \delta w_{ij}\rangle = -\eta\frac{\partial H}{\partial w_{ij}} = \frac{\eta}{T}\sum_t \Lambda(i,i_0)(x_j^{(t)}-w_{ij}). \tag{9.26}$$

Figure 9.7 shows how Kohonen's network learns by unfolding the elastic net of weight vectors until the shape of the net resembles the form of the input distribution

$$P_{\text{data}}(\boldsymbol{x}) = \begin{cases} 1 & \text{for } \boldsymbol{x} \text{ in the parallelogram in Figure 9.7(\textbf{a})}, \\ 0 & \text{otherwise}. \end{cases} \tag{9.27}$$

In other words, Kohonen's algorithm learns by distributing the weight vectors to reflect the distribution of input patterns. For the distribution (9.27) of inputs one may hope that the weights end up distributed uniformly in the parallelogram. This is roughly how it works (Figure 9.7), but there are problems at the boundaries. Why this happens is quite clear: for the distribution (9.27) there are no patterns outside the parallelogram that can draw the elastic net very close to the boundary.

To analyse how the boundaries affect learning for Kohonen's rule, we consider the steady-state condition

$$\langle \delta \boldsymbol{w}_i \rangle = \frac{\eta}{T} \sum_t \Lambda(i, i_0)\big(\boldsymbol{x}^{(t)} - \boldsymbol{w}_i^*\big) = 0\,. \tag{9.28}$$

This condition is more complicated than it looks at first sight, because i_0 depends on the weights and on the patterns, as mentioned above. The steady-state condition (9.28) is very difficult to analyse in general. One of the reasons is that global geometric information is difficult to learn. It is usually much easier to learn local structures. This is particularly true in the *continuum limit* where we can analyse local learning progress using Taylor expansions.

The analysis of condition (9.28) in the continuum limit is described by Hertz, Krogh, and Palmer [1], see also Ref. [110]. The calculation goes as follows. One assumes that there is a very dense net of weights, so that one can approximate $i \to \boldsymbol{r}$, $i_0 \to \boldsymbol{r}_0$, $\boldsymbol{w}_i \to \boldsymbol{w}(\boldsymbol{r})$, $\Lambda(i, i_0) \to \Lambda\big(\boldsymbol{r} - \boldsymbol{r}_0(\boldsymbol{x})\big)$, and $\frac{1}{T}\sum_t \to \int d\boldsymbol{x}\, P_{\text{data}}(\boldsymbol{x})$. In this *continuum limit*, Equation (9.28) reads

$$\int \mathrm{d}\boldsymbol{x}\, P_{\text{data}}(\boldsymbol{x})\,\Lambda\big(\boldsymbol{r} - \boldsymbol{r}_0(\boldsymbol{x})\big)\big[\boldsymbol{x} - \boldsymbol{w}^*(\boldsymbol{r})\big] = 0\,. \tag{9.29}$$

This is a condition for the spatial map $\boldsymbol{w}^*(\boldsymbol{r})$. Equation (9.29) is still quite difficult to analyse. So we specialise to one input and one output dimension, with spatial output coordinate r. This has the added advantage that we can easily draw the spatial map $w^*(r)$. It is the solution of

$$\int \mathrm{d}x\, P_{\text{data}}(x)\,\Lambda\big(r - r_0(x)\big)\big[x - w^*(r)\big] = 0 \tag{9.30}$$

The neighbourhood function is sharply peaked at $r = r_0(x)$. This means that the condition (9.30) yields the local properties of $w^*(r)$ around r_0, where r_0 is the coordinate of the winning unit, $x = w^*(r_0)$. Equation (9.30) involves an integral over patterns x. Using $x = w^*(r_0)$, this integral is expressed as an integral over r_0. Specifically we consider how $w^*(r_0)$ changes in the vicinity of a given point r, as $r_0(x)$ changes. To this end we expand $w^*(r_0)$ around r. To simplify the notation let us drop the asterisk. Then the expansion reads:

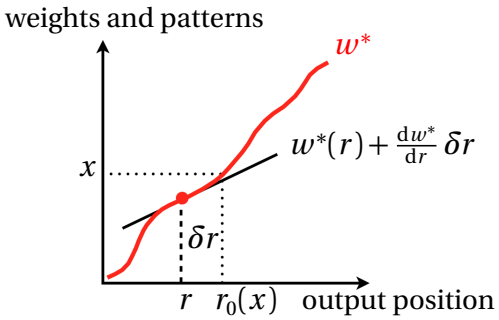


Figure 9.8: To find out how w^* varies near r, we expand w^* in δr around r. This gives $w^*(r)+\frac{dw^*}{dr}\delta r+\frac{1}{2}\frac{d^2w^*}{dr^2}\delta r^2+\ldots$.

$$w(r_0)=w(r)+w'(r)\delta r+\tfrac{1}{2}w'(r)'\delta r^2+\ldots \quad \text{with} \quad \delta r=r_0(x)-r\,. \tag{9.31}$$

Here w' denotes the derivative dw/dr evaluated at r, and I have dropped the asterisk. Using this expansion and $x=w(r_0)$ we express dx in Equation (9.30) in terms of $d\delta r$:

$$dx=dw(r_0)=dw(r+\delta r)\approx(w'+\delta r\,w'')d\delta r\,. \tag{9.32a}$$

To perform the integration we express the integrand in (9.30) in terms of δr:

$$P_{\text{data}}(x)=P\big(w(r_0)\big)\approx P_{\text{data}}(w)+\delta r\,w'\tfrac{d}{dw}P_{\text{data}}(w), \tag{9.32b}$$

and

$$x-w(r)=w'\delta r+\frac{1}{2}w''(r)\delta r^2+\ldots\,. \tag{9.32c}$$

Inserting these expressions into Equation (9.29) we find

$$\begin{aligned}0&=\int d\delta r\,(w'+\delta r\,w'')[P+\delta r\,w'\tfrac{d}{dw}P]\Lambda(\delta r)(\delta r\,w'+\tfrac{1}{2}\delta r^2w'')\\&=w'[\tfrac{3}{2}w''P_{\text{data}}(w)+w(w')^2\tfrac{d}{dw}P_{\text{data}}(w)]\int_{-\infty}^{\infty}d\delta r\,\delta r^2\Lambda(\delta r)\end{aligned} \tag{9.33}$$

Since the last integral in Equation (9.33) is non-zero, we must either have

$$w'=0 \quad \text{or} \quad \frac{3}{2}w''P_{\text{data}}(w)+(w')^2\tfrac{d}{dw}P_{\text{data}}(w)=0\,. \tag{9.34}$$

The first solution can be excluded because it corresponds to a singular weight distribution [see Equation (9.36)] that does not contain any geometrical information about the input distribution $P_{\text{data}}(x)$. The second solution gives

$$\frac{w''}{w'}=-\frac{2}{3}\frac{w'\frac{d}{dw}P_{\text{data}}(w)}{P_{\text{data}}(w)} \tag{9.35}$$

In other words, $\frac{\mathrm{d}}{\mathrm{d}r}\log|w'| = -\frac{2}{3}\frac{\mathrm{d}}{\mathrm{d}r}\log P_{\text{data}}(w)$. This means that $|w'| \propto [P_{\text{data}}(w)]^{-\frac{2}{3}}$. So the distribution ϱ of output weights is

$$\varrho(w) \equiv \left|\frac{\mathrm{d}r}{\mathrm{d}w}\right| = \frac{1}{|w'|} = [P_{\text{data}}(w)]^{\frac{2}{3}}. \tag{9.36}$$

This tells us that the Kohonen net learns the input distribution in the following way: the distribution of output weights in the steady state reflects the distribution of input patterns. Equation (9.36) tells us that the two distributions are not equal (equality would have been a perfect outcome). The distribution of weights is instead proportional to $P_{\text{data}}(w)^{\frac{2}{3}}$. This is a consequence of the fact that the elastic net has difficulties reaching the corners and edges of the domain where the input distribution is non-zero.

Let us finally discuss the convergence of Kohonen's algorithm. The update rule (9.23) can be rewritten as

$$(w_i - x) \leftarrow (1 - \eta\Lambda)(w_i - x). \tag{9.37}$$

For small enough η the factor $(1-\eta\Lambda)$ is positive. In this case it follows from Equation (9.37) that the order of weights in a monotonically increasing (decreasing) sequence does not change under an update [111**?**]. What happens when $w(r)$ has a maximum or a minimum (a *kink*)? Kinks can only disappear in two ways. Either they move to one of the boundaries, or two kinks (a minimum and a maximum) annihilate each other if they collide. Both processes are slow. This means that convergence to the steady state can be very slow. Therefore one usually starts with a larger learning rate, to get rid of kinks. After this *ordering phase*, one continues with a smaller step size to get the details of the distribution right (*convergence phase*).

9.4 Clustering with self-organising maps

What does Kohonen's algorithm achieve? Similar inputs activate neighbouring outputs (*semantic map*). Clustering. Alternative interpretation: weights as pointers to input space [112]: many weights point to dense regions in input space. Explain the term *self-organising map*. Refs. [2, 113, 114], linear vs. non-linear methods, k-means clustering, principal-component analysis. Fitting Gaussians to high-dimensional data [].

How to interpret the output of a self-organised map?

Vector quantisation

Examples: recognise MNIST digits. Tensor flow. MATLAB. Colours.

9.5 Other clustering algorithms

9.6 Summary

The unsupervised learning algorithms described above are based on Hebb's rule: certainly the Hebbian unsupervised learning rule (9.3) and Oja's rule (9.4). Also the competitive learning rule can be written in this form [Equation (9.22)]. Kohonen's algorithm is closely related to competitive learning, although the way in which Kohonen's rule learns spatial maps is better described by the notion of an elastic net that represents the values of the output weights, as well as their spatial location in the output array. Unsupervised learning rules can learn different features of the distribution of input patterns. They can discover which patterns occur most frequently, they can help to reduce the dimensionality of input space by finding the principal directions of the input distribution, detect clusters in the input data, compress data, and learn spatial input-output maps. The important point is that the algorithms learn without training, unlike the algorithms in Chapters 5 to 8.

Supervised-learning algorithms are now widely used for different applications. This is not really the case yet for unsupervised-learning algorithms, except that similar (sometimes equivalent) algorithms are used in Mathematical Statistics (k-means clustering) and Bioinformatics (`structure` [115]) where large data sets must be analysed, such as Human sequence data (HGDP) [116]. But the simple algorithms described in this Chapter provide a proof of concept: how machines can learn without feedback. In addition there is one significant application of unsupervised learning: where the network learns from incomplete feedback. This *reinforcement learning* is introduced in the next Chapter.

9.7 Exercises

9.1 Continuous Oja's rule. Using the ansatz $\boldsymbol{w} = \boldsymbol{q}/|\boldsymbol{q}|$ show that Equations (9.6) and (9.7) describe the same angular dynamics. The difference is just that $\boldsymbol{w}$ remains normalised to unity, whereas the norm of $\boldsymbol{q}$ may increase of decrease. See Ref. [108].

9.2 Data-covariance matrix. Determine the data-covariance matrix and the principal direction for the data shown in Figure 9.4.

9.3 Oja's rule. The aim of unsupervised learning is to construct a network that learns the properties of a distribution $P_{\text{data}}(\boldsymbol{x})$ of input patterns $\boldsymbol{x} = [x_1, \ldots, x_N]^\mathsf{T}$. Consider a network with one linear output that computes $y = \sum_{j=1}^{N} w_j x_j$. Under Oja's learning rule $\delta w_j = \eta y(x_j - y w_j)$ the weight vector $\boldsymbol{w}$ converges to a steady

state $\boldsymbol{w}^*$ with components w_j^*. The steady state has the following properties:

1. $|\boldsymbol{w}^*|^2 \equiv \sum_{j=1}^{N} (w_j^*)^2 = 1$.
2. $\boldsymbol{w}^*$ is the leading eigenvector of the matrix $\mathbb{C}'$ with elements $C'_{ij} = \langle x_i x_j \rangle$. Here $\langle \cdots \rangle$ denotes the average over $P_{\text{data}}(\boldsymbol{x})$.
3. $\boldsymbol{w}^*$ maximises $\langle y^2 \rangle$.

