

# 1

# Introduction

---

A first wave of interest in neural networks (also known as ‘connectionist models’ or ‘parallel distributed processing’) emerged after the introduction of simplified neurons by McCulloch and Pitts in 1943 (McCulloch & Pitts, 1943). These neurons were presented as models of biological neurons and as conceptual components for circuits that could perform computational tasks.

When Minsky and Papert published their book *Perceptrons* in 1969 (Minsky & Papert, 1969) in which they showed the deficiencies of perceptron models, most neural network funding was redirected and researchers left the field. Only a few researchers continued their efforts, most notably Teuvo Kohonen, Stephen Grossberg, James Anderson, and Kunihiro Fukushima.

The interest in neural networks re-emerged only after some important theoretical results were attained in the early eighties (most notably the discovery of error back-propagation), and new hardware developments increased the processing capacities. This renewed interest is reflected in the number of scientists, the amounts of funding, the number of large conferences, and the number of journals associated with neural networks. Nowadays most universities have a neural networks group, within their psychology, physics, computer science, or biology departments.

Artificial neural networks can be most adequately characterised as ‘computational models’ with particular properties such as the ability to adapt or learn, to generalise, or to cluster or organise data, and which operation is based on parallel processing. However, many of the above-mentioned properties can be attributed to existing (non-neural) models; the intriguing question is to which extent the neural approach proves to be better suited for certain applications than existing models. To date an equivocal answer to this question is not found.

Often parallels with biological systems are described. However, there is still so little known (even at the lowest cell level) about biological systems, that the models we are using for our artificial neural systems seem to introduce an oversimplification of the ‘biological’ models.

In this course we give an introduction to artificial neural networks. The point of view we take is that of a computer scientist. We are not concerned with the psychological implication of the networks, and we will at most occasionally refer to biological neural models. We consider neural networks as an alternative computational scheme rather than anything else.

These lecture notes start with a chapter in which a number of fundamental properties are discussed. In chapter 3 a number of ‘classical’ approaches are described, as well as the discussion on their limitations which took place in the early sixties. Chapter 4 continues with the description of attempts to overcome these limitations and introduces the back-propagation learning algorithm. Chapter 5 discusses recurrent networks; in these networks, the restraint that there are no cycles in the network graph is removed. Self-organising networks, which require no external teacher, are discussed in chapter 6. Then, in chapter 7 reinforcement learning is introduced. Chapters 8 and 9 focus on applications of neural networks in the fields of robotics and image processing respectively. The final chapters discuss implementational aspects.



# 2

## Fundamentals

---

The artificial neural networks which we describe in this course are all variations on the parallel distributed processing (PDP) idea. The architecture of each network is based on very similar building blocks which perform the processing. In this chapter we first discuss these processing units and discuss different network topologies. Learning strategies—as a basis for an adaptive system—will be presented in the last section.

### 2.1 A framework for distributed representation

An artificial network consists of a pool of simple processing units which communicate by sending signals to each other over a large number of weighted connections.

A set of major aspects of a parallel distributed model can be distinguished (cf. Rumelhart and McClelland, 1986 (McClelland & Rumelhart, 1986; Rumelhart & McClelland, 1986)):

- a set of processing units ('neurons,' 'cells');
- a state of activation  $y_k$  for every unit, which equivalent to the output of the unit;
- connections between the units. Generally each connection is defined by a weight  $w_{jk}$  which determines the effect which the signal of unit  $j$  has on unit  $k$ ;
- a propagation rule, which determines the effective input  $s_k$  of a unit from its external inputs;
- an activation function  $\mathcal{F}_k$ , which determines the new level of activation based on the effective input  $s_k(t)$  and the current activation  $y_k(t)$  (i.e., the update);
- an external input (aka bias, offset)  $\theta_k$  for each unit;
- a method for information gathering (the learning rule);
- an environment within which the system must operate, providing input signals and—if necessary—error signals.

Figure 2.1 illustrates these basics, some of which will be discussed in the next sections.

#### 2.1.1 Processing units

Each unit performs a relatively simple job: receive input from neighbours or external sources and use this to compute an output signal which is propagated to other units. Apart from this processing, a second task is the adjustment of the weights. The system is inherently parallel in the sense that many units can carry out their computations at the same time.

Within neural systems it is useful to distinguish three types of units: *input* units (indicated by an index  $i$ ) which receive data from outside the neural network, *output* units (indicated by

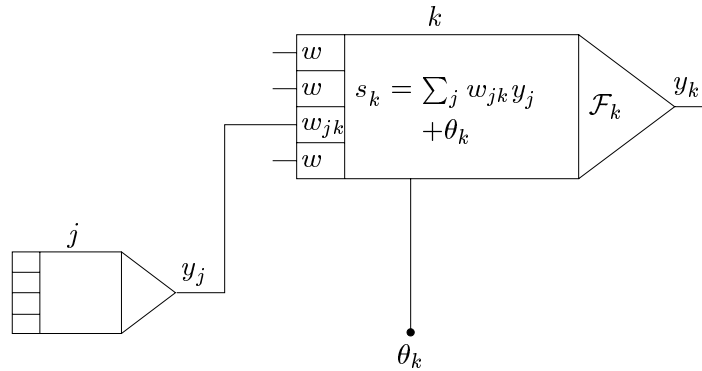


Figure 2.1: The basic components of an artificial neural network. The propagation rule used here is the ‘standard’ weighted summation.

an index  $o$ ) which send data out of the neural network, and *hidden* units (indicated by an index  $h$ ) whose input and output signals remain within the neural network.

During operation, units can be updated either *synchronously* or *asynchronously*. With synchronous updating, all units update their activation simultaneously; with asynchronous updating, each unit has a (usually fixed) probability of updating its activation at a time  $t$ , and usually only one unit will be able to do this at a time. In some cases the latter model has some advantages.

### 2.1.2 Connections between units

In most cases we assume that each unit provides an additive contribution to the input of the unit with which it is connected. The total input to unit  $k$  is simply the weighted sum of the separate outputs from each of the connected units plus a *bias* or *offset* term  $\theta_k$ :

$$s_k(t) = \sum_j w_{jk}(t) y_j(t) + \theta_k(t). \quad (2.1)$$

The contribution for positive  $w_{jk}$  is considered as an *excitation* and for negative  $w_{jk}$  as *inhibition*. In some cases more complex rules for combining inputs are used, in which a distinction is made between excitatory and inhibitory inputs. We call units with a propagation rule (2.1) *sigma units*.

A different propagation rule, introduced by Feldman and Ballard (Feldman & Ballard, 1982), is known as the propagation rule for the *sigma-pi unit*:

$$s_k(t) = \sum_j w_{jk}(t) \prod_m y_{j_m}(t) + \theta_k(t). \quad (2.2)$$

Often, the  $y_{j_m}$  are weighted before multiplication. Although these units are not frequently used, they have their value for gating of input, as well as implementation of lookup tables (Mel, 1990).

### 2.1.3 Activation and output rules

We also need a rule which gives the effect of the total input on the activation of the unit. We need a function  $\mathcal{F}_k$  which takes the total input  $s_k(t)$  and the current activation  $y_k(t)$  and produces a new value of the activation of the unit  $k$ :

$$y_k(t+1) = \mathcal{F}_k(y_k(t), s_k(t)). \quad (2.3)$$

Often, the activation function is a nondecreasing function of the total input of the unit:

$$y_k(t+1) = \mathcal{F}_k(s_k(t)) = \mathcal{F}_k \left( \sum_j w_{jk}(t) y_j(t) + \theta_k(t) \right), \quad (2.4)$$

although activation functions are not restricted to nondecreasing functions. Generally, some sort of threshold function is used: a hard limiting threshold function (a *sgn* function), or a linear or semi-linear function, or a smoothly limiting threshold (see figure 2.2). For this smoothly limiting function often a sigmoid (S-shaped) function like

$$y_k = \mathcal{F}(s_k) = \frac{1}{1 + e^{-s_k}} \quad (2.5)$$

is used. In some applications a hyperbolic tangent is used, yielding output values in the range  $[-1, +1]$ .

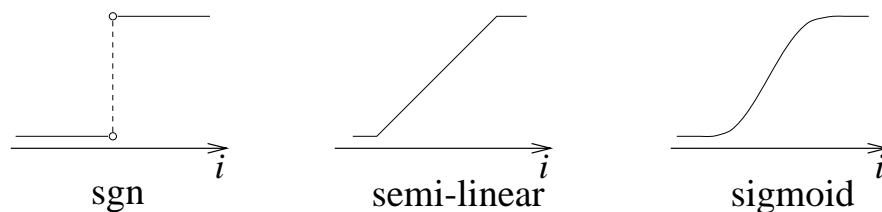


Figure 2.2: Various activation functions for a unit.

In some cases, the output of a unit can be a stochastic function of the total input of the unit. In that case the activation is not deterministically determined by the neuron input, but the neuron input determines the probability  $p$  that a neuron get a high activation value:

$$p(y_k \leftarrow 1) = \frac{1}{1 + e^{-s_k/T}}, \quad (2.6)$$

in which  $T$  (cf. temperature) is a parameter which determines the slope of the probability function. This type of unit will be discussed more extensively in chapter 5.

In all networks we describe we consider the output of a neuron to be identical to its activation level.

## 2.2 Network topologies

In the previous section we discussed the properties of the basic processing unit in an artificial neural network. This section focuses on the pattern of connections between the units and the propagation of data.

As for this pattern of connections, the main distinction we can make is between:

- *Feed-forward* networks, where the data flow from input to output units is strictly feed-forward. The data processing can extend over multiple (layers of) units, but no feedback connections are present, that is, connections extending from outputs of units to inputs of units in the same layer or previous layers.
- *Recurrent* networks that do contain feedback connections. Contrary to feed-forward networks, the dynamical properties of the network are important. In some cases, the activation values of the units undergo a relaxation process such that the network will evolve to a stable state in which these activations do not change anymore. In other applications, the change of the activation values of the output neurons are significant, such that the dynamical behaviour constitutes the output of the network (Pearlmutter, 1990).

Classical examples of feed-forward networks are the Perceptron and Adaline, which will be discussed in the next chapter. Examples of recurrent networks have been presented by Anderson (Anderson, 1977), Kohonen (Kohonen, 1977), and Hopfield (Hopfield, 1982) and will be discussed in chapter 5.

## 2.3 Training of artificial neural networks

A neural network has to be configured such that the application of a set of inputs produces (either ‘direct’ or via a relaxation process) the desired set of outputs. Various methods to set the strengths of the connections exist. One way is to set the weights explicitly, using *a priori* knowledge. Another way is to ‘train’ the neural network by feeding it teaching patterns and letting it change its weights according to some learning rule.

