An introduction to Neural Networks

Ben Kröse

Patrick van der Smagt

Eighth edition

November 1996

The authors can be reached at:

Ben Kröse
Faculty of Mathematics & Computer Science
University of Amsterdam
Kruislaan 403, NL–1098 SJ Amsterdam
THE NETHERLANDS
Phone: +31 20 525 7463
Fax: +31 20 525 7490
email: `krose@fwi.uva.nl`
URL: `http://www.fwi.uva.nl/research/neuro/`

Patrick van der Smagt
Institute of Robotics and System Dynamics
German Aerospace Research Establishment
P. O. Box 1116, D–82230 Wessling
GERMANY
Phone: +49 8153 282400
Fax: +49 8153 281134
email: `smagt@dlr.de`
URL: `http://www.op.dlr.de/FF-DR-RS/`

Contents

List of Figures

Preface

This manuscript attempts to provide the reader with an insight in artificial neural networks. Back in 1990, the absence of any state-of-the-art textbook forced us into writing our own. However, in the meantime a number of worthwhile textbooks have been published which can be used for background and in-depth information. We are aware of the fact that, at times, this manuscript may prove to be too thorough or not thorough enough for a complete understanding of the material; therefore, further reading material can be found in some excellent text books such as (Hertz, Krogh, & Palmer, 1991; Ritter, Martinetz, & Schulten, 1990; Kohonen, 1995; Anderson & Rosenfeld, 1988; DARPA, 1988; McClelland & Rumelhart, 1986; Rumelhart & McClelland, 1986).

Some of the material in this book, especially parts III and IV, contains timely material and thus may heavily change throughout the ages. The choice of describing robotics and vision as neural network applications coincides with the neural network research interests of the authors.

Much of the material presented in chapter 6 has been written by Joris van Dam and Anuj Dev at the University of Amsterdam. Also, Anuj contributed to material in chapter 9. The basis of chapter 7 was form by a report of Gerard Schram at the University of Amsterdam. Furthermore, we express our gratitude to those people out there in Net-Land who gave us feedback on this manuscript, especially Michiel van der Korst and Nicolas Maudit who pointed out quite a few of our goof-ups. We owe them many *kwartjes* for their help.

The seventh edition is not drastically different from the sixth one; we corrected some typing errors, added some examples and deleted some obscure parts of the text. In the eighth edition, symbols used in the text have been globally changed. Also, the chapter on recurrent networks has been (albeit marginally) updated. The index still requires an update, though.

Amsterdam/Oberpfaffenhofen, November 1996
Patrick van der Smagt
Ben Kröse

Part I

FUNDAMENTALS

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 Kunihiko 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) θ_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

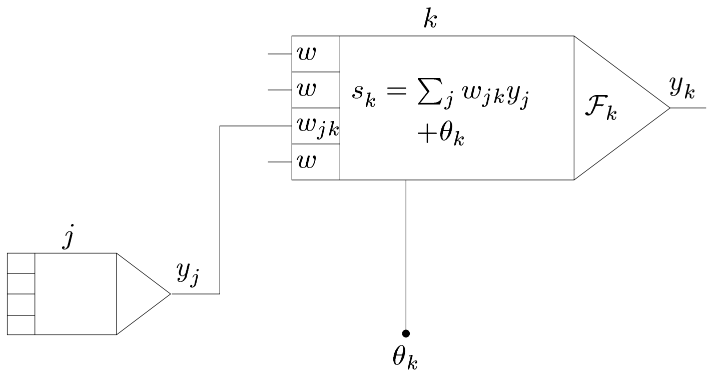


Figure 2.1: The basic components of an artificial neural network. The propagation rule used here is the 'standard' weighted summation.

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 θ_k:

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

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*:

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

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:

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

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

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

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

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

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

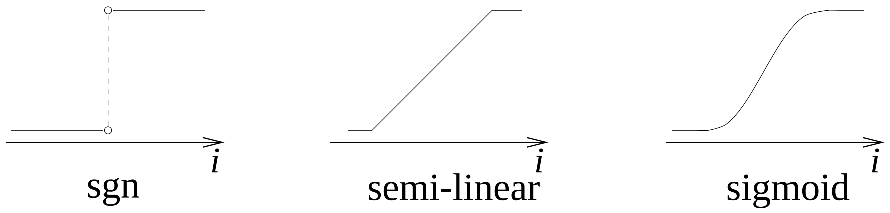


Figure 2.2: Various activation functions for a unit.

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:

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

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

$$\Delta w_{jk} = \gamma y_j y_k, \tag{2.7}$$

where γ 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:

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

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, ... the unit j, k, ...;

i an input unit;

h a hidden unit;

o an output unit;

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

x_j^p the jth element of the pth 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 jth 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;

γ_{jk} the learning rate associated with weight w_{jk};

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

θ_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 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

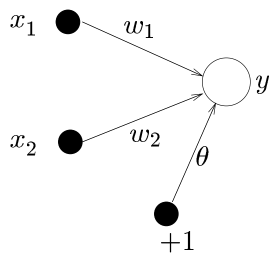


Figure 3.1: Single layer network with one output and two inputs.

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

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

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:

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

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:

$$w_1 x_1 + w_2 x_2 + \theta = 0 \tag{3.3}$$

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

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{\theta}{w_2}, \tag{3.4}$$

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.

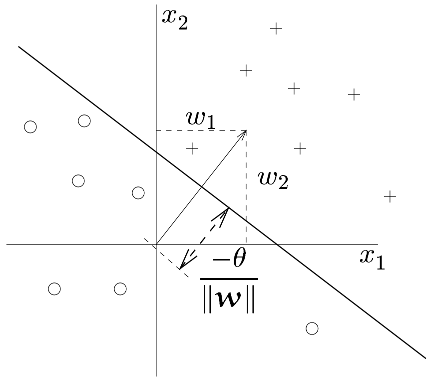


Figure 3.2: Geometric representation of the discriminant function and the weights.

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:

$$\begin{aligned} w_i(t+1) &= w_i(t) + \Delta w_i(t), & (3.5)\\ \theta(t+1) &= \theta(t) + \Delta\theta(t). & (3.6) \end{aligned}$$

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 θ. This θ 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:

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

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.

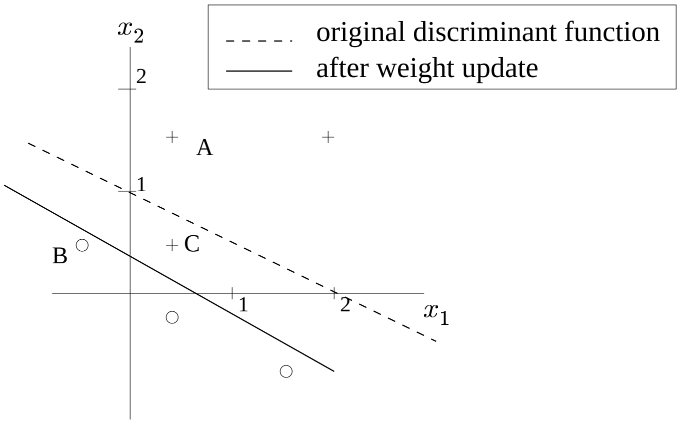


Figure 3.3: Discriminant function before and after weight update.

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 $\mathbf{x}$*[1]*. Now define $\cos\alpha \equiv \mathbf{w} \cdot \mathbf{w}^*/\|\mathbf{w}\|$. When according to the perceptron learning*

[1]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:*

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

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

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 $\lim_{t\to\infty} \cos\alpha(t) = \lim_{t\to\infty} \frac{\delta}{\sqrt{M}}\sqrt{t} = \infty$*, while by definition* $\cos\alpha \le 1$ *!*

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* $\mathbf{w} = \mathbf{0}$*,*

$$t_{max} = \frac{M}{\delta^2}. \tag{3.8}$$

□

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 ϕ_h, and a single output unit. The goal of the operation of the perceptron is to learn a given transformation $d : \{-1, 1\}^N \to \{-1, 1\}$ 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 ϕ_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 ϕ_h. Depending on the functions ϕ_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.

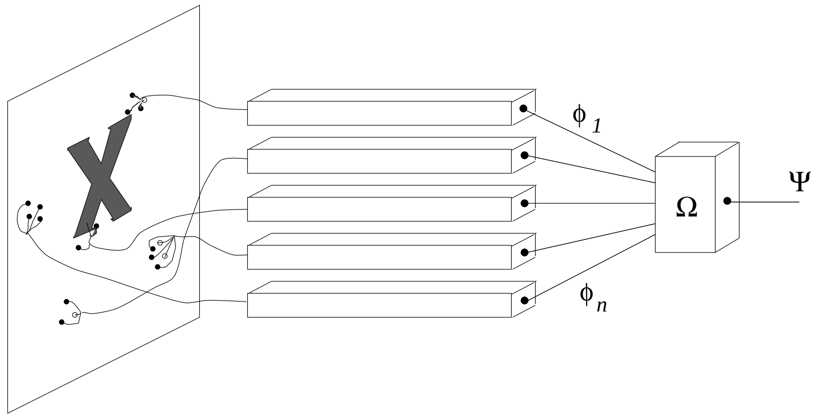


Figure 3.4: The Perceptron.

