Introduction to Neural Networks

MATTIAS WAHDE

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2007

Introduction to Neural Networks

MATTIAS WAHDE

Department of Applied Mechanics
Chalmers University of Technology
SE–412 96 Göteborg
Sweden
Telephone: +46 (0)31–772 1000

Contents

Chapter 1

Introduction

The study of artificial neural networks, whose history goes back to the early 1940s, is to a great extent inspired by biological considerations. Therefore, it is suitable to begin this chapter with a very brief introduction to biological neural networks.

1.1 Biological background

The elementary computational unit of a neural network is the **neuron**, which consists, essentially, of a cell body and wires connecting the neuron to other neurons. A schematic picture of a biological neuron is shown in Fig. 1.1. Starting from the cell body, a long filament known as the **axon** extends toward other neurons. This is the output wire of the neuron.

At some distance away from the cell body, the axon splits up in a delta of smaller wires which end on **synapses** that form the connections with other

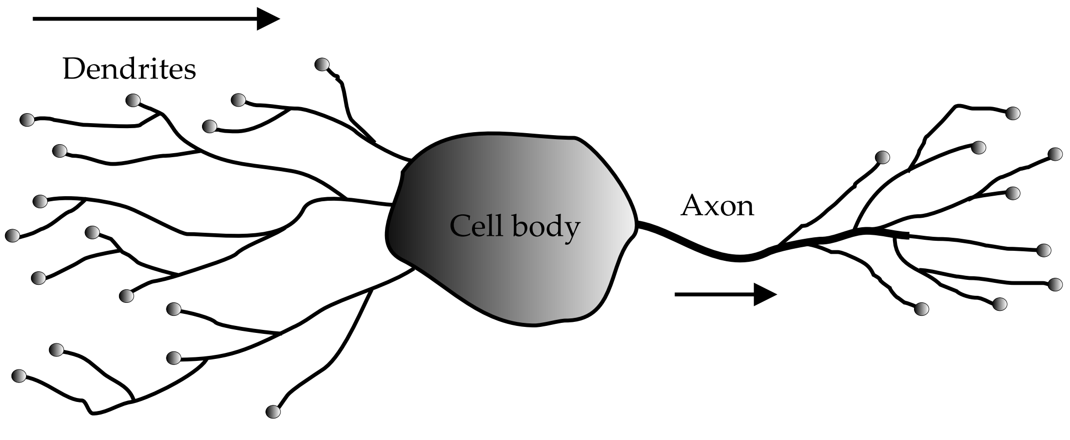

Figure 1.1: *A schematic illustration of a neuron.*

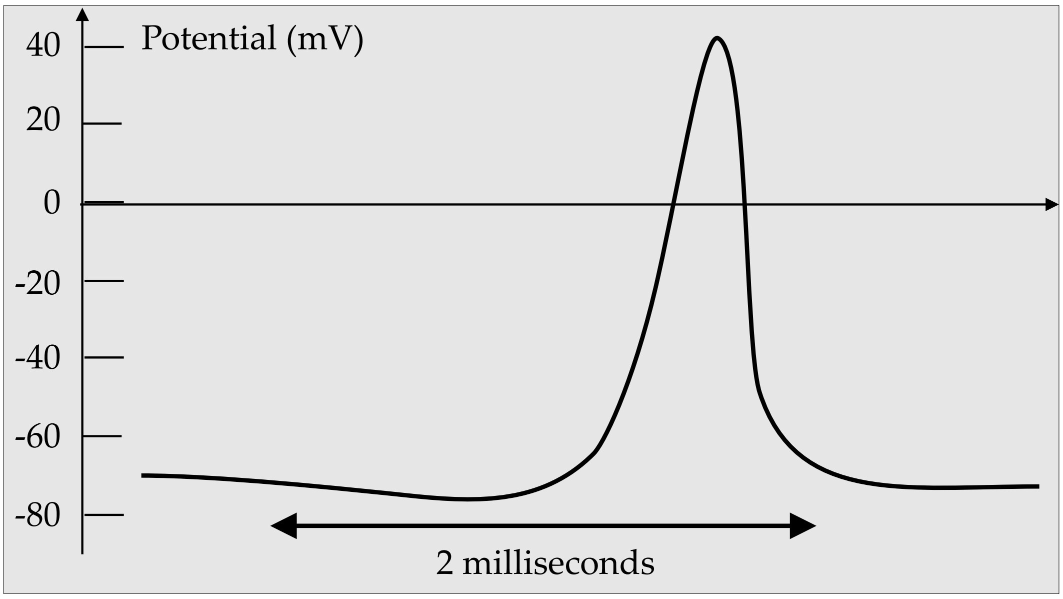

Figure 1.2: *A typical neural spike.*

neurons.

Input signals are received through the **dendrites**, which are connected to the axons of many other neurons via synapses. The transmission through the synapses is chemical for most neurons: **transmitter substances** are released on the axonal side of the synapse and diffuse toward the dendritic side, making the connection.

The signal transmission within a neuron, however, is electrical. When a neuron fires a **spike**, an electric potential propagates (with speeds of up to 100 m/s) along the axon, as shown in Fig. 1.2. The details of this process will not be considered here.

The synapses between neurons can be either excitatory or inhibitory. An **excitatory** synapse increases the propensity of the target neuron to fire, whereas an **inhibitory** synapse does the opposite. A neuron operates in a binary fashion: either it fires or it does not.

A typical animal brain consists of a very large number of neurons: a rat has of the order of 10^{10} neurons in its brain, and the human brain contains around 10^{12} neurons, and $10^{14} - 10^{15}$ synapses. Thus, each neuron is, on average, connected to hundreds of other neurons.

In addition, there are both short–range and long–range connections, as well as many feedback loops, making the brain an exceedingly complex structure, which is very far from being understood at this time.

As indicated in Fig. 1.2, the firing frequency of the neurons is not very high compared to the clock frequencies of digital computers. After each spike, a neuron needs a period of recovery or relaxation (known as the **refractory pe-**

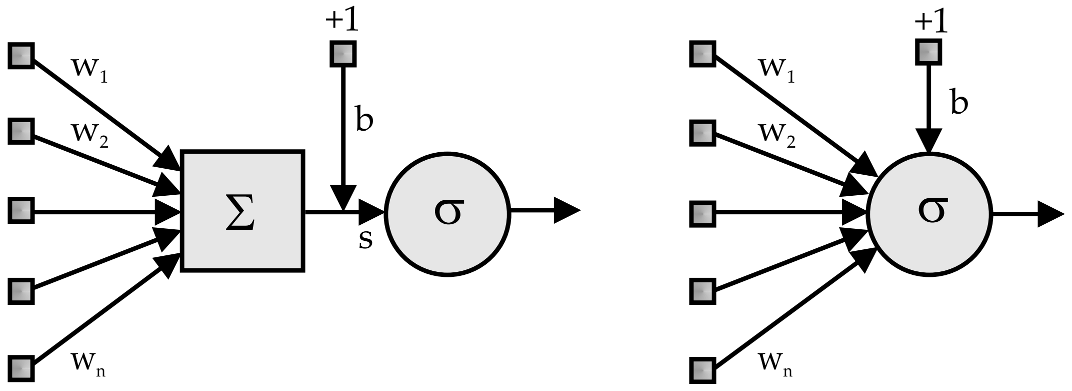

Figure 1.3: *The McCulloch–Pitts neuron is shown in the left panel. The signal flow is from left to right. The right panel shows the simplified notation which is normally used.*

riod) during which it is unable to fire, thus limiting the firing frequency to 1 kHz or less. This can be compared with the clock frequencies of a few GHz – more than a million times faster – of modern computers. How, then, can the brain perform such complex operations as it does given that it is, relatively speaking, rather slow? The answer lies in the massively parallel architecture of the brain, which allows it to carry out many operations per cycle.[1] In fact, even the brain of an *insect* generally performs many more operations per second than a computer.

1.2 Artificial neural networks

In order to distinguish them from biological neural networks (i.e. brains), the networks that are simulated in computers or implemented in computer hardware are usually called **artificial neural networks** or ANN for short.

A simple model of a single neuron, formulated by McCulloch and Pitts in the 1940s, is shown in the left panel of Fig. 1.3. The **McCulloch–Pitts** (MCP) **neuron** consists of a number of input connections, corresponding to the dendrites in the biological neuron, a summing device and a threshold device, corresponding to the cell body of the neuron which takes the decision of whether to fire, and an output connection which corresponds to the axon of the biological neuron.

The summing device sums up the influences of the input connections. The strength of a connection is represented by a number, known as the **connection weight**. A connection weight can be either positive (excitatory) or negative (inhibitory). In addition to the external inputs, one usually defines an additional

[1]The analogy with digital computers should not be taken too far. The brain does not have the equivalent of a single central processor which updates all the neurons synchronously. Instead, it operates in a decentralized, asynchronous way.

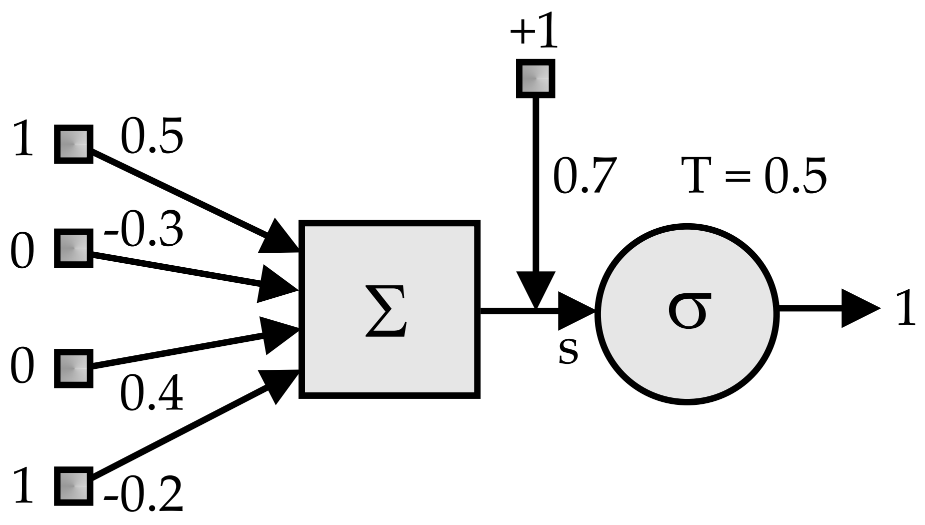

Figure 1.4: *An MCP neuron firing (see example 1.1).*

constant input known as the **bias** term (b), which measures the propensity of the neuron to fire in the absence of an external input. The output of the summing device becomes

$$s = \sum_{j=1}^{n} w_j x_j + b, \tag{1.1}$$

where n is the number of neurons, w_j are the connection weights, and x_j the input signals. In order to simplify the notation, the bias term is often written $w_0 x_0$, where $w_0 = b$ and x_0 is always equal to 1. With this notation, the output from the summing device can be written

$$s = \sum_{j=0}^{n} w_j x_j. \tag{1.2}$$

The output obtained from the summing device is passed through the threshold device which, in the case of the MCP neuron, is a **hard limiter** , i.e. its output takes the value 0 if the input is below the threshold (T), and 1 if it is above the threshold. Thus the output (y) of the MCP neuron satisfies the equation

$$y = \sigma(s) \equiv \sigma\left(\sum_{j=0}^{n} w_j x_j\right), \tag{1.3}$$

where $\sigma = 0$ if $s < T$, and 1 otherwise.

In most cases, the simplified notation shown in the right panel of Fig. 1.3 is used for representing neurons, rather than the complete notation shown in the left panel.

Example 1.1 If it is assumed that the input signals to an MCP neuron are also generated by similar neurons, they take the values 0 or 1. As they are fed to the summing device, they are multiplied by the corresponding connection weights. An MCP neuron is shown in Fig. 1.4. Here two of the inputs are on (=1) and two are off (=0). The signals are summed as follows

$$s = \sum_{j=0}^{4} w_j x_j = 0.7 \times 1 + 0.5 \times 1 - 0.3 \times 0 + 0.4 \times 0 - 0.2 \times 1 = 1.0, \tag{1.4}$$

and are then passed through the threshold causing, in this case, the neuron to fire:

$$y = \sigma(s) = 1, \tag{1.5}$$

since $s > T = 0.5$.

The information in an ANN is stored in the connection weights between the different neurons in the network. Thus, the computation (i.e. the functionality) of the network cannot be separated from its architecture. If the architecture changes by, for example, the addition or deletion of some neurons, the computation carried out by the network changes as well. **Learning** in ANNs is the process of determining and setting the weights connecting the neurons in the network so as to make the network perform as desired. Some of the learning algorithms for ANNs will be considered later in this book.

For now, let us examine a simple neural network, namely the one shown in the left panel of Fig. 1.5. The input signals are received by the network through the small squares to the left in the figure. Allowing only binary input signals (0 or 1) there are 4 possible input combinations to this simple network: 00, 01, 10, and 11. The corresponding output signals are, in order: 0, 0, 0, and 1. Thus, the network encodes the Boolean AND function. The AND function an other Boolean functions are usually represented in tabular form, using so called **truth tables**. The truth table for the AND function takes the following form

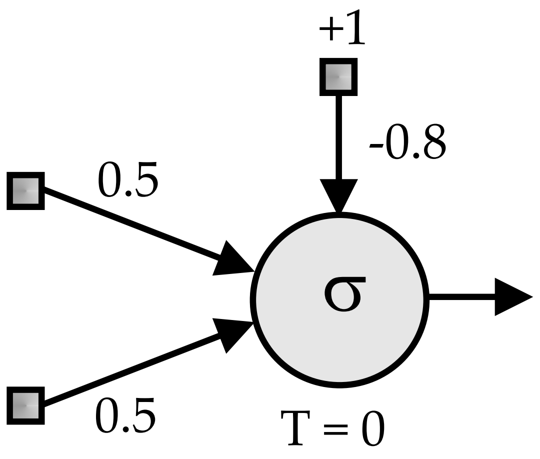

Figure 1.5: *An MCP neuron that implements the Boolean AND function.*

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

This network belongs to a class of networks known as **layered feedforward networks** (FFNNs), which are characterized by their particular structure in which the neurons are organized in separate layers without any intra–layer connections, and by the directionality – from input to output, without feedback connections – of the signal flow in the network. The network in Fig. 1.5 is a **one-layer** network. At a first glance one might say that it contains *two* layers of neurons. However, the input layer, represented by squares in Fig. 1.5, only serves to distribute the input; it does not perform any computation. The units in such a layer are called **input elements** to distinguish them from neurons, which perform a computation of the kind introduced in Eq. (1.3). Single–layer networks are rather limited in the computations they can perform, and below we shall also consider multi–layer ANNs.