### 2.3.1 Paradigms of learning

We can categorise the learning situations in two distinct sorts. These are:

- *Supervised learning* or *Associative learning* in which the network is trained by providing it with input and matching output patterns. These input-output pairs can be provided by an external teacher, or by the system which contains the network (*self-supervised*).
- *Unsupervised learning* or *Self-organisation* in which an (output) unit is trained to respond to clusters of pattern within the input. In this paradigm the system is supposed to discover statistically salient features of the input population. Unlike the supervised learning paradigm, there is no *a priori* set of categories into which the patterns are to be classified; rather the system must develop its own representation of the input stimuli.

### 2.3.2 Modifying patterns of connectivity

Both learning paradigms discussed above result in an adjustment of the weights of the connections between units, according to some modification rule. Virtually all learning rules for models of this type can be considered as a variant of the *Hebbian* learning rule suggested by Hebb in his classic book *Organization of Behaviour* (1949) (Hebb, 1949). The basic idea is that if two units  $j$  and  $k$  are active simultaneously, their interconnection must be strengthened. If  $j$  receives input from  $k$ , the simplest version of Hebbian learning prescribes to modify the weight  $w_{jk}$  with

$$\Delta w_{jk} = \gamma y_j y_k, \quad (2.7)$$

where  $\gamma$  is a positive constant of proportionality representing the *learning rate*. Another common rule uses not the actual activation of unit  $k$  but the difference between the actual and desired activation for adjusting the weights:

$$\Delta w_{jk} = \gamma y_j (d_k - y_k), \quad (2.8)$$

in which  $d_k$  is the desired activation provided by a teacher. This is often called the *Widrow-Hoff rule* or the *delta rule*, and will be discussed in the next chapter.

Many variants (often very exotic ones) have been published the last few years. In the next chapters some of these update rules will be discussed.

## 2.4 Notation and terminology

Throughout the years researchers from different disciplines have come up with a vast number of terms applicable in the field of neural networks. Our computer scientist point-of-view enables us to adhere to a subset of the terminology which is less biologically inspired, yet still conflicts arise. Our conventions are discussed below.

### 2.4.1 Notation

We use the following notation in our formulae. Note that not all symbols are meaningful for all networks, and that in some cases subscripts or superscripts may be left out (e.g.,  $p$  is often not necessary) or added (e.g., vectors can, contrariwise to the notation below, have indices) where necessary. Vectors are indicated with a bold non-slanted font:

$j, k, \dots$  the unit  $j, k, \dots$ ;

$i$  an input unit;

$h$  a hidden unit;

$o$  an output unit;

$\mathbf{x}^p$  the  $p$ th input pattern vector;

$x_j^p$  the  $j$ th element of the  $p$ th input pattern vector;

$\mathbf{s}^p$  the input to a set of neurons when input pattern vector  $p$  is clamped (i.e., presented to the network); often: the input of the *network* by clamping input pattern vector  $p$ ;

$\mathbf{d}^p$  the desired output of the *network* when input pattern vector  $p$  was input to the network;

$d_j^p$  the  $j$ th element of the desired output of the network when input pattern vector  $p$  was input to the network;

$\mathbf{y}^p$  the activation values of the *network* when input pattern vector  $p$  was input to the network;

$y_j^p$  the activation values of element  $j$  of the network when input pattern vector  $p$  was input to the network;

$W$  the matrix of connection weights;

$\mathbf{w}_j$  the weights of the connections which feed into unit  $j$ ;

$w_{jk}$  the weight of the connection from unit  $j$  to unit  $k$ ;

$\mathcal{F}_j$  the activation function associated with unit  $j$ ;

$\gamma_{jk}$  the learning rate associated with weight  $w_{jk}$ ;

$\boldsymbol{\theta}$  the biases to the units;

$\theta_j$  the bias input to unit  $j$ ;

$U_j$  the threshold of unit  $j$  in  $\mathcal{F}_j$ ;

$E^p$  the error in the output of the network when input pattern vector  $p$  is input;

$\mathcal{E}$  the energy of the network.

### 2.4.2 Terminology

**Output vs. activation of a unit.** Since there is no need to do otherwise, we consider the output and the activation value of a unit to be one and the same thing. That is, the output of each neuron equals its activation value.

**Bias, offset, threshold.** These terms all refer to a constant (i.e., independent of the network input but adapted by the learning rule) term which is input to a unit. They may be used interchangeably, although the latter two terms are often envisaged as a property of the activation function. Furthermore, this external input is usually implemented (and can be written) as a weight from a unit with activation value 1.

**Number of layers.** In a feed-forward network, the inputs perform no computation and their layer is therefore not counted. Thus a network with one input layer, one hidden layer, and one output layer is referred to as a network with *two* layers. This convention is widely though not yet universally used.

**Representation vs. learning.** When using a neural network one has to distinguish two issues which influence the performance of the system. The first one is the *representational* power of the network, the second one is the *learning* algorithm.

The representational power of a neural network refers to the ability of a neural network to represent a desired function. Because a neural network is built from a set of standard functions, in most cases the network will only *approximate* the desired function, and even for an *optimal* set of weights the approximation error is not zero.

The second issue is the learning algorithm. Given that there exist a set of optimal weights in the network, is there a procedure to (iteratively) find this set of weights?



Part II

**THEORY**



# 3

## Perceptron and Adaline

---

This chapter describes single layer neural networks, including some of the classical approaches to the neural computing and learning problem. In the first part of this chapter we discuss the representational power of the single layer networks and their learning algorithms and will give some examples of using the networks. In the second part we will discuss the representational limitations of single layer networks.

Two ‘classical’ models will be described in the first part of the chapter: the *Perceptron*, proposed by Rosenblatt (Rosenblatt, 1959) in the late 50’s and the *Adaline*, presented in the early 60’s by Widrow and Hoff (Widrow & Hoff, 1960).

### 3.1 Networks with threshold activation functions

A single layer feed-forward network consists of one or more output neurons  $o$ , each of which is connected with a weighting factor  $w_{io}$  to all of the inputs  $i$ . In the simplest case the network has only two inputs and a single output, as sketched in figure 3.1 (we leave the output index  $o$  out). The input of the neuron is the weighted sum of the inputs plus the bias term. The output

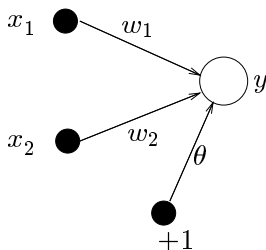


Figure 3.1: Single layer network with one output and two inputs.

of the network is formed by the activation of the output neuron, which is some function of the input:

$$y = \mathcal{F} \left( \sum_{i=1}^2 w_i x_i + \theta \right), \quad (3.1)$$

The activation function  $\mathcal{F}$  can be linear so that we have a linear network, or nonlinear. In this section we consider the threshold (or Heaviside or sgn) function:

$$\mathcal{F}(s) = \begin{cases} 1 & \text{if } s > 0 \\ -1 & \text{otherwise.} \end{cases} \quad (3.2)$$

The output of the network thus is either  $+1$  or  $-1$ , depending on the input. The network can now be used for a *classification* task: it can decide whether an input pattern belongs to one of two classes. If the total input is positive, the pattern will be assigned to class  $+1$ , if the

total input is negative, the sample will be assigned to class  $-1$ . The separation between the two classes in this case is a straight line, given by the equation:

$$w_1x_1 + w_2x_2 + \theta = 0 \quad (3.3)$$

The single layer network represents a *linear discriminant function*.

A geometrical representation of the linear threshold neural network is given in figure 3.2. Equation (3.3) can be written as

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{\theta}{w_2}, \quad (3.4)$$

and we see that the weights determine the slope of the line and the bias determines the ‘offset’, i.e. how far the line is from the origin. Note that also the weights can be plotted in the input space: the weight vector is always perpendicular to the discriminant function.

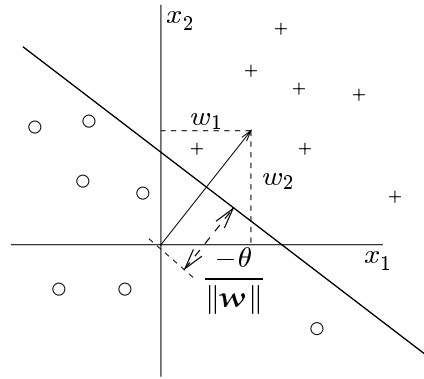


Figure 3.2: Geometric representation of the discriminant function and the weights.

Now that we have shown the representational power of the single layer network with linear threshold units, we come to the second issue: how do we *learn* the weights and biases in the network? We will describe two learning methods for these types of networks: the ‘perceptron’ learning rule and the ‘delta’ or ‘LMS’ rule. Both methods are iterative procedures that adjust the weights. A learning sample is presented to the network. For each weight the new value is computed by adding a correction to the old value. The threshold is updated in a same way:

$$w_i(t+1) = w_i(t) + \Delta w_i(t), \quad (3.5)$$

$$\theta(t+1) = \theta(t) + \Delta \theta(t). \quad (3.6)$$

The learning problem can now be formulated as: how do we compute  $\Delta w_i(t)$  and  $\Delta \theta(t)$  in order to classify the learning patterns correctly?

## 3.2 Perceptron learning rule and convergence theorem

Suppose we have a set of learning samples consisting of an input vector  $\mathbf{x}$  and a desired output  $d(\mathbf{x})$ . For a classification task the  $d(\mathbf{x})$  is usually  $+1$  or  $-1$ . The perceptron learning rule is very simple and can be stated as follows:

1. Start with random weights for the connections;
2. Select an input vector  $\mathbf{x}$  from the set of training samples;
3. If  $y \neq d(\mathbf{x})$  (the perceptron gives an incorrect response), modify all connections  $w_i$  according to:  $\Delta w_i = d(\mathbf{x})x_i$ ;

4. Go back to 2.

Note that the procedure is very similar to the Hebb rule; the only difference is that, when the network responds correctly, no connection weights are modified. Besides modifying the weights, we must also modify the threshold  $\theta$ . This  $\theta$  is considered as a connection  $w_0$  between the output neuron and a ‘dummy’ predicate unit which is always on:  $x_0 = 1$ . Given the perceptron learning rule as stated above, this threshold is modified according to:

$$\Delta\theta = \begin{cases} 0 & \text{if the perceptron responds correctly;} \\ d(\mathbf{x}) & \text{otherwise.} \end{cases} \quad (3.7)$$

### 3.2.1 Example of the Perceptron learning rule

A perceptron is initialized with the following weights:  $w_1 = 1, w_2 = 2, \theta = -2$ . The perceptron learning rule is used to learn a correct discriminant function for a number of samples, sketched in figure 3.3. The first sample A, with values  $\mathbf{x} = (0.5, 1.5)$  and target value  $d(\mathbf{x}) = +1$  is presented to the network. From eq. (3.1) it can be calculated that the network output is  $+1$ , so no weights are adjusted. The same is the case for point B, with values  $\mathbf{x} = (-0.5, 0.5)$  and target value  $d(\mathbf{x}) = -1$ ; the network output is negative, so no change. When presenting point C with values  $\mathbf{x} = (0.5, 0.5)$  the network output will be  $-1$ , while the target value  $d(\mathbf{x}) = +1$ . According to the perceptron learning rule, the weight changes are:  $\Delta w_1 = 0.5, \Delta w_2 = 0.5, \Delta\theta = 1$ . The new weights are now:  $w_1 = 1.5, w_2 = 2.5, \theta = -1$ , and sample C is classified correctly.