Show that (3) follows from (1) and (2). Prove (1) and (2), assuming that $\boldsymbol{w}^\star$ is a steady state. Write down a generalisation of Oja's rule that learns the first M principal components for zero-mean data, $\langle \boldsymbol{x} \rangle = 0$. Discuss: how does the rule ensure that the weight vectors remain normalised? What happens when $\langle \boldsymbol{x} \rangle$ is not zero?

9.4 Competitive learning of binary data. Explain how normalisation of patterns and weights works for paterns with 0/1 bits.

9.5 Kohonen net. Explain the meaning of the parameter σ in the neighbourhood function in Kohonen's learning rule, Equations (9.23) and (9.24). Discuss the nature of the update rule in the limit of $\sigma \to 0$. Discuss and explain the implementation of Kohonen's algorithm in a computer program. In the discussion, refer to and explain the following terms: *output array, neighbourhood function, ordering phase, convergence phase, kinks.*

9.6 Self-organising map. Write a computer program that implements Kohonen's algorithm with a two-dimensional output array, to learn the properties of a two-dimensional input distribution that is uniform inside an equilateral triangle with sides of unit length, and zero outside. *Hint*: to generate this distribution, sample at least 1000 points uniformly distributed over the smallest square that contains the triangle, and then accept only points that fall inside the triangle. Increase the number of weights and study how the two-dimensional density of weights near the boundary depends on the distance from the boundary.

9.7 Principal manifolds. Create a data set like the one shown in Figure 9.9, using $x_2 = x_1^2 + r$ where r is a Gaussian random number with mean zero and variance $\sigma_r^2 = 0.01$. Determine the principal component (blue line) of the data set (Section 6.3.1). Use Kohonen's algorithm to find a better approximation to the data, the *principal manifold* (red line). For both cases determine the variance of data that remains unexplained.

9.8 Iris data set. Write a computer program that combines a two-dimensional Kohonen net with a simple classifier to classify the Iris data set (Figure 5.1).

9.9 Steady state of two-dimensional Kohonen algorithm. Repeat the analysis of

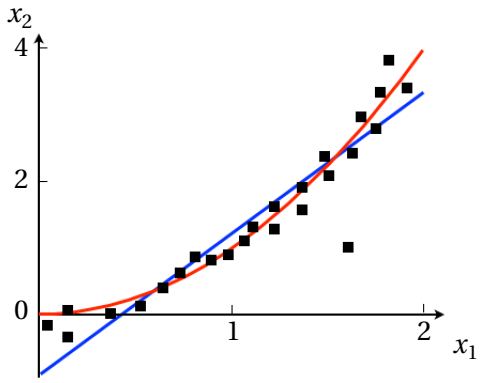


Figure 9.9: Kohnen's algorithm finds the *principal manifold* (red line) vs. principal-component analysis (Section 6.3.1), blue line (Exercise 9.7).

Equation (9.30), but for a two-dimensional Kohonen network. (a) Derive the equivalent of Equation (9.33) and determine a relation between the weight density ϱ and P_{data} assuming that the data distribution factorises $P_{\text{data}}(\boldsymbol{w}) = f(w_1)g(w)_2$ [110]. (b) Assume that $\boldsymbol{w}(\boldsymbol{r}) = u + i\,v$ can be written as an analytic function of $\boldsymbol{r} = x + i\,y$ and derive a relation between ρ and P_{data}.

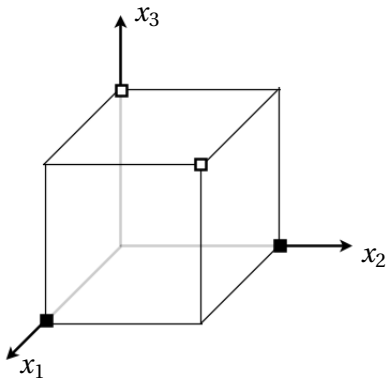


Figure 10.1: The XOR problem can be solved by embedding into a three-dimensional input space.

10 Radial basis-function networks

Problems that are not linearly separable can be solved by perceptrons with hidden layers, as we saw in Chapter 5. Figure 5.10, for example, shows a piecewise linear decision boundary that can be parameterised by hidden neurons.

Another approach is to *map* the coordinates of input space so that the problem becomes linearly separable. It is usually easier to separate patterns in higher dimensions. To see this consider the XOR problem. It is not linearly separable in two-dimensional input space. The problem becomes separable when we *embed* the points in three-dimensional space, for instance by assigning $x_3 = 0$ to the $t = +1$ patterns and $x_3 = 1$ to the $t = -1$ patterns (Figure 10.1). This example illustrates why it is often helpful to map input space to a higher-dimensional space – because it is more likely that the resulting problem is linearly separable. It may also be possible to achieve separability by a non-linear transformation of input space to a space of the same dimension. Figure 10.2 shows how the XOR problem can be transformed into a linearly separable problem by the transformation

$$u_1(\boldsymbol{x}) = (x_2 - x_1)^2 \quad \text{and} \quad u_2(\boldsymbol{x}) = x_2\,. \tag{10.1}$$

The Figure shows the non-separable problem in the x_1-x_2 plane, and in the new coordinates u_1 and u_2. Since the problem is linearly separable in the u_1-u_2 plane we can solve it by a single McCulloch-Pitts neuron with weights $\boldsymbol{W}$ and threshold Θ, parameterising the decision boundary as $\boldsymbol{W} \cdot \boldsymbol{u}(\boldsymbol{x}) = \Theta$. In fact, one does not need the threshold Θ because the function $\boldsymbol{u}$ can have a constant part. For instance, we could choose $u_1(\boldsymbol{x}) = 2(x_2 - x_1)^2 - 1$. In the following we therefore set $\Theta = 0$.

We expect that it should be easier to achieve linear separability the higher the *embedding dimension* is. This statement is quantified by Cover's theorem, discussed

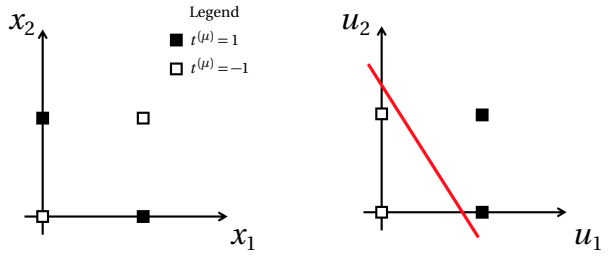


Figure 10.2: Left: input plane for the XOR function (Figure 5.8). The problem is not linearly separable. Right: in the u_1-u_2 plane the problem is linearly separable.

in Section 10.1. The question is of course how to find the non-linear mapping $\boldsymbol{u}(\boldsymbol{x})$. One possibility is to use radial basis functions. This is a way of parameterising the functions $u_j(\boldsymbol{x})$ in terms of weight vectors $\boldsymbol{w}_j$, and to determine suitable weight vectors iteratively. How this works is summarised in Section 10.2.

10.1 Separating capacity of a surface

In its simplest form, *Cover's theorem* [117] concerns a classification problem given by p points with coordinate vectors $\boldsymbol{u}^{(\mu)}$ in m-dimensional space.

$$t^{(\mu)} = \begin{cases} +1 & \text{with probability } \frac{1}{2}, \\ -1 & \text{with probability } \frac{1}{2}. \end{cases} \tag{10.2}$$

This random classification problem is *homogeneously* linearly separable if we can find an m-dimensional weight vector $\boldsymbol{W}$ with components W_j,

$$\boldsymbol{W} = \begin{bmatrix} W_1 \\ \vdots \\ W_m \end{bmatrix}, \tag{10.3}$$

so that $\boldsymbol{W} \cdot \boldsymbol{u} = 0$ is a valid decision boundary that goes through the origin:

$$\boldsymbol{W} \cdot \boldsymbol{u}^{(\mu)} > 0 \quad \text{if} \quad t^{(\mu)} = 1 \quad \text{and} \quad \boldsymbol{W} \cdot \boldsymbol{u}^{(\mu)} < 0 \quad \text{if} \quad t^{(\mu)} = -1\,. \tag{10.4}$$

So *homogeneously linearly separable* problems are classification problems that are linearly separable by a hyperplane that contains the origin (zero threshold, Chapter 5).

Now assume that the points (including the origin) are in *general position* (Figure 10.3). In this case Cover's theorem states the probability that the random classification problem of p patterns in dimension m is homogeneously linearly separable:

Figure 10.3: Left: 5 points in general position in the plane. Right: these points are not in general position because three points lie on a straight line.

$$P(p,m) = \begin{cases} \left(\frac{1}{2}\right)^{p-1} \sum_{k=0}^{m-1} \binom{p-1}{k} & \text{for } p > m\,, \\ 1 & \text{otherwise}\,. \end{cases} \tag{10.5}$$

Here $\binom{l}{k} = \frac{l!}{(l-k)!k!}$ are the binomial coefficients, for $l \geq k \geq 0$. If one *defines* $\binom{l}{k} = 0$ for $l < k$ then Equation (10.5) can be written as $P(p,m) = (\frac{1}{2})^{p-1} \sum_{k=0}^{m-1} \binom{p-1}{k}$. Equation (10.5) is proven by recursion, starting from a set of $p-1$ points in general position. Assume that the number $C(p-1,m)$ of homogeneously linearly separable classification problems given these points is known. After adding one more point, one can compute the $C(p,m)$ in terms of $C(p-1,m)$. Recursion yields Equation (10.5).

To connect the result (10.5) to the discussion at the beginning of this Chapter, we take $\boldsymbol{u}^{(\mu)} = \boldsymbol{u}(\boldsymbol{x})$ where $\boldsymbol{x}^{(\mu)}$ are p patterns in N-dimensional input space

$$\boldsymbol{x}^{(\mu)} = \begin{bmatrix} x_1^{(\mu)} \\ \vdots \\ x_N^{(\mu)} \end{bmatrix} \quad \text{for} \quad \mu = 1, \ldots, p\,, \tag{10.6}$$

and we assume that $\boldsymbol{u}$ is a set of m polynomial functions of finite order. Then the probability that the problem of the p points $\boldsymbol{x}^{(\mu)}$ in N-dimensional input space is separable by a polynomial decision boundary is given by Equation (10.5) [2, 117]. Note that the probability $P(p,m)$ is independent of the dimension N of the input space. Figure 10.4 shows this probability for $p = \lambda m$ as a function of λ for different values of m. Note that $P(2m,m) = \frac{1}{2}$. In the limit of large m, the function $P(\lambda m, m)$ approaches a step function. In this limit one can separate at most $2m$ patterns (*separability threshold*).

Now consider a random sequence of patterns $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$ and targets $t_1, t_2, \ldots$ and ask [117]: what is the distribution of the largest integer so that the problem $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ is separable in embedding dimension m, but $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n, \boldsymbol{x}_{n+1}$ is not? $P(n,m)$ is the probability that n patterns are linearly separable in embedding dimension m. We can write $P(n+1,m) = q(n+1|n)P(n,m)$ where $q(n+1|n)$ is the conditional probability that $n+1$ patterns are linearly separable if the n patterns were. Then

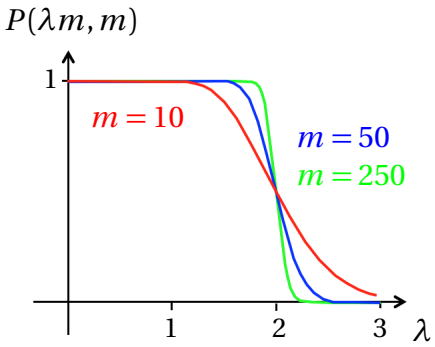


Figure 10.4: Probability (10.5) of separability for $p = \lambda m$ as a function of λ for three different values of the embedding dimension m. Note the pronounced threshold near $\lambda = 2$, for large values of m.

the probability that $n+1$ patterns are not separable (but n patterns are) reads $(1-q)P(n,m) = P(n,m) - P(n+1,m)$. We can interpret the right-hand side of this Equation as a distribution p_n of the random variable n, the maximal number of separable patterns in embedding dimension m:

$$p_n = P(n,m) - P(n+1,m) = \left(\frac{1}{2}\right)^n \binom{n-1}{m-1} \quad \text{for} \quad n = 0, 1, 2, \ldots .$$

It follows that the expected maximal number of separable patterns is

$$\langle n \rangle = \sum_{n=0}^{\infty} n p_n = 2m . \tag{10.7}$$

So the expected maximal number of separable patterns is twice the embedding dimension. This quantifies the notion that it is easier to separate patterns in higher embedding dimensions.