So far, we have only considered neurons which give a binary output. In some cases a graded response, taking any value between 0 and 1, can be useful. Such a response can easily be obtained by changing the thresholding function σ a bit. The most common choice of graded **response function**[2] is the **sigmoid**, defined by

$$\sigma(z) = \frac{1}{1 + \mathrm{e}^{-cz}}, \tag{1.6}$$

[2]The response function is sometimes called the **activation function** or the **squashing function**.

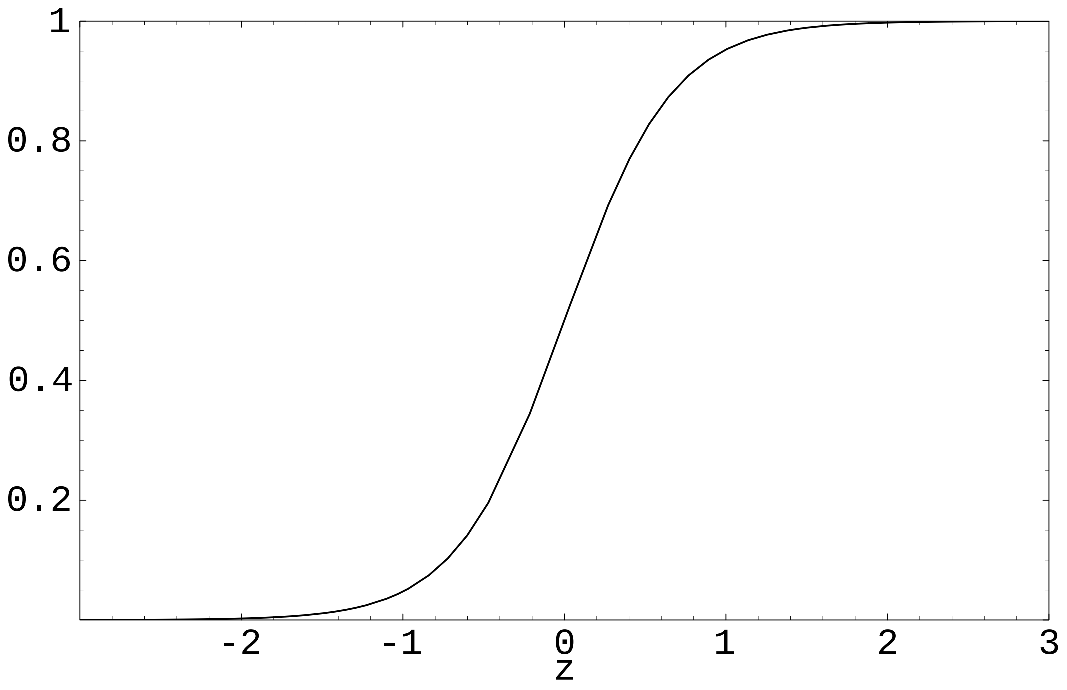

Figure 1.6: *A sigmoid with* $c = 3$.

where c is a constant parameter that determines the slope of the transition between the two limiting values 0 (as z approaches $-\infty$) and 1 (as z approaches plus $+\infty$). As $c \to \infty$, the sigmoid turns into a hard limiter with threshold 0. The sigmoid with $c = 3$ is shown in Fig. 1.6. Another graded response function, with output in the interval $[-1, 1]$ rather than $[0, 1]$ is provided by the hyperbolic tangent function.

How can a graded response be reconciled with the fact that biological neurons give a binary response to their input signals? The simplest answer is to note that there is no need for a reconciliation; biological neural networks form the background for ANNs, but one does not have to follow biology exactly. In fact, even with the binary response, the MCP neuron is only a very crude approximation of a biological neuron. However, the graded response can be motivated biologically by a slight modification of the interpretation of the output of a neuron. Instead of considering the output to represent individual spikes, we interpret the output of the neuron as the *average* firing frequency in an interval of time, containing sufficiently many spikes to form an accurate average. This is illustrated in Fig. 1.7: the uppermost part of the figure shows a neuron firing with a varying frequency, the time average of which is listed in the lower graph.

The layered feedforward architecture is not the only possible architecture for ANNs. A feedforward network cannot remember previously encountered

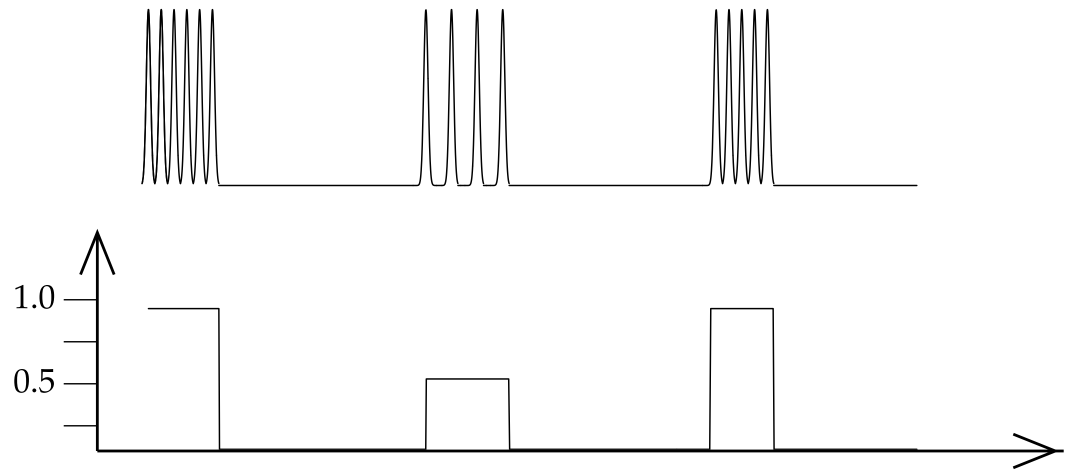

Figure 1.7: *A spike train from a neuron (upper graph) and the corresponding graded response.*

input signals, and thus has no dynamic short–term memory. It does, of course, have a static long–term memory provided by the learning procedure and encoded in its connection weights. However, it will always give the same output for any given input, regardless of the signals that preceded that input. **Recurrent neural networks** (RNNs), on the other hand, have feedback connections providing the network with a limited short–term memory. The response of such a network depends not only on the connection weights, but also on the previous input signals. Recurrent neural networks need not be layered at all. In Chapter 3 we shall consider a type of fully connected, non–layered ANNs which can be used as neural memories. An FFNN is shown in the left panel of Fig. 1.8 and an RNN is shown in the right panel of the same figure. For the FFNN, the output y of a neuron is simply computed as in Eq. (1.3), using e.g. the sigmoid function defined in Eq. (1.6). Thus, using the signals from the input elements, the outputs from the first layer of neurons are computed, and are then used as inputs to the next layer etc. For the RNN, the computation of the output is carried out in a different way. It is common to apply RNNs e.g. for steering autonomous robots, which operate in continuous (rather than discrete) time. In such cases, the RNN must also be capable of delivering continuous signals. Thus, for an RNN operating in continuous time, the output of neuron i in the network is given by

$$\tau_i \dot{x}_i(t) + x_i(t) = \sigma\left(\sum_{j=1}^{n} w_{ij} x_j(t) + \sum_{j=1}^{n^{\text{in}}} w_{ij}^{\text{in}} I_j(t) + b_i\right), \quad i = 1, \ldots, n \qquad (1.7)$$

where w_{ij} are the weights connecting neurons to each other, w_{ij}^{in} are weights

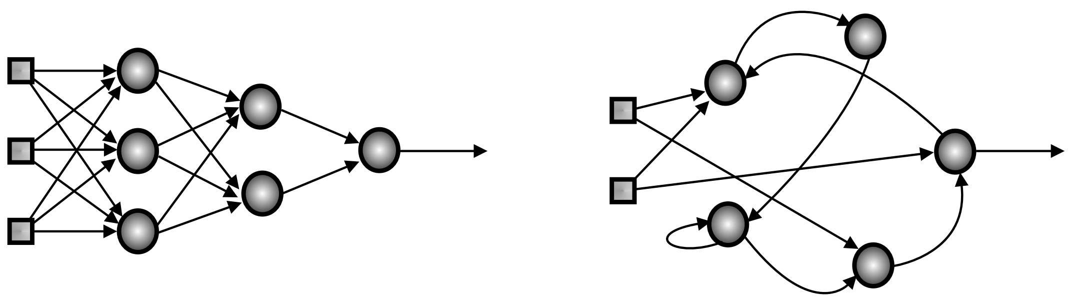

Figure 1.8: *A layered neural network (left) and a recurrent neural network. The signal flow is from left to right, except for the recurrent connections in the right panel.*

connecting input j to neuron i, b_i are the bias terms, τ_i are time constants, n is the number of neurons, and $n^{\rm in}$ is the number of input elements.

In practice, when integrating the network equations for a continuous-time RNN, a discretization procedure is applied so that

$$\tau_i \frac{x_i(t+\Delta t) - x_i(t)}{\Delta t} + x_i(t) = \sigma\left(\sum_{j=1}^{n} w_{ij} x_j(t) + \sum_{j=1}^{n^{\rm in}} w_{ij}^{\rm in} I_j(t) + b_i\right), \tag{1.8}$$

from which the equation for $x(t+\Delta t)$

$$x_i(t+\Delta t) = x_i(t) + \frac{\Delta t}{\tau_i}\left[-x_i(t) + \sigma\left(\sum_{j=1}^{n} w_{ij} x_j(t) + \sum_{j=1}^{n^{\rm in}} w_{ij}^{\rm in} I_j(t) + b_i\right)\right] \tag{1.9}$$

easily can be obtained. For numerical stability, the integration time step Δt should be much smaller (by a factor 10, say) than the smallest time constant $\tau_{\rm min}$ in the network.

Neural networks have many advantages. With the aid of learning algorithms, they are able to form their own representation of the data in the inner (hidden) layers of the network, most often in a manner not provided explicitly by the user. Their non–linear computing elements allow them to represent very complex, non–linear mappings. The distributed nature of the computation in ANNs makes them fault tolerant; In a large network, the failure of one or a few neurons does not degrade the performance of the network catastrophically. This **graceful degradation** is important, particularly in hardware implementations of ANNs, which can be made very robust. The distributed computation may, however, also be a disadvantage, at least from the point of view of a human observer. Neural networks are often very difficult to interpret, and must in some cases be used as black boxes, delivering correct output to any input by carrying out computations in ways that are not easily accessible to the user.

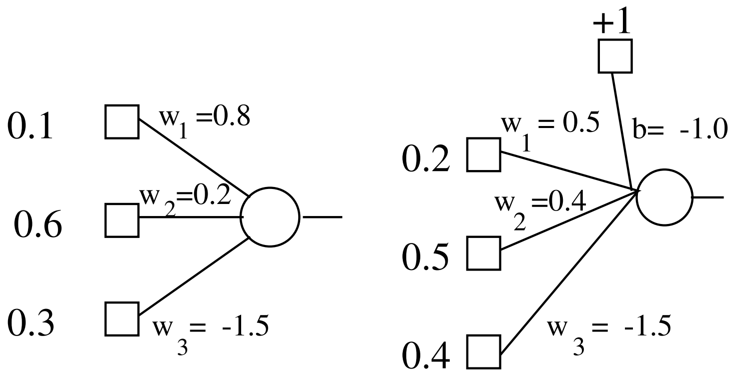

Figure 1.9: *Networks for Problem 1.1.*

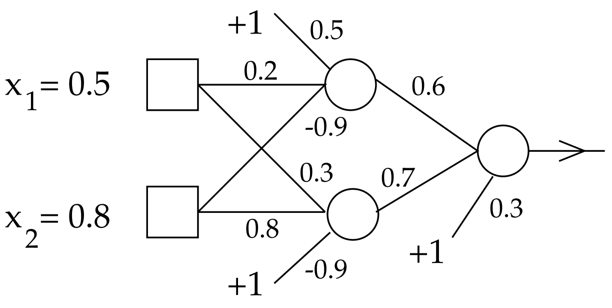

Figure 1.10: *Network for Problem 1.3.*

Problems

1.1 Compute the output signals of the two networks in Fig. 1.9. Assume a linear response function.

1.2 Design (by hand) a neural network with two input elements (x_1 and x_2) and one output neuron y, with the following input–output characteristics: $x_1 = 0, x_2 = 0 \to y = 1$, $x_1 = 0, x_2 = 1 \to y = 1$, $x_1 = 1, x_2 = 0 \to y = 0$, $x_1 = 1, x_2 = 1 \to y = 0$. For the response function of the neuron, use a **hard limiter**, i.e. a step function, with threshold $T = 0$.

1.3 Find the output of the network in Fig. 1.10. For the response function, use the sigmoid defined in Eq.(1.6), with $c = 3$.

Answers to exercises

1.1 -0.25 (left network) and -1.3 (right network).
1.2 For example, $w_1 = -1, w_2 = 0, b = 0.5$ (There are other solutions).
1.3 $y = 0.9254$.

Chapter 2

Backpropagation

There exists several different methods for training neural networks, i.e. for setting the connection weights of the network so as to make it perform the desired computation. In this chapter we shall introduce a method, known as **backpropagation** for training multilayer FFNNs. This method belongs to a class of **learning methods** known as **supervised learning methods**, which are characterized by the presence of a teacher who provides the network with information about its performance, thus providing it with the means to make improvements. In order to introduce the concepts needed for the discussion on backpropagation, we shall first consider learning in simple single–layer ANNs.