In figure 3.3 the discriminant function before and after this weight update is shown.

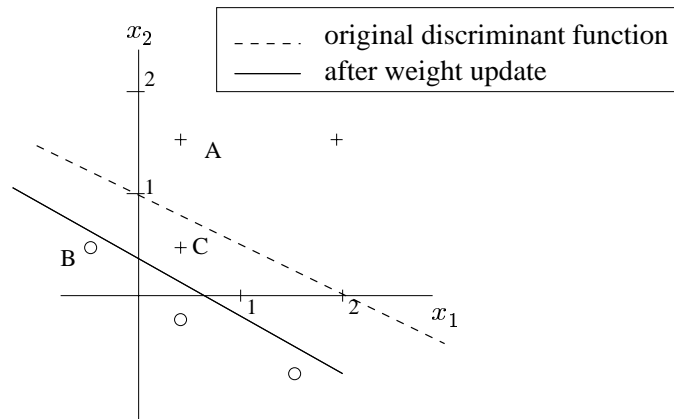


Figure 3.3: Discriminant function before and after weight update.

### 3.2.2 Convergence theorem

For the perceptron learning rule there exists a convergence theorem, which states the following:

**Theorem 1** *If there exists a set of connection weights  $\mathbf{w}^*$  which is able to perform the transformation  $y = d(\mathbf{x})$ , the perceptron learning rule will converge to some solution (which may or may not be the same as  $\mathbf{w}^*$ ) in a finite number of steps for any initial choice of the weights.*

**Proof** *Given the fact that the length of the vector  $\mathbf{w}^*$  does not play a role (because of the sgn operation), we take  $\|\mathbf{w}^*\| = 1$ . Because  $\mathbf{w}^*$  is a correct solution, the value  $|\mathbf{w}^* \cdot \mathbf{x}|$ , where  $\cdot$  denotes dot or inner product, will be greater than 0 or: there exists a  $\delta > 0$  such that  $|\mathbf{w}^* \cdot \mathbf{x}| > \delta$  for all inputs  $\mathbf{x}$ <sup>1</sup>. Now define  $\cos \alpha \equiv \mathbf{w} \cdot \mathbf{w}^* / \|\mathbf{w}\|$ . When according to the perceptron learning*

<sup>1</sup>Technically this need not to be true for any  $\mathbf{w}^*$ ;  $\mathbf{w}^* \cdot \mathbf{x}$  could in fact be equal to 0 for a  $\mathbf{w}^*$  which yields no misclassifications (look at definition of  $\mathcal{F}$ ). However, another  $\mathbf{w}^*$  can be found for which the quantity will not be 0. (Thanks to: Terry Regier, Computer Science, UC Berkeley)

rule, connection weights are modified at a given input  $\mathbf{x}$ , we know that  $\Delta\mathbf{w} = d(\mathbf{x})\mathbf{x}$ , and the weight after modification is  $\mathbf{w}' = \mathbf{w} + \Delta\mathbf{w}$ . From this it follows that:

$$\begin{aligned}\mathbf{w}' \cdot \mathbf{w}^* &= \mathbf{w} \cdot \mathbf{w}^* + d(\mathbf{x}) \cdot \mathbf{w}^* \cdot \mathbf{x} \\ &= \mathbf{w} \cdot \mathbf{w}^* + \text{sgn}(\mathbf{w}^* \cdot \mathbf{x}) \mathbf{w}^* \cdot \mathbf{x} \\ &> \mathbf{w} \cdot \mathbf{w}^* + \delta\end{aligned}$$

$$\begin{aligned}\|\mathbf{w}'\|^2 &= \|\mathbf{w} + d(\mathbf{x})\mathbf{x}\|^2 \\ &= \mathbf{w}^2 + 2d(\mathbf{x})\mathbf{w} \cdot \mathbf{x} + \mathbf{x}^2 \\ &< \mathbf{w}^2 + \mathbf{x}^2 \quad (\text{because } d(\mathbf{x}) = -\text{sgn}[\mathbf{w} \cdot \mathbf{x}] !!) \\ &= \mathbf{w}^2 + M.\end{aligned}$$

After  $t$  modifications we have:

$$\begin{aligned}\mathbf{w}(t) \cdot \mathbf{w}^* &> \mathbf{w} \cdot \mathbf{w}^* + t\delta \\ \|\mathbf{w}(t)\|^2 &< \mathbf{w}^2 + tM\end{aligned}$$

such that

$$\begin{aligned}\cos \alpha(t) &= \frac{\mathbf{w}^* \cdot \mathbf{w}(t)}{\|\mathbf{w}(t)\|} \\ &> \frac{\mathbf{w}^* \cdot \mathbf{w} + t\delta}{\sqrt{\mathbf{w}^2 + tM}}.\end{aligned}$$

From this follows that  $\lim_{t \rightarrow \infty} \cos \alpha(t) = \lim_{t \rightarrow \infty} \frac{\delta}{\sqrt{M}} \sqrt{t} = \infty$ , while by definition  $\cos \alpha \leq 1$  !

The conclusion is that there must be an upper limit  $t_{\max}$  for  $t$ . The system modifies its connections only a limited number of times. In other words: after maximally  $t_{\max}$  modifications of the weights the perceptron is correctly performing the mapping.  $t_{\max}$  will be reached when  $\cos \alpha = 1$ . If we start with connections  $\mathbf{w} = \mathbf{0}$ ,

$$t_{\max} = \frac{M}{\delta^2}. \quad (3.8)$$

□

### 3.2.3 The original Perceptron

The Perceptron, proposed by Rosenblatt (Rosenblatt, 1959) is somewhat more complex than a single layer network with threshold activation functions. In its simplest form it consist of an  $N$ -element input layer ('retina') which feeds into a layer of  $M$  'association,' 'mask,' or 'predicate' units  $\phi_h$ , and a single output unit. The goal of the operation of the perceptron is to learn a given transformation  $d : \{-1, 1\}^N \rightarrow \{-1, 1\}$  using learning samples with input  $\mathbf{x}$  and corresponding output  $y = d(\mathbf{x})$ . In the original definition, the activity of the predicate units can be any function  $\phi_h$  of the input layer  $\mathbf{x}$  but the learning procedure only adjusts the connections to the output unit. The reason for this is that no recipe had been found to adjust the connections between  $\mathbf{x}$  and  $\phi_h$ . Depending on the functions  $\phi_h$ , perceptrons can be grouped into different *families*. In (Minsky & Papert, 1969) a number of these families are described and properties of these families have been described. The output unit of a perceptron is a linear threshold element. Rosenblatt (1959) (Rosenblatt, 1959) proved the remarkable theorem about perceptron learning and in the early 60s perceptrons created a great deal of interest and optimism. The initial euphoria was replaced by disillusion after the publication of Minsky and Papert's *Perceptrons* in 1969 (Minsky & Papert, 1969). In this book they analysed the perceptron thoroughly and proved that there are severe restrictions on what perceptrons can represent.

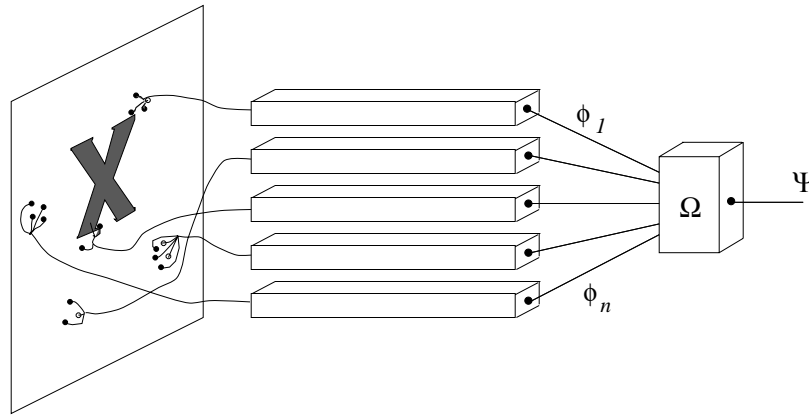


Figure 3.4: The Perceptron.

### 3.3 The adaptive linear element (Adaline)

An important generalisation of the perceptron training algorithm was presented by Widrow and Hoff as the ‘least mean square’ (LMS) learning procedure, also known as the *delta rule*. The main functional difference with the perceptron training rule is the way the output of the system is used in the learning rule. The perceptron learning rule uses the output of the threshold function (either  $-1$  or  $+1$ ) for learning. The delta-rule uses the net output without further mapping into output values  $-1$  or  $+1$ .

The learning rule was applied to the ‘adaptive linear element,’ also named *Adaline*<sup>2</sup>, developed by Widrow and Hoff (Widrow & Hoff, 1960). In a simple physical implementation (fig. 3.5) this device consists of a set of controllable resistors connected to a circuit which can sum up currents caused by the input voltage signals. Usually the central block, the summer, is also followed by a quantiser which outputs either  $+1$  or  $-1$ , depending on the polarity of the sum.

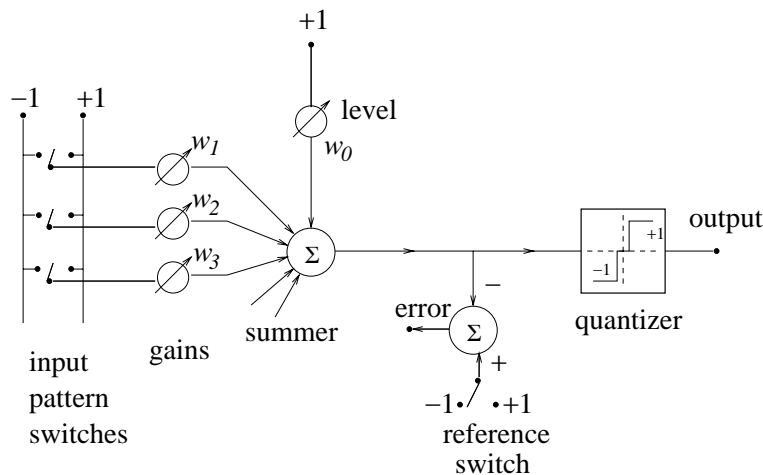


Figure 3.5: The Adaline.

Although the adaptive process is here exemplified in a case when there is only one output, it may be clear that a system with many parallel outputs is directly implementable by multiple units of the above kind.