Comparing with the discussion of linear separability in Chapter 5 we see that Cover's theorem determines the separation capacity of a single-layer perceptron [1].

10.2 Radial basis-function networks

The idea behind radial basis-function networks is to parameterise the functions $u_j(\boldsymbol{x})$ in terms of weight vectors $\boldsymbol{w}_j$, and to use an unsupervised-learning algorithm (Chapter 9) to find weights that separate the input data. A common choice [2] are *radial basis functions* of the form:

$$u_j(\boldsymbol{x}) = \exp\Big(-\frac{1}{2s_j^2}\left|\boldsymbol{x} - \boldsymbol{w}_j\right|^2\Big). \tag{10.8}$$

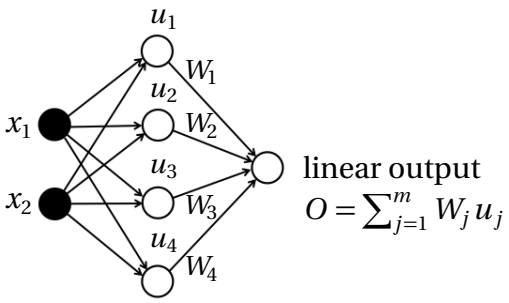


Figure 10.5: Radial basis-function network for $N = 2$ inputs and $m = 4$ radial basis functions (10.8). The output neuron has a linear activation function, weights $\boldsymbol{W}$, and zero threshold.

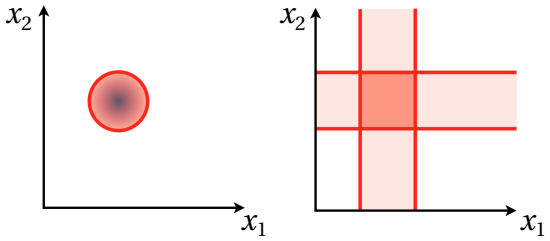


Figure 10.6: Comparison between radial-basis function network and perceptron. Left: the output of a radial basis function is localised in input space. Right: to achieve a localised output with sigmoid units one needs two hidden layers (Figure 7.5).

These functions are not of the finite-order polynomial form that was assumed in Cover's theorem. This means that the theorem does not strictly apply. The parameters s_j parameterise the *widths* of the radial basis functions. In the simplest version of the algorithm they are set to unity. When they are allowed to vary they reflect different widths of the radial basis functions. Other choices for radial basis functions are given by Haykin [2].

Figure 10.5 shows a radial basis-function network for $N = 2$ and $m = 4$. The four neurons in the hidden layer stand for the four radial basis functions (10.8) that map the inputs to four-dimensional $\boldsymbol{u}$-space. The network looks like a perceptron (Chapter 5). But here the hidden layers work in a different way. Perceptrons have hidden McCulloch-Pitts neurons that compute *non-local* outputs $\sigma(\boldsymbol{w}_j \cdot \boldsymbol{x} - \theta)$. The output of radial basis functions $u_j(\boldsymbol{x})$, by contrast, is *localised* in input space [Figure 10.6(left)]. We saw in Section 7.1 how to make localised basis functions out of McCulloch-Pitts neurons with sigmoid activation functions $\sigma(b)$, but one needs two hidden layers to do that [Figure 10.6(right)].

Radial basis functions produce localised outputs with a single hidden layer, and this makes it possible to divide up input space into localised regions, each corresponding to one radial basis function. Imagine for a moment that we have as many

radial basis functions as input patterns. In this case we can simply take $\boldsymbol{w}_\nu = \boldsymbol{x}^{(\nu)}$ for $\nu = 1, \ldots, p$. Then the classification problem can be written as

$$\sum_\mu U_{\nu\mu} W_\mu = t^{(\nu)}, \tag{10.9}$$

with $U_{\nu\mu} = u_\nu(\boldsymbol{x}^{(\mu)})$. Equation (10.9) uses that the output unit is linear and computes $O^{(\mu)} = \boldsymbol{W} \cdot \boldsymbol{u}(\boldsymbol{x}^{(\mu)})$ (Figure 10.5). If all patterns are pairwise different, $\boldsymbol{x}^{(\mu)} \neq \boldsymbol{x}^{(\nu)}$ for $\mu \neq \nu$, then the matrix $\mathbb{U}$ is invertible [2]. In this case the solution of the classification problem reads

$$W_\mu = \sum_\nu [\mathbb{U}^{-1}]_{\mu\nu} t^{(\nu)}, \tag{10.10}$$

where $\mathbb{U}$ is the symmetric $p \times p$ matrix with entries $U_{\mu\nu}$. In practice one can get away with fewer radial basis functions by choosing their weights to point in the directions of clusters of input data. To this end one can use unsupervised competitive learning (Algorithm 12), where the *winning unit* is defined to be the one with largest u_j. How are the widths s_j determined? The width s_j of radial basis function $u_j(\boldsymbol{x})$ is taken to be equal to the minimum distance between $\boldsymbol{w}_j$ and the centers of the surrounding radial basis functions. Once weights and widths of the radial basis functions are found, the weights of the output neuron are determined by minimising

$$H = \frac{1}{2} \sum_\mu \left(t^{(\mu)} - O^{(\mu)} \right)^2 \tag{10.11}$$

with respect to $\boldsymbol{W}$. An approximate solution can be obtained by stochastic gradient descent on H keeping the parameters of the radial basis functions fixed. Cover's theorem indicates that the problem is more likely to be separable if the embedding dimension m is higher. Figure 10.7 shows an example for a non-linear decision boundary found by this algorithm.

10.3 Summary

Radial basis-function networks are similar to the perceptrons described in Chapters 5 to 7, in that they are feed-forward networks designed to solve classification problems. The outputs work in similar ways.

A fundamental difference is that the parameters of the radial basis functions are determined by unsupervised learning, whereas perceptrons are trained using supervised learning for *all* units.

While McCulloch-Pitts neurons compute weights to minimise their output from given targets, the radial basis functions compute weights by maximising u_j as a

Algorithm 12 radial basis functions

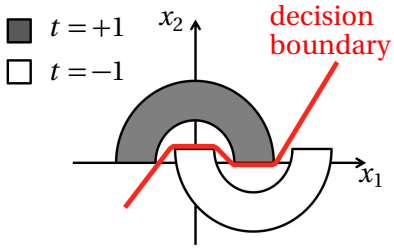
initialise the weights w_{jk} independently randomly from $[-1,1]$;
set all widths to $s_j = 0$;
for $t = 1,\ldots,T$ **do**
    feed randomly chosen pattern $\boldsymbol{x}^{(\mu)}$;
    determine winning unit j_0: $u_{j_0} \geq u_j$ for all values of j;
    update widths: $s_j = \min_{j\neq k} |\boldsymbol{w}_j - \boldsymbol{w}_k|$;
    update only winning unit: $\delta \boldsymbol{w}_{j_0} = \eta(\boldsymbol{x}^{(\mu)} - \boldsymbol{w}_{j_0})$;
end for
end;

function of j. The algorithm for finding the weights of the radial basis functions is summarised in Algorithm 12.

Further, as opposed to the deep networks from Chapter 7, radial basis-function networks have only one hidden layer, and a linear output neuron. Radial-basis function nets learn using a *hybrid* scheme: unsupervised learning for the parameters of the radial basis functions, and supervised learning for the weights of the output neuron.

10.4 Further reading

For a proof of Cover's theorem see Ref. [118]. Radial-basis function networks are discussed by Haykin in Chapter 5 of his book [2]. Hertz, Krogh and Palmer [1] have a brief Section on radial basis-function nets too (their Section 9.7). They use



m	classification error
5	0.26
20	0.06

Figure 10.7: Left: non-linear decision boundary found by a radial basis-function network for $m = 20$ (schematic). Right: classification error for different embedding dimensions. The decision boundary is not yet quite perfect for $m = 20$, it still cuts the corners a little bit. Make an OpenTA task and update the Table.

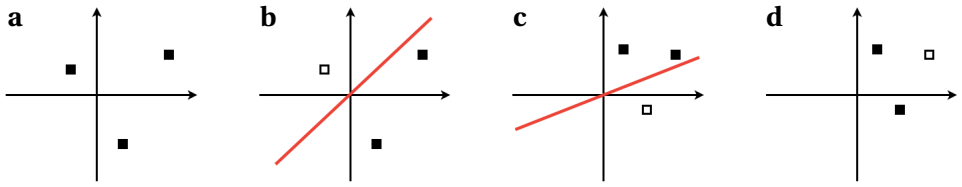


Figure 10.8: Cover's theorem for $p = 3$ and $m = 2$. Examples for problems that are homogeneously linearly separable, (**b**) and (**c**), and for problems that are not homogeneously linearly separable (**a**) and (**d**).

normalised radial basis functions

$$u_j(\boldsymbol{x}) = \frac{\exp\left(-\frac{1}{2s_j^2}\left|\boldsymbol{x}-\boldsymbol{w}_j\right|^2\right)}{\sum_{k=1}^{m}\exp\left(-\frac{1}{2s_j^2}\left|\boldsymbol{x}-\boldsymbol{w}_j\right|^2\right)}, \tag{10.12}$$

instead of Equation (10.8).

It has been argued that radial-basis function nets do not generalise as well as perceptrons do [119]. To solve this problem, Poggio and Girosi [120] suggested to determine the parameters $\boldsymbol{w}_j$ of the radial basis function by supervised learning, using stochastic gradient descent.

10.5 Exercises

10.1 Expected maximal number of separable patterns. Show that the sum in Equation (10.7) sums to $2m$.

10.2 Cover's theorem. Prove that $P(3,2) = \frac{3}{4}$ by complete enumeration of all cases. Some cases (not all) are shown in Figure 10.8.

10.3 Radial basis functions for XOR. Show that the two-dimensional Boolean XOR problem can be solved using the two radial basis functions $u_1(\boldsymbol{x}^{(\mu)}) = \exp(-|\boldsymbol{x}^{(\mu)}-\boldsymbol{w}_1|^2)$ and $u_2(\boldsymbol{x}^{(\mu)}) = \exp(-|\boldsymbol{x}^{(\mu)}-\boldsymbol{w}_2|^2)$ with $\boldsymbol{w}_1 = (1,1)^\mathsf{T}$ and $\boldsymbol{w}_2 = [0,0]^\mathsf{T}$. Draw the positions of the four input patterns in the transformed space with coordinates u_1 and u_2, encoding the different target values. *Hint*: to compute the states of input patterns in the u_1-u_2-plane, use the following approximations: $\exp(-1) \approx 0.37$, $\exp(-2) \approx 0.14$. Explain the term *decision boundary*, draw a decision boundary for the XOR problem, and give the corresponding weights and thresholds for the simple perceptron.

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	0

Table 10.1: Inputs and target values for the XOR problem. Exercise 10.3.

μ	$x_1^{(\mu)}$	$x_2^{(\mu)}$	$x_3^{(\mu)}$	$t^{(\mu)}$
1	-1	-1	-1	1
2	-1	-1	1	1
3	-1	1	-1	1
4	-1	1	1	-1
5	1	-1	-1	1
6	1	-1	1	1
7	1	1	-1	-1
8	1	1	1	1

Table 10.2: Inputs and target values. Exercise 10.4.