2.1 Single–layer networks and the delta rule

Consider a very simple ANN consisting of only one neuron, and n input elements as shown in Fig. 2.1. The neuron receives the signals from the input elements and produces an output according to

$$y = \sigma(\sum_{j=1}^{n} w_j x_j + b) \equiv \sigma(\sum_{j=0}^{n} w_j x_j), \tag{2.1}$$

where σ is the response function, and b is the bias term. To simplify the discussion, let us assume a linear activation function, i.e. $\sigma(s) = s$, so that the output from the neuron is simply given by

$$y = \sum_{j=0}^{n} w_j x_j. \tag{2.2}$$

Now, assume also that there is a set of training data consisting of M input–output pairs. Thus, for each member of the training set, the input signals, as

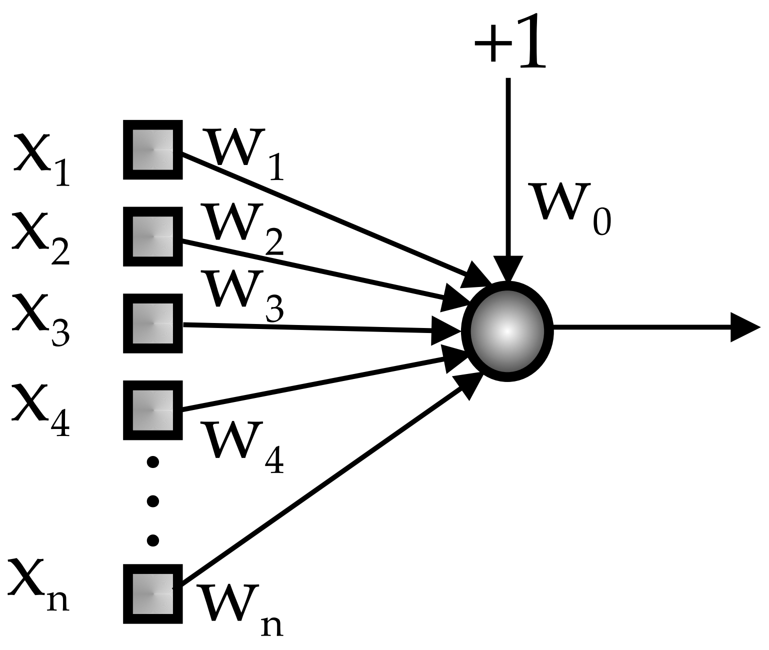

Figure 2.1: *A single–neuron network with n input elements.*

well as the desired output signal, are given. The task of the learning method is to set the weights w_j in such a way as to approximate the desired output signals as closely as possible; in other words to minimize the deviation, over the whole training set, between the desired output and the output from the network. The desired output, denoted $o^{(m)}, m = 1, \ldots, M$, is known for each configuration of input signals in the training data set. The input signals are denoted $\mathbf{x}^{(m)} = (x_1^{(m)}, x_2^{(m)}, \ldots, x_n^{(m)})$, $m = 1, \ldots, M$. Thus, we can form the **error** $e^{(m)}$, defined as $o^{(m)} - y^{(m)}$, where $y^{(m)}$ is the actual output of the network when presented with the input $\mathbf{x}^{(m)}$. Using Eq. (2.2), the error can be written as

$$e^{(m)} = o^{(m)} - \sum_{j=0}^{n} w_j x_j^{(m)}. \tag{2.3}$$

Given the errors $e^{(m)}$ for each member of the training data set, the **total squared error**, denoted E, is computed as

$$E = \frac{1}{2} \sum_{m=1}^{M} (e^{(m)})^2 = \frac{1}{2} \sum_{m=1}^{M} (o^{(m)} - \sum_{j=0}^{n} w_j x_j^{(m)})^2. \tag{2.4}$$

The factor $\frac{1}{2}$ is introduced to simplify the expressions for the weight modifications derived below. For any given training set, the only variables in Eq. (2.4) are the weights w_j. Expanding the sum, we note that there are three kinds of terms: those that are constant, those that are linear in the w_j, and those that are quadratic in the w_j. Collecting terms, we can therefore write the expression for

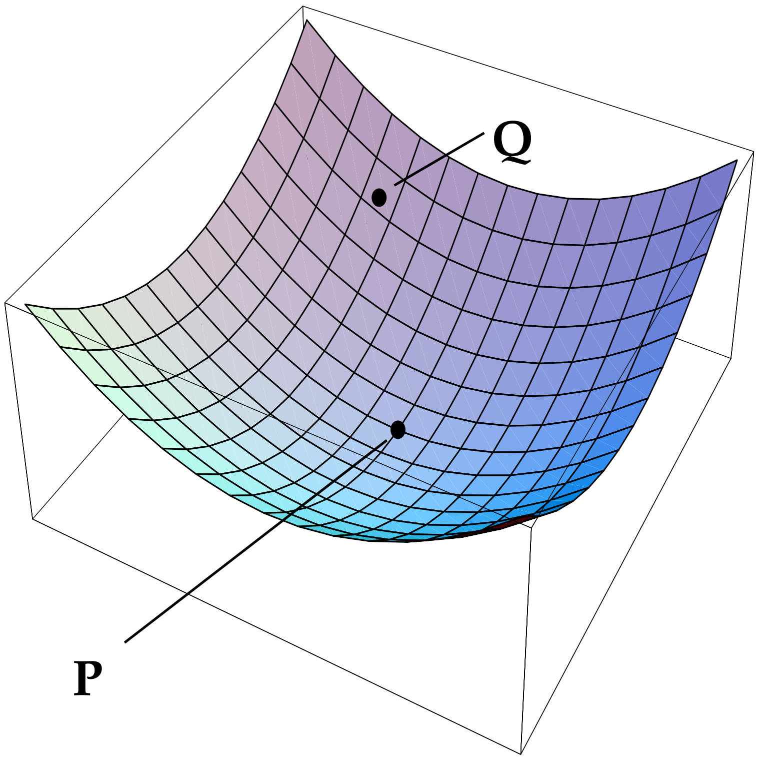

Figure 2.2: *The error surface for a single–neuron network with* $n = 1$.

E as a quadratic form in w_j:

$$E = A + \sum_{j=0}^{n} B_j w_j + \sum_{j=0}^{n} \sum_{k=0}^{n} C_{jk} w_j w_k, \tag{2.5}$$

where $A = \frac{1}{2}\sum_{m=1}^{M}(o^{(m)})^2$, $B_j = -\sum_{m=1}^{M} o^{(m)} x_j^{(m)}$, and $C_{jk} = \frac{1}{2}\sum_{m=1}^{M} x_j^{(m)} x_k^{(m)}$. The surface described by Eq. (2.5) is an n–dimensional paraboloid in weight space. The case $n = 1$ is shown in Fig. 2.2. Thus, there exists a set of weights for which E has a unique (global) minimum, denoted P in the figure. How can this minimum be reached? Imagine that the weights (in a case with $n = 1$) are such that the network is located at point Q in the figure. Since the existence of a unique minimum is known, a sensible strategy to reach point P is to determine in which direction the surface is steepest (downhill) at point Q, and then move in this direction.

The direction of steepest descent at point Q is simply given by the negative of the **gradient** of the surface, i.e.

$$-\nabla_w E = -(\frac{\partial E}{\partial w_0}, \ldots, \frac{\partial E}{\partial w_n}). \tag{2.6}$$

In order for the weights to converge toward the global minimum P, they should, after each evaluation of the training set, be changed according to

$$w_j \to w_j + \Delta w_j,\ 0 = 1, \ldots, n, \tag{2.7}$$

where

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j},\ j = 0, \ldots, n, \tag{2.8}$$

where η is the **learning rate** parameter. The procedure of following the negative of the gradient of the error surface toward the global minimum is known as **gradient descent**. Using the form of the total error E as given in Eq. (2.4), the expression for the components of the gradient of E can be simplified:

$$\begin{aligned}\frac{\partial E}{\partial w_j} &= \frac{1}{2}\frac{\partial}{\partial w_j}\sum_{m=1}^{M}(e^{(m)})^2 = \sum_{m=1}^{M} e^{(m)}\frac{\partial e^{(m)}}{\partial w_j} = \\ &= \sum_{m=1}^{M} e^{(m)}\frac{\partial(o^{(m)} - \sum_{j'=0}^{n} w_{j'}x_{j'}^{(m)})}{\partial w_j} = -\sum_{m=1}^{M} e^{(m)}x_j^{(m)} = \\ &= -\sum_{m=1}^{M}(o^{(m)} - y^{(m)})x_j^{(m)}.\end{aligned} \tag{2.9}$$

The rule for the change in the weights can now be written

$$\Delta w_j = -\eta\frac{\partial E}{\partial w_j} = \eta\sum_{m=1}^{M}(o^{(m)} - y^{(m)})x_j^{(m)}. \tag{2.10}$$

This rule is known as the **delta rule** or the **Widrow–Hoff rule** in honor of its discoverers.

If the training set is large, the computation of the weight modifications is rather time–consuming, and the convergence toward the global minimum becomes slow. In order to speed up the convergence, one normally carries out what is known as **stochastic gradient descent**, in which an approximate error surface is formed by considering just one input–output pair, selected at random from the training set. The rule for weight modification in stochastic gradient descent takes the form

$$\Delta w_j = \eta(o^{(m)} - y^{(m)})x_j^{(m)},\ m \in [1, M]. \tag{2.11}$$

Thus, in stochastic gradient descent, a random permutation of the training data set is formed (in order to make sure that all patterns are tested), and the weights are updated according to Eq. (2.11) for every input–output pair. A pass through the entire data set is known as an **epoch**. When a training epoch is completed, i.e. when all patterns have been considered, a new random permutation of the training data set is formed, and the process is repeated again, etc.

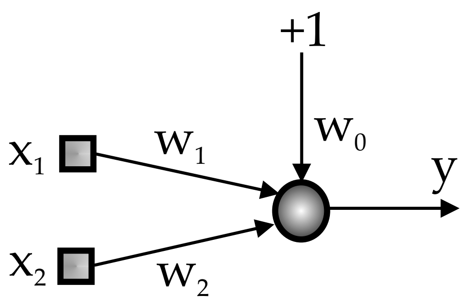

Figure 2.3: *A single–neuron network with two input elements.*

x_1	x_2	o
0	0	low
1	0	high
0	1	high
1	1	low

Table 2.1: *A set of input–output pairs. The* low *and* high *notation for the outputs simply indicate that the exact values are not important for the particular example discussed in the text.*

2.1.1 Limitations

The rule for updating weights defined by Eq. (2.11) is neat and useful. However, there are limits on the representational powers of single–layer networks. As an example, consider a case in which the input–output pairs in Table 2.1 are given. The output signals are listed as *low* and *high* rather than as numerical values. This is so, because the exact values are not important to the example we will discuss. The numerical values could be *low* = 0.1 and *high* = 0.9, or *low* = 0.49 and *high* = 0.51. Now, consider the case of a single–layer network with two input elements and one neuron, as shown in Fig. 2.3. The output values produced by the network for each of the four input output pairs are

$$y^{(1)} = w_0 \quad \text{for } x_1 = 0, x_2 = 0, \tag{2.12}$$
$$y^{(2)} = w_0 + w_1 \quad \text{for } x_1 = 1, x_2 = 0, \tag{2.13}$$
$$y^{(3)} = w_0 + w_2 \quad \text{for } x_1 = 0, x_2 = 1, \tag{2.14}$$
$$y^{(4)} = w_0 + w_1 + w_2 \quad \text{for } x_1 = 1, x_2 = 1. \tag{2.15}$$

Now, let us set a value of a **separator** α which distinguishes low output from high output. For instance, for both of the $\{$*low*, *high*$\}$ sets above, $\alpha = 0.5$ would be an acceptable separator. Introducing $\beta = \alpha - w_0$ it is easy to see that, in order to reproduce the desired output data in Table 2.1, the output of the network would have to satisfy the equations

$$0 < \beta, \tag{2.16}$$

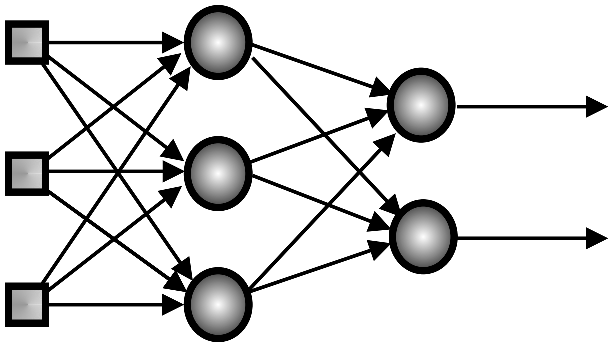

Figure 2.4: *A multi–layer neural network. The biases are not shown.*

$$w_1 > \beta, \tag{2.17}$$
$$w_2 > \beta, \tag{2.18}$$
$$w_1 + w_2 < \beta. \tag{2.19}$$

Clearly, the last equation cannot be satisfied if the two equations immediately above it are satisfied, and vice versa. Thus, the single–layer network is unable to form a representation of the data set in Table 2.1. For the case in which the *low* values are set to 0, and the *high* values set to 1, the data set in Table 2.1 is known as the Boolean **exclusive or** or the **XOR** function, and is just one of many Boolean functions that cannot be represented by single–layer networks. It is easy to see that a change in the response function $\sigma(s) = s$ to any other monotonous function would not help either.

2.2 Multi–layer networks and backpropagation

We have just seen that there are limits to what a single–layer neural network can accomplish, and we will now consider multi–layer networks, which have considerably higher computational power. A network consisting of one layer of input elements and two layers of neurons is shown in Fig. 2.4. The middle layer is also called a **hidden layer**. A multi–layer neural network may have more than one hidden layer but, for simplicity, we shall only consider the case of a network with a single hidden layer in the derivation below.

However, regardless of the number of hidden layers, multi–layer networks with *linear* activation functions will not do; In the case of a linear two–layer neural network of the kind shown in Fig. 2.4, the output $\mathbf{y}^H$ of the hidden layer can be written $\mathbf{y}^H = \mathbf{w}^{I\to H}\mathbf{x}$, where $\mathbf{w}^{I\to H}$ is a constant matrix. The output from the network, for which we henceforth use the notation $\mathbf{y}^O$ is thus given by $\mathbf{y}^O = \mathbf{w}^{H\to O}\mathbf{y}^H = \mathbf{w}^{H\to O}\mathbf{w}^{I\to H}\mathbf{x} \equiv \mathbf{w}\mathbf{x}$, where $\mathbf{w} = \mathbf{w}^{H\to O}\mathbf{w}^{I\to H}$. Thus, a multi–layer linear network can be reduced to a single–layer network.

This does not apply, however, if the activation function σ is non–linear, which we will assume is the case in the derivation below.

We shall now derive a learning rule for multi–layer neural networks. Consider again the network shown in Fig. 2.4. It is assumed that there is a training data set for which the desired output vector $\mathbf{o}^{(m)}, m = 1, \ldots, M$ is known for each input vector $\mathbf{x}^{(m)}$. For any input vector $\mathbf{x} = \mathbf{x}^{(m)}$, the corresponding output vector can be computed and an error signal $\mathbf{e}^{(m)}$ whose components $e_l = e_l^{(m)}$ are defined as

$$e_l = o_l - y_l^O, \tag{2.20}$$

can be computed for each neuron $l, l = 1, \ldots, n^{(O)}$, where $n^{(O)}$ is the number of neurons in the output layer. In Eq. (2.20), the term o_l denotes the lth component of the desired output vector $\mathbf{o} = \mathbf{o}^{(m)}$, and in most of the equations that will follow in this section, the index m that enumerates the input–output pairs will be dropped, in order to minimize (for clarity) the number of indices on the variables.