If the input conductances are denoted by  $w_i$ ,  $i = 0, 1, \dots, n$ , and the input and output signals

<sup>2</sup>ADALINE first stood for ADaptive LInear NEuron, but when artificial neurons became less and less popular this acronym was changed to ADaptive LInear Element.

by  $x_i$  and  $y$ , respectively, then the output of the central block is defined to be

$$y = \sum_{i=1}^n w_i x_i + \theta, \quad (3.9)$$

where  $\theta \equiv w_0$ . The purpose of this device is to yield a given value  $y = d^p$  at its output when the set of values  $x_i^p$ ,  $i = 1, 2, \dots, n$ , is applied at the inputs. The problem is to determine the coefficients  $w_i$ ,  $i = 0, 1, \dots, n$ , in such a way that the input-output response is correct for a large number of arbitrarily chosen signal sets. If an exact mapping is not possible, the average error must be minimised, for instance, in the sense of least squares. An adaptive operation means that there exists a mechanism by which the  $w_i$  can be adjusted, usually iteratively, to attain the correct values. For the *Adaline*, Widrow introduced the delta rule to adjust the weights. This rule will be discussed in section 3.4.

### 3.4 Networks with linear activation functions: the delta rule

For a single layer network with an output unit with a *linear* activation function the output is simply given by

$$y = \sum_j w_j x_j + \theta. \quad (3.10)$$

Such a simple network is able to represent a linear relationship between the value of the output unit and the value of the input units. By thresholding the output value, a classifier can be constructed (such as Widrow's Adaline), but here we focus on the linear relationship and use the network for a *function approximation* task. In high dimensional input spaces the network represents a (hyper)plane and it will be clear that also multiple output units may be defined.

Suppose we want to train the network such that a hyperplane is fitted as well as possible to a set of training samples consisting of input values  $x^p$  and desired (or *target*) output values  $d^p$ . For every given input sample, the output of the network differs from the target value  $d^p$  by  $(d^p - y^p)$ , where  $y^p$  is the actual output for this pattern. The delta-rule now uses a cost- or error-function based on these differences to adjust the weights.

The error function, as indicated by the name least mean square, is the summed squared error. That is, the total error  $E$  is defined to be

$$E = \sum_p E^p = \frac{1}{2} \sum_p (d^p - y^p)^2, \quad (3.11)$$

where the index  $p$  ranges over the set of input patterns and  $E^p$  represents the error on pattern  $p$ . The LMS procedure finds the values of all the weights that minimise the error function by a method called *gradient descent*. The idea is to make a change in the weight proportional to the negative of the derivative of the error as measured on the current pattern with respect to each weight:

$$\Delta_p w_j = -\gamma \frac{\partial E^p}{\partial w_j} \quad (3.12)$$

where  $\gamma$  is a constant of proportionality. The derivative is

$$\frac{\partial E^p}{\partial w_j} = \frac{\partial E^p}{\partial y^p} \frac{\partial y^p}{\partial w_j}. \quad (3.13)$$

Because of the linear units (eq. (3.10)),

$$\frac{\partial y^p}{\partial w_j} = x_j \quad (3.14)$$



and

$$\frac{\partial E^p}{\partial y^p} = -(d^p - y^p) \quad (3.15)$$

such that

$$\Delta_p w_j = \gamma \delta^p x_j \quad (3.16)$$

where  $\delta^p = d^p - y^p$  is the difference between the target output and the actual output for pattern  $p$ .

The delta rule modifies weight appropriately for target and actual outputs of either polarity and for both continuous and binary input and output units. These characteristics have opened up a wealth of new applications.

### 3.5 Exclusive-OR problem

In the previous sections we have discussed two learning algorithms for single layer networks, but we have not discussed the limitations on the *representation* of these networks.

$x_0$	$x_1$	$d$
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1

Table 3.1: Exclusive-or truth table.

One of Minsky and Papert's most discouraging results shows that a single layer perceptron cannot represent a simple exclusive-or function. Table 3.1 shows the desired relationships between inputs and output units for this function.

In a simple network with two inputs and one output, as depicted in figure 3.1, the net input is equal to:

$$s = w_1 x_1 + w_2 x_2 + \theta. \quad (3.17)$$

According to eq. (3.1), the output of the perceptron is zero when  $s$  is negative and equal to one when  $s$  is positive. In figure 3.6 a geometrical representation of the input domain is given. For a constant  $\theta$ , the output of the perceptron is equal to one on one side of the dividing line which is defined by:

$$w_1 x_1 + w_2 x_2 = -\theta \quad (3.18)$$

and equal to zero on the other side of this line.

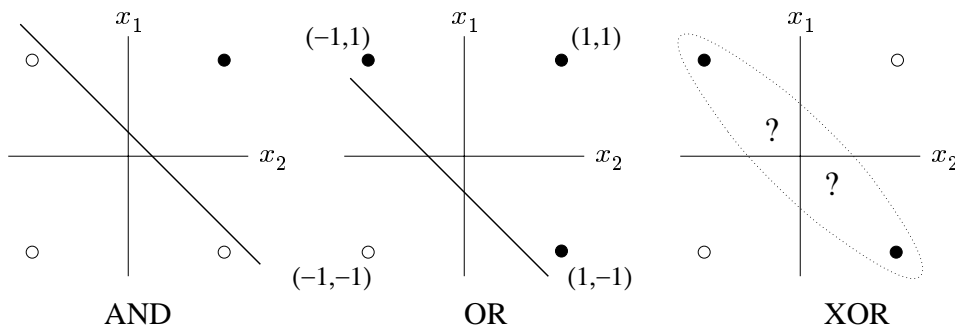


Figure 3.6: Geometric representation of input space.

To see that such a solution cannot be found, take a look at figure 3.6. The input space consists of four points, and the two solid circles at  $(1, -1)$  and  $(-1, 1)$  cannot be separated by a straight line from the two open circles at  $(-1, -1)$  and  $(1, 1)$ . The obvious question to ask is: How can this problem be overcome? Minsky and Papert prove in their book that for binary inputs, any transformation can be carried out by adding a layer of predicates which are connected to all inputs. The proof is given in the next section.

For the specific XOR problem we geometrically show that by introducing *hidden units*, thereby extending the network to a *multi-layer perceptron*, the problem can be solved. Fig. 3.7a demonstrates that the four input points are now embedded in a three-dimensional space defined by the two inputs plus the single hidden unit. These four points are now easily separated by

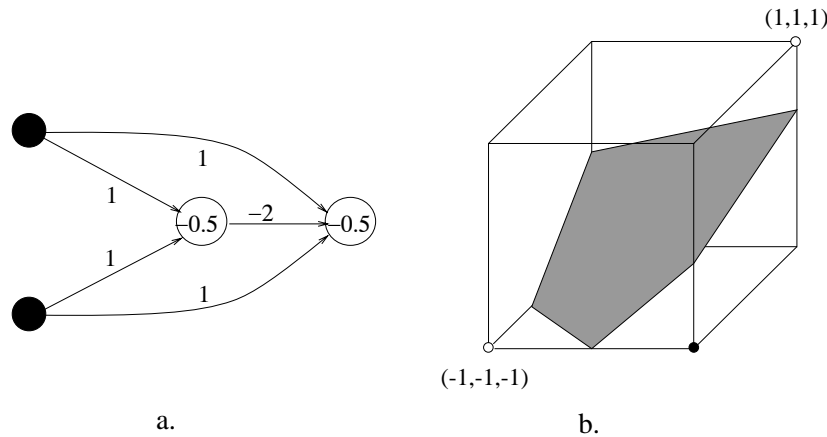


Figure 3.7: Solution of the XOR problem.

a) The perceptron of fig. 3.1 with an extra hidden unit. With the indicated values of the weights  $w_{ij}$  (next to the connecting lines) and the thresholds  $\theta_i$  (in the circles) this perceptron solves the XOR problem. b) This is accomplished by mapping the four points of figure 3.6 onto the four points indicated here; clearly, separation (by a linear manifold) into the required groups is now possible.

a linear manifold (plane) into two groups, as desired. This simple example demonstrates that adding hidden units increases the class of problems that are soluble by feed-forward, perceptron-like networks. However, by this generalisation of the basic architecture we have also incurred a serious loss: we no longer have a learning rule to determine the optimal weights!

### 3.6 Multi-layer perceptrons can do everything

In the previous section we showed that by adding an extra hidden unit, the XOR problem can be solved. For binary units, one can prove that this architecture is able to perform any transformation given the correct connections and weights. The most primitive is the next one. For a given transformation  $y = d(\mathbf{x})$ , we can divide the set of all possible input vectors into two classes:

$$X^+ = \{ \mathbf{x} \mid d(\mathbf{x}) = 1 \} \quad \text{and} \quad X^- = \{ \mathbf{x} \mid d(\mathbf{x}) = -1 \}. \quad (3.19)$$

Since there are  $N$  input units, the total number of possible input vectors  $\mathbf{x}$  is  $2^N$ . For every  $\mathbf{x}^p \in X^+$  a hidden unit  $h$  can be reserved of which the activation  $y_h$  is 1 if and only if the specific pattern  $p$  is present at the input: we can choose its weights  $w_{ih}$  equal to the specific pattern  $\mathbf{x}^p$  and the bias  $\theta_h$  equal to  $1 - N$  such that

$$y_h^p = \text{sgn} \left( \sum_i w_{ih} x_i^p - N + \frac{1}{2} \right) \quad (3.20)$$

is equal to 1 for  $\mathbf{x}^p = \mathbf{w}_h$  only. Similarly, the weights to the output neuron can be chosen such that the output is one as soon as one of the  $M$  predicate neurons is one:

$$y_o^p = \text{sgn} \left( \sum_{h=1}^M y_h + M - \frac{1}{2} \right). \quad (3.21)$$

This perceptron will give  $y_o = 1$  only if  $\mathbf{x} \in X^+$ : it performs the desired mapping. The problem is the large number of predicate units, which is equal to the number of patterns in  $X^+$ , which is maximally  $2^N$ . Of course we can do the same trick for  $X^-$ , and we will always take the minimal number of mask units, which is maximally  $2^{N-1}$ . A more elegant proof is given in (Minsky & Papert, 1969), but the point is that for complex transformations the number of required units in the hidden layer is exponential in  $N$ .

### 3.7 Conclusions

In this chapter we presented single layer feedforward networks for classification tasks and for function approximation tasks. The representational power of single layer feedforward networks was discussed and two learning algorithms for finding the optimal weights were presented. The simple networks presented here have their advantages and disadvantages. The disadvantage is the limited representational power: only linear classifiers can be constructed or, in case of function approximation, only linear functions can be represented. The advantage, however, is that because of the linearity of the system, the training algorithm will converge to the optimal solution. This is not the case anymore for nonlinear systems such as multiple layer networks, as we will see in the next chapter.