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*[2], 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 $+1$ of -1, depending on the polarity of the sum.

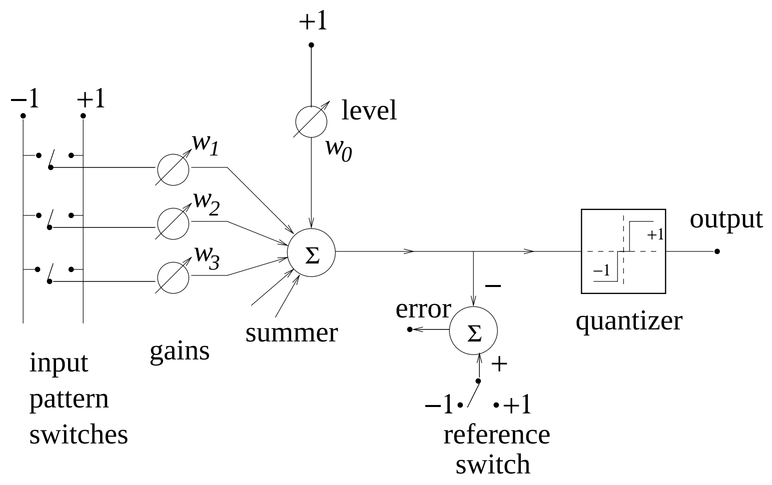


Figure 3.5: The Adaline.

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, $i = 0, 1, \ldots, n$, and the input and output signals

[2]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

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

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, $i = 1, 2, \ldots, n$, is applied at the inputs. The problem is to determine the coefficients w_i, $i = 0, 1, \ldots, n$, 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

$$y = \sum_j w_j x_j + \theta. \tag{3.10}$$

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

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

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:

$$\Delta_p w_j = -\gamma \frac{\partial E^p}{\partial w_j} \tag{3.12}$$

where γ is a constant of proportionality. The derivative is

$$\frac{\partial E^p}{\partial w_j} = \frac{\partial E^p}{\partial y^p} \frac{\partial y^p}{\partial w_j}. \tag{3.13}$$

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

$$\frac{\partial y^p}{\partial w_j} = x_j \tag{3.14}$$

and

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

such that

$$\Delta_p w_j = \gamma \delta^p x_j \tag{3.16}$$

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:

$$s = w_1 x_1 + w_2 x_2 + \theta. \tag{3.17}$$

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 θ, the output of the perceptron is equal to one on one side of the dividing line which is defined by:

$$w_1 x_1 + w_2 x_2 = -\theta \tag{3.18}$$

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

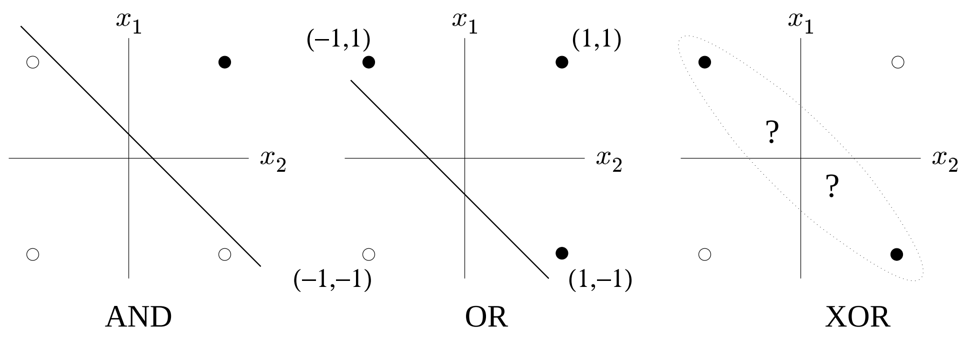
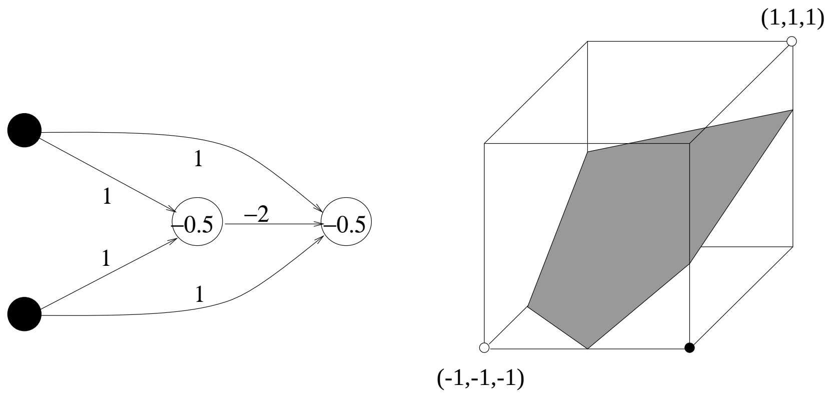


Figure 3.6: Geometric representation of input space.

To see that such a solution cannot be found, take a loot 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



a. b.

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 θ_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:

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

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 θ_h equal to $1-N$ such that

$$y_h^p = \operatorname{sgn}\left(\sum_i w_{ih} x_i^p - N + \tfrac{1}{2} \right) \tag{3.20}$$

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:

$$y_o^p = \mathrm{sgn}\left(\sum_{h=1}^{M} y_h + M - \tfrac{1}{2}\right). \tag{3.21}$$

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[1] 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

[1] Of course, when linear activation functions are used, a multi-layer network is not more powerful than a single-layer network.

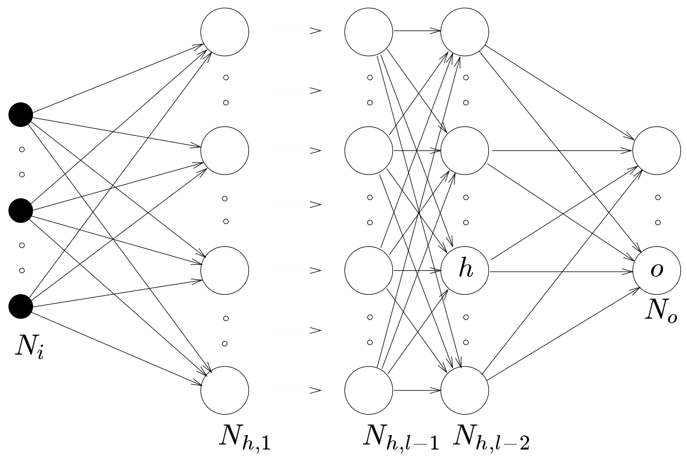


Figure 4.1: A multi-layer network with l layers of units.

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

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

in which

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

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

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

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

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

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

$$\frac{\partial E^p}{\partial w_{jk}} = \frac{\partial E^p}{\partial s_k^p} \frac{\partial s_k^p}{\partial w_{jk}}. \tag{4.5}$$

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

$$\frac{\partial s_k^p}{\partial w_{jk}} = y_j^p. \tag{4.6}$$

When we define

$$\delta_k^p = -\frac{\partial E^p}{\partial s_k^p}, \tag{4.7}$$

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:

$$\Delta_p w_{jk} = \gamma \delta_k^p y_j^p. \tag{4.8}$$

The trick is to figure out what δ_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 δ's which can be implemented by propagating error signals backward through the network.

To compute δ_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

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

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

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

which is simply the derivative of the squashing function $\mathcal{F}$ for the kth 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

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

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

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

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; $E^p = E^p(s_1^p, s_2^p, \ldots, s_j^p, \ldots)$ and we use the chain rule to write

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

Substituting this in equation (4.9) yields

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

Equations (4.12) and (4.14) give a recursive procedure for computing the δ'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

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

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 δ 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 δ_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 δ, on the unit k receiving the input and the output of the unit j sending this signal along the connection:
$$\Delta_p w_{jk} = \gamma \delta_k^p y_j^p. \tag{4.16}$$
- If the unit is an output unit, the error signal is given by
$$\delta_o^p = (d_o^p - y_o^p)\, \mathcal{F}'(s_o^p). \tag{4.17}$$
Take as the activation function $\mathcal{F}$ the 'sigmoid' function as defined in chapter 2:
$$y^p = \mathcal{F}(s^p) = \frac{1}{1 + e^{-s^p}}. \tag{4.18}$$
In this case the derivative is equal to
$$\begin{aligned} \mathcal{F}'(s^p) &= \frac{\partial}{\partial s^p} \frac{1}{1 + e^{-s^p}} \\ &= \frac{1}{(1 + e^{-s^p})^2} (-e^{-s^p}) \\ &= \frac{1}{(1 + e^{-s^p})} \frac{e^{-s^p}}{(1 + e^{-s^p})} \\ &= y^p(1 - y^p). \end{aligned} \tag{4.19}$$

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

$$\delta_o^p = (d_o^p - y_o^p)\, y_o^p(1 - y_o^p). \tag{4.20}$$

- 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:

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

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 γ. 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 γ, is to make the change in weight dependent of the past weight change by adding a *momentum* term:

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

where t indexes the presentation number and α 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.