10.4 Radial basis functions. Table 10.2 describes a classification problem. Show that this problem can be solved as follows. Transform the inputs x_1, x_2, x_3 to two-dimensional coordinates u_1, u_2 using radial basis functions:

$$u_1^{(\mu)} = \exp(-\tfrac{1}{4}|\boldsymbol{x}^{(\mu)} - \boldsymbol{w}_1|^2), \text{ with } \boldsymbol{w}_1 = (\text{-}1,1,1)^\mathsf{T}, \tag{10.13}$$

$$u_2^{(\mu)} = \exp(-\tfrac{1}{4}|\boldsymbol{x}^{(\mu)} - \boldsymbol{w}_2|^2), \text{ with } \boldsymbol{w}_2 = (1,1,\text{-}1)^\mathsf{T}, \tag{10.14}$$

with $\boldsymbol{x}^{(\mu)} = [x_1^{(\mu)}, x_2^{(\mu)}, x_3^{(\mu)}]^\mathsf{T}$. Plot the positions of the eight input patterns in the u_1-u_2-plane. *Hint*: to compute $u_j^{(\mu)}$ use the following approximations: exp(-1) $\approx$ 0.37, exp(-2) $\approx$ 0.14, exp(-3) $\approx$ 0.05. The transformed data is used as input to a simple perceptron (no hidden layers) with one output unit $O^{(\mu)} = \mathrm{sgn}\big(\sum_{i=1}^{2} W_j\, u_j{}^{(\mu)} - \Theta\big)$. Draw a decision boundary in the u_1-u_2-plane and determine the corresponding weight vector $\boldsymbol{W}$ and the threshold Θ.

10.5 Half moons. Figure 10.2 illustrates the half-moon problem introduced by Haykin [2]. Make your own input data set by distributing inputs in the two regions shown with targets $t = +1$ in the filled region, and $t = -1$ in the empty one. Find approximate decision boundaries using a radial-basis function network with m radial basis functions, for $m = 5, 10, 20$ and 100 and plot them in the input plane. Determine the corresponding classification errors.

11 Reinforcement learning

Supervised learning requires labeled data, where each input comes with a target pattern that the network is supposed to learn. Unsupervised learning, by contrast, does not require labeled data. *Reinforcement learning* lies in between these extremes. The term reinforcement describes the principle of learning with only incomplete feedback, in the form of *penalty* or *reward*. For a network with a vector of outputs, for instance, the feedback may consist only of a single bit of information: *reward* (all outputs correct) or *penalty* (some or all bits are wrong):

$$r = \begin{cases} +1, & \text{reward}\,, \\ -1, & \text{penalty}\,. \end{cases} \tag{11.1}$$

The net is rewarded ($r = 1$) when its output matches the targets well (but perhaps not perfectly), but it receives negative feedback ($r = -1$) when the match is not good. The goal is to learn to produce outputs that receive positive feedback (reward) more frequently than those that trigger a penalty: rewarded outputs are reinforced.

More generally, the feedback may be random, given by a distribution initially unknown to the network. At any rate, this rules out learning by gradient descent on an energy function like (5.23) or (7.31).

One distinguishes two fundamental kinds of reinforcement problems, *associative* and *non-associative* problems. An example for a non-associative task is the N-armed bandit problem. Imagine N slot machines with different reward distributions, initially unknown to the player. Given a finite amount of money, the question is in which order to play the machines so as to maximise the overall winst. The dilemma is whether to stick with a machine that gives a high reward, or whether to try out other machines that may yield a low reward initially, but could give high rewards eventually (exploit versus explore dilemma). In this type of problems the network receives no other inputs apart from the reinforcement signal. There is no dynamics, and the goal is to find a sequence of *actions* that maximises the reward.

In associative tasks, by contrast, the net receives inputs (*stimuli*). The network learns to *associate* the optimal output with each input, namely the output that yields the highest reward. Such tasks occur in behavioural psychology, where the problem is to discriminate between different stimuli, and to associate the right behaviour with each stimulus. The simplest algorithms try to maximise the immediate reward. An example is the associative reward-penalty algorithm described in Section 11.1. It uses stochastic neurons with weights that are trained using partial feedback from the environment. There are close connections to Boltzmann machines (Chapter **??**) which operate with stochastic neurons too. For a single output neuron, the associative reward-penalty algorithm converges to the optimal strategy. Numerical

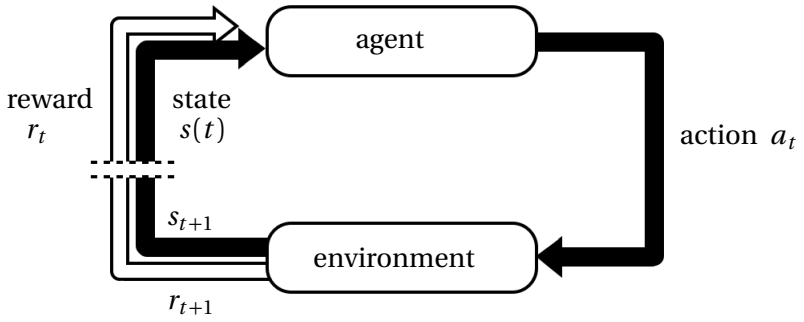


Figure 11.1: Sequential reinforcement learning (schematic). Adapted from Figure 3.1 in Ref. [121].

examples show that networks of stochastic neurons can learn optimal strategies, but there is no general proof of convergence. This is the main reason why recent research has adopted alternative schemes for which convergence can be proven. By now you will not be surprised to learn that in most applications the stringent convergence conditions are not satisfied.

Sequential reinforcement learning (Figure 11.1) allows to learn sequential decision tasks [121, 122]. A common example is an agent that learns to navigate a complex environment, for instance to a motile microorganism in the turbulent ocean that should swim to the surface as quickly as possible [123]. The agent determines its state by observing the local environment. The microorganism, for example, might measure the local strain and vorticity in its vicinity. The enviroment provides a reinforcement signal (the distance to the surface for example), and the agent determines which action to take, given its state and the reinforcement signal. Should it turn, stop to swim, or accelerate?

So the agent learns to associate optimal actions with certain states. This sounds quite similar to associating optimal outputs with different inputs. The conceptual difference is that the action of the agent modifies the enviroment: its actions take it to a different place in the turbulent flow, with different vorticity and different strain. The goal is generally to optimise the reward in the long run, to achieve this it is necessary to explore actions that do not give an immediate high reward. Improved algorithms therefore optimise the expected future reward, given the information collected so far. At each time step the agent takes the action determined by its current *policy*, for instance the action that maximises the expected future reward (*greedy* policy), or a policy that allows to sometimes take sub-optimal steps in order to explore potentially better alternatives.

Temporal difference learning allows to estimate the expected future reward, after

T iterations say, by breaking up the learning into time steps $t = 1, \ldots, T$. The basic idea is that it is better to adjust the prediction of a future reward as one iterates, rather than waiting for T iterations before updating the prediction. The original version of the algorithm expressed the reward at time T in terms of differences at time steps $t+1$ and t (*telescoping sum*) [124]. More recent versions look a bit different [121, 125]. At any rate, the algorithm builds up a lookup table that summarises the best actions for each state, the *Q-matrix*. The elements of the Q-matrix contain estimates of the expected future reward for each state-action pair. In general it is difficult to prove convergence of algorithms for temporal difference learning.

A simplified scheme is *Q-learning*, an approximation to the temporal difference algorithm. In Q learning, the Q-matrix is updated assuming that the agent always follows the greedy policy, even though it might follow a different policy. Q-learning allows agents to learn to play strategic games [8]. A simple example is the game of tic-tac-toe (Section 11.5). Games such as chess or go require to keep track of a very large number of states, so large that Q-learning in its simplest form becomes impractical (*curse of dimension*). An alternative is to represent the mapping from states to actions by a deep neural network (deep Q learning [8]).

11.1 Associative reward-penalty algorithm

The associative reward-penalty algorithm is one of the simplest examples that shows how neural networks can learn from incomplete feedback. This algorithm uses *stochastic neurons* which are trained to maximise the average immediate reward.

In Chapters 5 to 8 the output neurons were deterministic functions of their inputs, either sigmoid, tanh, or softmax functions. Reinforcement learning, by contrast, works with stochastic neurons. The idea is the same as in Chapters 3, 4, and **??**: stochastic neurons can explore a wider range of possible states which may in the end lead to a better solution. The state y_i of neuron i is given by the *stochastic update rule* (3.2):

$$y_i = \begin{cases} +1, & \text{with probability } p(b_i), \\ -1, & \text{with probability } 1-p(b_i), \end{cases} \tag{11.2}$$

where $b_i = \sum_j w_{ij} x_j$ is the local field, and $p(b) = (1+e^{-2\beta b})^{-1}$ as before. The parameter β^{-1} is the noise level, as in Chapters 3, 4, and **??**. Since the output can assume only two values, $y_i = \pm 1$, Equation (11.2) describes a *binary* stochastic neuron.

To illustrate how the associate reward-penalty algorithm works, consider a sequence $\boldsymbol{x}(t)$ of patterns or stimuli drawn from a distribution of inputs, and assume that all $\boldsymbol{x}$ occur with equal probability. In each step $t = 1, 2, \ldots$ the neurons compute outputs y_i (for $i = 1, \ldots, M$) according to (3.2). The environment provides a *rein-*

forcement signal $r(\boldsymbol{x}, y) = \pm 1$, a reward ($r = 1$) with probability $p_{\text{reward}}(\boldsymbol{x}, \boldsymbol{y})$, and a penalty ($r = -1$) with probability $1 - p_{\text{reward}}(\boldsymbol{x}, \boldsymbol{y})$. The goal is to adjust the weights so that neuron produces outputs that are rewarded with high probability. To derive a *learning rule* we need a cost function. One possibility is to take an average of the immediate reward $r(\boldsymbol{x}(t), \boldsymbol{y})$. The idea then is to find a learning rule for the weight increments δw_{mn} that increases the average immediate reward [126]:

$$\langle \delta w_{mn} \rangle = \alpha \frac{\partial \langle r \rangle}{\partial w_{mn}} \tag{11.3}$$

with *learning rate*$\alpha > 0$. Here the average is over the stochastic output of the neurons, determined by Equation (11.2), over the reward distribution $p_{\text{reward}}(\boldsymbol{x}, \boldsymbol{y})$, and over the random sequence of input patterns $\boldsymbol{x}(t)$. It is assumed that the reward distribution is stationary, just like the distributions of inputs and Equation (11.2).The average immediate reward is given by

$$\langle r \rangle = \frac{1}{T} \sum_{t=1}^{T} \sum_{y_1, \ldots, y_M} \Big\langle r(\boldsymbol{x}(t), \boldsymbol{y}) P(\boldsymbol{y}; \boldsymbol{x}(t), \{w_{ij}\}) \Big\rangle_{\text{reward}} \tag{11.4}$$

where

$$P(\boldsymbol{y}; \boldsymbol{x}, \{w_{ij}\}) = \prod_{i=1}^{M} \begin{cases} p(b_i) & \text{if} \quad y_i = 1\,, \\ 1 - p(b_i) & \text{if} \quad y_i = -1 \end{cases} \tag{11.5}$$

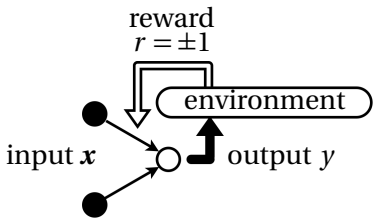
is the probability that the network produces the output $\boldsymbol{y}$ given its weights w_{ij} and the input $\boldsymbol{x}$, and $\langle \cdots \rangle_{\text{reward}}$ is an average over the response of the environment, determined by the reward distribution $p_{\text{reward}}(\boldsymbol{x}, \boldsymbol{y})$. To compute the gradient in Equation (11.3) one uses the chain rule, Equation (6.16) as well as Equation (3.8). The result is:

$$\frac{\partial P(\boldsymbol{y}; \boldsymbol{x}, \{w_{ij}\})}{\partial w_{mn}} = \beta (y_m - \langle y_m \rangle) x_n\,, \tag{11.6}$$