# 4

## Back-Propagation

---

As we have seen in the previous chapter, a single-layer network has severe restrictions: the class of tasks that can be accomplished is very limited. In this chapter we will focus on feed-forward networks with layers of processing units.

Minsky and Papert (Minsky & Papert, 1969) showed in 1969 that a two layer feed-forward network can overcome many restrictions, but did not present a solution to the problem of how to adjust the weights from input to hidden units. An answer to this question was presented by Rumelhart, Hinton and Williams in 1986 (Rumelhart, Hinton, & Williams, 1986), and similar solutions appeared to have been published earlier (Werbos, 1974; Parker, 1985; Cun, 1985).

The central idea behind this solution is that the errors for the units of the hidden layer are determined by back-propagating the errors of the units of the output layer. For this reason the method is often called the *back-propagation learning rule*. Back-propagation can also be considered as a generalisation of the delta rule for non-linear activation functions<sup>1</sup> and multi-layer networks.

### 4.1 Multi-layer feed-forward networks

A feed-forward network has a layered structure. Each layer consists of units which receive their input from units from a layer directly below and send their output to units in a layer directly above the unit. There are no connections within a layer. The  $N_i$  inputs are fed into the first layer of  $N_{h,1}$  *hidden* units. The input units are merely ‘fan-out’ units; no processing takes place in these units. The activation of a hidden unit is a function  $\mathcal{F}_i$  of the weighted inputs plus a bias, as given in in eq. (2.4). The output of the hidden units is distributed over the next layer of  $N_{h,2}$  hidden units, until the last layer of hidden units, of which the outputs are fed into a layer of  $N_o$  *output* units (see figure 4.1).

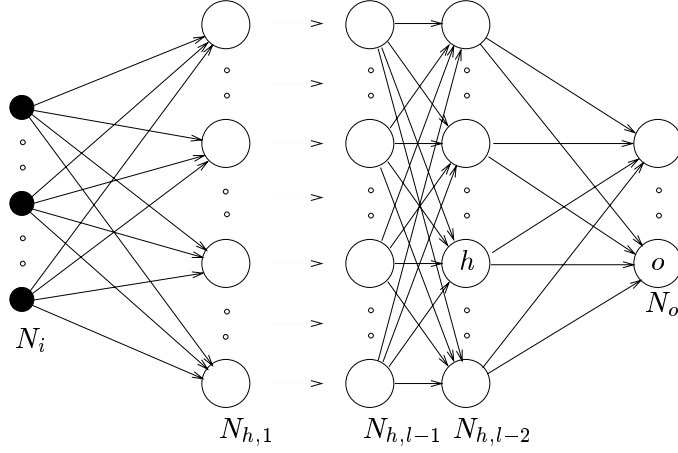
Although back-propagation can be applied to networks with any number of layers, just as for networks with binary units (section 3.6) it has been shown (Hornik, Stinchcombe, & White, 1989; Funahashi, 1989; Cybenko, 1989; Hartman, Keeler, & Kowalski, 1990) that only one layer of hidden units suffices to approximate any function with finitely many discontinuities to arbitrary precision, provided the activation functions of the hidden units are non-linear (the *universal approximation theorem*). In most applications a feed-forward network with a single layer of hidden units is used with a sigmoid activation function for the units.

### 4.2 The generalised delta rule

Since we are now using units with nonlinear activation functions, we have to generalise the delta rule which was presented in chapter 3 for linear functions to the set of non-linear activation

---

<sup>1</sup>Of course, when linear activation functions are used, a multi-layer network is not more powerful than a single-layer network.

Figure 4.1: A multi-layer network with  $l$  layers of units.

functions. The activation is a differentiable function of the total input, given by

$$y_k^p = \mathcal{F}(s_k^p), \quad (4.1)$$

in which

$$s_k^p = \sum_j w_{jk} y_j^p + \theta_k. \quad (4.2)$$

To get the correct generalisation of the delta rule as presented in the previous chapter, we must set

$$\Delta_p w_{jk} = -\gamma \frac{\partial E^p}{\partial w_{jk}}. \quad (4.3)$$

The error measure  $E^p$  is defined as the total quadratic error for pattern  $p$  at the output units:

$$E^p = \frac{1}{2} \sum_{o=1}^{N_o} (d_o^p - y_o^p)^2, \quad (4.4)$$

where  $d_o^p$  is the desired output for unit  $o$  when pattern  $p$  is clamped. We further set  $E = \sum_p E^p$  as the *summed squared error*. We can write

$$\frac{\partial E^p}{\partial w_{jk}} = \frac{\partial E^p}{\partial s_k^p} \frac{\partial s_k^p}{\partial w_{jk}}. \quad (4.5)$$

By equation (4.2) we see that the second factor is

$$\frac{\partial s_k^p}{\partial w_{jk}} = y_j^p. \quad (4.6)$$

When we define

$$\delta_k^p = -\frac{\partial E^p}{\partial s_k^p}, \quad (4.7)$$

we will get an update rule which is equivalent to the delta rule as described in the previous chapter, resulting in a gradient descent on the error surface if we make the weight changes according to:

$$\Delta_p w_{jk} = \gamma \delta_k^p y_j^p. \quad (4.8)$$

The trick is to figure out what  $\delta_k^p$  should be for each unit  $k$  in the network. The interesting result, which we now derive, is that there is a simple recursive computation of these  $\delta$ 's which can be implemented by propagating error signals backward through the network.

To compute  $\delta_k^p$  we apply the chain rule to write this partial derivative as the product of two factors, one factor reflecting the change in error as a function of the output of the unit and one reflecting the change in the output as a function of changes in the input. Thus, we have

$$\delta_k^p = -\frac{\partial E^p}{\partial s_k^p} = -\frac{\partial E^p}{\partial y_k^p} \frac{\partial y_k^p}{\partial s_k^p}. \quad (4.9)$$

Let us compute the second factor. By equation (4.1) we see that

$$\frac{\partial y_k^p}{\partial s_k^p} = \mathcal{F}'(s_k^p), \quad (4.10)$$

which is simply the derivative of the squashing function  $\mathcal{F}$  for the  $k$ th unit, evaluated at the net input  $s_k^p$  to that unit. To compute the first factor of equation (4.9), we consider two cases. First, assume that unit  $k$  is an output unit  $k = o$  of the network. In this case, it follows from the definition of  $E^p$  that

$$\frac{\partial E^p}{\partial y_o^p} = -(d_o^p - y_o^p), \quad (4.11)$$

which is the same result as we obtained with the standard delta rule. Substituting this and equation (4.10) in equation (4.9), we get

$$\delta_o^p = (d_o^p - y_o^p) \mathcal{F}_o'(s_o^p) \quad (4.12)$$

for any output unit  $o$ . Secondly, if  $k$  is not an output unit but a hidden unit  $k = h$ , we do not readily know the contribution of the unit to the output error of the network. However, the error measure can be written as a function of the net inputs from hidden to output layer;  $E^p = E^p(s_1^p, s_2^p, \dots, s_j^p, \dots)$  and we use the chain rule to write

$$\frac{\partial E^p}{\partial y_h^p} = \sum_{o=1}^{N_o} \frac{\partial E^p}{\partial s_o^p} \frac{\partial s_o^p}{\partial y_h^p} = \sum_{o=1}^{N_o} \frac{\partial E^p}{\partial s_o^p} \frac{\partial}{\partial y_h^p} \sum_{j=1}^{N_h} w_{ko} y_j^p = \sum_{o=1}^{N_o} \frac{\partial E^p}{\partial s_o^p} w_{ho} = - \sum_{o=1}^{N_o} \delta_o^p w_{ho}. \quad (4.13)$$

Substituting this in equation (4.9) yields

$$\delta_h^p = \mathcal{F}'(s_h^p) \sum_{o=1}^{N_o} \delta_o^p w_{ho}. \quad (4.14)$$

Equations (4.12) and (4.14) give a recursive procedure for computing the  $\delta$ 's for all units in the network, which are then used to compute the weight changes according to equation (4.8). This procedure constitutes the generalised delta rule for a feed-forward network of non-linear units.

### 4.2.1 Understanding back-propagation

The equations derived in the previous section may be mathematically correct, but what do they actually mean? Is there a way of understanding back-propagation other than reciting the necessary equations?

The answer is, of course, yes. In fact, the whole back-propagation process is intuitively very clear. What happens in the above equations is the following. When a learning pattern is clamped, the activation values are propagated to the output units, and the actual network output is compared with the desired output values, we usually end up with an error in each of the output units. Let's call this error  $e_o$  for a particular output unit  $o$ . We have to bring  $e_o$  to zero.

The simplest method to do this is the greedy method: we strive to change the connections in the neural network in such a way that, next time around, the error  $e_o$  will be zero for this particular pattern. We know from the delta rule that, in order to reduce an error, we have to adapt its incoming weights according to

$$\Delta w_{ho} = (d_o - y_o)y_h. \quad (4.15)$$

That's step one. But it alone is not enough: when we only apply this rule, the weights from input to hidden units are never changed, and we do not have the full representational power of the feed-forward network as promised by the universal approximation theorem. In order to adapt the weights from input to hidden units, we again want to apply the delta rule. In this case, however, we do not have a value for  $\delta$  for the hidden units. This is solved by the chain rule which does the following: distribute the error of an output unit  $o$  to all the hidden units that is it connected to, weighted by this connection. Differently put, a hidden unit  $h$  receives a delta from each output unit  $o$  equal to the delta of that output unit weighted with (= multiplied by) the weight of the connection between those units. In symbols:  $\delta_h = \sum_o \delta_o w_{ho}$ . Well, not exactly: we forgot the activation function of the hidden unit;  $\mathcal{F}'$  has to be applied to the delta, before the back-propagation process can continue.

### 4.3 Working with back-propagation

The application of the generalised delta rule thus involves two phases: During the first phase the input  $\mathbf{x}$  is presented and propagated forward through the network to compute the output values  $y_o^p$  for each output unit. This output is compared with its desired value  $d_o$ , resulting in an error signal  $\delta_o^p$  for each output unit. The second phase involves a backward pass through the network during which the error signal is passed to each unit in the network and appropriate weight changes are calculated.