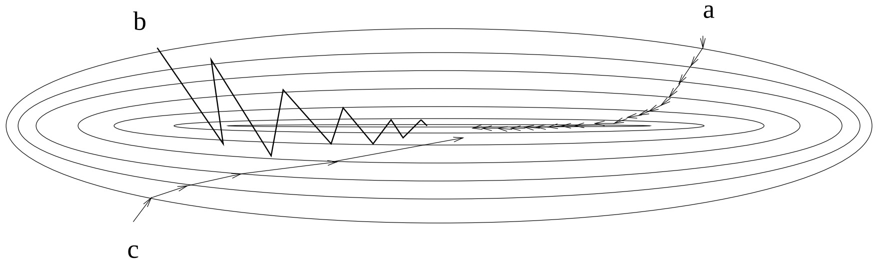


Figure 4.2: The descent in weight space. a) for small learning rate; b) for large learning rate: note the oscillations, and c) with large learning rate and momentum term added.

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 ($p = 1, 2, \ldots, P$). 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

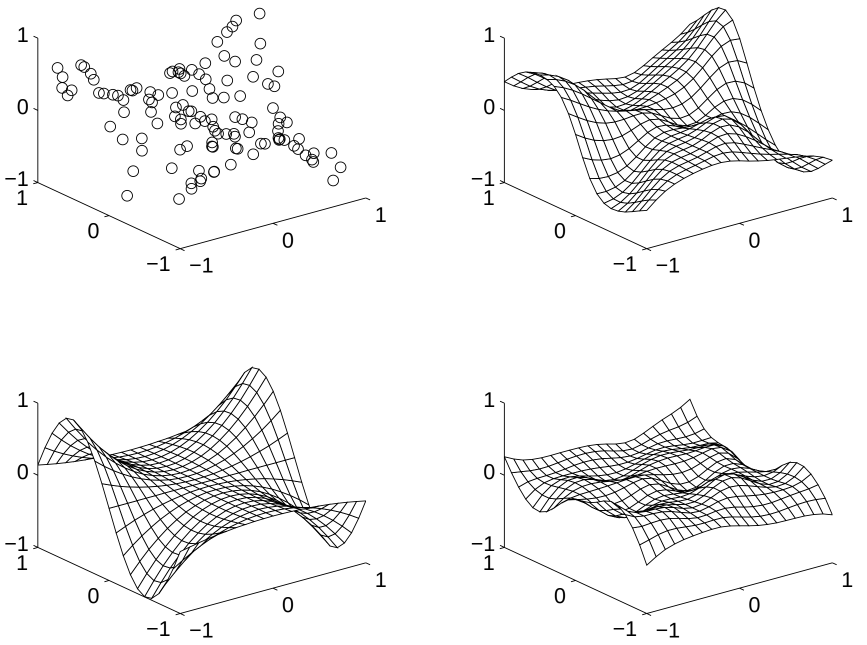


Figure 4.3: Example of function approximation with a feedforward network. Top left: The original learning samples; Top right: The approximation with the network; Bottom left: The function which generated the learning samples; Bottom right: The error in the approximation.

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):

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

We can rewrite this as a summation of sine terms

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

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 weighs from hidden to output unit; the phase factor θ_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 the 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.

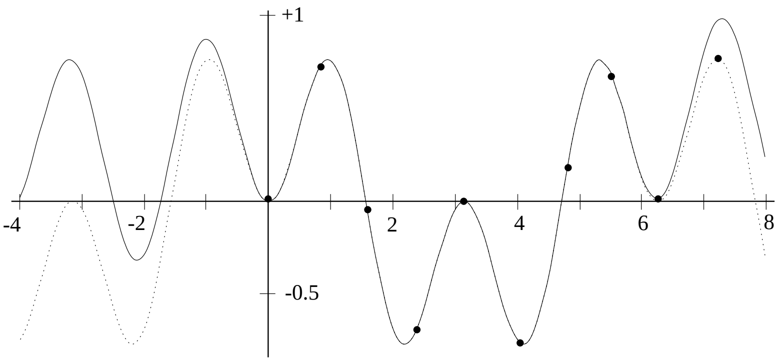


Figure 4.4: The periodic function $f(x) = \sin(2x)\sin(x)$ approximated with sine activation functions. (Adapted from (Dastani, 1991).)

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

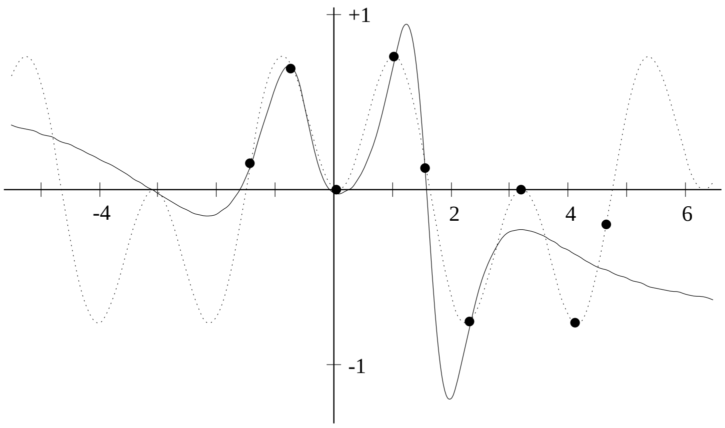


Figure 4.5: The periodic function $f(x) = \sin(2x)\sin(x)$ approximated with sigmoid activation functions.
(Adapted from (Dastani, 1991).)

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

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

where T denotes transpose, and

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

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

$$\nabla f = \mathbf{A}\mathbf{x} - \mathbf{b}, \tag{4.27}$$

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

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

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.,

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

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.,

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

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

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

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

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

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

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

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

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[4] of steepest descent.

---

[2] A matrix $\mathbf{A}$ is called *positive definite* if $\forall \mathbf{y} \neq \mathbf{0}$,

$$\mathbf{y}^T \mathbf{A} \mathbf{y} > 0. \tag{4.26}$$

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

[4] 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*.

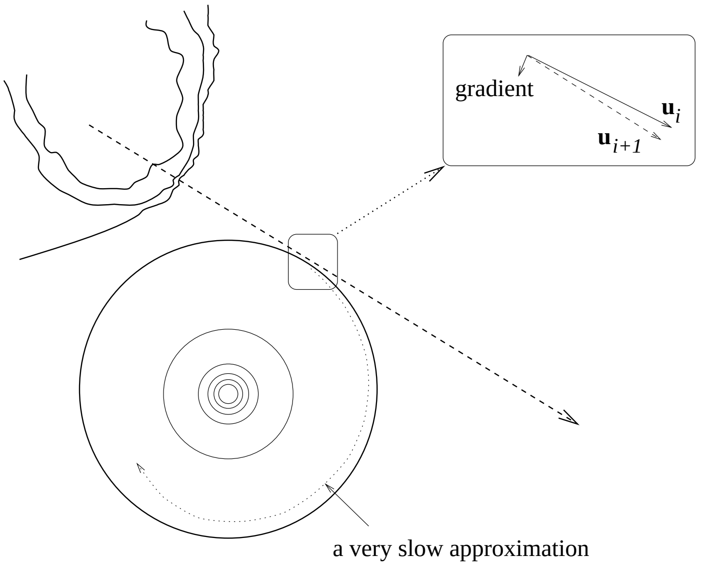


Figure 4.6: Slow decrease with conjugate gradient in non-quadratic systems. The hills on the left are very steep, resulting in a large search vector $\mathbf{u}_i$. When the quadratic portion is entered the new search direction is constructed from the previous direction and the gradient, resulting in 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

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

in which γ 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 γ 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:

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

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 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:

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

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_{learning} is small (the network output goes perfectly through the learning samples) but E_{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_{learning} is larger than in the case of 5 learning samples, but the E_{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

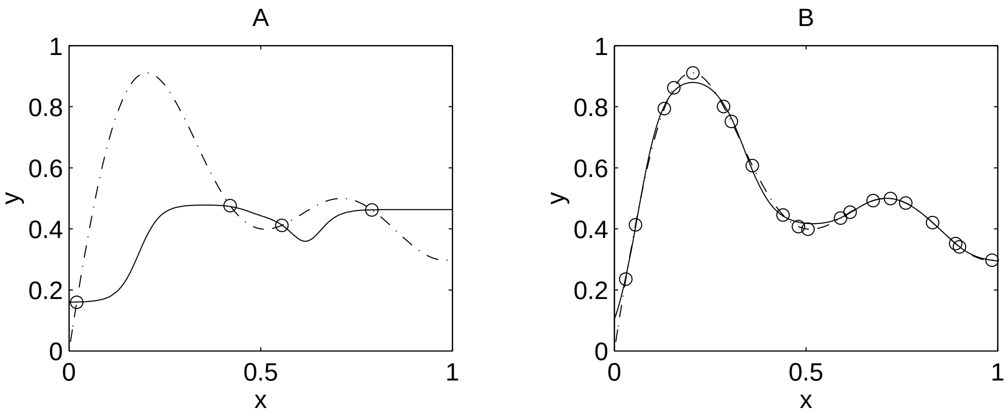


Figure 4.7: Effect of the learning set size on the generalization. The dashed line gives the desired function, the learning samples are depicted as circles and the approximation by the network is shown by the drawn line. 5 hidden units are used. a) 4 learning samples. b) 20 learning samples.