Thus, for an output neuron, the corresponding error signal can easily be computed. But what about the neurons in the *hidden* layer(s)? For these neurons, no simple error signal can be formed, since it is not obvious how to reward (or punish) a hidden neuron for a result that appears in the output layer. We are therefore faced with a **credit assignment problem** for the neurons in the hidden layer(s).

2.2.1 Output neurons

To begin with, however, let us deal with output neurons. Given the error signal for each neuron j, the total error can be defined as

$$\mathcal{E}^{(m)} = \frac{1}{2}\sum_{l=1}^{n^{(O)}} (e_l^{(m)})^2. \tag{2.21}$$

This equation yields the error for *one* input–output pair, with index m. The mean square error over the whole training set can be computed as

$$\overline{\mathcal{E}} = \frac{1}{M}\sum_{m=1}^{M} \mathcal{E}^{(m)}. \tag{2.22}$$

However, just as in the case of the single–layer neural network, we will use stochastic gradient descent, and thus update the weights of the network after each input signal. For a neuron i in the output layer, the partial derivative of the error $\mathcal{E} \equiv \mathcal{E}^{(m)}$ with respect to the weight $w_{ij}^{H\to O}$, connecting neuron j in the hidden layer to neuron i (see Fig. 2.5), can be written

$$\frac{\partial \mathcal{E}}{\partial w_{ij}^{H\to O}} = \frac{\partial}{\partial w_{ij}^{H\to O}}\left(\frac{1}{2}\sum_{l=1}^{n^{(O)}} e_l^2\right) = e_i \frac{\partial e_i}{\partial w_{ij}^{H\to O}} =$$

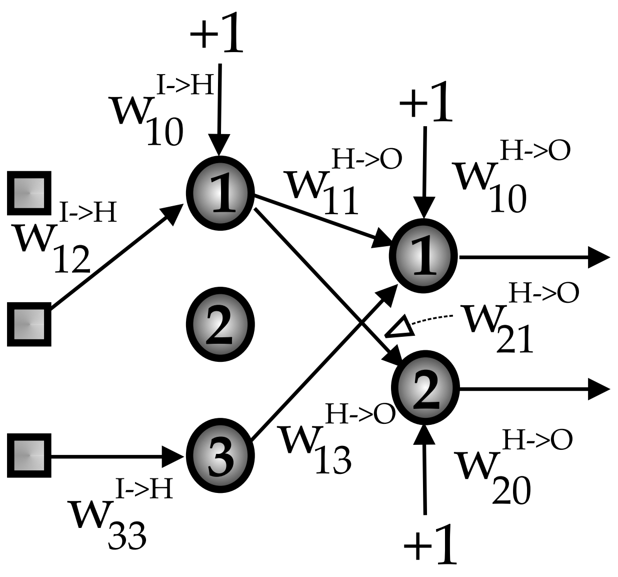

Figure 2.5: *The notation used in the derivation of the backpropagation rule, here shown for the case $n^{(I)} = 3, n^{(H)} = 3, n^{(O)} = 2$. $w_{ij}^{H\rightarrow O}$ is the weight connecting neuron j in the hidden layer to neuron i in the output layer. Similarly, $w_{ij}^{I\rightarrow H}$ connects input element j to neuron i in the hidden layer. For clarity, only a few connections have been drawn in the figure.*

$$\begin{aligned}
&= e_i \frac{\partial}{\partial w_{ij}^{H\rightarrow O}} \left(o_i - \sigma \left(\sum_{s=0}^{n^{(H)}} w_{is}^{H\rightarrow O} y_s^H \right) \right) = \\
&= -e_i \sigma' \left(\sum_{s=0}^{n^{(H)}} w_{is}^{H\rightarrow O} y_s^H \right) \frac{\partial \left(\sum_{s=0}^{n^{(H)}} w_{is}^{H\rightarrow O} y_s^H \right)}{\partial w_{ij}^{H\rightarrow O}} = \\
&= -e_i \sigma' y_j^H, \qquad (2.23)
\end{aligned}$$

where, in the last step, the argument of the derivative of the response function σ was not displayed. In analogy with the delta rule, Eq.(2.10), the weight modifications for the output neurons are now given by

$$\Delta w_{ij}^{H\rightarrow O} = -\eta \frac{\partial \mathcal{E}}{\partial w_{ij}^{H\rightarrow O}} = \eta \delta_i y_j^H, \qquad (2.24)$$

where $\delta_i = e_i \sigma'$ is the **local gradient**. Thus, with the exception of the factor σ', which was equal to 1 in the derivation of the delta rule, the result is the same as that given by Eq. (2.11).

2.2.2 Hidden neurons

Consider now a neuron in the hidden layer in Fig. 2.4 or Fig. 2.5, with output signal y_i^H. Since this neuron is connected to all the neurons in the output layer, via the connection weights $w_{ij}^{H\to O}$, it will give a contribution to the error of all those neurons. Formally, we can also in this case write the weight modification rule as

$$\Delta w_{ij}^{I\to H} = -\eta \frac{\partial \mathcal{E}}{\partial w_{ij}^{I\to H}}. \tag{2.25}$$

For this expression to be useful, we must compute the partial derivative of $\mathcal{E}$ with respect to $w_{ij}^{I\to H}$. Proceeding in a way similar to that used for the output neurons considered above, we get

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_{ij}^{I\to H}} &= \frac{\partial \mathcal{E}}{\partial y_i^H}\frac{\partial y_i^H}{\partial w_{ij}^{I\to H}} = \frac{\partial \mathcal{E}}{\partial y_i^H}\frac{\partial}{\partial w_{ij}^{I\to H}}\sigma\left(\sum_{p=0}^{n^{(I)}} w_{ip}^{I\to H} y_p^I\right) = \\ &= \frac{\partial \mathcal{E}}{\partial y_i^H}\sigma'\left(\sum_{p=0}^{n^{(I)}} w_{ip}^{I\to H} y_p^I\right) y_j^I, \end{aligned}\tag{2.26}$$

where, for the two–layer network, y_p^I is the output of the elements in the input layer, i.e. $y_p^I = x_p$. Continuing the calculation, we get

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial y_i^H} &= \frac{\partial}{\partial y_i^H}\left(\frac{1}{2}\sum_{l=1}^{n^{(O)}} e_l^2\right) = \sum_{l=1}^{n^{(O)}} e_l \frac{\partial e_l}{\partial y_i^H} = \sum_{l=1}^{n^{(O)}} e_l \frac{\partial (o_l - y_l^O)}{\partial y_i^H} = \\ &= -\sum_{l=1}^{n^{(O)}} e_l \frac{\partial}{\partial y_i^H}\sigma\left(\sum_{s=0}^{n^{(H)}} w_{ls}^{H\to O} y_s^H\right) = \\ &= -\sum_{l=1}^{n^{(O)}} e_l \sigma'\left(\sum_{s=0}^{n^{(H)}} w_{ls}^{H\to O} y_s^H\right) w_{li}^{H\to O} = \\ &= -\sum_{l=1}^{n^{(O)}} \delta_l w_{li}^{H\to O}, \end{aligned}\tag{2.27}$$

where, in the final step, δ_l is defined as in Eq. (2.24). Combining Eqs. (2.25), (2.26), and (2.27), the weight modification can be written

$$\Delta w_{ij}^{I\to H} = \eta \kappa_i y_j^I, \tag{2.28}$$

where

$$\kappa_i = \sigma'\left(\sum_{p=0}^{n^{(I)}} w_{ip}^{I\to H} y_p^I\right) \sum_{l=1}^{n^{(O)}} \delta_l w_{li}^{H\to O}. \tag{2.29}$$

From Eq. (2.28) it is clear that the expression for the weight modification in the hidden layer is similar to the corresponding expression for the output layer. The main difference is that δ is replaced by κ which, in turn, is a weighted sum of the δ–terms obtained for the output layer.

The weight modification rules defined in Eqs. (2.24) and (2.28) constitute the backpropagation algorithm. It derives its name from the fact that errors are propagated backward in the network; in order to compute the weight change in the hidden layer, the local gradient δ from the output layer must first be known.

Note that the modifications Δw_{ij} are computed for *all* weights in the network before *any* modifications are actually applied. Thus, in the computation of the modifications for the weights connecting the input layer to the hidden layer, the old values are used for the weights $w_{ij}^{H \to O}$.

In the derivation above, we limited ourselves to the case of two–layer networks. It is easy to realize, though, that the formulae for the weight changes can be used even in cases where there are more than one hidden layer. The weight modification in the first hidden layer (i.e. the one that follows immediately after the input elements) would then be computed as $\eta \gamma_i y_j^I$, where the γ_i are computed using the κ_i which, in turn, are obtained from the δ_i.

2.2.3 Improving backprogagation using a momentum term

When the backpropagation algorithm is used, the weights are first initialized to small, random values. The speed by which learning progresses is dependent on the value of the learning rate parameter η. If η is chosen too large, the mean square training error $\overline{\mathcal{E}}$ will not follow the gradient towards the minimum, but will instead jump around wildly on the error surface. On the other hand, if η is chosen too small, convergence towards the minimal $\overline{\mathcal{E}}$ will be stable but very slow. In order to increase the learning rate without causing instabilities, one may introduce a **momentum term**, consisting of the weight change in the previous backpropagation step multiplied by a constant ϵ, and change the weight modification rule according to

$$\Delta w_{ij}^{H \to O}(t) = \epsilon \Delta w_{ij}^{H \to O}(t-1) + \eta \delta_i y_j^{(H)}, \tag{2.30}$$

and similarly for the weights between the input layer and the hidden layer.

2.2.4 Varying the learning rate

The value of the learning rate parameter η should, of course, be chosen so as to generate the fastest possible convergence of the algorithm. However, as indicated above, a trade-off between reliable convergence and fast convergence has to be found. The problem of choosing a value of η is made even more difficult by the fact that the optimal value of this parameter will not stay constant during a run. Initially, when the weights are randomly assigned, there are many directions in weight space leading to a reduction in $\overline{\mathcal{E}}$, so that a large value of η can be used. As the training progresses, and the error approaches its minimum, smaller values of η should be used.

In the literature, several different methods for selecting a functional form for the variation of η have been suggested. A common choice is to let η vary with the training epoch (denoted n) according to

$$\eta(n) = \eta_0 \frac{1 + \frac{c}{\eta_0}\frac{n}{T}}{1 + \frac{c}{\eta_0}\frac{n}{T} + T\frac{n^2}{T^2}}. \tag{2.31}$$

Using this equation, a near-constant $\eta \approx \eta_0$ is obtained for small n. For $n >> T$, η varies as c/n.

The variation method introduced in Eq. (2.31) has the learning rate parameter decreasing in a pre-specified way, regardless of $\overline{\mathcal{E}}$. More complex variation schemes can be derived, in which the variation in η is determined by the performance of the algorithm. In such schemes η is reduced if $\Delta\overline{\mathcal{E}}$, i.e. the change in the training error from one training epoch to the next, is consistently positive (over a few training epochs). Similarly, η is increased if $\Delta\overline{\mathcal{E}}$ is consistently negative. If $\Delta\overline{\mathcal{E}}$ oscillates, i.e. if the variation is negative for some epochs and positive for others, η is left unchanged. Thus, a functional form for the variation $\Delta\eta$ is given by

$$\Delta\eta = \begin{cases} a & \text{if } \Delta\overline{\mathcal{E}} < 0 \text{ consistently;} \\ -b\eta & \text{if } \Delta\overline{\mathcal{E}} > 0 \text{ consistently;} \\ 0 & \text{otherwise,} \end{cases} \tag{2.32}$$

where a and b are constants.

Example 2.1 In order to illustrate how the backpropagation algorithm works, let us apply it to the network shown in Fig. 2.6. Initially, the weights are set as shown in the figure. The input signal is $x_1 = 1.0$ and $x_2 = 0.0$, and the desired output signal is $o_1 = 1.0$. The activation function is a sigmoid with $c = 1$, and the learning rate parameter η is equal to 1. Let us first compute the error. The output from the neurons in the hidden layer becomes

$$\begin{aligned} y_1^H &= \sigma\left(\sum_{p=0}^{2} w_{1p}^{I\to H} y_p^I\right) = \sigma(0.2 \times 1.0 + 0.3 \times 1.0 - 0.1 \times 0) = \\ &= \sigma(0.5) = \frac{1}{1 + \mathrm{e}^{-0.5}} = 0.6225, \end{aligned} \tag{2.33}$$

$$\begin{aligned} y_2^H &= \sigma\left(\sum_{p=0}^{2} w_{2p}^{I\to H} y_p^I\right) = \sigma(-0.2 \times 1.0 - 0.2 \times 1.0 + 0.1 \times 0) = \\ &= \sigma(-0.4) = \frac{1}{1 + \mathrm{e}^{0.4}} = 0.4013. \end{aligned} \tag{2.34}$$