**Weight adjustments with sigmoid activation function.** The results from the previous section can be summarised in three equations:

- The weight of a connection is adjusted by an amount proportional to the product of an error signal  $\delta$ , on the unit  $k$  receiving the input and the output of the unit  $j$  sending this signal along the connection:

$$\Delta_p w_{jk} = \gamma \delta_k^p y_j^p. \quad (4.16)$$

- If the unit is an output unit, the error signal is given by

$$\delta_o^p = (d_o^p - y_o^p) \mathcal{F}'(s_o^p). \quad (4.17)$$

Take as the activation function  $\mathcal{F}$  the 'sigmoid' function as defined in chapter 2:

$$y^p = \mathcal{F}(s^p) = \frac{1}{1 + e^{-s^p}}. \quad (4.18)$$

In this case the derivative is equal to

$$\begin{aligned} \mathcal{F}'(s^p) &= \frac{\partial}{\partial s^p} \frac{1}{1 + e^{-s^p}} \\ &= \frac{1}{(1 + e^{-s^p})^2} (-e^{-s^p}) \\ &= \frac{1}{(1 + e^{-s^p})} \frac{e^{-s^p}}{(1 + e^{-s^p})} \\ &= y^p(1 - y^p). \end{aligned} \quad (4.19)$$



such that the error signal for an output unit can be written as:

$$\delta_o^p = (d_o^p - y_o^p) y_o^p (1 - y_o^p). \quad (4.20)$$

- The error signal for a hidden unit is determined recursively in terms of error signals of the units to which it directly connects and the weights of those connections. For the sigmoid activation function:

$$\delta_h^p = \mathcal{F}'(s_h^p) \sum_{o=1}^{N_o} \delta_o^p w_{ho} = y_h^p (1 - y_h^p) \sum_{o=1}^{N_o} \delta_o^p w_{ho}. \quad (4.21)$$

**Learning rate and momentum.** The learning procedure requires that the change in weight is proportional to  $\partial E^p / \partial w$ . True gradient descent requires that infinitesimal steps are taken. The constant of proportionality is the learning rate  $\gamma$ . For practical purposes we choose a learning rate that is as large as possible without leading to oscillation. One way to avoid oscillation at large  $\gamma$ , is to make the change in weight dependent of the past weight change by adding a *momentum* term:

$$\Delta w_{jk}(t+1) = \gamma \delta_k^p y_j^p + \alpha \Delta w_{jk}(t), \quad (4.22)$$

where  $t$  indexes the presentation number and  $\alpha$  is a constant which determines the effect of the previous weight change.

The role of the momentum term is shown in figure 4.2. When no momentum term is used, it takes a long time before the minimum has been reached with a low learning rate, whereas for high learning rates the minimum is never reached because of the oscillations. When adding the momentum term, the minimum will be reached faster.

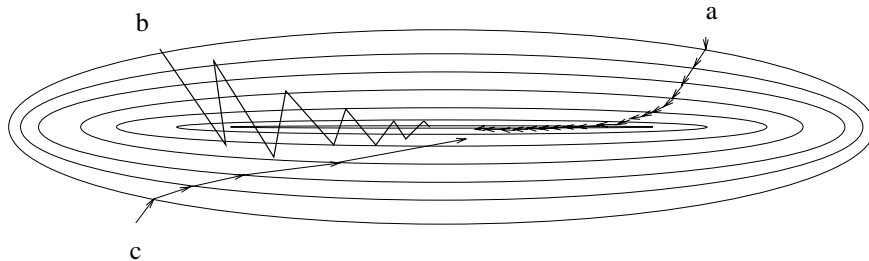


Figure 4.2: The descent in weight space. a) for small learning rate; b) for large learning rate: note the oscillations, and c) with large learning rate and momentum term added.

**Learning per pattern.** Although, theoretically, the back-propagation algorithm performs gradient descent on the total error only if the weights are adjusted after the full set of learning patterns has been presented, more often than not the learning rule is applied to each pattern separately, i.e., a pattern  $p$  is applied,  $E^p$  is calculated, and the weights are adapted ( $p = 1, 2, \dots, P$ ). There exists empirical indication that this results in faster convergence. Care has to be taken, however, with the order in which the patterns are taught. For example, when using the same sequence over and over again the network may become focused on the first few patterns. This problem can be overcome by using a permuted training method.

## 4.4 An example

A feed-forward network can be used to approximate a function from examples. Suppose we have a system (for example a chemical process or a financial market) of which we want to know

the characteristics. The input of the system is given by the two-dimensional vector  $\mathbf{x}$  and the output is given by the one-dimensional vector  $d$ . We want to estimate the relationship  $d = f(\mathbf{x})$  from 80 examples  $\{\mathbf{x}^p, d^p\}$  as depicted in figure 4.3 (top left). A feed-forward network was

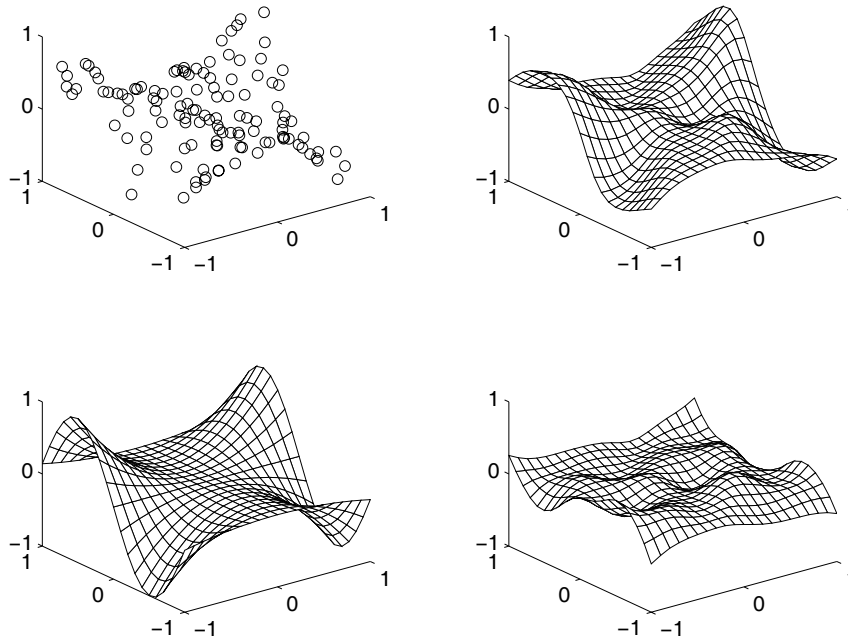


Figure 4.3: Example of function approximation with a feedforward network. Top left: The original learning samples; Top right: The approximation with the network; Bottom left: The function which generated the learning samples; Bottom right: The error in the approximation.

programmed with two inputs, 10 hidden units with sigmoid activation function and an output unit with a linear activation function. Check for yourself how equation (4.20) should be adapted for the linear instead of sigmoid activation function. The network weights are initialized to small values and the network is trained for 5,000 learning iterations with the back-propagation training rule, described in the previous section. The relationship between  $\mathbf{x}$  and  $d$  as represented by the network is shown in figure 4.3 (top right), while the function which generated the learning samples is given in figure 4.3 (bottom left). The approximation error is depicted in figure 4.3 (bottom right). We see that the error is higher at the edges of the region within which the learning samples were generated. The network is considerably better at interpolation than extrapolation.

## 4.5 Other activation functions

Although sigmoid functions are quite often used as activation functions, other functions can be used as well. In some cases this leads to a formula which is known from traditional function approximation theories.

For example, from Fourier analysis it is known that any periodic function can be written as a infinite sum of sine and cosine terms (Fourier series):

$$f(x) = \sum_{n=0}^{\infty} (a_n \cos nx + b_n \sin nx). \quad (4.23)$$

We can rewrite this as a summation of sine terms

$$f(x) = a_0 + \sum_{n=1}^{\infty} c_n \sin(nx + \theta_n), \quad (4.24)$$

with  $c_n = \sqrt{a_n^2 + b_n^2}$  and  $\theta_n = \arctan(b/a)$ . This can be seen as a feed-forward network with a single input unit for  $x$ ; a single output unit for  $f(x)$  and hidden units with an activation function  $\mathcal{F} = \sin(s)$ . The factor  $a_0$  corresponds with the bias of the output unit, the factors  $c_n$  correspond with the weights from hidden to output unit; the phase factor  $\theta_n$  corresponds with the bias term of the hidden units and the factor  $n$  corresponds with the weights between the input and hidden layer.

The basic difference between the Fourier approach and the back-propagation approach is that in the Fourier approach the ‘weights’ between the input and the hidden units (these are the factors  $n$ ) are fixed integer numbers which are analytically determined, whereas in the back-propagation approach these weights can take any value and are typically learning using a learning heuristic.

To illustrate the use of other activation functions we have trained a feed-forward network with one output unit, four hidden units, and one input with ten patterns drawn from the function  $f(x) = \sin(2x) \sin(x)$ . The result is depicted in Figure 4.4. The same function (albeit with other learning points) is learned with a network with eight (!) sigmoid hidden units (see figure 4.5). From the figures it is clear that it pays off to use as much knowledge of the problem at hand as possible.

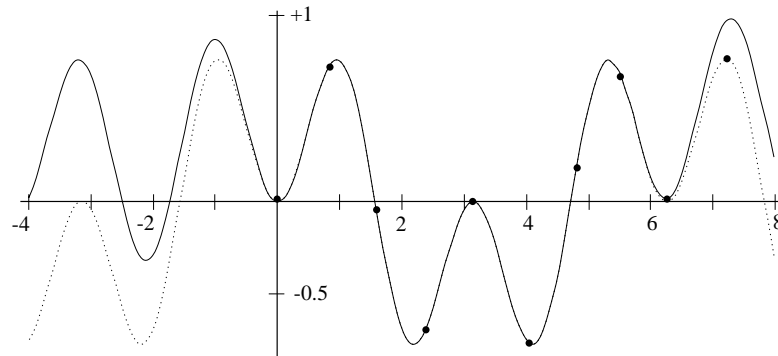


Figure 4.4: The periodic function  $f(x) = \sin(2x) \sin(x)$  approximated with sine activation functions. (Adapted from (Dastani, 1991).)

## 4.6 Deficiencies of back-propagation

Despite the apparent success of the back-propagation learning algorithm, there are some aspects which make the algorithm not guaranteed to be universally useful. Most troublesome is the long training process. This can be a result of a non-optimum learning rate and momentum. A lot of advanced algorithms based on back-propagation learning have some optimised method to adapt this learning rate, as will be discussed in the next section. Outright training failures generally arise from two sources: network paralysis and local minima.

**Network paralysis.** As the network trains, the weights can be adjusted to very large values. The total input of a hidden unit or output unit can therefore reach very high (either positive or negative) values, and because of the sigmoid activation function the unit will have an activation very close to zero or very close to one. As is clear from equations (4.20) and (4.21), the weight

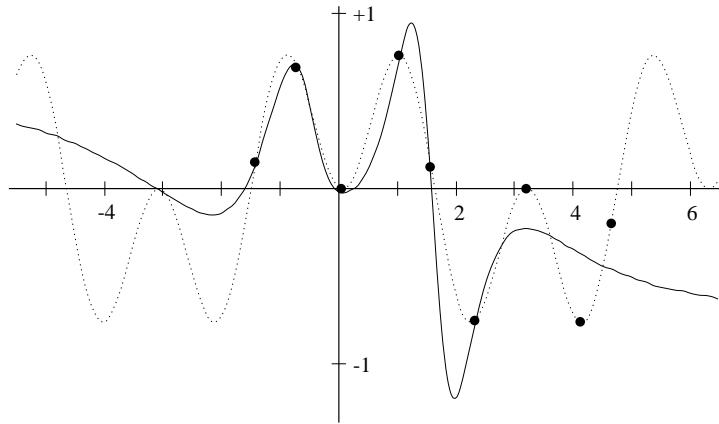


Figure 4.5: The periodic function  $f(x) = \sin(2x)\sin(x)$  approximated with sigmoid activation functions.