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.

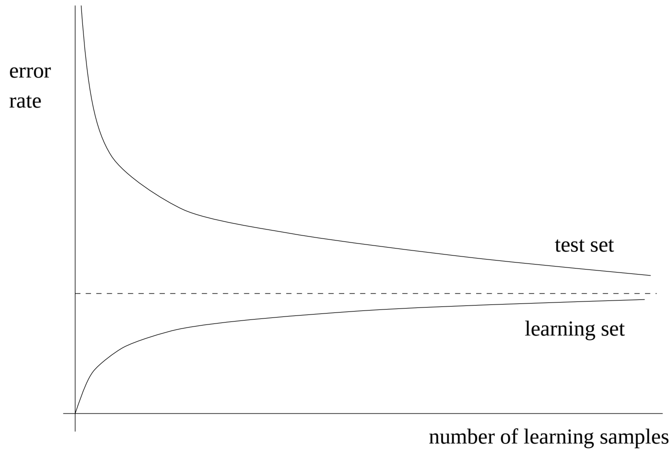


Figure 4.8: Effect of the learning set size on the error rate. The average error rate and the average test error rate as a function of the number of learning samples.

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.

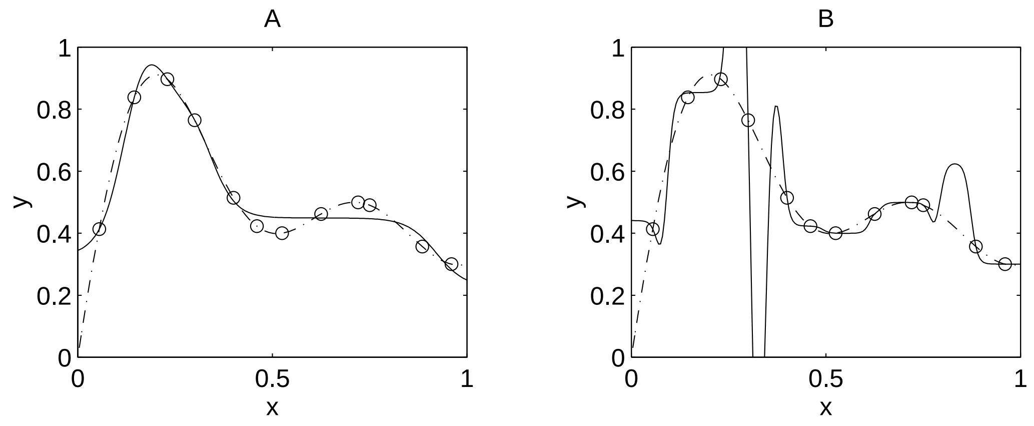


Figure 4.9: Effect of the number of hidden units on the network performance. The dashed line gives the desired function, the circles denote the learning samples and the drawn line gives the approximation by the network. 12 learning samples are used. a) 5 hidden units. b) 20 hidden units.

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_{learning}. However, adding hidden units will first lead to a reduction of the E_{test}, but then lead to an increase of E_{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.

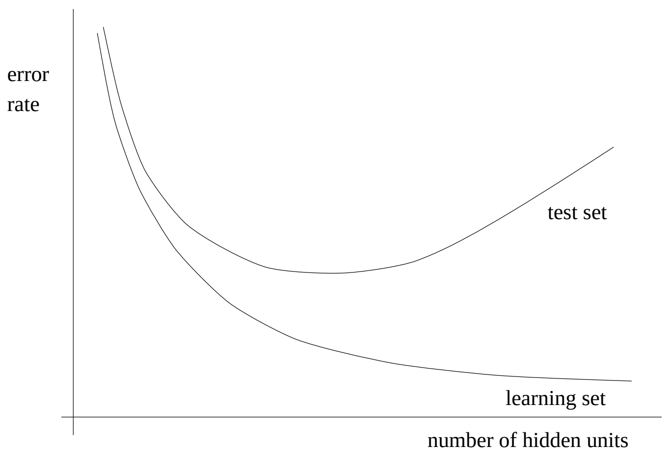


Figure 4.10: The average learning error rate and the average test error rate as a function of the number of hidden units.

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 presentation 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).

5 Recurrent Networks

The learning algorithms discussed in the previous chapter were applied to feed-forward networks: all data flows in a network in which no cycles are present.

But what happens when we introduce a cycle? For instance, we can connect a hidden unit with itself over a weighted connection, connect hidden units to input units, or even connect all units with each other. Although, as we know from the previous chapter, the approximational capabilities of such networks do not increase, we *may* obtain decreased complexity, network size, etc. to solve the same problem.

An important question we have to consider is the following: what do we want to learn in a recurrent network? After all, when one is considering a recurrent network, it is possible to continue propagating activation values *ad infinitum*, or until a stable point (attractor) is reached. As we will see in the sequel, there exist recurrent network which are attractor based, i.e., the activation values in the network are repeatedly updated until a stable point is reached after which the weights are adapted, but there are also recurrent networks where the learning rule is used after each propagation (where an activation value is transversed over each weight only once), while external inputs are included in each propagation. In such networks, the recurrent connections can be regarded as extra inputs to the network (the values of which are computed by the network itself).

In this chapter recurrent extensions to the feed-forward network introduced in the previous chapters will be discussed—yet not to exhaustion. The theory of the dynamics of recurrent networks extends beyond the scope of a one-semester course on neural networks. Yet the basics of these networks will be discussed.

Subsequently some special recurrent networks will be discussed: the Hopfield network in section 5.2, which can be used for the representation of binary patterns; subsequently we touch upon Boltzmann machines, therewith introducing stochasticity in neural computation.

5.1 The generalised delta-rule in recurrent networks

The back-propagation learning rule, introduced in chapter 4, can be easily used for training patterns in recurrent networks. Before we will consider this general case, however, we will first describe networks where some of the hidden unit activation values are fed back to an extra set of input units (the *Elman network*), or where output values are fed back into hidden units (the *Jordan network*).

A typical application of such a network is the following. Suppose we have to construct a network that must generate a control command depending on an external input, which is a time series $\mathbf{x}(t)$, $\mathbf{x}(t-1)$, $\mathbf{x}(t-2)$, ... With a feed-forward network there are two possible approaches:

1. create inputs x_1, x_2, ..., x_n which constitute the last n values of the input vector. Thus a 'time window' of the input vector is input to the network.

2. create inputs x, x', x", ... Besides only inputting $\mathbf{x}(t)$, we also input its first, second, etc.

derivatives. Naturally, computation of these derivatives is not a trivial task for higher-order derivatives.

The disadvantage is, of course, that the input dimensionality of the feed-forward network is multiplied with n, leading to a very large network, which is slow and difficult to train. The Jordan and Elman networks provide a solution to this problem. Due to the recurrent connections, a window of inputs need not be input anymore; instead, the network is supposed to learn the influence of the previous time steps itself.

5.1.1 The Jordan network

One of the earliest recurrent neural network was the Jordan network (Jordan, 1986a, 1986b). An exemplar network is shown in figure 5.1. In the Jordan network, the activation values of the

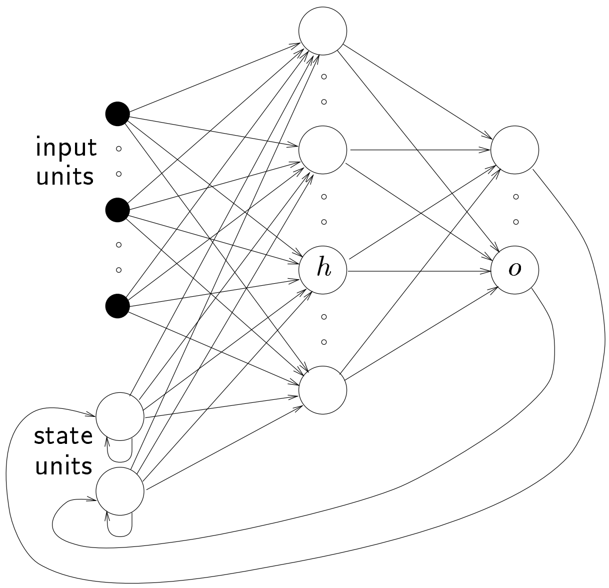


Figure 5.1: The Jordan network. Output activation values are fed back to the input layer, to a set of extra neurons called the *state units*.

output units are fed back into the input layer through a set of extra input units called the *state units*. There are as many state units as there are output units in the network. The connections between the output and state units have a fixed weight of $+1$; learning takes place only in the connections between input and hidden units as well as hidden and output units. Thus all the learning rules derived for the multi-layer perceptron can be used to train this network.

5.1.2 The Elman network

The Elman network was introduced by Elman in 1990 (Elman, 1990). In this network a set of *context units* are introduced, which are extra input units whose activation values are fed back from the hidden units. Thus the network is very similar to the Jordan network, except that (1) the hidden units instead of the output units are fed back; and (2) the extra input units have no self-connections.