The calculation is similar to the one for the learning rule of Boltzmann machines (Chapter **??**). The average $\langle y_m \rangle$ in Equation (11.6) is over the stochastic output of the neurons, determined by Equation (11.2). The average is a deterministic function of $\boldsymbol{x}$ and of the weights: $\langle y_m \rangle = \tanh(\beta b_m)$ with $b_m = \sum_j w_{mj} x_j$. Comparing Equations (11.6) and (11.4) we deduce that the learning rule

$$\delta w_{mn} = \eta r_t \big[y_m(t) - \langle y_m \rangle \big] x_n(t) \tag{11.7}$$

satisifies (11.3) with $\alpha = \beta \eta$. The average in Equation (11.7) is over the stochasticity of the neurons, (11.2). It is plausible that this rule maximises the average immediate reward because it tends to increase this quantity on average, and because the weight increments tend to zero as the network learns to produce output that always



p_{reward}	$y=-1$	$y=+1$
$\boldsymbol{x}_1=[1,0]^{\mathsf{T}}$	0.6	0.8
$\boldsymbol{x}_2=[1,1]^{\mathsf{T}}$	0.3	0.1

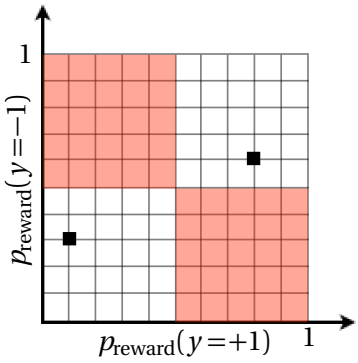


Figure 11.2: Binary stochastic neuron for reward-penalty algorithm [127]. The Table gives the reward probabilities for two different inputs or stimuli, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, with two components each. Also shown is the *contingency space* of the problem, representing inputs $\boldsymbol{x}$ in terms of the coordinates $p_{\text{reward}}(\boldsymbol{x}, y=-1)$ and $p_{\text{reward}}(\boldsymbol{x}, y=+1)$. Show plots for r_t versus t for different values of λ, show individual realisations as well as ensemble average.

produces $\max_{\boldsymbol{y}}\{p_{\text{reward}}(\boldsymbol{x},\boldsymbol{y})\}$, independent of $\boldsymbol{y}$ so that $\boldsymbol{y}-\langle\boldsymbol{y}\rangle$ averages to zero. In practice slightly different rules are found to work a bit better, for instance [1]

$$\delta w_{mn}=\eta\big[r_t\,y_m(t)-\langle y_m\rangle\big]x_n(t)\,. \tag{11.8}$$

Alternatively, asymmetric learning rules [127] tend to work well, such as:

$$\delta w_{mn}(t)=\eta\begin{cases}\big[y_m(t)-\langle y_m\rangle\big]x_n(t) & \text{if} \quad r_t=+1\,,\\ \lambda\big[-y_m(t)-\langle y_m\rangle\big]x_n(t) & \text{if} \quad r_t=-1\,.\end{cases} \tag{11.9}$$

One takes $0<\lambda<1$, so that the learning rate for $r=1$ is larger than that for $r=-1$. The learning rules (11.8) and (11.9) give the same weight increment for $r=1$, but differ in the weight updates when $r=-1$.

An example with only one binary neuron is shown in Figure 11.1. An agent encounters two different inputs or stimuli, $\boldsymbol{x}_1=[1,0]^{\mathsf{T}}$ and $\boldsymbol{x}_2=[1,1]^{\mathsf{T}}$ which occur with equal probabilities. Suppose that the optimal response to $\boldsymbol{x}_1$ is $y=1$ whereas the optimal response to $\boldsymbol{x}_2$ is $y=-1$. Corresponding reward probabilities $p_{\text{reward}}(\boldsymbol{x},y)$ are listed in Figure 11.1. If the agent had learned to always give the optimal response, then the average reward would take the maximal value

$$r_{\max}=p_{\text{reward}}(\boldsymbol{x}_1,y=+1)+p_{\text{reward}}(\boldsymbol{x}_2,y=-1)-1=0.1\,. \tag{11.10}$$

Here we used that $\langle r(\boldsymbol{x},y)\rangle=p_{\text{reward}}(\boldsymbol{x},y)-[1-p_{\text{reward}}(\boldsymbol{x},y)]$, and that $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ appear with equal propabilities.

The initial reward vanishes on average if one initialises the weights of the neuron to zero. Figure 11.1 shows simulation results for $\langle r_t \rangle$ as a function of t, for $\eta = 0.5$ and different values of λ. We see that the average reward approaches $r_{\max}$ as $t \to \infty$ and $\lambda \to 0$. In other words, the associative reward-penalty algorithm converges to the optimal solution in the limit of small λ; the algorithm maximises the average immediate reward. The resulting weight vector determines a decision boundary separating the input patterns, for instance $\boldsymbol{w} \approx [9, -20]^{\mathsf{T}}$. As expected, the first stimulus component has a positive weight for both stimuli, but the second component of the second stimulus is strongly inhibited to produce the output $y = -1$.

Comparing one single realisation of a learning curve with the ensemble average (Fig. 11.1), we see that there are significant fluctuations, in particular when λ is small.

Figure shows the *contingency space* of the problem, representing the inputs $\boldsymbol{x}$ in a plane with coordinates $p_{\text{reward}}(\boldsymbol{x}, y = +1)$ and $p_{\text{reward}}(\boldsymbol{x}, y = -1)$. It is easier to learn to associate the correct output with inputs that lie in the red regions, because $p_{\text{reward}}(y = +1) > \frac{1}{2}$ and $p_{\text{reward}}(y = -1) < \frac{1}{2}$ or vice versa. In this case one can solve the problem by fixing $y = +1$ and sampling $p_{\text{reward}}(\boldsymbol{x}, y = +1)$ for all $\boldsymbol{x}$. If $p_{\text{reward}}(\boldsymbol{x}, y = +1) > \frac{1}{2}$ then $y = +1$ is the correct output for $\boldsymbol{x}$. Otherwise it is $y = -1$. This does not work outside the red region. For example, if both reward probabilities are larger than one half, then one must sample both $p_{\text{reward}}(\boldsymbol{x}, y = -1)$ and $p_{\text{reward}}(\boldsymbol{x}, y = +1)$ sufficiently often to determine which one is larger, one must find *the greater of two goods* according to Barto [127]. This illustrates the fundamental dilemma of reinforcement learning mentioned above: an output that appears at first to yield a high reward may not be the optimal one in the long run. To find the optimal output it is necessary to estimate both reward probabilities precisely. This means that one must try all possible outputs frequently, not only the one that appears to be optimal at.

For one binary neuron the algorithm can be proven [128] to converge as $t \to \infty$ in the asymmetric limit of small λ, where the weights remain almost the same when $r = -1$. In this limit the agent learns mainly from positive feedback. In general the convergence becomes quite slow in this limit. Furthermore, the the proof assumes that the input patterns are linearly independent (page 72), and this means that the number of patterns cannot exceed N. Associative reinforcement problems with linearly dependent inputs can be solved with hidden neurons [127], just like the classification problems described in Chapter 5. However, there is no mathematical proof of convergence for nets with more than one unit. Alternatively one can increase N, embedding the input patterns in a higher-dimensional input space (Chapter 10).

In summary, we have seen how the associative reward-penalty algorithm learns from partial feedback by reinforcement. The method has several shortcomings.

First, convergence is proven only for one single binary stochastic unit, and only in the asymmetric limit $\lambda \to 0$ where convergence tends to be slow. Second, since the proof assumes that the inputs are linearly independent, the algorithm can learn to associate optimal actions with at most N inputs (where N is the input dimension). Third, the learning rule is based upon the *immediate reward* r_t at time step t. It is a model for how an animal may learn to respond in different ways to different stimuli. But when learning to win in chess, the reward is not immediate: only at the end of the game do we learn whether it is won, lost, or drawn. More generally, an agent navigating an complex environment should not only consider which immediate reward a certain action gives, but also how this action affects possible future rewards. One way of estimating future rewards is temporal difference learning, discussed next.

11.2 Temporal difference learning

Temporal difference learning is a method to predict time series from earlier observations [124]. Given a time series of observations $\boldsymbol{s}_t$ at times $t = 0, \ldots, T-1$, the task is to estimate $F(\boldsymbol{s}_T)$ where $F(\boldsymbol{s})$ is a given scalar function. For instance, one might try to predict the temperature in one weeks' time from observations of local temperature, barometric pressure, as well as wind speed and direction.

Assume that the estimator $P(\boldsymbol{s}_t, \boldsymbol{w})$ depends only on the observation $\boldsymbol{s}_t$ at time t (not on previous observations), and on a weight vector $\boldsymbol{w}$ to be optimised. In general P can be a non-linear function of the weights, and one could arrange the weights into several layers of neurons, each with one weight vector. In the simplest case one can just write a linear model for the prediction, just like in Equation (5.19):

$$P(\boldsymbol{s}_t, \boldsymbol{w}) = \boldsymbol{w} \cdot \boldsymbol{s}_t \,. \tag{11.11}$$

The weight vector $\boldsymbol{w}$ with components w_j is determined so that $P(\boldsymbol{s}_t, \boldsymbol{w})$ approximates $F(\boldsymbol{s}_T)$. This can be achieved by gradient descent with the energy function $H = \frac{1}{2}\sum_{t=0}^{T-1}[F - P(\boldsymbol{s}_t)]^2$:

$$\delta w_m = \alpha \sum_{t=0}^{T-1} [F - P(\boldsymbol{s}_t)] \frac{\partial P}{\partial w_m} \,. \tag{11.12}$$

The idea of temporal difference learning [124] is to express the error $f - P_t$ in terms of temporal differences $P(\boldsymbol{s}_{\tau+1}) - P(\boldsymbol{s}_\tau)$ by writing $F - P(\boldsymbol{s}_t) = \sum_{\tau=t}^{T-1} P(\boldsymbol{s}_{\tau+1}) - P(\boldsymbol{s}_\tau)$ with $P(\boldsymbol{s}_T) \equiv F$. Rearranging the sums one obtains the weight-update rule

$$\delta \boldsymbol{w} = \alpha \sum_{t=0}^{T-1} [P(\boldsymbol{s}_{t+1}) - P(\boldsymbol{s}_t)] \sum_{\tau=0}^{t} \boldsymbol{s}_\tau \,. \tag{11.13}$$

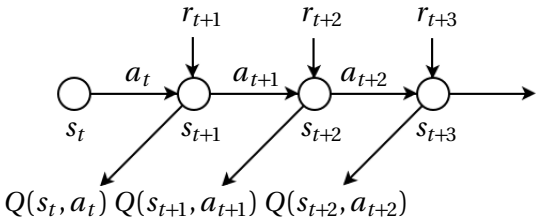


Figure 11.3: Sequence of states s_t in sequential reinforcement learning. The action a_t leads from s_t to s_{t+1} where the agent receives reinforcement r_{t+1}. The Q-matrix with elements $Q(s_t,a_t)$ estimates the future discounted reward.

A more general rule with weighting factor $0 \leq \lambda \leq 1$ reads:

$$\delta \boldsymbol{w} = \alpha \sum_{t=0}^{T-1} [P(\boldsymbol{s}_{t+1}) - P(\boldsymbol{s}_t)] \sum_{\tau=0}^{t} \lambda^{t-\tau} \boldsymbol{s}_\tau \,. \tag{11.14}$$

Smaller values of λ reduce the weight of past states in the sum. The corresponding single-step weight update is:

$$\delta \boldsymbol{w}_t = \alpha [P(\boldsymbol{s}_{t+1}) - P(\boldsymbol{s}_t)] \sum_{\tau=0}^{t} \lambda^{t-\tau} \boldsymbol{s}_\tau \,. \tag{11.15}$$

This learning rule is called temporal difference rule, denoted by TD(λ). Substituting $\lambda = 0$ yields a learning rule that depends only on the present state:

$$\delta \boldsymbol{w}_t = \alpha [P(\boldsymbol{s}_{t+1}) - P(\boldsymbol{s}_t)] \boldsymbol{s}_t \,. \tag{11.16}$$

This is just Hebb's rule (6.7) with target $P(\boldsymbol{s}_{t+1})$. This choice of target allows to learn one-step prediction of the time series.