(Adapted from (Dastani, 1991).)

adjustments which are proportional to  $y_k^p(1 - y_k^p)$  will be close to zero, and the training process can come to a virtual standstill.

**Local minima.** The error surface of a complex network is full of hills and valleys. Because of the gradient descent, the network can get trapped in a local minimum when there is a much deeper minimum nearby. Probabilistic methods can help to avoid this trap, but they tend to be slow. Another suggested possibility is to increase the number of hidden units. Although this will work because of the higher dimensionality of the error space, and the chance to get trapped is smaller, it appears that there is some upper limit of the number of hidden units which, when exceeded, again results in the system being trapped in local minima.

## 4.7 Advanced algorithms

Many researchers have devised improvements of and extensions to the basic back-propagation algorithm described above. It is too early for a full evaluation: some of these techniques may prove to be fundamental, others may simply fade away. A few methods are discussed in this section.

Maybe the most obvious improvement is to replace the rather primitive steepest descent method with a *direction set* minimisation method, e.g., conjugate gradient minimisation. Note that minimisation along a direction  $\mathbf{u}$  brings the function  $f$  at a place where its gradient is perpendicular to  $\mathbf{u}$  (otherwise minimisation along  $\mathbf{u}$  is not complete). Instead of following the gradient at every step, a set of  $n$  directions is constructed which are all conjugate to each other such that minimisation along one of these directions  $\mathbf{u}_j$  does not spoil the minimisation along one of the earlier directions  $\mathbf{u}_i$ , i.e., the directions are non-interfering. Thus one minimisation in the direction of  $\mathbf{u}_i$  suffices, such that  $n$  minimisations in a system with  $n$  degrees of freedom bring this system to a minimum (provided the system is quadratic). This is different from gradient descent, which directly minimises in the direction of the steepest descent (Press, Flannery, Teukolsky, & Vetterling, 1986).

Suppose the function to be minimised is approximated by its Taylor series

$$\begin{aligned} f(\mathbf{x}) &= f(\mathbf{p}) + \sum_i \left. \frac{\partial f}{\partial x_i} \right|_{\mathbf{p}} x_i + \frac{1}{2} \sum_{i,j} \left. \frac{\partial^2 f}{\partial x_i \partial x_j} \right|_{\mathbf{p}} x_i x_j + \cdots \\ &\approx \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x} + c \end{aligned}$$

where  $T$  denotes transpose, and

$$c \equiv f(\mathbf{p}) \quad \mathbf{b} \equiv -\nabla f|_{\mathbf{p}} \quad [\mathbf{A}]_{ij} \equiv \left. \frac{\partial^2 f}{\partial x_i \partial x_j} \right|_{\mathbf{p}}. \quad (4.25)$$

$\mathbf{A}$  is a symmetric positive definite<sup>2</sup>  $n \times n$  matrix, the *Hessian* of  $f$  at  $\mathbf{p}$ . The gradient of  $f$  is

$$\nabla f = \mathbf{A}\mathbf{x} - \mathbf{b}, \quad (4.27)$$

such that a change of  $\mathbf{x}$  results in a change of the gradient as

$$\delta(\nabla f) = \mathbf{A}(\delta\mathbf{x}). \quad (4.28)$$

Now suppose  $f$  was minimised along a direction  $\mathbf{u}_i$  to a point where the gradient  $-\mathbf{g}_{i+1}$  of  $f$  is perpendicular to  $\mathbf{u}_i$ , i.e.,

$$\mathbf{u}_i^T \mathbf{g}_{i+1} = 0, \quad (4.29)$$

and a new direction  $\mathbf{u}_{i+1}$  is sought. In order to make sure that moving along  $\mathbf{u}_{i+1}$  does not spoil minimisation along  $\mathbf{u}_i$  we require that the gradient of  $f$  remain perpendicular to  $\mathbf{u}_i$ , i.e.,

$$\mathbf{u}_i^T \mathbf{g}_{i+2} = 0; \quad (4.30)$$

otherwise we would once more have to minimise in a direction which has a component of  $\mathbf{u}_i$ . Combining (4.29) and (4.30), we get

$$0 = \mathbf{u}_i^T (\mathbf{g}_{i+1} - \mathbf{g}_{i+2}) = \mathbf{u}_i^T \delta(\nabla f) = \mathbf{u}_i^T \mathbf{A} \mathbf{u}_{i+1}. \quad (4.31)$$

When eq. (4.31) holds for two vectors  $\mathbf{u}_i$  and  $\mathbf{u}_{i+1}$  they are said to be *conjugate*.

Now, starting at some point  $\mathbf{p}_0$ , the first minimisation direction  $\mathbf{u}_0$  is taken equal to  $\mathbf{g}_0 = -\nabla f(\mathbf{p}_0)$ , resulting in a new point  $\mathbf{p}_1$ . For  $i \geq 0$ , calculate the directions

$$\mathbf{u}_{i+1} = \mathbf{g}_{i+1} + \gamma_i \mathbf{u}_i, \quad (4.32)$$

where  $\gamma_i$  is chosen to make  $\mathbf{u}_i^T \mathbf{A} \mathbf{u}_{i+1} = 0$  and the successive gradients perpendicular, i.e.,

$$\gamma_i = \frac{\mathbf{g}_{i+1}^T \mathbf{g}_{i+1}}{\mathbf{g}_i^T \mathbf{g}_i} \quad \text{with} \quad \mathbf{g}_k = -\nabla f|_{\mathbf{p}_k} \quad \text{for all } k \geq 0. \quad (4.33)$$

Next, calculate  $\mathbf{p}_{i+2} = \mathbf{p}_{i+1} + \lambda_{i+1} \mathbf{u}_{i+1}$  where  $\lambda_{i+1}$  is chosen so as to minimise  $f(\mathbf{p}_{i+2})$ <sup>3</sup>.

It can be shown that the  $\mathbf{u}$ 's thus constructed are all mutually conjugate (e.g., see (Stoer & Bulirsch, 1980)). The process described above is known as the *Fletcher-Reeves* method, but there are many variants which work more or less the same (Hestenes & Stiefel, 1952; Polak, 1971; Powell, 1977).

Although only  $n$  iterations are needed for a quadratic system with  $n$  degrees of freedom, due to the fact that we are not minimising quadratic systems, as well as a result of round-off errors, the  $n$  directions have to be followed several times (see figure 4.6). Powell introduced some improvements to correct for behaviour in non-quadratic systems. The resulting cost is  $O(n)$  which is significantly better than the linear convergence<sup>4</sup> of steepest descent.

---

<sup>2</sup>A matrix  $\mathbf{A}$  is called *positive definite* if  $\forall \mathbf{y} \neq \mathbf{0}$ ,

$$\mathbf{y}^T \mathbf{A} \mathbf{y} > 0. \quad (4.26)$$

<sup>3</sup>This is not a trivial problem (see (Press et al., 1986).) However, line minimisation methods exist with super-linear convergence (see footnote 4).

<sup>4</sup>A method is said to converge linearly if  $E_{i+1} = cE_i$  with  $c < 1$ . Methods which converge with a higher power, i.e.,  $E_{i+1} = c(E_i)^m$  with  $m > 1$  are called *super-linear*.

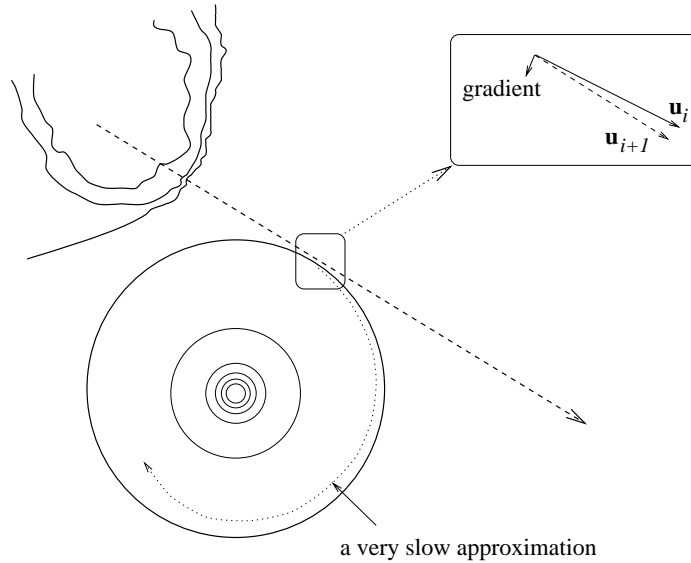


Figure 4.6: Slow decrease with conjugate gradient in non-quadratic systems. The hills on the left are very steep, resulting in a large search vector  $\mathbf{u}_i$ . When the quadratic portion is entered the new search direction is constructed from the previous direction and the gradient, resulting in a spiraling minimisation. This problem can be overcome by detecting such spiraling minimisations and restarting the algorithm with  $\mathbf{u}_0 = -\nabla f$ .

Some improvements on back-propagation have been presented based on an independent adaptive learning rate parameter for each weight.

Van den Boomgaard and Smeulders (Boomgaard & Smeulders, 1989) show that for a feed-forward network without hidden units an incremental procedure to find the optimal weight matrix  $W$  needs an adjustment of the weights with

$$\Delta W(t+1) = \gamma(t+1) (\mathbf{d}(t+1) - W(t) \mathbf{x}(t+1)) \mathbf{x}(t+1), \quad (4.34)$$

in which  $\gamma$  is not a constant but an variable  $(N_i + 1) \times (N_i + 1)$  matrix which depends on the input vector. By using *a priori* knowledge about the input signal, the storage requirements for  $\gamma$  can be reduced.

Silva and Almeida (Silva & Almeida, 1990) also show the advantages of an independent step size for each weight in the network. In their algorithm the learning rate is adapted after every learning pattern:

$$\gamma_{jk}(t+1) = \begin{cases} u\gamma_{jk}(t) & \text{if } \frac{\partial E(t+1)}{\partial w_{jk}} \text{ and } \frac{\partial E(t)}{\partial w_{jk}} \text{ have the same signs;} \\ d\gamma_{jk}(t) & \text{if } \frac{\partial E(t+1)}{\partial w_{jk}} \text{ and } \frac{\partial E(t)}{\partial w_{jk}} \text{ have opposite signs.} \end{cases} \quad (4.35)$$

where  $u$  and  $d$  are positive constants with values slightly above and below unity, respectively. The idea is to decrease the learning rate in case of oscillations.