The schematic structure of this network is shown in figure 5.2.

Again the hidden units are connected to the context units with a fixed weight of value $+1$. Learning is done as follows:

1. the context units are set to 0; $t = 1$;

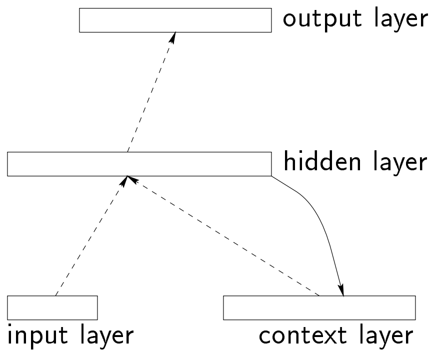


Figure 5.2: The Elman network. With this network, the hidden unit activation values are fed back to the input layer, to a set of extra neurons called the *context units*.

2. pattern $\mathbf{x}^t$ is clamped, the forward calculations are performed once;
3. the back-propagation learning rule is applied;
4. $t \leftarrow t + 1$; go to 2.

The context units at step t thus always have the activation value of the hidden units at step $t - 1$.

Example

As we mentioned above, the Jordan and Elman networks can be used to train a network on reproducing time sequences. The idea of the recurrent connections is that the network is able to 'remember' the previous states of the input values. As an example, we trained an Elman network on controlling an object moving in 1D. This object has to follow a pre-specified trajectory $\mathbf{x}_d$. To control the object, forces F must be applied, since the object suffers from friction and perhaps other external forces.

To tackle this problem, we use an Elman net with inputs x and x_d, one output F, and three hidden units. The hidden units are connected to three context units. In total, five units feed into the hidden layer.

The results of training are shown in figure 5.3. The same test can be done with an ordinary

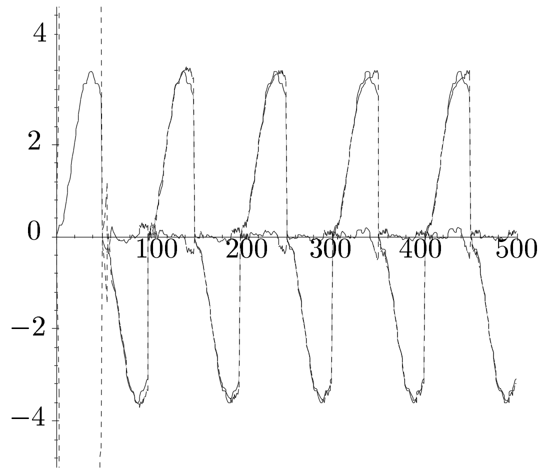


Figure 5.3: Training an Elman network to control an object. The solid line depicts the desired trajectory $\mathbf{x}_d$; the dashed line the realised trajectory. The third line is the error.

feed-forward network with sliding window input. We tested this with a network with five inputs, four of which constituted the sliding window x_{-3}, x_{-2}, x_{-1}, and x_0, and one the desired next position of the object. Results are shown in figure 5.4. The disappointing observation is that

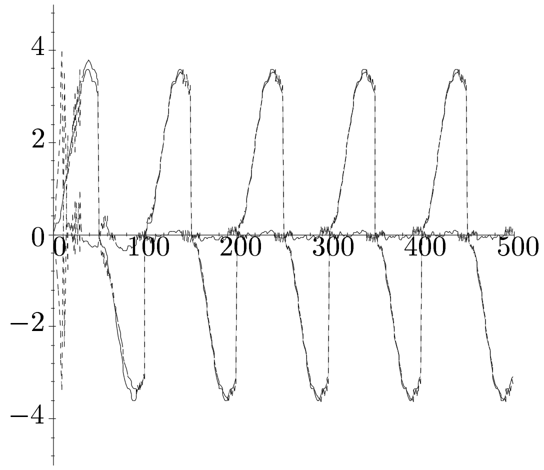


Figure 5.4: Training a feed-forward network to control an object. The solid line depicts the desired trajectory $\mathbf{x}_d$; the dashed line the realised trajectory. The third line is the error.

the results are actually better with the ordinary feed-forward network, which has the same complexity as the Elman network.

5.1.3 Back-propagation in fully recurrent networks

More complex schemes than the above are possible. For instance, independently of each other Pineda (Pineda, 1987) and Almeida (Almeida, 1987) discovered that error back-propagation is in fact a special case of a more general gradient learning method which can be used for training attractor networks. However, also when a network does not reach a fixpoint, a learning method can be used: back-propagation through time (Pearlmutter, 1989, 1990). This learning method, the discussion of which extents beyond the scope of our course, can be used to train a multi-layer perceptron to follow trajectories in its activation values.

5.2 The Hopfield network

One of the earliest recurrent neural networks reported in literature was the *auto-associator* independently described by Anderson (Anderson, 1977) and Kohonen (Kohonen, 1977) in 1977. It consists of a pool of neurons with connections between each unit i and j, $i \neq j$ (see figure 5.5). All connections are weighted.

In 1982, Hopfield (Hopfield, 1982) brings together several earlier ideas concerning these networks and presents a complete mathematical analysis based on Ising spin models (Amit, Gutfreund, & Sompolinsky, 1986). It is therefore that this network, which we will describe in this chapter, is generally referred to as the *Hopfield network*.

5.2.1 Description

The Hopfield network consists of a set of N interconnected neurons (figure 5.5) which update their activation values asynchronously and independently of other neurons. All neurons are both input and output neurons. The activation values are binary. Originally, Hopfield chose activation values of 1 and 0, but using values $+1$ and -1 presents some advantages discussed below. We will therefore adhere to the latter convention.

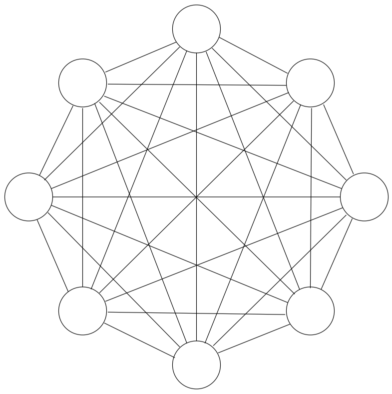


Figure 5.5: The auto-associator network. All neurons are both input and output neurons, i.e., a pattern is clamped, the network iterates to a stable state, and the output of the network consists of the new activation values of the neurons.

The state of the system is given by the activation values[1] $\mathbf{y} = (y_k)$. The net input $s_k(t+1)$ of a neuron k at cycle $t+1$ is a weighted sum

$$s_k(t+1) = \sum_{j \neq k} y_j(t)\, w_{jk} + \theta_k. \tag{5.1}$$

A simple threshold function (figure 2.2) is applied to the net input to obtain the new activation value $y_i(t+1)$ at time $t+1$:

$$y_k(t+1) = \begin{cases} +1 & \text{if } s_k(t+1) > U_k \\ -1 & \text{if } s_k(t+1) < U_k \\ y_k(t) & \text{otherwise,} \end{cases} \tag{5.2}$$

i.e., $y_k(t+1) = \text{sgn}(s_k(t+1))$. For simplicity we henceforth choose $U_k = 0$, but this is of course not essential.

A neuron k in the Hopfield network is called *stable* at time t if, in accordance with equations (5.1) and (5.2),

$$y_k(t) = \text{sgn}(s_k(t-1)). \tag{5.3}$$

A state α is called stable if, when the network is in state α, all neurons are stable. A pattern $\mathbf{x}^p$ is called stable if, when $\mathbf{x}^p$ is clamped, all neurons are stable.

When the extra restriction $w_{jk} = w_{kj}$ is made, the behaviour of the system can be described with an energy function

$$\mathcal{E} = -\tfrac{1}{2} \sum_{j \neq k} \sum y_j y_k w_{jk} - \sum_k \theta_k y_k. \tag{5.4}$$

Theorem 2 *A recurrent network with connections $w_{jk} = w_{kj}$ in which the neurons are updated using rule (5.2) has stable limit points.*

Proof *First, note that the energy expressed in eq. (5.4) is bounded from below, since the y_k are bounded from below and the w_{jk} and θ_k are constant. Secondly, $\mathcal{E}$ is monotonically decreasing when state changes occur, because*

$$\Delta\mathcal{E} = -\Delta y_k \left(\sum_{j \neq k} y_j w_{jk} + \theta_k \right) \tag{5.5}$$

is always negative when y_k changes according to eqs. (5.1) and (5.2).

□

[1] Often, these networks are described using the symbols used by Hopfield: V_k for activation of unit k, T_{jk} for the connection weight between units j and k, and U_k for the external input of unit k. We decided to stick to the more general symbols y_k, w_{jk}, and θ_k.

The advantage of a $+1/-1$ model over a $1/0$ model then is symmetry of the states of the network. For, when some pattern $\mathbf{x}$ is stable, its inverse is stable, too, whereas in the $1/0$ model this is not always true (as an example, the pattern $00\cdots00$ is always stable, but $11\cdots11$ need not be). Similarly, both a pattern and its inverse have the same energy in the $+1/-1$ model.