Temporal difference learning allows a machine to learnig to play backgammon [129], using a deep layered network and backpropagation (Section 6.1) to determine the weights.

11.3 Sequential reinforcement learning

In sequential reinforcement learning we distinguish *episodic* from *continuous* tasks. In episodic tasks, the learning problem is divided into *episodes* during which an agent visits a finite sequence of T states, $\boldsymbol{s}_0, \ldots, \boldsymbol{s}_{T-1}$, and collects the rewards $r_1, \ldots, r_T$. The question is how to estimate the average of the future reward

$$R_t = \sum_{\tau=t}^{T-1} r_{\tau+1} \,. \tag{11.17}$$

associated with a certain move. Games are examples of episodic tasks, each round corresponds to an episode. Often the number of steps per round (episode length T) varies from round to round. In order to estimate the expected reward one usually averages over many episodes.

Continuous tasks, by contrast, do not have a defined end point. Since the sum in (11.17) might diverge as $T \to \infty$, it is customary to introduce a weighting factor $0 < \gamma \leq 1$ in the sum:

$$R_t = \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_{\tau+1} \,. \tag{11.18}$$

The weighting factor reduces the contribution of the far future to the estimate. Smaller values of γ is give more weight to the immediate future. The sum in Equation (11.18) is called *future discounted reward*.

Let us now discuss how temporal difference learning can allow an agent to navigate a complex enviroment in an optimal fashion. To optimise the expected future reward (11.17). we need to consider how the agent explores state space. As mentioned above this is affected by the reinforcement signal at a given time, by the current state of the agent, $\boldsymbol{s}_t$, by possible actions, and by the policy [2] that determines what the agent should do. For a given state, the agent may for instance always take the action that maximises the reward (*greedy* policy). The ε-greedy policy does mainly that, but with a small probability ε it takes a suboptimal action.

We follow the steps outlined in Section 11.2, but we shall see that the resulting learning rule is slightly different because the target R_t to estimate is a sum over t. We denote the estimator of R_t by $P(\boldsymbol{s}_t)$. As before one expresses the error $R_t - P(\boldsymbol{s}_t)$ as a sum of temporal differences, using $\sum_{\tau=t}^{T-1} r_{\tau+1} - P(\boldsymbol{s}_t) = \sum_{\tau=t}^{T-1} [r_{\tau+1} - P(\boldsymbol{s}_{\tau+1}) - P(\boldsymbol{s}_\tau)]$. Using the gradient-descent rule (11.12) one obtains

$$\delta \boldsymbol{w}_t = \alpha [r_{t+1} + P(\boldsymbol{s}_{t+1}) - P(\boldsymbol{s}_t)] \boldsymbol{s}_t \,. \tag{11.19}$$

This rule corresponds to the following update rule for P:

$$P_{t+1}(\boldsymbol{s}_t) = P_t(\boldsymbol{s}_t) + \alpha [r_{t+1} + P_t(\boldsymbol{s}_{t+1}) - P_t(\boldsymbol{s}_t)] \,. \tag{11.20}$$

The notation P_t emphasises that estimator is updated recursively. The update rule (11.20) applies to estimating the future reward (11.17) for episodic tasks. The corresponding rule for estimating the future discounted reward (11.18) for continuous tasks reads

$$P_{t+1}(\boldsymbol{s}_t) = P_t(\boldsymbol{s}_t) + \alpha [r_{t+1} + \gamma P_t(\boldsymbol{s}_{t+1}) - P_t(\boldsymbol{s}_t)] \,. \tag{11.21}$$

When the sequence $\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{s}_2, \ldots$ is a Markov chain (Section 4.4), then the rule (11.21) can be shown to converge if one uses a time-dependent learning rate α_t that satisfies

$$\sum_{t=0}^{\infty}\alpha_t=\infty \quad \text{and} \quad \sum_{t=0}^{\infty}\alpha_t^2<\infty\,. \tag{11.22}$$

Returning to the problem outlined in the beginning of this Chapter, consider an agent exploring a complex environment. The task might be to get from location A to location B as quickly as possible, or expending as little energy as possible. At time t the agent is at position $\boldsymbol{x}_t$, it may have velocity $\boldsymbol{v}_t$. These variables as well as the local state of the environment are summarised in the state vector $\boldsymbol{s}_t$. Given $\boldsymbol{s}_t$ the agent can act in certain ways: it can slow down, speed up, or turn for example. These possible actions are summarised in a vector $\boldsymbol{a}_t$. At each time step, the agent take the action $\boldsymbol{a}_t$ that optimises the expected future discounted reward (11.18), given its present state $\boldsymbol{s}_t$. The estimated expected future reward for any given state-action pair is summarised in a matrix, the *Q-matrix* with elements $Q(\boldsymbol{s}_t,\boldsymbol{a}_t)$, the analogue of P in the previous Section. Different rows of the Q-matrix correspond to different states, and different columns to different actions. We use temporal difference learning with $\lambda=0$ to determine Q:

$$Q_{t+1}(\boldsymbol{s}_t,\boldsymbol{a}_t)=Q_t(\boldsymbol{s}_t,\boldsymbol{a}_t)+\alpha_t\big[r_{t+1}+\gamma Q_t(\boldsymbol{s}_{t+1},\boldsymbol{a}_{t+1})-Q_t(\boldsymbol{s}_t,\boldsymbol{a}_t)\big]. \tag{11.23}$$

This algorithm is called SARSA, because one needs $\boldsymbol{s}_t,\boldsymbol{a}_t,r_{t+1},\boldsymbol{s}_{t+1}$, and $\boldsymbol{a}_{t+1}$ to update the Q-matrix (Figure 11.3). It is commonly assumed that the algorithm converges if all state-action pairs $[\boldsymbol{s}_t,\boldsymbol{a}_t]$ are visited sufficiently often. However, there is no general convergence proof. The difficulty is that the update rule depends not only on the present state-action pair $[\boldsymbol{s}_t,\boldsymbol{a}_t]$, but also on the next action $\boldsymbol{a}_{t+1}$, and thus indirectly upon the policy. Sometimes this is indicated by writing Q_π for the Q-matrix given policy π. In general the policy can change as the algorithm is iterated.

11.4 Q-learning

The one-step Q-learning rule [130] is an approximation to Eq. (11.23) that does not depend on $\boldsymbol{a}_{t+1}$. Instead one assumes that the next action is the optimal one

$$Q_{t+1}(\boldsymbol{s}_t,\boldsymbol{a}_t)=Q_t(\boldsymbol{s}_t,\boldsymbol{a}_t)+\alpha_t\big[r_{t+1}+\gamma\max_{\boldsymbol{a}}Q(\boldsymbol{s}_{t+1},\boldsymbol{a})-Q(\boldsymbol{s}_t,\boldsymbol{a}_t)\big], \tag{11.24}$$

regardless of the policy that is currently followed. For the greedy policy Eq. (11.24) is equivalent to (11.23), but in general the two algorithms differ.

The Q-learning algorithm is summarised in Algorithm 13. Now assume that the agent is in state $\boldsymbol{s}_t$. Given $Q(\boldsymbol{s}_t,\boldsymbol{a}_t)$ the agent chooses an action $\boldsymbol{a}_t$ according to a given policy. According to the ε-greedy policy it chooses the action $\boldsymbol{a}_t$ that yields

Algorithm 13 Q-learning

initialise $Q(\boldsymbol{s}_0, \boldsymbol{a}_0)$;
for $n = 1, \ldots, N$ **do**
  initialise $\boldsymbol{s}_0$;
  for $t = 0, \ldots, T$ **do**
    choose $\boldsymbol{a}_t$ from $Q(\boldsymbol{a}_t, \boldsymbol{s}_t)$ according to ε-greedy policy;
    compute $\boldsymbol{s}_{t+1}$ and record r_{t+1};
    update $Q(\boldsymbol{s}_t, \boldsymbol{a}_t) \leftarrow Q(\boldsymbol{s}_t, \boldsymbol{a}_t) + \alpha[r_{t+1} + \gamma \max_{\boldsymbol{a}} Q(\boldsymbol{s}_{t+1}, \boldsymbol{a}) - Q(\boldsymbol{s}_t, \boldsymbol{a}_t)]$;
  end for
end for
end;

the largest Q given $\boldsymbol{s}_t$ with probability $1-\varepsilon$. With probability ε it randomly picks an action. The choice of action $\boldsymbol{a}_t$ determines the next state $\boldsymbol{s}_{t+1}$, and this in turn allows to update the Q-matrix using Eq. (11.24).

Let us see how this works for a very simple example, the associative task described in Fig. 11.1. One episode corresponds to computing the output of the neuron given its initial state $\boldsymbol{s}_0$, so $T = 1$. There is no sequence of states, and the task is to estimate the immediate reward. In this case the update rule (11.24) simplifies to

$$Q_{t+1}(\boldsymbol{s}_0, a_0) = Q_t(\boldsymbol{s}_0, a_0) + \alpha_t \big[r - Q(\boldsymbol{s}_0, a_0) \big], \tag{11.25}$$

analogous to (5.26). There are only two states in this problem $\boldsymbol{s}_0 = \boldsymbol{x}_1$ and $\boldsymbol{s}_0 = \boldsymbol{x}_2$, and two actions, $a_0 = \pm 1$. The steady-state condition for the learning rule is $\langle r - Q^*(\boldsymbol{s}_0, a_0) \rangle = 0$, where $\langle \cdots \rangle$ is the average over many episodes, that is realisations of $\boldsymbol{s}_0$ and a_0. This means that

$$\begin{bmatrix} Q^*_{11} & Q^*_{12} \\ Q^*_{21} & Q^*_{22} \end{bmatrix} = \begin{bmatrix} 2p_{\text{reward}}(\boldsymbol{x}_1, y = -1) & 2p_{\text{reward}}(\boldsymbol{x}_1, y = +1) \\ 2p_{\text{reward}}(\boldsymbol{x}_2, y = -1) & 2p_{\text{reward}}(\boldsymbol{x}_2, y = +1) \end{bmatrix} = \begin{bmatrix} 0.2 & 0.6 \\ -0.4 & -0.8 \end{bmatrix} \tag{11.26}$$

for the task described in Figure 11.1. Figure 11.4 demonstrates how the algorithm (11.25) approaches the maximal immediate reward $r_{\max}$. Panel (a) shows 100 realisations of the learning process together with the average, for...

What is the role of the term $\gamma \max_{\boldsymbol{a}} Q(\boldsymbol{s}_{t+1}, \boldsymbol{a})$ in the Q-learning rule (11.24)?

Problems. Slow convergence. How to determine suitable states and/or actions? Need to discretise. Use plankton as an example. Curse of dimension. Deep Q-learning [8].

Figure 11.4: Q-learning for the task described in Figure 11.1. Three panels. Plot reward as a function of time for 100 realisations (grey) and average reward for $\varepsilon = 0.2$, $\alpha_t = t^{-1}$, $\varepsilon = 0.2$, $\alpha_t = 5t^{-1}$, and $\varepsilon = 0.7$, $\alpha_t = 5t^{-1}$. What do we want to illustrate with these parameter choices?

11.5 Tic-tac-toe

11.6 Summary

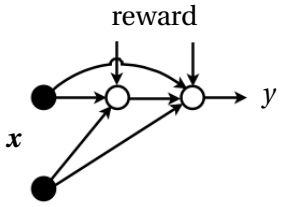
In Chapters 5 to 8 we discusse how to train networks to correctly associate input/output pairs $(\boldsymbol{x}^{(\mu)}, \boldsymbol{t}^{(\mu)})$ in a training set (*supervised learning*). Chapters 9 and 10 described *unsupervised* learning where the network learns without any feedback. The reinforcement algorithm introduced in the present Chapter is in between. It learns from incomplete feedback. Not all targets are known to the network. The feedback may simply be $r = \pm 1$ (good or bad solution). The associative reward-penalty algorithm uses this feedback to build up a training set that allows the network to learn.