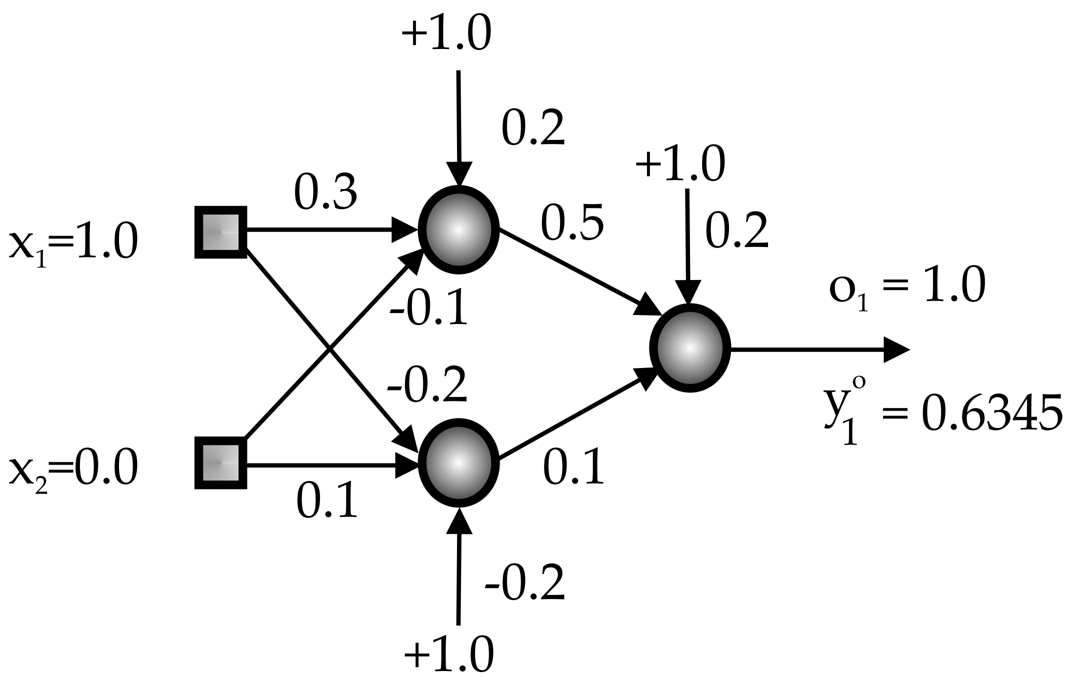

Figure 2.6: *The neural network considered in Example 2.1.*

Similarly, the output of the neuron in the output layer becomes

$$\begin{aligned} y_1^O &= \sigma\left(\sum_{s=0}^{2} w_{1s}^{H\to O} y_s^H\right) = \sigma(0.2\times 1.0 + 0.5\times 0.6225 + 0.1\times 0.4013) = \\ &= \sigma(0.5514) = 0.6345. \end{aligned} \tag{2.35}$$

Using Eq. (2.20), the error signal can be computed as

$$e_1 = o_1 - y_1^O = 1 - 0.6345 = 0.3655. \tag{2.36}$$

The derivative of the activation function is given by

$$\sigma'\left(\sum_{s=0}^{2} w_{1s}^{H\to O} y_s^H\right) = \sigma'(0.5514) = 0.2319. \tag{2.37}$$

Thus,

$$\delta_1 = e_1 \sigma'\left(\sum_{s=0}^{2} w_{1s}^{H\to O} y_s^H\right) = 0.3655\times 0.2319 = 0.0848. \tag{2.38}$$

Using Eq. (2.24), the modification of the weight $w_{10}^{H\to O}$ is obtained as

$$\Delta w_{10}^{H\to O} = \eta \delta_1 y_0^H = 1\times 0.0848\times 1 = 0.0848. \tag{2.39}$$

Using the same procedure, the change in the other two weights connecting the hidden layer to the output layer are obtained

$$\Delta w_{11}^{H\to O} = \eta \delta_1 y_1^H = 1 \times 0.0848 \times 0.6225 = 0.0528, \tag{2.40}$$
$$\Delta w_{12}^{H\to O} = \eta \delta_1 y_2^H = 1 \times 0.0848 \times 0.4013 = 0.0340. \tag{2.41}$$

Proceeding now to the hidden layer, we first compute κ_1 as follows

$$\begin{aligned} \kappa_1 &= \sigma'\left(\sum_{p=0}^{2} w_{1p}^{I\to H} y_p^I\right) \sum_{l=1}^{1} \delta_l w_{l1}^{H\to O} = \\ &= \sigma'(0.5)\delta_1 w_{11}^{H\to O} = 0.2350 \times 0.0848 \times 0.5 = 0.00996. \end{aligned} \tag{2.42}$$

The change in the weight $w_{10}^{I\to H}$ is now given by

$$\Delta w_{10}^{I\to H} = \eta \kappa_1 y_0^I = 1 \times 0.00996 \times 1 = 0.00996. \tag{2.43}$$

The modifications to the weights $w_{11}^{I\to H}$ and $w_{12}^{I\to H}$ are computed in the same way, resulting in (check!)

$$\Delta w_{11}^{I\to H} = 0.00996, \tag{2.44}$$
$$\Delta w_{12}^{I\to H} = 0. \tag{2.45}$$

Using the same method, κ_2 can be obtained, and thereby also the modifications of the weights entering the second neuron in the hidden layer. The result is (check!)

$$\Delta w_{20}^{I\to H} = 0.00204, \tag{2.46}$$
$$\Delta w_{21}^{I\to H} = 0.00204, \tag{2.47}$$
$$\Delta w_{22}^{I\to H} = 0. \tag{2.48}$$

With the updated weights, the output of the network becomes, using the same input signal ($x_1 = 1, x_2 = 0$)

$$y_1^O = 0.6649. \tag{2.49}$$

Thus, with the new weights, the error is reduced from 0.3655 to

$$e_1 = o_1 - y_1^O = 1 - 0.6649 = 0.3351. \tag{2.50}$$

2.3 Applications of backpropagation

Feedforward networks consistute the most common architecture for artificial neural networks, and the use of backpropagation for training feedforward networks is widespread. In this section, some examples of applications of backpropagation will be given. These examples represent only a small fraction of the possible applications, and you are requested to search for further information on the internet. We will start, however, with a brief discussion concerning the suitability of backpropagation for various problems.

2.3.1 Using backpropagation

The selection of a suitable architecture for an adaptive system is a difficult issue. While no general method can be provided, some guidelines can be given. First of all, it should be remembered that a neural network often represents a black-box solution to a problem: the network can be trained to learn almost any input-output mapping, but the internal representation of the mapping is often very difficult to analyze, mainly because the computation in neural networks is distributed, with each neuron performing only a small part of the overall computation. The distributed computation is a great advantage in hardware implementations, where the neural network must be able to provide correct predictions even if one (or a few) neuron fails.

Neural networks are also usually very good at interpolating between the input values used for training, and thus to generate correct output even for previously unseen input values (even in non-linear problems).

In industrial applications, it is often needed to find a model of e.g. a mechanical or an electrical system, which can generate the same output as the actual system for any input. The task of finding such a model is known as **system identification** and is a common application of neural networks. In cases where an exact physical model exists, it is often better to fit the parameters of this model rather than using a neural network. However, it is commonly so that the physical model is too complex for the available parameter-fitting techniques. In such cases, the use of a neural network is well motivated. Of course, some systems are so complex that there is no model available at all, and in such cases a neural network is a natural choice of representation.

It should also be noted that, even though the training of neural network may be complex, the computation of output from the network (once the training is completed) is almost instantaneous. This property is often particularly useful in system identification of complex dynamical systems; even if a physical model exists, it may be so complex that the computation of output cannot be performed in real time, thus motivating the use of a neural network instead of the physical model.

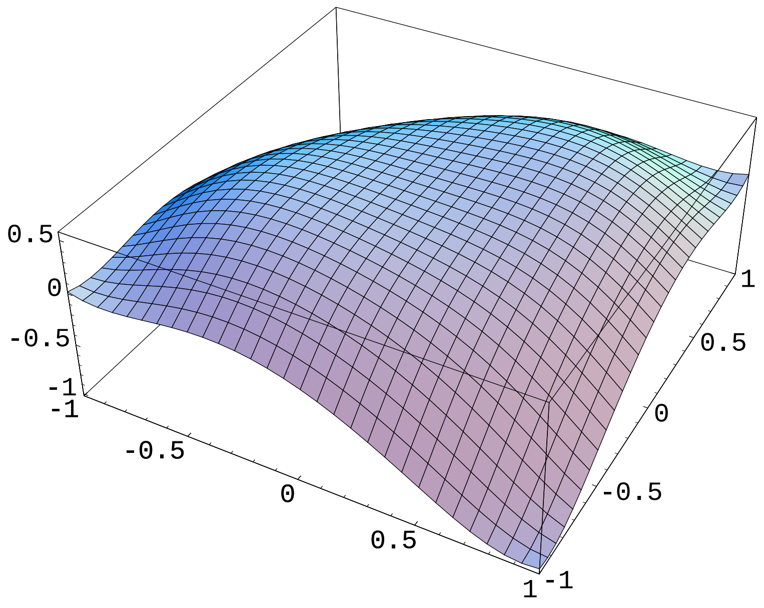

Figure 2.7: *The function* $f(x, y) = \frac{1}{2}\left(\sin \sqrt{2}xy + \cos \sqrt{3}(x^2 + y^2)\right)$.

2.3.2 Function approximation

Feedforward neural networks (FFNNs) are often used for approximating mathematical functions. As a specific example, consider the control of exhaust gas temperature in a car engine. Here, it has been common to use **lookup tables** to relate the various variables involved and to provide a good prediction of the temperature. However, this is a strongly non-linear problem with many variables involved, and thus the lookup tables must be large to provide a sufficiently fine-grained coverage of the input space. With restrictions on the memory storage space available, a neural network may provide a more compact representation of the function.

As an example, we will now consider an abstract example of function approximation. Consider the function

$$f(x, y) = \frac{1}{2}\left(\sin \sqrt{2}xy + \cos \sqrt{3}(x^2 + y^2)\right) \tag{2.51}$$

in the range $\{x, y\} \in [-1, 1]$. A training data set containing the values of x and y, as well as the corresponding value $f(x, y)$ at $441 = 21^2$ different points in $[-1, 1]$ was generated, as well as a validation set containing $400 = 20^2$ points. A backpropagation program, applied to a 2-4-1 FFNN, was used for the training. As both x and y, and the function $f(x, y)$ take values in $[-1, 1]$, $\tanh(cz)$ was used as the activation function.

The training error fell from an initial value of around 0.38 to a final value of 0.03, and the validation error fell from around 0.36 to around 0.03. The training

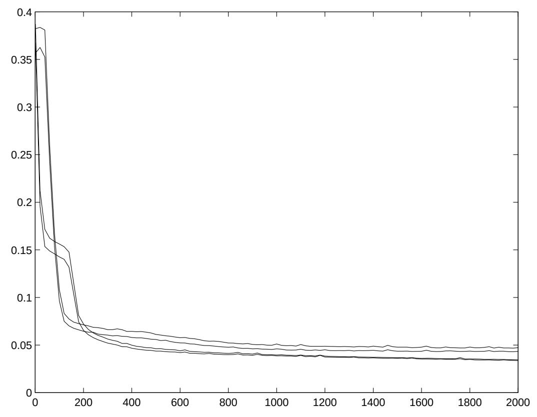

Figure 2.8: *Training and validation errors for two training cases: the two curves with a plateau stem from a case with* 10 *hidden neurons, and the remaining two curves stem from a case with* 4 *hidden neurons. In both cases, the validation error was slightly smaller than the training error.*

and validation errors for the first 2,000 epochs are shown in Fig. 2.8. The figure also shows the training an validation errors for a case with $n_H = 10$ hidden neurons. As can be seen, the error obtained in the case of $n_H = 10$ initially falls faster than for the smaller network. Then, however, the error reaches a plateau before starting to fall again. At the last epoch shown, the larger network has achieved a slightly smaller error than the network with $n_H = 4$.

An example of the performance of the final 2-4-1 FFNN is shown in Fig. 2.9. As can be seen, the output from the network (diamonds) at the training points approximate the function rather well, even though the training could have been extended further in this particular case.

2.3.3 Time series prediction

Consider a series of values $x(i), i = 1, \ldots, n$. **Time series prediction** is the problem of predicting $x(j + j_+)$, given a set of N earlier values $\{x(j - j_1), x(j - j_2), \ldots, x(j - j_N)\}$, for all relevant values of j. Commonly, the prediction is based on consecutive earlier values, i.e. $j_1 = 1, j_2 = 2$ etc., and the prediction of concerns the next value in the series i.e. $j_+ = 0$.

Time series prediction appears in many different contexts. Common applications include prediction of macroeconomic and financial time series, weather prediction, and earthquake prediction. Feedforward networks can be used for predicting time series, by using N earlier values of the time series as inputs to a neural network with a single output, whose desired value is the next element of the series. An example of such a network is shown in Fig. 2.10. N is sometimes referred to as the **lookback**.

In many cases, an FFNN can perform successful prediction of a time se-

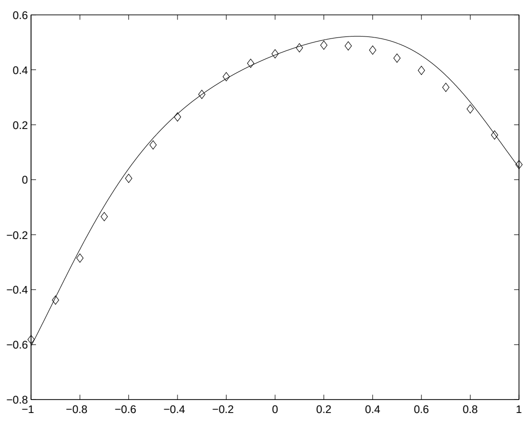

Figure 2.9: *A slice through the surface defined by* $z = f(x, y)$*, for* $x = 0.5$, $y \in [-1, 1]$*. The solid line shows* $f(0.5, y)$*, and the diamonds show the output from the 2-4-1 FFNN obtained after 20,000 training epochs.*

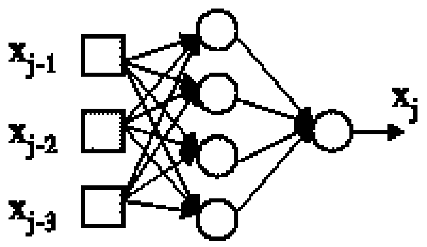

Figure 2.10: *A feedforward network for time series prediction. For this network, the lookback* N *is equal to 3.*

ries. However, such networks should not be applied uncritically to all time series prediction problems: the weights of a trained FFNN constitute a permanent memory that stores a representation of the data set used for the training. However, such a network has no *short-term* memory, other than that obtained from the fact that the input signals represent N consecutive earlier data points. An example should suffice to illustrate the problem. Consider the time series shown in Fig. 2.11. In the left panel, a prediction (disk) is made based on a given lookback (3, in this case). For any given value of the lookback, an FFNN will be completely oblivious to earlier values, which may lead to problems, as illustrated in the right panel: here, two different situations are superposed. In both cases the input values within the lookback window are the same, but the output that the FFNN is supposed to generate is different in the two cases. Of course, the lookback can be increased, but that just shifts the problem; a figure similar to Fig. 2.11 can be drawn for any value of the lookback.