Removing the restriction of bidirectional connections (i.e., $w_{jk} = w_{kj}$) results in a system that is not guaranteed to settle to a stable state.

5.2.2 Hopfield network as associative memory

A primary application of the Hopfield network is an associative memory. In this case, the weights of the connections between the neurons have to be thus set that the states of the system corresponding with the patterns which are to be stored in the network are stable. These states can be seen as 'dips' in energy space. When the network is cued with a noisy or incomplete test pattern, it will render the incorrect or missing data by iterating to a stable state which is in some sense 'near' to the cued pattern.

The Hebb rule can be used (section 2.3.2) to store P patterns:

$$w_{jk} = \begin{cases} \sum_{p=1}^{P} x_j^p x_k^p & \text{if } j \neq k \\ 0 & \text{otherwise,} \end{cases} \tag{5.6}$$

i.e., if x_j^p and x_k^p are equal, w_{jk} is increased, otherwise decreased by one (note that, in the original Hebb rule, weights only increase). It appears, however, that the network gets saturated very quickly, and that about $0.15N$ memories can be stored before recall errors become severe.

There are two problems associated with storing too many patterns:

1. the stored patterns become unstable;
2. spurious stable states appear (i.e., stable states which do not correspond with stored patterns).

The first of these two problems can be solved by an algorithm proposed by Bruce *et al.* (Bruce, Canning, Forrest, Gardner, & Wallace, 1986):

Algorithm 1 *Given a starting weight matrix* $W = \left[w_{jk}\right]$, *for each pattern* $\mathbf{x}^p$ *to be stored and each element* x_k^p *in* $\mathbf{x}^p$ *define a correction* ϵ_k *such that*

$$\epsilon_k = \begin{cases} 0 & \textit{if } y_k \textit{ is stable and } \mathbf{x}^p \textit{ is clamped;} \\ 1 & \textit{otherwise.} \end{cases} \tag{5.7}$$

Now modify w_{jk} *by* $\Delta w_{jk} = y_j y_k (\epsilon_j + \epsilon_k)$ *if* $j \neq k$. *Repeat this procedure until all patterns are stable.*

It appears that, in practice, this algorithm usually converges. There exist cases, however, where the algorithm remains oscillatory (try to find one)!

The second problem stated above can be alleviated by applying the Hebb rule *in reverse* to the spurious stable state, but with a low learning factor (Hopfield, Feinstein, & Palmer, 1983). Thus these patterns are weakly *un*stored and will become unstable again.

5.2.3 Neurons with graded response

The network described in section 5.2.1 can be generalised by allowing continuous activation values. Here, the threshold activation function is replaced by a sigmoid. As before, this system can be proved to be stable when a symmetric weight matrix is used (Hopfield, 1984).

Hopfield networks for optimisation problems

An interesting application of the Hopfield network with graded response arises in a heuristic solution to the NP-complete travelling salesman problem (Garey & Johnson, 1979). In this problem, a path of minimal distance must be found between n cities, such that the begin- and end-points are the same.

Hopfield and Tank (Hopfield & Tank, 1985) use a network with $n \times n$ neurons. Each row in the matrix represents a city, whereas each column represents the position in the tour. When the network is settled, each row and each column should have one and only one active neuron, indicating a specific city occupying a specific position in the tour. The neurons are updated using rule (5.2) with a sigmoid activation function between 0 and 1. The activation value $y_{Xj} = 1$ indicates that city X occupies the j^{th} place in the tour.

An energy function describing this problem can be set up as follows. To ensure a correct solution, the following energy must be minimised:

$$\begin{aligned} \mathcal{E} \;=\; & \tfrac{A}{2} \sum_X \sum_j \sum_{k \neq j} y_{Xj} y_{Xk} \\ & + \tfrac{B}{2} \sum_j \sum_X \sum_{X \neq Y} y_{Xj} y_{Yj} \\ & + \tfrac{C}{2} \left(\sum_X \sum_j y_{Xj} - n \right)^2 \end{aligned} \tag{5.8}$$

where A, B, and C are constants. The first and second terms in equation (5.8) are zero if and only if there is a maximum of one active neuron in each row and column, respectively. The last term is zero if and only if there are exactly n active neurons.

To minimise the distance of the tour, an extra term

$$\tfrac{D}{2} \sum_X \sum_{Y \neq X} \sum_j d_{XY} y_{Xj} (y_{Y,j+1} + y_{Y,j-1}) \tag{5.9}$$

is added to the energy, where d_{XY} is the distance between cities X and Y and D is a constant. For convenience, the subscripts are defined modulo n.

The weights are set as follows:

$$\begin{aligned} w_{Xj,Yk} = \; & -A\delta_{XY}(1 - \delta_{jk}) && \text{inhibitory connections within each row} \\ & -B\delta_{jk}(1 - \delta_{XY}) && \text{inhibitory connections within each column} \\ & -C && \text{global inhibition} \\ & -Dd_{XY}(\delta_{k,j+1} + \delta_{k,j-1}) && \text{data term} \end{aligned} \tag{5.10}$$

where $\delta_{jk} = 1$ if $j = k$ and 0 otherwise. Finally, each neuron has an external bias input Cn.

Discussion

Although this application is interesting from a theoretical point of view, the applicability is limited. Whereas Hopfield and Tank state that, in a ten city tour, the network converges to a valid solution in 16 out of 20 trials while 50% of the solutions are optimal, other reports show less encouraging results. For example, (Wilson & Pawley, 1988) find that in only 15% of the runs a valid result is obtained, few of which lead to an optimal or near-optimal solution. The main problem is the lack of global information. Since, for an N-city problem, there are $N!$ possible tours, each of which may be traversed in two directions as well as started in N points, the number of different tours is $N!/2N$. Differently put, the N-dimensional hypercube in which the solutions are situated is $2N$ degenerate. The degenerate solutions occur evenly within the hypercube, such that all but one of the final $2N$ configurations are redundant. The competition between the degenerate tours often leads to solutions which are piecewise optimal but globally inefficient.

5.3 Boltzmann machines

The Boltzmann machine, as first described by Ackley, Hinton, and Sejnowski in 1985 (Ackley, Hinton, & Sejnowski, 1985) is a neural network that can be seen as an extension to Hopfield networks to include hidden units, and with a stochastic instead of deterministic update rule. The weights are still symmetric. The operation of the network is based on the physics principle of *annealing*. This is a process whereby a material is heated and then cooled very, very slowly to a freezing point. As a result, the crystal lattice will be highly ordered, without any impurities, such that the system is in a state of very low energy. In the Boltzmann machine this system is mimicked by changing the deterministic update of equation (5.2) in a stochastic update, in which a neuron becomes active with a probability p,

$$p(y_k \leftarrow +1) = \frac{1}{1 + e^{-\Delta \mathcal{E}_k / T}} \tag{5.11}$$

where T is a parameter comparable with the (synthetic) temperature of the system. This stochastic activation function is not to be confused with neurons having a sigmoid deterministic activation function.

In accordance with a physical system obeying a Boltzmann distribution, the network will eventually reach 'thermal equilibrium' and the relative probability of two global states α and β will follow the Boltzmann distribution

$$\frac{P_\alpha}{P_\beta} = e^{-(\mathcal{E}_\alpha - \mathcal{E}_\beta)/T} \tag{5.12}$$

where P_α is the probability of being in the α^{th} global state, and $\mathcal{E}_\alpha$ is the energy of that state. Note that at thermal equilibrium the units still change state, but the probability of finding the network in any global state remains constant.

At low temperatures there is a strong bias in favour of states with low energy, but the time required to reach equilibrium may be long. At higher temperatures the bias is not so favourable but equilibrium is reached faster. A good way to beat this trade-off is to start at a high temperature and gradually reduce it. At high temperatures, the network will ignore small energy differences and will rapidly approach equilibrium. In doing so, it will perform a search of the coarse overall structure of the space of global states, and will find a good minimum at that coarse level. As the temperature is lowered, it will begin to respond to smaller energy differences and will find one of the better minima within the coarse-scale minimum it discovered at high temperature.

As multi-layer perceptrons, the Boltzmann machine consists of a non-empty set of visible and a possibly empty set of hidden units. Here, however, the units are binary-valued and are updated stochastically and asynchronously. The simplicity of the Boltzmann distribution leads to a simple learning procedure which adjusts the weights so as to use the hidden units in an optimal way (Ackley et al., 1985). This algorithm works as follows.

First, the input and output vectors are clamped. The network is then annealed until it approaches thermal equilibrium at a temperature of 0. It then runs for a fixed time at equilibrium and each connection measures the fraction of the time during which both the units it connects are active. This is repeated for all input-output pairs so that each connection can measure $\langle y_j y_k \rangle^{\text{clamped}}$, the expected probability, averaged over all cases, that units j and k are simultaneously active at thermal equilibrium when the input and output vectors are clamped.

Similarly, $\langle y_j y_k \rangle^{\text{free}}$ is measured when the output units are not clamped but determined by the network.