11.7 Further reading

Reinforcement learning has developed substantially in recent years [125, 131, 132]. Many recent studies on reinforcement learning use the Q-learning algorithm [125, 132]. A recent application of reinforcement learning is AlphaGo, an algorithm that learnt to play the game of Go. How it works is described in this blog.

11.8 Exercises

11.1 Binary stochastic neurons. A layer has binary stochastic neurons y_i with update rule $y_i = +1$ with probability $p(b_i)$ and $y_i = -1$ otherwise. Here $b_i = \sum_j w_{ij} x_j$, and $p(b) = (1 + \mathrm{e}^{-2\beta b})^{-1}$. The parameter β^{-1} is the noise level, x_j are inputs, and w_{ij} are weights. Consider the the energy function $H = \frac{1}{2}\sum_{i\mu}\left(t_i^{(\mu)} - y_i^{(\mu)}\right)^2$ with targets $t_i^{(\mu)} = \pm 1$. Show that the average of y_i equals $\langle y_i^{(\mu)} \rangle = \tanh\left(\beta\, b_i^{(\mu)}\right)$. Stochastic neurons can be trained by gradient descent on the energy function $H' = \frac{1}{2}\sum_{i\mu}\left(t_i^{(\mu)} - \langle y_i^{(\mu)} \rangle\right)^2$, defined in terms of the average outputs $\langle y_i^{(\mu)} \rangle$. The error $\delta_m^{(\mu)}$ is defined by $\delta\, w_{mn} \equiv -\eta \frac{\partial H'}{\partial w_{mn}} = \eta \sum_\mu \delta_m^{(\mu)} x_n^{(\mu)}$. Show that $\delta_m^{(\mu)} = (t_m^{(\mu)} - \langle y_m^{(\mu)} \rangle)\beta(1 - \langle y_m^{(\mu)} \rangle^2)$. Compare this result for $\delta_m^{(\mu)}$ with the update rule for standard output neurons with activation func-



p_{reward}	$y=-1$	$y=+1$
$\boldsymbol{x}_1=[0,0]^\mathsf{T}$	0.8	0.1
$\boldsymbol{x}_2=[0,1]^\mathsf{T}$	0.1	0.8
$\boldsymbol{x}_3=[1,0]^\mathsf{T}$	0.1	0.8
$\boldsymbol{x}_4=[1,1]^\mathsf{T}$	0.8	0.1

Figure 11.5: Stochastic XOR problem [127].

Figure 11.6: Three-armed bandit problem [131]. Three slot machines have Gaussian return distributions shown, with means and standard deviations $\mu_1=5,\sigma_1=1$, $\mu_2=1,\sigma_2=10$, and $\mu_3=2,\sigma_3=4$. Shifted Gaussians. Analyse convergence for different values of ε.

tion $g(b)$ and energy function (5.23): explain differences and similarities between the two expressions.

11.2 Gradient of average reward. Derive Equation (11.6).

11.3 Klopf's self-interested neuron. Klopf's self-interested neuron [133] is a binary stochastic neuron with outputs 0 and 1. Derive a learning rule that is equivalent to Eq. (11.7).

11.4 Associate reward-penalty algorithm. Barto [127] explains how to solve association tasks with linearly dependent inputs using hidden neurons. One of his examples is illustrated in Fig. 11.5. Using the network shown in Fig. 11.5, implement the associate reward-penalty algorithm and analyse its convergence for the XOR task. Then verify that the task can be solved by a single stochastic neuron if you embed the input data in a four-dimensional input space.

11.5 Three-armed bandit problem. Implement the Q-learning algorithm for the three-armed bandit problem with reward distributions described in Fig. 11.6. Goal: exploitation vs. exploration.

11.6 Tic tac toe. Describe OpenTA task. Discuss symmetries.

Bibliography

[1] HERTZ, J, KROGH, A & PALMER, R 1991 *Introduction to the Theory of Neural Computation*. Addison-Wesley.

[2] HAYKIN, S 1999 *Neural Networks: a comprehensive foundation*, 2nd edn. New Jersey: Prentice Hall.

[3] HORNER, H, Neuronale Netze, [Online; accessed 8-November-2018].

[4] GOODFELLOW, I, BENGIO, Y & COURVILLE, A, Deep Learning, [Online; accessed 5-September-2018].

[5] NIELSEN, M, Neural Networks and Deep Learning, [Online; accessed 13-August-2018].

[6] LECUN, A, BENGIO, Y & HINTON, G 2015 Deep learning. *Nature* **521**, 463.

[7] MACHEREY, W, KRIKUN, M, CAO, Y, GAO, Q, MACHEREY, K, KLINGNER, J, SHAH, A, JOHNSON, M, LIU, X, KAISER, L, GOUWS, S, KATO, Y, KUDO, T, KAZAWA, H, STEVENS, K, KURIAN, G, PATIL, N, WANG, W, YOUNG, C, SMITH, J, RIESA, J, RUDNICK, A, VINYALS, O, CORRADO, G, HUGHES, M & DEAN, J 2016 Google's neural machine translation system: bridging the gap between Human and machine translation arxiv:1609.08144.

[8] DEEPMIND, AlphaGo, [Online; accessed: 20-August-2018].

[9] BRAIN MAPS, An interactive multiresolution brain atlas; SMI32-immunoreactive pyramidal neuron in medial prefrontal cortex of macaque, brainmaps.org, [Online; accessed 14-August-2018].

[10] GABBIANI, F & METZNER, W 1999 Encoding and processing of sensory information in neuronal spike trains. *Journal of Experimental Biology* **202** (10), 1267.

[11] MCCULLOCH, W & PITTS, W 1943 A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys.* **5**, 115.

[12] ROSENBLATT, F 1958 A probabilistic model for information storage and organization in the brain. *Psychological Review* **65**, 386.

[13] ROSENBLATT, F 1958 The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Rev.* **65**, 386.

[14] LITTLE, W 1974 The existence of persistent states in the brain. *Mathematical Biosciences* **19**, 101 – 120.

[15] HOPFIELD, J. J, Hopfield network, [Online; accessed 14-August-2018].

[16] FISCHER, A & IGEL, C 2014 Training restricted Boltzmann machines: An introduction. *Pattern Recognition* **47** (1), 25–39.

[17] LIPPMANN, R. P 1987 An introduction to computing with neural nets **3**, 9–44.

[18] MATHEWS, J & WALKER, R. L 1964 *Mathematical Methods of Physics*. New York: W.A. Benjamin.

[19] WolframMathWorld, [Online; accessed 17-September-2019].

[20] KADANOFF, L. P, More is the same: phjase transitions and mean field theories, https://arxiv.org/abs/0906.0653, [Online; accessed 3-September-2020].

[21] HOPFIELD, J. J 1982 Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences* **79** (8), 2554–2558.

[22] HOPFIELD, J. J 1984 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.

[23] MÜLLER, B, REINHARDT, J & STRICKLAND, M. T 1999 *Neural Networks: An Introduction*. Heidelberg: Springer.

[24] AMIT, D. J, GUTFREUND, H & SOMPOLINSKY, H 1985 Spin-glass models of neural networks. *Phys. Rev. A* **32**, 1007.

[25] GESZTI, T 1990 *Physical models of neural networks*. World Scientific.

[26] AMIT, D. J & GUTFREUND, H 1987 Statistical mechanics of neural networks near saturation. *Ann. Phys.* **173**, 30.

[27] ZIRNBAUER, M 1994 Another critique of the replica trick arxiv:cond-mat/9903338.

[28] STEFFAN, H & KUEHN, R 1994 Replica symmetry breaking in attractor neural network models arxiv:cond-mat/9404036.

[29] VOLK, D 1998 On the phase transition of Hopfield networks – another Monte Carlo study. *Int. J. Mod. Phys. C* **9**, 693.

[30] LÖWE, M 1998 In the storage capcaity of Hopfield models with correlated patterns. *Ann. Prob.* **8**, 1216.

[31] KIRKPATRICK, S, GELATT, C. D & VECCHI, M. P 1983 Optimization by simulated annealing. *Science* **220**, 671–680.

[32] POTVIN, J. Y & SMITH, K. A 1999 Artificial neural networks for combinatorial optimization. In *Handbook of Metaheuristics* (ed. G F. & G Kochenberger). Heidelberg: Springer.

[33] MANDZIUK, J 2002 Neural networks for the n-Queens problem: a review. *Control and Cybernetics* **31**, 217, Special issue on neural networks for optimization and control.

[34] WATERMAN, M 1995 *Introduction to Bioinformatics*. Prentice Hall.

[35] HOPFIELD, J. J & TANK, D. W 1985 Neural computation of decisions in optimisation problems. *Biol. Cybern.* **52**, 141.

[36] PRESS, W, TEUKOLSKY, S, VETTERLING, W & FLANNERY, W 1992 *Numerical Recipes in C: The Art of Scientific Computing, second edition*. New York: Cambridge University Press.

[37] KAMPEN, N. V 2007 *Stochastic processes in physics and chemistry*. North Holland.

[38] MEHLIG, B, HEERMANN, D. W & FORREST, B. M 1992 Hybrid Monte Carlo method for condensed-matter systems. *Phys. Rev. B* **45**, 679–685.

[39] BINDER, K, ed. 1986 *Monte Carlo Methods in Statistical Physics*. Heidelberg: Springer.

[40] NEWMAN, M. E. J & BARKEMA, G. T 1999 *Monte Carlo Methods in Statistical Physics*. Oxford: Clarendon Press.

[41] SALAMON, P, SIBANI, P & FROST, R 2002 *Facts, conjectures, and improvements for simulated annealing*. SIAM.

[42] MACHINE LEARNING REPOSITORY UNIVERSITY OF CALIFORNIA IRVINE, archive.ics.uci.edu/ml, [Online; accessed 18-August-2018].

[43] FISHER, R. A 1936 The use of multiple measurements in taxonomic problems. *Ann. Eugenics* **7**, 179.

[44] MINSKY, M & PAPERT, S 1969 *Perceptrons. An Introduction to Computational Geometry*. MIT Press.

[45] KANAL, L. N 2001 Perceptrons. In *International Encyclopedia of the Social and Behavioral Sciences*.

[46] GREUB, W 1981 *Mathematical Methods of Physics*. New York: Springer.

[47] LECUN, Y, BOTTOU, L, ORR, G. B & MÜLLER, K.-R 1998 Efficient back prop. In *Neural networks: tricks of the trade* (ed. G. B Orr & K.-R Müller). Springer.

[48] NESTEROV, Y 1983 A method of solving a convex programming problem with convergence rate o$(1/k^2)$. *Soviet Mathematics Doklady* **27**, 372.

[49] SUTSKEVER, I 2013 Training recurrent neural networks. PhD thesis, University of Toronto, [Online; accessed 27-October-2018].

[50] HANSON, S. J & PRATT, L. Y 1989 Comparing biases for minimal network construction with backpropagation. In *Advances in Neural Information Processing Systems 1*.

[51] KROGH, A & HERTZ, J. A 1992 A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems 4*.

[52] HASSIBI, B & G.STORK, D 1993 Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems 5*.

[53] LECUN, Y, DENKTER, J. S & SOLLA, S 1990 Optimal brain damage. In *Advances in Neural Information Processing Systems 2* (ed. D. S Touretzky), p. 598.

[54] RUMELHART, D. E, HINTON, G. E & WILLIAMS, R. J 1986 Learning internal representations by error propagation. In *Parallel distributed processing: explorations in the microstructure of cognition* (ed. D. E Rumelhart & J. L MCClelland).

[55] HORNIK, K, STINCHOME, M & WHITE, H 1989 Neural networks are universal approximators. *Neural Networks* **2**, 359.

[56] CVITANOVIC, P, ARTUSO, G, MAINIERI, R, TANNER, G & VATTAY, G, chaosbook.org/version15 (Niels Bohr Institute, Copenhagen 2015), Lyapunov exponents, [Online; accessed 30-September-2018].

[57] PENNINGTON, J, SCHOENHOLZ, S. S & GANGULI, S 2017 Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. In *Advances in Neural Information Processing Systems 30*.