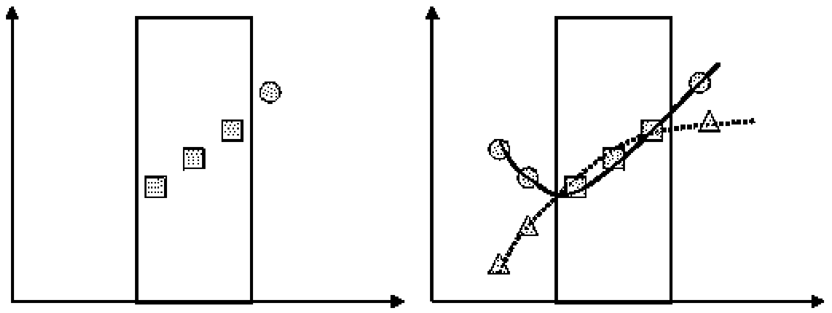

Figure 2.11: *An illustration of the potential problems caused by the lack of dynamic memory in FFNN. In the left panel, the prediction of an FFNN is shown for a given input, based on a lookback of 3. In the right panel, two different situations (with different desired network output) are shown. In both cases, the FFNN would give the same output.*

Thus, the lack of a short-term memory may, in some cases, limit the usefulness of FFNN for time series prediction. The problem can be mitigated by the introduction of feedback couplings in the network, i.e. by transforming the network to an RNN. On the other hand, such a solution renders (ordinary) backpropagation useless, and also increases the number of parameters available in the network, thus also increasing the risk of overfitting.

In cases where a mathematical model exists of the system that is to be predicted, it is often better to use this model than to introduce a neural network. However, in many applications (e.g. financial and macroeconomic time series) there exists no accurate mathematical model or, if there is a model, it is strongly non-linear (earthquake and weather prediction). In such cases, a neural network may be useful, as it will form its own model of the data set through the tuning of the network weights. However, the solution obtained from the neural network should preferably be compared with solutions obtained from simpler methods, such as e.g. a linear model. In fact, in weather (temperature) prediction, the best prediction is often to simply assume that the next value will be equal to the last available data point.

Weather prediction

The data set shown in Fig. 2.12 contains measurements of the annual rainfall in Fortaleza, Brazil, from 1849 to 1979. As is evident from the figure, there is a strong variation from year to year in the amount of rainfall. The lookback was set to 5, and the resulting set was divided into a training set with 100 elements (input-output pairs) and a validation set with 26 elements. The number of hidden neurons was set to 5 in the backpropagation run. Fig. 2.13 shows the training errors (bottom curve) and validation errors (top curve) ob-

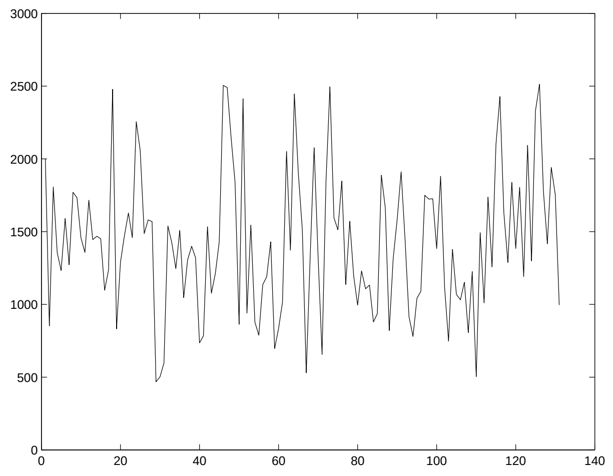

Figure 2.12: *Annual rainfall (cm) in Fortaleza, Brazil, 1849-1979.*

tained during the training. As is evident from the figure, overfitting occurs very quickly for this data set (using the selected FFNN setup and training parameters). From Fig. 2.13 it is clear that the training should be stopped around 7,000 epochs, at which point the training error is around 0.2. Note that the data set has been rescaled by first subtracting 400 from each data point, and then dividing the result by 2200, giving values in the range $[0,1]$. For comparison, the simple strategy of using the present value as the prediction of the next value yields (with the same rescaling parameters) an error of 0.20, i.e. the same as the FFNN. Scaling back to original units, this corresponds to a prediction error of around 440.0 cm.

Several tests were performed using different number of hidden neurons. However, no significant improvement could be obtained.

The choice of an optimal number of input elements (i.e. an optimal lookback) is also a non-trivial problem. However, a lower bound on the lookback can be obtained: given a certain input $\{x_1,\ldots,x_N\}$, an FFNN will always produce the same output, as illustrated in Fig. 2.11. Thus, if N is chosen too small there may be several cases with similar input and different desired output. In this case, a larger lookback should be used.

2.3.4 Image recognition

Image recognition has many applications. A common example is handwritten digit recognition, which is of importance e.g. for postal services, where the sorting of letters often is dependent on accurate (and fast) automatic reading of zip codes. In personal digital assistants (PDAs), recognition of handwritten letters and digits is also important for the functionality of the PDA.

In security applications, the recognition of facial features plays an essential

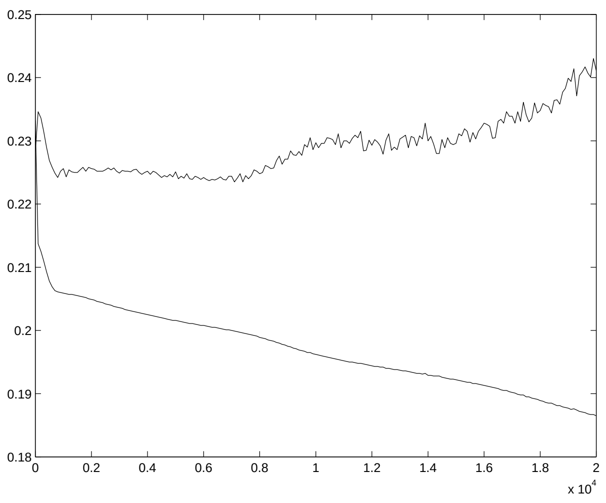

Figure 2.13: *Training error (bottom curve) and prediction error (top curve) from a backpropagation run using the rainfall data. Note the clear case of overfitting.*

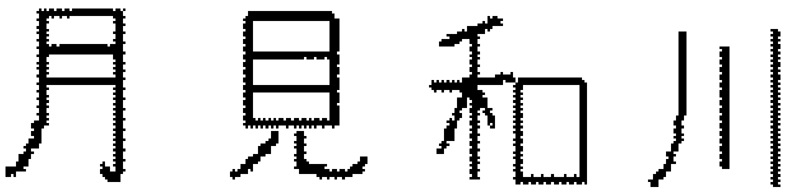

Figure 2.14: *Examples of kanji signs. From left to right, the signs are* tsuki *(moon),* miru *(to see),* wa *(peace, harmony), and* kawa *(river).*

role. The decision of whether or not to open a door can be based on the output from a system for automatic face recognition. Such a system must, however, be able to cope with variations in facial features such as e.g. small changes in the viewing angle, different hair cuts, presence or absence of glasses etc. Often, systems for person identification are not based on facial images but instead on other unique features of an individual, such as e.g. fingerprints or retina scans.

In all applications described above, it is possible to use FFNN. In realistic applications, the FFNNs often contain several (more than 2) layers, and are usually not fully connected. Instead, subnetworks acting as feature detectors are introduced in the network. The exact procedure for doing so will not be described here. Instead, a specific example will be presented.

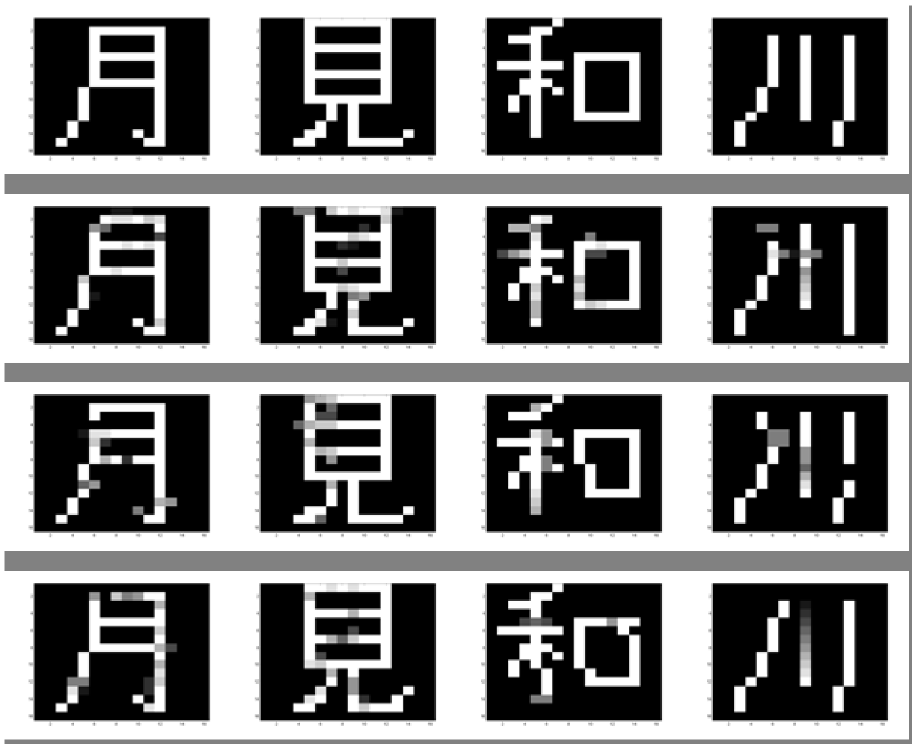

Figure 2.15: *A $256-4-2$ FFNN was trained to distinguish between the four kanji signs in the first row. The resulting FFNN produced the correct output for each kanji sign, including the 12 noisy versions shown in the bottom three rows.*

2.3.5 Automatic recognition of Kanji signs

Some languages, e.g. Japanese, are built up from pictograms (called kanji, in the case of Japanese). Here, each kanji sign (or combination of kanji signs) carries a specific meaning and a specific pronounciation[1]. Some examples of Kanji signs are shown in Fig. 2.14. In tasks involving image recognition, the number of input elements equals the total number of pixels in the image. Thus, if the image is n_x by n_y pixels, the number of input elements will be $n_x n_y$. The number of output elements can be chosen in several different ways. For example, a binary representation can be used, with several neurons that give either 0 or 1 as output. Since the output from neurons is a decimal number rather than an integer, **thresholding** is generally used to obtain a binary number as output. For example, in thresholding, any output signal below a threshold T would be set to 0, and any output signal above $1-T$ would be set to 1.

The binary number thus generated can encode the identity of the output signal via a lookup table. Alternatively, a single output neuron can be used, with each output encoded as a specific range within the interval $[0, 1]$.

[1]Often, however, each kanji sign can be pronounced in several different ways

An advantage with neural networks is their ability to generalize; An FFNN trained on a set of input-output pairs, is often able to produce the correct output for other input vectors than those previously encountered. This is illustrated in Fig. 2.15: an FFNN was trained to distinguish between the four kanji signs in shown in the top row. The image resolution was 256 pixels, and the backpropagation algorithm was applied to a $256 - 4 - 2$ network. The desired outputs for the different kanji signs were encoded as two-bit binary numbers, with 00 representing the first kanji sign, 01 the second etc. The network was then applied to the 12 kanji signs shown in the three bottom rows. In all cases, the network performed well, identifying the 12 signs with approximately the same (low) error rate as that obtained when applying the network to the training data.

Input		Desired output
0.0	0.0	0.027
0.3	0.0	0.343
0.7	0.2	0.807
0.1	0.4	0.302
0.5	0.4	0.672
0.8	0.6	1.080
0.2	0.8	0.551
0.4	0.9	0.794

Table 2.2: *Data table for Problem 2.1.*

Problems

2.1 Use stochastic gradient descent to find the weights and the bias term of the single–neuron network (with 2 input elements) that best represents the data in Table 2.2. For the response function, use $\sigma(x) = x$.

2.2 From the derivation above, we know that a one–layer network cannot represent the XOR function. Assume, anyway, that an attempt is made to represent this function using a network with one neuron and two input elements. What are the values of the weights and the bias term that minimize the sum (over all the input–output pairs) of the squared error?

2.3 Show that the derivative of the response function

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-cz}}, \tag{2.52}$$

satisfies the equation

$$\sigma'(x) = c\,\sigma(x)(1 - \sigma(x)). \tag{2.53}$$

2.4 Perform (by hand) a backpropagation step and determine the new weights and biases for the network shown in Fig. 2.16, assuming that the desired output signal is equal to 0.5. Set the learning rate parameter η to 0.3. For the response function, use a sigmoid, Eq. (2.52), with $c = 3$. Note: show *clearly* each step in the calculation.

2.5 Use backpropagation on a 1-n_{H}-1 neural network to approximate the function $f(x) = \sin(x)(1 - \mathrm{e}^{-x})$ on the interval $x \in [0, 1]$. Generate one training data set and one validation set by sampling the function. Test the backpropagation algorithm for various values of n_{H}. Show the training error and the

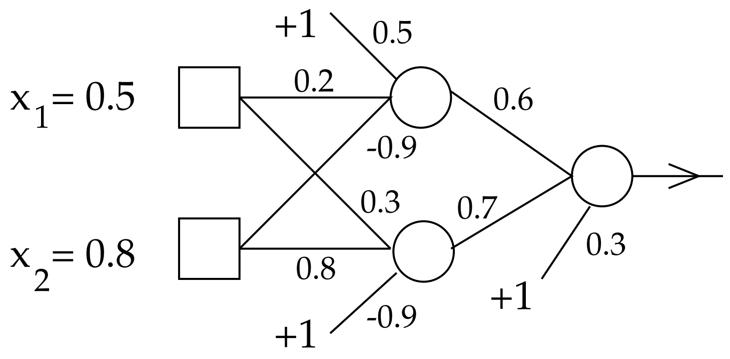

Figure 2.16: *Network for Problem 2.4.*

validation error, as well as the final network, for each n_{H}. Also, for networks with $n_{\mathrm{H}} >> 1$, investigate the effect of losing one neuron in the middle layer, by computing (by hand) the output and the error over the training set.

2.6 Collect a time series data set x on the internet (e.g. a macroeconomic or meteorological data set), containing measurements for at least 100 time steps. Divide the data set into a training set and a validation set. The latter should contain at least 20% of the data. Next, compute the RMS error, defined as

$$\sqrt{\frac{1}{L}\sum_i (o(i) - y(i))^2}, \tag{2.54}$$

over both the training data set and the validation data set, assuming that the simplest possible prediction is made, namely $x(i+1) = x(i)$. L denotes the number of predictions, $o(i)$ the desired output, and $y(i)$ the actual output.