In order to determine optimal weights in the network, an error function must be determined. Now, the probability $P^{\text{free}}(\mathbf{y}^p)$ that the visible units are in state $\mathbf{y}^p$ when the system is running freely can be measured. Also, the *desired* probability $P^{\text{clamped}}(\mathbf{y}^p)$ that the visible units are in state $\mathbf{y}^p$ is determined by clamping the visible units and letting the network run. Now, if the weights in the network are correctly set, both probabilities are equal to each other, and the error E in the network must be 0. Otherwise, the error must have a positive value measuring the discrepancy between the network's internal mode and the environment. For this effect, the 'asymmetric divergence' or 'Kullback information' is used:

$$E = \sum_p P^{\text{clamped}}(\mathbf{y}^p) \log \frac{P^{\text{clamped}}(\mathbf{y}^p)}{P^{\text{free}}(\mathbf{y}^p)}, \tag{5.13}$$

Now, in order to minimise E using gradient descent, we must change the weights according to

$$\Delta w_{jk} = -\gamma \frac{\partial E}{\partial w_{jk}}. \tag{5.14}$$

It is not difficult to show that

$$\frac{\partial E}{\partial w_{jk}} = -\frac{1}{T} \left(\langle y_j y_k \rangle^{\text{clamped}} - \langle y_j y_k \rangle^{\text{free}} \right). \tag{5.15}$$

Therefore, each weight is updated by

$$\Delta w_{jk} = \gamma \left(\langle y_j y_k \rangle^{\text{clamped}} - \langle y_j y_k \rangle^{\text{free}} \right). \tag{5.16}$$

6 Self-Organising Networks

In the previous chapters we discussed a number of networks which were trained to perform a mapping $F : \Re^n \to \Re^m$ by presenting the network 'examples' $(\mathbf{x}^p, \mathbf{d}^p)$ with $\mathbf{d}^p = F(\mathbf{x}^p)$ of this mapping. However, problems exist where such training data, consisting of input and desired output pairs are not available, but where the only information is provided by a set of input patterns $\mathbf{x}^p$. In these cases the relevant information has to be found within the (redundant) training samples $\mathbf{x}^p$.

Some examples of such problems are:

- clustering: the input data may be grouped in 'clusters' and the data processing system has to find these inherent clusters in the input data. The output of the system should give the cluster label of the input pattern (discrete output);
- vector quantisation: this problem occurs when a continuous space has to be discretised. The input of the system is the n-dimensional vector $\mathbf{x}$, the output is a discrete representation of the input space. The system has to find optimal discretisation of the input space;
- dimensionality reduction: the input data are grouped in a subspace which has lower dimensionality than the dimensionality of the data. The system has to learn an optimal mapping, such that most of the variance in the input data is preserved in the output data;
- feature extraction: the system has to extract features from the input signal. This often means a dimensionality reduction as described above.

In this chapter we discuss a number of neuro-computational approaches for these kinds of problems. Training is done without the presence of an external teacher. The unsupervised weight adapting algorithms are usually based on some form of global competition between the neurons.

There are very many types of self-organising networks, applicable to a wide area of problems. One of the most basic schemes is *competitive learning* as proposed by Rumelhart and Zipser (Rumelhart & Zipser, 1985). A very similar network but with different emergent properties is the topology-conserving map devised by Kohonen. Other self-organising networks are ART, proposed by Carpenter and Grossberg (Carpenter & Grossberg, 1987a; Grossberg, 1976), and Fukushima's cognitron (Fukushima, 1975, 1988).

6.1 Competitive learning

6.1.1 Clustering

Competitive learning is a learning procedure that divides a set of input patterns in clusters that are inherent to the input data. A competitive learning network is provided only with input vectors $\mathbf{x}$ and thus implements an *unsupervised* learning procedure. We will show its equivalence to a class of 'traditional' clustering algorithms shortly. Another important use of these networks is vector quantisation, as discussed in section 6.1.2.

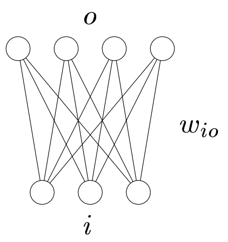


Figure 6.1: A simple competitive learning network. Each of the four outputs o is connected to all inputs i.

An example of a competitive learning network is shown in figure 6.1. All output units o are connected to all input units i with weights w_{io}. When an input pattern $\mathbf{x}$ is presented, only a single output unit of the network (the *winner*) will be activated. In a correctly trained network, all $\mathbf{x}$ in one cluster will have the same winner. For the determination of the winner and the corresponding learning rule, two methods exist.

Winner selection: dot product

For the time being, we assume that both input vectors $\mathbf{x}$ and weight vectors $\mathbf{w}_o$ are normalised to unit length. Each output unit o calculates its activation value y_o according to the dot product of input and weight vector:

$$y_o = \sum_i w_{io} x_i = \mathbf{w}_o{}^T \mathbf{x}. \tag{6.1}$$

In a next pass, output neuron k is selected with maximum activation

$$\forall o \neq k : \qquad y_o \leq y_k. \tag{6.2}$$

Activations are reset such that $y_k = 1$ and $y_{o \neq k} = 0$. This is is the *competitive* aspect of the network, and we refer to the output layer as the *winner-take-all* layer. The winner-take-all layer is usually implemented in software by simply selecting the output neuron with highest activation value. This function can also be performed by a neural network known as MAXNET (Lippmann, 1989). In MAXNET, all neurons o are connected to other units o' with inhibitory links and to itself with an excitatory link:

$$w_{o,o'} = \begin{cases} -\epsilon & \text{if } o \neq o' \\ +1 & \text{otherwise.} \end{cases} \tag{6.3}$$

It can be shown that this network converges to a situation where only the neuron with highest initial activation survives, whereas the activations of all other neurons converge to zero. From now on, we will simply assume a winner k is selected without being concerned which algorithm is used.

Once the winner k has been selected, the weights are updated according to:

$$\mathbf{w}_k(t+1) = \frac{\mathbf{w}_k(t) + \gamma(\mathbf{x}(t) - \mathbf{w}_k(t))}{\|\mathbf{w}_k(t) + \gamma(\mathbf{x}(t) - \mathbf{w}_k(t))\|} \tag{6.4}$$

where the divisor ensures that all weight vectors $\mathbf{w}$ are normalised. Note that only the weights of winner k are updated.

The weight update given in equation (6.4) effectively rotates the weight vector $\mathbf{w}_o$ towards the input vector $\mathbf{x}$. Each time an input $\mathbf{x}$ is presented, the weight vector closest to this input is

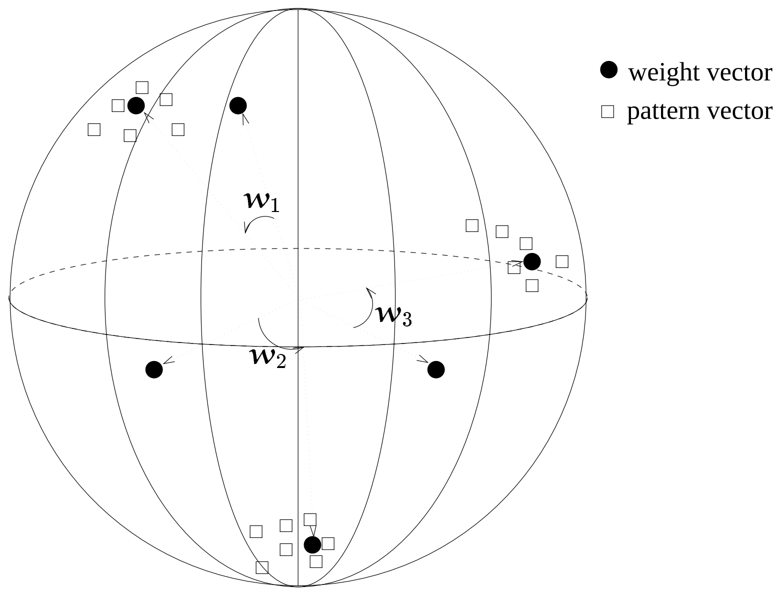


Figure 6.2: Example of clustering in 3D with normalised vectors, which all lie on the unity sphere. The three weight vectors are rotated towards the centres of gravity of the three different input clusters.

selected and is subsequently rotated towards the input. Consequently, weight vectors are rotated towards those areas where many inputs appear: the clusters in the input. This procedure is visualised in figure 6.2.

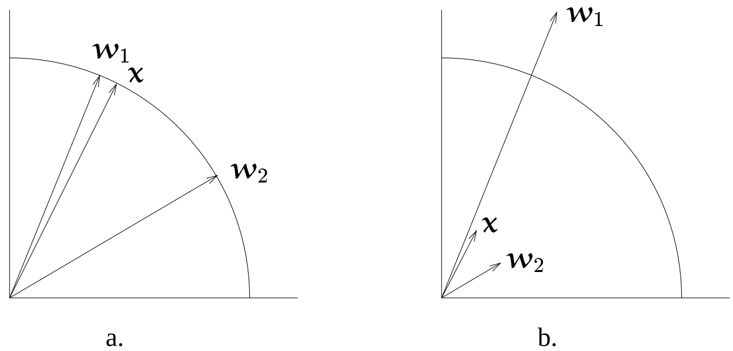


Figure 6.3: Determining the winner in a competitive learning network. a. Three normalised vectors. b. The three vectors having the same directions as in a., but with different lengths. In a., vectors x and w_1 are nearest to each other, and their dot product $x^T w_1 = |x||w_1| \cos\alpha$ is larger than the dot product of x and w_2. In b., however, the pattern and weight vectors are not normalised, and in this case w_2 should be considered the 'winner' when x is applied. However, the dot product $x^T w_1$ is still larger than $x^T w_2$.