[58] GLOROT, X, BORDES, A & BENGIO, Y 2011 Deep sparse rectifier neural networks. In *Proceedings of Machine Learning Research*.

[59] SUTSKEVER, I, MARTENS, J, DAHL, G & HINTON, G 2013 On the importance of initialization and momentum in deep learning ACM Digital Library.

[60] GLOROT, X & BENGIO, Y 2010 Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of Machine Learning Research*.

[61] SRIVASTAV, N, HINTON, G, KRISHEVSKY, A, SUTSKEVER, I & SALKHUTDINOV, R 2014 Dropout: A simple way to prevent neural networks from overfitting Journal of Machine Learning Research.

[62] FRANKLE, J & CARBIN, M 2018 The lottery ticket hypothesis: Finding small, trainable neural networks arxiv:1803.03635.

[63] IMAGENET, image-net.org, [Online; accessed 3-September-2018].

[64] IOFFE, S & SZEGEDY, C 2015 Batch normalization: Accelerating deep network training by reducing internal covariate shift arxiv:1502.03167.

[65] SANTURKAR, S, TSIPRAS, D, ILYAS, A & MADRY, A 2018 How does batch normalization help optimization? (No, it is not about internal covariate shift) arxiv:1805.11604.

[66] KRIZHEVSKY, A, SUTSKEVER, I & HINTON, G. E 2012 Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*.

[67] TENSORFLOW, tensorflow.org, [Online; accessed 3-September-2018].

[68] LECUN, Y & CORTES, C, MNIST, [Online; accessed 3-September-2018].

[69] KERAS, Keras, [Online; accessed 26-September-2018].

[70] THEANO, Theano, [Online; accessed 26-September-2018].

[71] PYTORCH, PyTorch, [Online; accessed 28-October-2018].

[72] SMITH, L. N 2015 Cyclical learning rates for training neural networks arxiv:1506.01186.

[73] CIRESAN, D, MEIER, U & SCHMIDHUBER, J 2012 Multi-column deep neural networks for image classification arxiv:1202.2745.

[74] PICASSO, J. P, Pre-processing before digit recognition for nn and cnn trained with mnist dataset, stackexchange, [Online; accessed 26-September-2018].

[75] KOZIELSKI, M, FORSTER, J & NEY, H 2012 Moment-based image normalization for handwritten text recognition. In *Proceedings of the 2012 International Conference on Frontiers in Handwriting Recognition*.

[76] PASCAL VOC DATA SET, Pascal VOC, [Online; accessed 6-September-2018].

[77] REDMON, J, DIVVALA, S, GIRSHICK, R & FARHADI, A, You only look once: Unified, real-time object detection, arxiv:1506.02640.

[78] RUSSAKOVSKY, O, DENG, J, SU, H, KRAUSE, J, SATHEESH, S, MA, S, HUANG, Z, KARPATHY, A, KHOSLA, A, BERNSTEIN, M, BERG, A. C & FEI-FEI, L 2014 Imagenet large scale visual recognition challenge arxiv:1409.0575.

[79] HE, K, ZHANG, X, REN, S & SUN, J 2015 Deep residual learning for image recognition arxiv:1512.03385.

[80] LI, F, JOHNSON, J & YEUNG, S, CNN architectures, CNN architectures, [Online; accessed 23-September-2018].

[81] HU, J, SHEN, L & SUN, G 2018 Squeeze-and-excitation networks arxiv:1709.01507.

[82] SEIF, G, Deep learning for image recognition: why it's challenging, where we've been, and what's next, Towards Data Science, [Online; accessed 26-September-2018].

[83] Tensor processing unit, github.com/tensorflow/tpu, [Online; accessed 23-September-2018].

[84] SZEGEDY, C, LIU, W, JIA, Y, SERMANET, P, REED, S, ANGUELOV, D, ERHAN, D, VANHOUCKE, V & RABINOVICH, A 2014 Going deeper with convolutions arxiv:1409.4842.

[85] ZENG, X, OUYANG, W, YAN, J, LI, H, XIAO, T, WANG, K, LIU, Y, ZHOU, Y, YANG, B, WANG, Z, ZHOU, H & WANG, X 2016 Crafting GBD-net for object detection arxiv:1610.02579.

[86] HERN, A, Computers now better than humans at recognising and sorting images, The Guardian, [Online; accessed 26-September-2018].

[87] KARPATHY, A, What I learned from competing against a convnet on imagenet, blog, [Online; accessed 26-September-2018].

[88] KHURSHUDOV, A, Suddenly, a leopard print sofa appears, [Online; accessed 23-August-2018].

[89] KARPATHY, A, ILRSVC labeling interface, [Online; accessed 26-September-2018].

[90] GEIRHOS, R, TEMME, C. R. M, RAUBER, J, SCHÜTT, H. H, BETHGE, M & WICHMANN, F. A 2018 Generalisation in Humans and deep neural networks arxiv:1808.08750.

[91] SZEGEDY, C, ZAREMBA, W, SUTSKEVER, I, BRUNA, J, ERBAN, D. A ND GOODFELLOW, I & FERGUS, R 2013 Intriguing properties of neural networks arxiv:1312.6199.

[92] NGUYEN, A, YOSINSKI, J & CLUNE, J 2015 Deep neural networks are easily fooled: high confidence predictions for unrecognisable images arxiv:1412.1897.

[93] YOSINSKI, J, CLUNE, J, NGUYEN, A, FUCHS, T & LIPSON, H 2015 Understanding neural networks through deep visualization arxiv:1506.06579.

[94] CHOROMANSKA, A, HENAFF, M, MATHIEU, M, BEN AROUS, G & LECUN, Y 2014 The loss surfaces of multilayer networks arxiv:1412.0233.

[95] KIRKPATRICK, J, PASCANU, R, RABINOWITZ, N, VENESS, J, DESJARDINS, G, RUSU, A. A, MILAN, K, QUAN, J, RAMALHO, T, GRABSKA-BARWINSKA, A, HASSABIS, D, CLOPATH, C, KUMARAN, D & HADSELL, R 2016 Overcoming catastrophic forgetting in neural networks arxiv:1612.00796.

[96] SETTLES, B 2009 Active Learning Literature Survey. *Tech. Rep.* 1648. University of Wisconsin–Madison.

[97] SUTSKEVER, I, VINYALS, O & LE, Q. V 2014 Sequence to sequence learning with neural networks arxiv:1409.3215.

[98] LIPTON, Z. C, BERKOWITZ, J & ELKAN, C 2015 A critical review of recurrent neural networks for sequence learning arxiv:1506.00019.

[99] PASCANU, R, MIKOLOV, T & BENGIO, Y 2012 On the difficulty of training recurrent neural networks arxiv:1211.5063.

[100] HOCHREITER, S & SCHMIDHUBER, J 1997 Long short-term memory. *Neural Computation* **9**, 1735.

[101] PAPINENI, K, ROUKOS, S, WARD, T & ZHU, W.-J 2002 Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, p. 311.

[102] LUKOSEVICIUS, M & JAEGER, H 2009 Reservoir computing approaches to recurrent neural network training. *Computer Science Review* **3**, 127.

[103] SCHRAUWEN, B, VERSTRAETEN, D & VAN CAMPENHOUT, J 2007 An overview of reservoir computing: theory, applications and implementations. In *ESANN2007 Proceedings - Europen Symposium on Artificial Neural Networks*.

[104] TANAKA, G, YAMANE, T, HÉROUX, J. B, NAKANE, R, KANAZAWA, N, TAKEDA, S, NUMATA, H, NAKANO, D & HIROSE, A 2019 Recent advances in physical reservoir computing: A review. *Neural Networks* **115**, 100 – 123.

[105] PATHAK, J, HUNT, B, GIRVAN, M, LU, Z & OTT, E 2018 Model-free prediction of large spatiotemporally chaotic systems from data: A reservoir computing approach. *Phys. Rev. Lett.* **120**, 024102.

[106] DOYA, K 1993 Bifurcations of recurrent neural networks in gradient descent learning. *IEEE Transactions on Neural Networks* **1**, 75.

[107] KARPATHY, A, The unreasonable effectiveness of recurrent neural networks, webpage, [Online; accessed 4-October-2018].

[108] WILKINSON, M, BEZUGLYY, V & MEHLIG, B 2009 Fingerprints of random flows? *Phys. Fluids* **21**, 043304.

[109] OJA, E 1982 A simplified neuron model as a principal component analyzer. *J. Math. Biol.* **15**, 267.

[110] RITTER, H & SCHULTEN, K 1986 On the stationary state of kohonen's self-organizing sensory mapping. *Biological Cybernetics* **54**, 99–106.

[111] ERWIN, E, OBERMAYER, K & SCHULTEN, K 1992 Self-organizing maps: ordering, convergence properties and energy functions. *Biological Cybernetics* **67**, 47–55.

[112] KOHONEN, T 2013 Essentials of the self-organizing map. *Neural Networks* **37**, 52 – 65.

[113] MARTIN, R & OBERMAYER, K 2009 Self-organizing maps. In *Encyclopedia of Neuroscience* (ed. L. R Squire), p. 551. Oxford: Academic Press.

[114] SNYDER, W, NISSMAN, D, VAN DEN BOUT, D & BILGRO, G 1990 Kohonen networks and clustering: Comparative performance in color clustering. In *Advances in Neural Information Processing Systems 3*.

[115] PRITCHARD, J. K, STEPHENS, M & DONNELLY, P 2000 Inference of population structure using multilocus genotype data. *Genetics* **155**, 945.

[116] Human genome diversity project, HGDP, [Online; accessed 10-October-2018].

[117] COVER, T. M 1965 Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Trans. on electronic computers* p. 326.

[118] SOMPOLINSKY, H, Introduction: the perceptron, The perceptron, [Online; accessed 9-October-2018].

[119] WETTSCHERECK, D & DIETTERICH, T 1992 Improving the performance of radial basis function networks by learning center locations. In *Advances in Neural Information Processing Systems 4*, pp. 1133–1140. Morgan Kaufmann.

[120] POGGIO, T & GIROSI, F 1990 Networks for approximation and learning. In *Proceedings of the IEEE*, , vol. 78, p. 1481.

[121] SUTTON, R. S & BARTO, A. G 2018 *Reinforcement Learning: An Introduction*, 2nd edn. The MIT Press.

[122] BARTO, A. G & DIETTERICH, T. G 2004 *Reinforcement Learning and Its Relationship to Supervised Learning*, pp. 45–63. Wiley-IEEE Press.

[123] COLABRESE, S, GUSTAVSSON, K, CELANI, A & BIFERALE, L 2017 Flow navigation by smart microswimmers via reinforcement learning. *Phys. Rev. Lett.* **118**, 158004.

[124] SUTTON, R. S 1988 Learning to predict by the methods of temporal differences. *Machine Learning* **3**, 9–44.

[125] SZEPESVARI, C 2010 Algorithms for reinforcement learning. In *Synthesis Lectures on Artificial Intelligence and Machine Learning* (ed. R. J Brachmann & T Dietterich). Morgan and Claypool Publishers.

[126] WILLIAMS, R. J 1992 Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* **8**, 229–256.

[127] BARTO, A. G 1985 Learning by statistical cooperation of self-interested neuron-like computing elements. *Hum. Neurobiol.* **4**, 229–56.

[128] BARTO, A. G & ANANDAN, P 1985 Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernectis* **15**, 360–375.

[129] MCCLELLAND, J. L 2015 *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises*. New Jersey: Prentice Hall, [Online; accessed 19-November-2019].

[130] WATKINS, C. J. C. H 1989 Learning from delayed rewards. PhD thesis, University of Cambridge, [Online; accessed 25-December-2019].

[131] SUTTON, R. S, ed. 1992 *A special issue of machine learning on reinforcement learning, reprinted from Machine Learning Vol. 8, Nos. 3–4*. Springer.

[132] WIERING, M & VAN OTTERLO, M 1999 *Reinforcement learning*. Heidelberg: Springer.

[133] KLOPF, A. H 1982 *The Hedonistic Neuron: Theory of Memory, Learning and Intelligence*. Taylor and Francis.

Index