Next, design an adequate FFNN, and train it using backpropagation until the RMS error over both the training data set and the validation data set fall below the RMS error obtained using simple prediction $x(i+1) = x(i)$.

Answers to selected exercises

2.1 $w_1 = 0.99, w_2 = 0.40, b = 0.033.$

Chapter 3

Neural memories

To have a **memory**, i.e. the ability to remember previously encountered things or situations is an important property of many living systems, and also many robotic systems, particularly those that go beyond simple reactive behaviors.

While the methods for learning in feedforward networks that were studied in the previous section do store information in the connection weights of the respective network, they do not involve the explicit storage *and* recall of prespecified patterns. In other words, a network trained by those methods does not function as an **associative memory**, i.e. a memory where a cue or a **stimulus** leads to **recall** of a specific stored memory.

We will begin our discussion of neural memories and algorithms for information storage by considering linear memories, and will then continue with a brief description of a class of non–linear recurrent memories known as Hopfield networks.

3.1 Feedforward memories

In the late 1940s, Donald Hebb introduced a learning method for biological neural networks known as the **Hebb rule**, which can be described as follows: if two neurons fire together (or, in any case, within a very short period of time), the connection between them will be strengthened. This type of learning has been observed in the brain, even though the simple Hebb rule cannot account for all types of learning in biological neural networks. A formalized version of Hebb's idea has turned out to be very useful for artificial neural networks as well, as we shall now see.

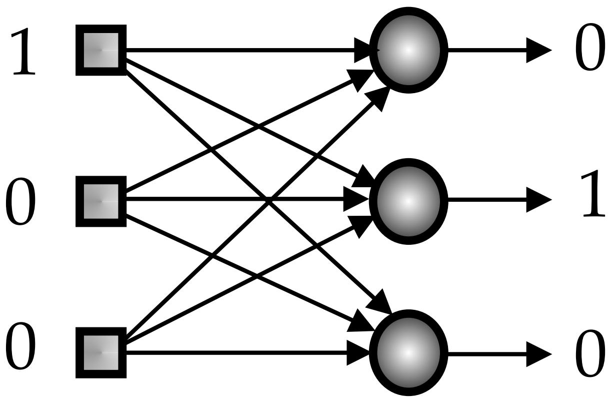

Figure 3.1: *A simple linear associative memory. The activation function of the three output neurons is given by $\sigma(z) = z$.*

3.1.1 Linear memories

Consider the simple one–layer neural network shown in Fig. 3.1. We wish to store the vector

$$\mathbf{r} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}. \tag{3.1}$$

in this network, in such a way that it is recalled, i.e. appears in the output neurons, when the network is presented with the cue

$$\mathbf{c} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}. \tag{3.2}$$

In the figure, the input pattern is shown on the left side of the network, and the output pattern on the right side, and our task is to determine the network weights that will produce this situation. Let us assume that the initial network weights are all zero, and let us then apply Hebb's idea. Thus, for neurons that fire together, the connection weight is strengthened – let us put it equal to 1, whereas no change occurs in connection weights where the input and output neurons are not firing together. With this method, only weight w_{21} will be set to one, and all other weights will remain equal to zero. Thus, the corresponding weight matrix $\mathbf{W}$ will take the form

$$\mathbf{W} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \tag{3.3}$$

Now, if the network is presented with the cue $\mathbf{c}^T = (1,0,0)$, we obtain the output

$$\mathbf{Wc} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \tag{3.4}$$

i.e. the stored vector $\mathbf{r}$.

We can now formalize the method for memory storage a bit further. Consider a general input vector (cue) $\mathbf{c}$ with n components, and a corresponding output vector $\mathbf{r}$ with m components. In accordance with Hebb's rule, we set the elements of the memory matrix of a **linear memory** according to

$$\mathbf{W} = \mathbf{rc}^T = \begin{pmatrix} r_1c_1 & r_1c_2 & \cdots & r_1c_n \\ r_2c_1 & r_2c_2 & \cdots & \vdots \\ \vdots & \ddots & & \vdots \\ r_mc_1 & & \ddots & r_mc_n \end{pmatrix}. \tag{3.5}$$

Note that the product in Eq. (3.5) is *not* a scalar product. Instead, it is known as the **outer product** or **tensor product** between the two vectors $\mathbf{r}$ and $\mathbf{c}$ and is also written $\mathbf{r}\otimes\mathbf{c}$. Now, when the n input neurons of the network are presented with the input vector $\mathbf{r}$, the m output neurons will give the output

$$\mathbf{Wc} = \mathbf{rc^Tc} = \mathbf{r}, \tag{3.6}$$

provided that the cue vector $\mathbf{c}$ has been normalized such that $|\mathbf{c}| = 1$, which we will assume to be the case from now on.

Normally, of course, several vectors need to be stored, rather than just one. In this case, the memory matrix **M** is obtained as the sum of the memory matrices corresponding to the individual vectors that are stored

$$\mathbf{M} = \sum_k \mathbf{W}_k = \sum_k \mathbf{r_k c_k^T}, \tag{3.7}$$

where the index k enumerates the stored patterns. If the memory matrix **M** is presented with the cue $\mathbf{c_j}$, the output vector is obtained as

$$\begin{aligned} \mathbf{o} &= \mathbf{Mc}_j = \sum_k (\mathbf{r}_k\mathbf{c}_k^{\mathrm{T}})\mathbf{c}_j = \sum_k (\mathbf{c}_k^{\mathrm{T}}\mathbf{c}_j)\mathbf{r}_k = \\ &= (\mathbf{c}_j^{\mathrm{T}}\mathbf{c}_j)\mathbf{r}_j + \sum_{k\neq j}(\mathbf{c}_k^{\mathrm{T}}\mathbf{c}_j)\mathbf{r}_k = \mathbf{r}_j + \sum_{k\neq j}(\mathbf{c}_k^{\mathrm{T}}\mathbf{c}_j)\mathbf{r}_k. \end{aligned} \tag{3.8}$$

From this equation, it is evident that the memory exhibits **perfect recall** such that it gives exactly the stored pattern $\mathbf{r}_j$ when presented with $\mathbf{c}_j$, *only* if the cue vectors are orthogonal to each other. If not, there will be some **interference** between the different memories. Clearly, demanding perfect recall, one cannot store more than n distinct memories in a linear memory.

If the network is presented with a cue that deviates only slightly from one of the $\mathbf{c}_j$-vectors, the output will also deviate only slightly from $\mathbf{r}_j$:

$$\mathbf{M}(\mathbf{c}_j + \delta\mathbf{c}_j) = \mathbf{r}_j + \delta\mathbf{r}_j, \tag{3.9}$$

where $\delta\mathbf{r}_j = \mathbf{M}\delta\mathbf{c}_j$, and perfect recall has been assumed.

Example 3.1 Construct a linear memory with the following properties:

$$\begin{aligned} \mathbf{c}_1 = (1,0,0)^{\mathrm{T}} &\rightarrow \mathbf{r}_1 = (2,0,3)^{\mathrm{T}}, && (3.10)\\ \mathbf{c}_2 = (0,1,0)^{\mathrm{T}} &\rightarrow \mathbf{r}_2 = (-2,4,1)^{\mathrm{T}}, && (3.11)\\ \mathbf{c}_3 = (0,0,1)^{\mathrm{T}} &\rightarrow \mathbf{r}_3 = (0,-2,-3)^{\mathrm{T}}. && (3.12) \end{aligned}$$

Check that the matrix memory exhibits perfect recall for the three stored patterns. What will be the output vector if the cue is chosen as an approximation to $\mathbf{c}_1$, namely $(0.9, 0.1, -0.1)^T/|(0.9, 0.1, -0.1)|$?

Solution: According to Eq. (3.7), the memory matrix $\mathbf{M}$ is given by

$$\begin{aligned} \mathbf{M} &= \mathbf{r}_1 \otimes \mathbf{c}_1 + \mathbf{r}_2 \otimes \mathbf{c}_2 + \mathbf{r}_3 \otimes \mathbf{c}_3 = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 3 & 0 & 0 \end{pmatrix} + \\ &+ \begin{pmatrix} 0 & -2 & 0 \\ 0 & 4 & 0 \\ 0 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -2 \\ 0 & 0 & -3 \end{pmatrix} = \begin{pmatrix} 2 & -2 & 0 \\ 0 & 4 & -2 \\ 3 & 1 & -3 \end{pmatrix}. \end{aligned} \tag{3.13}$$

Using $\mathbf{M}$, we get

$$\mathbf{M}\mathbf{c}_1 = \begin{pmatrix} 2 \\ 0 \\ 3 \end{pmatrix} \equiv \mathbf{r}_1, \; \mathbf{M}\mathbf{c}_2 = \begin{pmatrix} -2 \\ 4 \\ 1 \end{pmatrix} \equiv \mathbf{r}_2, \; \mathbf{M}\mathbf{c}_3 = \begin{pmatrix} 0 \\ -2 \\ -3 \end{pmatrix} \equiv \mathbf{r}_3. \tag{3.14}$$

Thus, the memory exhibits perfect recall.

Finally, with the cue vector $(0.9, 0.1, -0.1)^T/|(0.9, 0.1, -0.1)|$, the output is equal to $(1.76, 0.66, 3.40)^T$, a fairly good approximation of the vector $\mathbf{r}_1$.

3.1.2 Memories with threshold

In the presentation above, the activation function was linear. In order to reduce the effects of interference between different stored patterns, a threshold can be introduced so that the output of the network is given by

$$\mathbf{o}' = (o'_1, o'_2, \ldots, o'_m)^T = (\sigma(o_1), \sigma(o_2), \ldots, \sigma(o_m))^T, \tag{3.15}$$

where the vector $\mathbf{o} = (o_1, o_2, \ldots, o_m)^T$ is equal to $\mathbf{Mc}$, and the threshold function is given by

$$\sigma(z) = \begin{cases} 1 & \text{if z} \geq T \\ 0 & \text{otherwise} \end{cases}, \tag{3.16}$$

where T is the threshold. Using a threshold function, interference below the threshold can be eliminated, provided that the allowed values in the output are 0 or 1. Furthermore, a memory with threshold can exhibit both **error correction** (directing the system toward correct recall of a stored pattern $\mathbf{r}_j$ when presented with a slightly erroneous cue $\mathbf{c}'_j$) and **memory completion** (recalling the correct memory even if the signals from one or a few input neurons are absent), which is really a special case of error correction).

Example 3.2: Construct a memory matrix M that stores the following cue–recall pairs:

$$\mathbf{c}_1 = \frac{1}{2}(1,1,1,1)^{\mathrm{T}} \quad , \quad \mathbf{r}_1 = (1,0)^{\mathrm{T}}, \tag{3.17}$$

$$\mathbf{c}_2 = (1,0,0,0)^{\mathrm{T}} \quad , \quad \mathbf{r}_2 = (0,1)^{\mathrm{T}}. \tag{3.18}$$

Show that, in the absence of threshold, there is some interference between the two stored memories. Show also that a threshold of 0.75 eliminates the interference. Using the same threshold, investigate what happens if the network is presented with a slightly incorrect version of the cue vector $\mathbf{c}_2$, namely $(0.9, 0.1, 0, 0)^T/|0.9, 0.1, 0, 0|$. Show also that the network can perform memory completion on an incomplete version of $\mathbf{c}_1$: $(1,0,1,1)^T/\sqrt{3}$.
Solution: Forming the memory matrix M using Eq. (3.5) we obtain

$$\mathbf{M} = \begin{pmatrix} 0.5 & 0.5 & 0.5 & 0.5 \\ 1 & 0 & 0 & 0 \end{pmatrix}. \tag{3.19}$$

Applying the cue $\mathbf{c}_1$ the output $\mathbf{o}_1 = \mathbf{Mc}_1 = (1, 0.5)^T$ is obtained, and for the cue $\mathbf{c}_2$, the output becomes $\mathbf{o}_2 = (0.5, 1)^T$. Hence, the recall is not perfect: some interference is present. Introducing the threshold $T = 0.75$, the output signals become $\mathbf{o}'_1 = (\sigma(1), \sigma(0.5))^T = (1,0)^T = \mathbf{r}_1$ and $\mathbf{o}'_2 = (\sigma(0.5), \sigma(1))^T = (0,1)^T = \mathbf{r}_2$.

Presenting the noisy input signal $(0.9, 0.1, 0, 0)^T/|0.9, 0.1, 0, 0|$, the unthresholded output becomes $(0.552, 0.994)$, and after thresholding we obtain the output $(0,1) = \mathbf{r}_2$. Using instead the incomplete version of $\mathbf{c}_1$, $(1,0,1,1)^T/\sqrt{3}$, the unthresholded output is $(0.866, 0.577)$ which, after thresholding, becomes $(1, 0) = \mathbf{r}_1$.

3.2 Hopfield memories

A very important property of biological memories (human or otherwise) is their ability to perform both error correction and memory completion. The thresholded neural memories listed above were able to do this, at least to some extent. However, biological memory is much more complex than a simple input–output mapping, and there exists other ANN models of memory which tries to take this into account. One such model is the **Hopfield model** , which will now be introduced.

3.2.1 The Hopfield algorithm

We begin the discussion of Hopfield memories (or Hopfield networks – the two terms will be used interchangeably) by presenting the relevant equations. Consider a situation where N memory patterns are to be stored in a memory consisting of n neurons. In the case of a Hopfield memory, the network is almost **fully connected**: each neuron is coupled to all other neurons (except itself). The output of a neuron in a Hopfield network is computed as

$$y_i = \sigma\left(\sum_{j=1}^{n} m_{ij} y_j\right), \tag{3.20}$$

where σ is the response function, and m_{ij} are the connection weights ($m_{ii} \equiv 0$). The updating rule for such a neuron is similar to the rule for the MCP neuron (see Chapter 1), except that it uses the output of all other neurons in the network as its input. Assume now that each pattern is represented by a vector **r** with n binary elements. The elements of the memory vector could, for instance, be chosen from the set $\{0, 1\}$, but the equations become more elegant if instead they are chosen from the set $\{-1, 1\}$ which we will use here. With this choice, the function σ must be taken as the signum function, with the properties

$$\operatorname{sgn}(z) = \begin{cases} \text{-1} & \text{if} \quad z < 0, \\ 1 & \text{if} \quad z > 0. \end{cases} \tag{3.21}$$

In accordance with the Hebb rule, the weights of the network are set according to

$$m_{ij} = \frac{1}{N}\sum_{\nu=1}^{N} r_i^{(\nu)} r_j^{(\nu)}, \;\; m_{ii} = 0. \tag{3.22}$$

Now, once the patterns have been stored in the Hopfield memory, the recall is obtained as follows: First, the network is initialized by presenting it with the cue $\mathbf{c}$ (which must also be n–dimensional)

$$y_i(0) = c_i, \; i = 1, \ldots, n. \tag{3.23}$$

Then, the network is updated **asynchronously**, i.e. one neuron at a time. A neuron k is chosen at random. Its output is computed as

$$y_k \to y'_k = \operatorname{sgn}\left(\sum_{j=1}^{n} m_{kj} y_j\right). \tag{3.24}$$

If the input to the sgn function is exactly equal to zero, y_k is left unchanged. This iteration, consisting of the selection of a random neuron and output modification according to Eq. (3.24), is repeated until none of the output signals changes anymore. The values of the components of the vector $\mathbf{y}$ are then taken as the output of the network.