## 4.8 How good are multi-layer feed-forward networks?

From the example shown in figure 4.3 is clear that the approximation of the network is not perfect. The resulting approximation error is influenced by:

1. The learning algorithm and number of iterations. This determines how good the error on the training set is minimized.

2. The number of learning samples. This determines how good the training samples represent the actual function.
3. The number of hidden units. This determines the ‘expressive power’ of the network. For ‘smooth’ functions only a few number of hidden units are needed, for wildly fluctuating functions more hidden units will be needed.

In the previous sections we discussed the learning rules such as back-propagation and the other gradient based learning algorithms, and the problem of finding the minimum error. In this section we particularly address the effect of the number of learning samples and the effect of the number of hidden units.

We first have to define an adequate error measure. All neural network training algorithms try to minimize the error of the set of *learning* samples which are available for training the network. The average error per learning sample is defined as the *learning error rate* error rate:

$$E_{\text{learning}} = \frac{1}{P_{\text{learning}}} \sum_{p=1}^{P_{\text{learning}}} E^p,$$

in which  $E^p$  is the difference between the desired output value and the actual network output for the learning samples:

$$E^p = \frac{1}{2} \sum_{o=1}^{N_o} (d_o^p - y_o^p)^2.$$

This is the error which is measurable during the training process.

It is obvious that the actual error of the network will differ from the error at the locations of the training samples. The difference between the desired output value and the actual network output should be integrated over the entire input domain to give a more realistic error measure. This integral can be estimated if we have a large set of samples: the *test* set. We now define the *test* error rate as the average error of the test set:

$$E_{\text{test}} = \frac{1}{P_{\text{test}}} \sum_{p=1}^{P_{\text{test}}} E^p.$$

In the following subsections we will see how these error measures depend on learning set size and number of hidden units.

#### 4.8.1 The effect of the number of learning samples

A simple problem is used as example: a function  $y = f(x)$  has to be approximated with a feed-forward neural network. A neural network is created with an input, 5 hidden units with sigmoid activation function and a linear output unit. Suppose we have only a small number of learning samples (e.g., 4) and the networks is trained with these samples. Training is stopped when the error does not decrease anymore. The original (desired) function is shown in figure 4.7A as a dashed line. The learning samples and the approximation of the network are shown in the same figure. We see that in this case  $E_{\text{learning}}$  is small (the network output goes perfectly through the learning samples) but  $E_{\text{test}}$  is large: the test error of the network is large. The approximation obtained from 20 learning samples is shown in figure 4.7B. The  $E_{\text{learning}}$  is larger than in the case of 5 learning samples, but the  $E_{\text{test}}$  is smaller.

This experiment was carried out with other learning set sizes, where for each learning set size the experiment was repeated 10 times. The average learning and test error rates as a function of the learning set size are given in figure 4.8. Note that the learning error increases with an increasing learning set size, and the test error decreases with increasing learning set size. A low

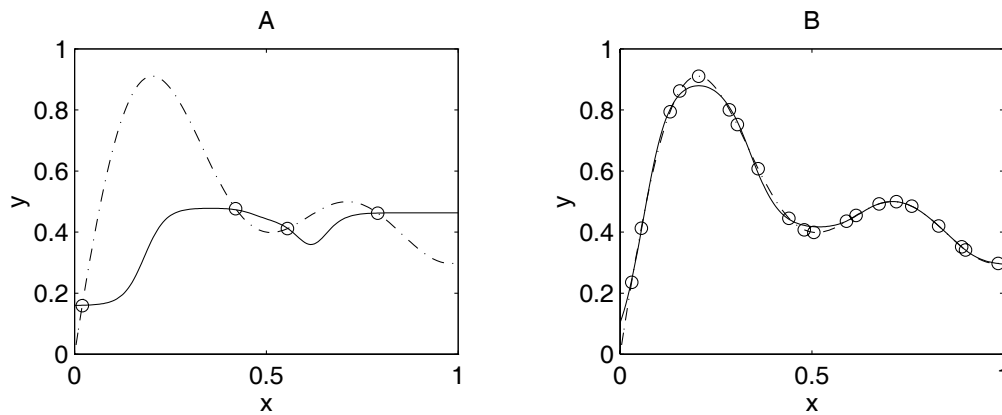


Figure 4.7: Effect of the learning set size on the generalization. The dashed line gives the desired function, the learning samples are depicted as circles and the approximation by the network is shown by the drawn line. 5 hidden units are used. a) 4 learning samples. b) 20 learning samples.

learning error on the (small) learning set is no guarantee for a good network performance! With increasing number of learning samples the two error rates converge to the same value. This value depends on the *representational* power of the network: given the optimal weights, how good is the approximation. This error depends on the number of hidden units and the activation function. If the learning error rate does not converge to the test error rate the learning procedure has not found a global minimum.

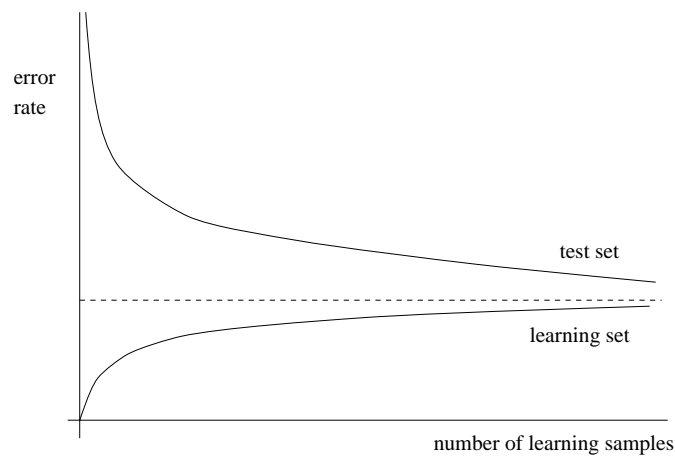


Figure 4.8: Effect of the learning set size on the error rate. The average error rate and the average test error rate as a function of the number of learning samples.

#### 4.8.2 The effect of the number of hidden units

The same function as in the previous subsection is used, but now the number of hidden units is varied. The original (desired) function, learning samples and network approximation is shown in figure 4.9A for 5 hidden units and in figure 4.9B for 20 hidden units. The effect visible in figure 4.9B is called *overtraining*. The network fits exactly with the learning samples, but because of the large number of hidden units the function which is actually represented by the network is far more wild than the original one. Particularly in case of learning samples which contain a certain amount of noise (which all real-world data have), the network will ‘fit the noise’ of the learning samples instead of making a smooth approximation.



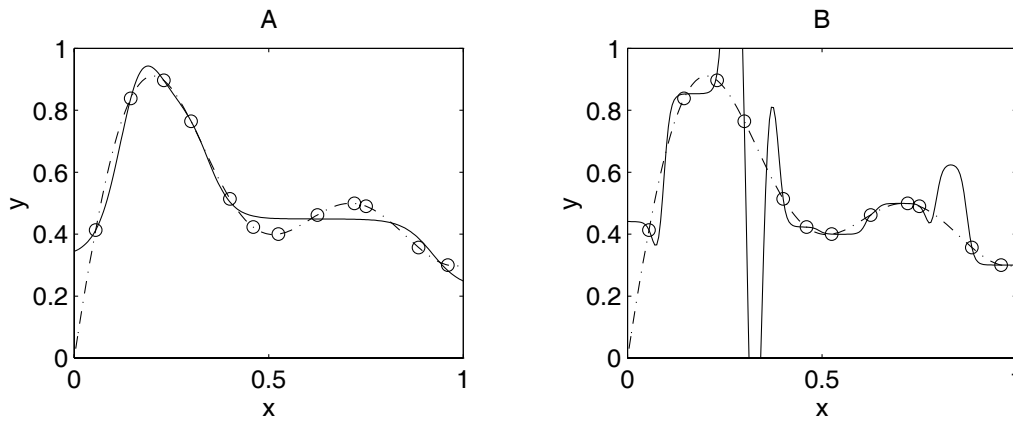


Figure 4.9: Effect of the number of hidden units on the network performance. The dashed line gives the desired function, the circles denote the learning samples and the drawn line gives the approximation by the network. 12 learning samples are used. a) 5 hidden units. b) 20 hidden units.

This example shows that a large number of hidden units leads to a small error on the training set but not necessarily leads to a small error on the test set. Adding hidden units will always lead to a reduction of the  $E_{\text{learning}}$ . However, adding hidden units will first lead to a reduction of the  $E_{\text{test}}$ , but then lead to an increase of  $E_{\text{test}}$ . This effect is called the *peaking effect*. The average learning and test error rates as a function of the learning set size are given in figure 4.10.

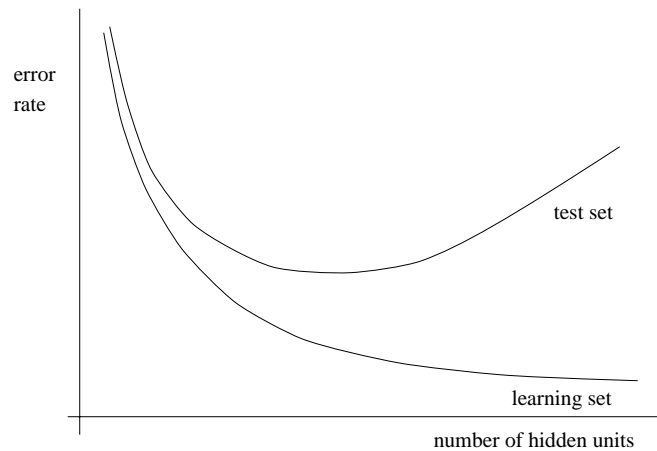


Figure 4.10: The average learning error rate and the average test error rate as a function of the number of hidden units.

## 4.9 Applications

Back-propagation has been applied to a wide variety of research applications. Sejnowski and Rosenberg (1987) (Sejnowski & Rosenberg, 1986) produced a spectacular success with NETtalk, a system that converts printed English text into highly intelligible speech.

A feed-forward network with one layer of hidden units has been described by Gorman and Sejnowski (1988) (Gorman & Sejnowski, 1988) as a classification machine for sonar signals.

Another application of a multi-layer feed-forward network with a back-propagation training algorithm is to learn an unknown function between input and output signals from the presen-

tation of examples. It is hoped that the network is able to generalise correctly, so that input values which are not presented as learning patterns will result in correct output values. An example is the work of Josin (Josin, 1988), who used a two-layer feed-forward network with back-propagation learning to perform the inverse kinematic transform which is needed by a robot arm controller (see chapter 8).