Example 3.3: Store the vectors $\mathbf{r}^{(1)} = (1,1,1)^T$ and $\mathbf{r}^{(2)} = (-1,-1,-1)^T$ in a Hopfield network. How does the corresponding weight matrix look? Which memory is recalled if the network is subjected to the input signal $\mathbf{c} = (1,1,-1)^T$?
Solution: According to Eq. (3.22), the memory matrix $\mathbf{m}$ will have the following elements:

$$m_{11} = 0, \tag{3.25}$$

$$m_{12} = \frac{1}{2}\sum_{\nu=1}^{2} r_1^{(\nu)} r_2^{(\nu)} = \frac{1}{2}\left(1 \times 1 + (-1) \times (-1)\right) = 1, \tag{3.26}$$

etc. The complete matrix looks as follows

$$\mathbf{m} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}. \tag{3.27}$$

With $\mathbf{c} = (1,1,-1)^T$, the output signals of the two first neurons are unchanged

$$y_1 \to y'_1 = \operatorname{sgn}(0 \times 1 + 1 \times 1 + 1 \times (-1)) = \operatorname{sgn}(0) \Rightarrow \text{no change}, \tag{3.28}$$

$$y_2 \to y'_2 = \operatorname{sgn}(1 \times 1 + 0 \times 1 + 1 \times (-1)) = \operatorname{sgn}(0) \Rightarrow \text{no change}. \tag{3.29}$$

However, for the third neuron (whenever it is selected by the random selection procedure), the output becomes

$$y_3 \to y'_3 = \operatorname{sgn}(1 \times 1 + 1 \times 1 + 0 \times (-1)) = \operatorname{sgn}(2) = 1. \tag{3.30}$$

Thus, when the third neuron is selected, the output of the network changes according to $(1,1,-1)^T \rightarrow (1,1,1)^T$. When this state has been reached, no further change occurs (check!). The Hopfield memory was able to correct the error in the cue **c** and converge on the stored memory closest to the noisy cue.

3.2.2 Interpretation of the Hopfield network

The Hopfield model is, in fact, inspired more by physics than biology, and it has strong analogies with the theory of spin glasses in statistical physics. The analysis of this analogy is beyond the scope of this book, but we shall nevertheless briefly discuss how the Hopfield memory operates. Borrowing from physics, we can define an **energy function** for the Hopfield network as

$$E = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} m_{ij} y_i y_j. \tag{3.31}$$

Let us now investigate the behavior of E as the Hopfield algorithm is applied to the network. Consider a neuron k whose output undergoes a change according to the Hopfield algorithm in Eq. (3.24). There are two possible changes, namely $\Delta y_k = y'_k - y_k = 1-(-1) = 2$ and $\Delta y_k = y'_k - y_k = -1-1 = -2$. The new value of the energy function E becomes

$$\begin{aligned}
E' &= -\frac{1}{2}\sum_{\substack{i=1\\i\neq k}}^{n}\sum_{\substack{j=1\\j\neq k}}^{n} m_{ij}y_iy_j - \frac{1}{2}\sum_{\substack{j=1\\j\neq k}}^{n} m_{kj}y'_ky_j - \frac{1}{2}\sum_{\substack{i=1\\i\neq k}}^{n} m_{ik}y_iy'_k = \\
&= -\frac{1}{2}\sum_{\substack{i=1\\i\neq k}}^{n}\sum_{\substack{j=1\\j\neq k}}^{n} m_{ij}y_iy_j - \sum_{\substack{j=1\\j\neq k}}^{n} m_{jk}y_jy'_k = \\
&= -\frac{1}{2}\sum_{\substack{i=1\\i\neq k}}^{n}\sum_{\substack{j=1\\j\neq k}}^{n} m_{ij}y_iy_j - \sum_{\substack{j=1\\j\neq k}}^{n} m_{jk}y_j(\Delta y_k + y_k) = \\
&= -\frac{1}{2}\sum_{\substack{i=1\\i\neq k}}^{n}\sum_{\substack{j=1\\j\neq k}}^{n} m_{ij}y_iy_j - \sum_{\substack{j=1\\j\neq k}}^{n} m_{jk}y_jy_k - \sum_{\substack{j=1\\j\neq k}}^{n} m_{jk}y_j\Delta y_k = \\
&= -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} m_{ij}y_iy_j - \sum_{j=1}^{n} m_{jk}y_j\Delta y_k = E - \Delta y_k\sum_{j=1}^{n} m_{kj}y_j,
\end{aligned} \tag{3.32}$$

where, in the last step, the condition $j \neq k$ can be dropped, since $m_{ii} \equiv 0$ anyway. Furthermore, the fact that the weight matrix is symmetric has also been used. Thus, the change in the energy function equals

$$\Delta E = E' - E = -\Delta y_k \sum_{j=1}^{n} m_{kj} y_j. \tag{3.33}$$

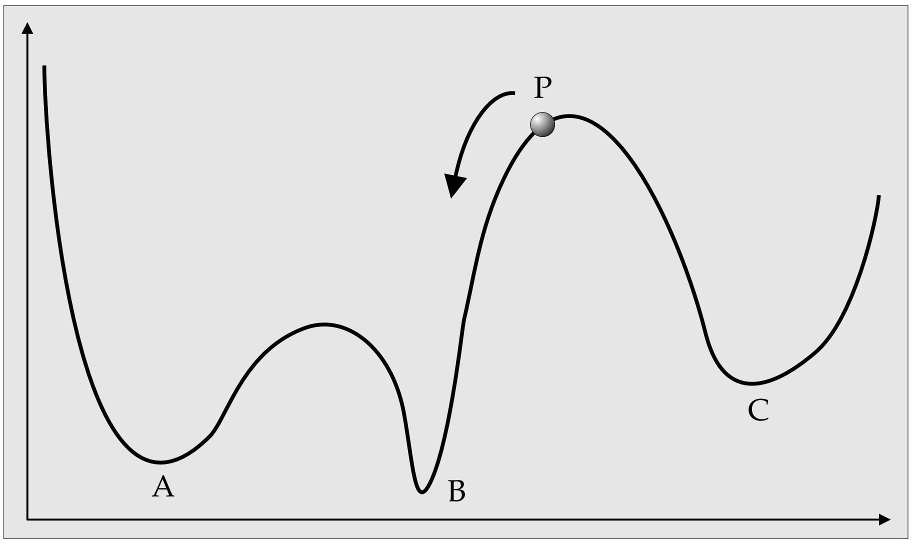

Figure 3.2: *A one–dimensional energy landscape. Three attractors (A,B, and C) are shown. The arrow shows the motion of the point P under the influence of the Hopfield algorithm.*

Now, if $\Delta y_k < 0$ the input to the signum function must have been negative. However, the input to the signum function is exactly equal to $\sum_{j=1}^{n} m_{kj} y_j$, i.e. the sum in Eq. (3.33). One the other hand, if $\Delta y_k > 0$, the sum in Eq. (3.33) must have been positive. Either way, the product of Δy_k and $\sum_{j=1}^{n} m_{kj} y_j$ will be positive, and ΔE will therefore be negative.

Thus, when the Hopfield algorithm is applied, the energy E will always decrease, until a minimum has been reached. Thereafter, $\Delta E = 0$. The function E is an example of a so called **Lyapunov function**, and an interpretation in terms of an **energy landscape** can now be introduced. An example is shown in Fig. 3.2. For simplicity, only a one–dimensional landscape has been drawn. Every vector that is presented to the network is represented by a point in the energy landscape. The Hopfield algorithm will move the point toward an **attractor** i.e. a local minimum in the energy landscape, as shown in Fig.3.2.

However, there is no guarantee that the algorithm will converge on one of the stored memory patterns. The energy landscape may contain **false attractors**, i.e. local minima in the energy landscape that do not correspond to any stored vector. An example of a false attractor is shown in Fig. 3.3.

Even though the possible presence of false attractors complicates the situation a bit, we have seen that a Hopfield network is able to converge on the correct stored pattern even when it is presented with a noisy or incomplete version of that pattern. However, the storage capacity of Hopfield networks is not very large, and several extensions and modifications of the Hopfield model have been suggested. One such extension is the **Boltzmann machine** which, however, will not be treated here.

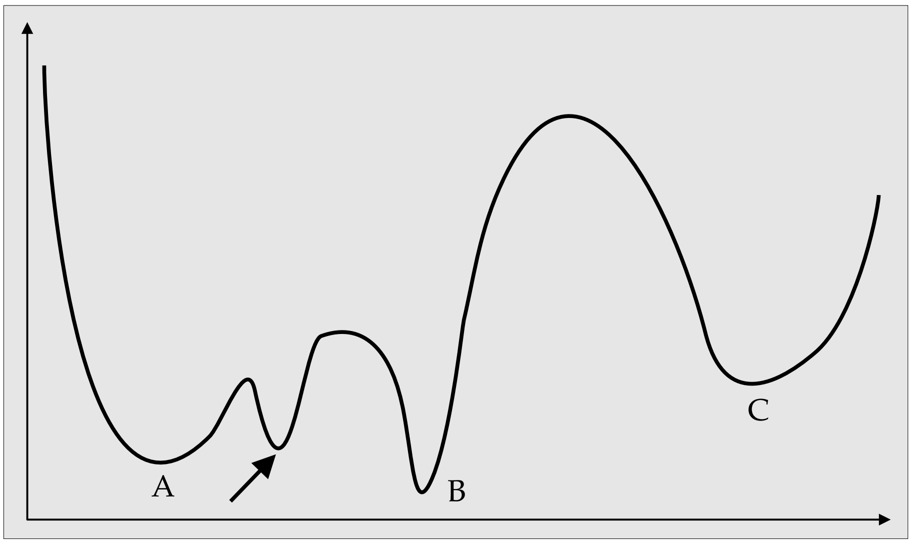

Figure 3.3: *One-dimensional energy landscape with one false attractor, indicated by an arrow.*

Problems

3.1 a) Generate a linear memory that stores the following associations:

$$\mathbf{c}_1 = \frac{1}{\sqrt{2}}(1,1,0)^{\mathrm{T}} \rightarrow \mathbf{r}_1 = (1,-3,5)^{\mathrm{T}}, \tag{3.34}$$

$$\mathbf{c}_2 = \frac{1}{\sqrt{2}}(-1,1,0)^{\mathrm{T}} \rightarrow \mathbf{r}_2 = (-2,4,-6)^{\mathrm{T}}, \tag{3.35}$$

$$\mathbf{c}_3 = (0,0,1)^{\mathrm{T}} \rightarrow \mathbf{r}_3 = (3,-3,2)^{\mathrm{T}}. \tag{3.36}$$

Check that the linear memory associates perfectly when presented with the three input signals $\mathbf{c}_1, \mathbf{c}_1$, and $\mathbf{c}_3$.
b) What is the output if the input is chosen as an approximation of the vector $\mathbf{c}_3$, namely $(0, 0.2425, 0.9701)^{\mathrm{T}}$?

3.2 a) The two vectors $\mathrm{r}^{(1)} = (-1,-1,-1,-1,-1)^{\mathrm{T}}$ and $\mathrm{r}^{(1)} = (1,1,-1,1,-1)^{\mathrm{T}}$ are to be stored in a Hopfield network. Determine the weight matrix $\mathbf{m}$.
b) The Hopfield network with the matrix $\mathbf{m}$ from $\mathbf{a}$) is exposed to the input signal $\mathbf{c} = (1,-1,1,-1,-1)^{\mathrm{T}}$. The Hopfield algorithm generally updates the output signal for one randomly chosen neuron at a time. Assume that the neurons, in this case, happen to be updated in the order $1,2,3,4,5$, i.e. neuron 1 is updated first, followed by neuron 2 etc. Determine the output signal from all 5 neurons after *each* of the 5 updating steps.

Answers to exercises

3.1 a)

$$M = \frac{1}{\sqrt{2}} \begin{pmatrix} 3 & \text{-1} & 3\sqrt{2} \\ \text{-7} & 1 & \text{-}3\sqrt{2} \\ 11 & \text{-1} & 2\sqrt{2} \end{pmatrix}. \tag{3.37}$$

b) $\mathbf{r} = (2.739, -2.739, 1.769)$.

3.2 a) The weight matrix takes the following form

$$\mathbf{m} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

b) Starting from $\mathbf{y}^{(0)} = \mathrm{c} = (1, -1, 1, -1, -1)^{\mathrm{T}}$, the output from the network, after the first step, is

$$\begin{aligned} y_1 \to y_1' &= \operatorname{sgn}\left(\sum_{j=1}^{5} m_{1j} y_j\right) = \operatorname{sgn}(-2) = -1 \Rightarrow \\ &\Rightarrow \mathbf{y}^{(1)} = (-1, -1, 1, -1, -1)^{\mathrm{T}} \end{aligned} \tag{3.38}$$

For the subsequent steps, the output vectors are

$$y_2' = -1 \Rightarrow \mathbf{y}^{(2)} = (-1, -1, 1, -1, -1)^{\mathrm{T}} \tag{3.39}$$

$$y_3' = -1 \Rightarrow \mathbf{y}^{(3)} = (-1, -1, -1, -1, -1)^{\mathrm{T}} = \mathbf{r}^{(1)} \tag{3.40}$$

$$y_4' = -1 \Rightarrow \mathbf{y}^{(4)} = (-1, -1, -1, -1, -1)^{\mathrm{T}} \tag{3.41}$$

$$y_5' = -1 \Rightarrow \mathbf{y}^{(5)} = (-1, -1, -1, -1, -1)^{\mathrm{T}} \tag{3.42}$$