Winner selection: Euclidean distance

Previously it was assumed that both inputs x and weight vectors w were normalised. Using the the activation function given in equation (6.1) gives a 'biological plausible' solution. In figure 6.3 it is shown how the algorithm would fail if unnormalised vectors were to be used. Naturally one would like to accommodate the algorithm for unnormalised input data. To this end, the winning neuron k is selected with its weight vector w_k *closest* to the input pattern x, using the

Euclidean distance measure:

$$k : \|\mathbf{w}_k - \mathbf{x}\| \leq \|\mathbf{w}_o - \mathbf{x}\| \quad \forall o. \tag{6.5}$$

It is easily checked that equation (6.5) reduces to (6.1) and (6.2) if all vectors are normalised. The Euclidean distance norm is therefore a more general case of equations (6.1) and (6.2). Instead of *rotating* the weight vector towards the input as performed by equation (6.4), the weight update must be changed to implement a *shift* towards the input:

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \gamma(\mathbf{x}(t) - \mathbf{w}_k(t)). \tag{6.6}$$

Again only the weights of the winner are updated.

A point of attention in these recursive clustering techniques is the initialisation. Especially if the input vectors are drawn from a large or high-dimensional input space, it is not beyond imagination that a randomly initialised weight vector $\mathbf{w}_o$ will never be chosen as the winner and will thus never be moved and never be used. Therefore, it is customary to initialise weight vectors to a set of input patterns $\{\mathbf{x}\}$ drawn from the input set at random. Another more thorough approach that avoids these and other problems in competitive learning is called *leaky learning*. This is implemented by expanding the weight update given in equation (6.6) with

$$\mathbf{w}_l(t+1) = \mathbf{w}_l(t) + \gamma'(\mathbf{x}(t) - \mathbf{w}_l(t)) \qquad \forall l \neq k \tag{6.7}$$

with $\gamma' \ll \gamma$ the *leaky learning rate*. A somewhat similar method is known as *frequency sensitive competitive learning* (Ahalt, Krishnamurthy, Chen, & Melton, 1990). In this algorithm, each neuron records the number of times it is selected winner. The more often it wins, the less sensitive it becomes to competition. Conversely, neurons that consistently fail to win increase their chances of being selected winner.

Cost function

Earlier it was claimed, that a competitive network performs a clustering process on the input data. I.e., input patterns are divided in disjoint clusters such that similarities between input patterns in the same cluster are much bigger than similarities between inputs in different clusters. Similarity is measured by a distance function on the input vectors, as discussed before. A common criterion to measure the quality of a given clustering is the *square error criterion*, given by

$$E = \sum_p \|\mathbf{w}_k - \mathbf{x}^p\|^2 \tag{6.8}$$

where k is the winning neuron when input $\mathbf{x}^p$ is presented. The weights $\mathbf{w}$ are interpreted as *cluster centres*. It is not difficult to show that competitive learning indeed seeks to find a minimum for this square error by following the negative gradient of the error-function:

Theorem 3 *The error function for pattern* $\mathbf{x}^p$

$$E^p = \tfrac{1}{2}\sum_i (w_{ki} - x_i^p)^2, \tag{6.9}$$

where k is the winning unit, is minimised by the weight update rule in eq. (6.6).

Proof *As in eq. (3.12), we calculate the effect of a weight change on the error function. So we have that*

$$\Delta_p w_{io} = -\gamma \frac{\partial E^p}{\partial w_{io}} \tag{6.10}$$

where γ is a constant of proportionality. Now, we have to determine the partial derivative of E^p:

$$\frac{\partial E^p}{\partial w_{io}} = \begin{cases} w_{io} - x_i^p & \textit{if unit } o \textit{ wins} \\ 0 & \textit{otherwise} \end{cases} \tag{6.11}$$

such that

$$\Delta_p w_{io} = -\gamma(w_{io} - x_i^p) = \gamma(x_o^p - w_{io}) \tag{6.12}$$

which is eq. (6.6) written down for one element of $\mathbf{w}_o$.

Therefore, eq. (6.8) is minimised by repeated weight updates using eq. (6.6).

□

An almost identical process of moving cluster centres is used in a large family of conventional clustering algorithms known as *square error clustering methods*, e.g., k-means, FORGY, ISODATA, CLUSTER.

Example

In figure 6.4, 8 clusters of each 6 data points are depicted. A competitive learning network using Euclidean distance to select the winner was initialised with all weight vectors $\mathbf{w}_o = 0$. The network was trained with $\gamma = 0.1$ and a $\gamma' = 0.001$ and the positions of the weights after 500 iterations are shown.

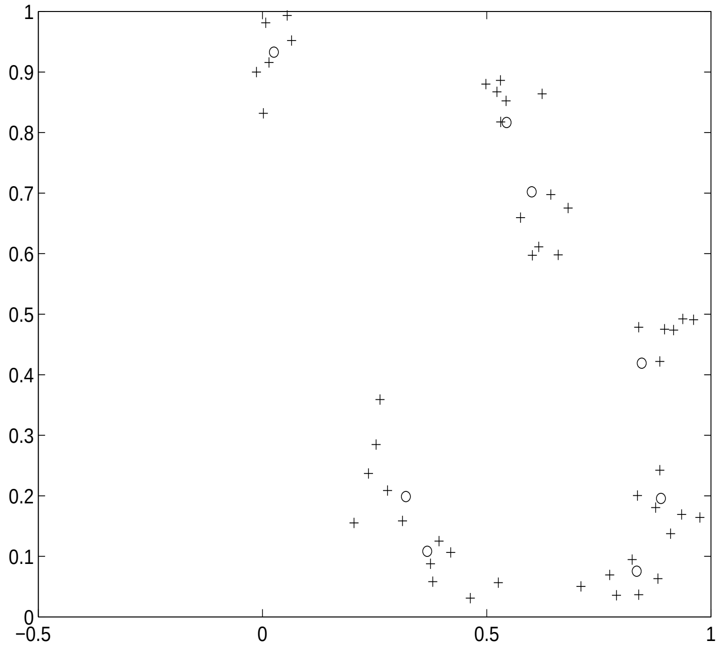


Figure 6.4: Competitive learning for clustering data. The data are given by "+". The positions of the weight vectors after 500 iterations is given by "o".

6.1.2 Vector quantisation

Another important use of competitive learning networks is found in *vector quantisation*. A vector quantisation scheme divides the input space in a number of disjoint subspaces and represents each input vector $\mathbf{x}$ by the label of the subspace it falls into (i.e., index k of the winning neuron). The difference with clustering is that we are not so much interested in finding clusters of similar data, but more in quantising the entire input space. The quantisation performed by the competitive learning network is said to 'track the input probability density function': the density of neurons and thus subspaces is highest in those areas where inputs are most likely to appear, whereas a more coarse quantisation is obtained in those areas where inputs are scarce. An example of tracking the input density is sketched in figure 6.5. Vector quantisation through competitive

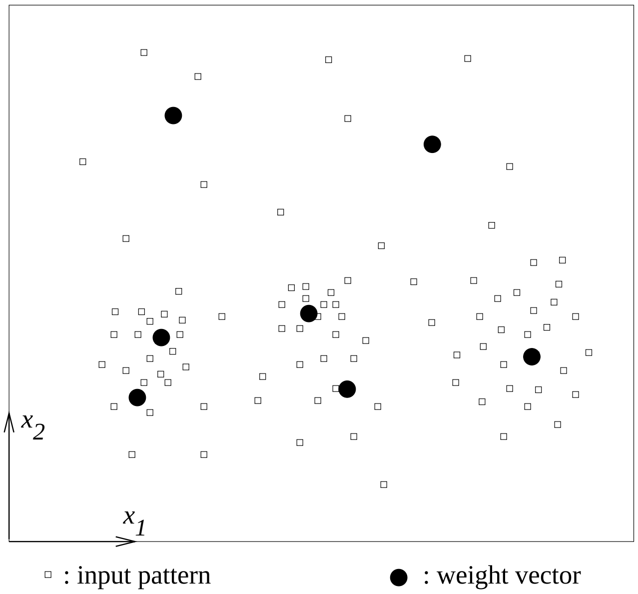


Figure 6.5: This figure visualises the tracking of the input density. The input patterns are drawn from $\Re^2$; the weight vectors also lie in $\Re^2$. In the areas where inputs are scarce, the upper part of the figure, only few (in this case two) neurons are used to discretise the input space. Thus, the upper part of the input space is divided into two large separate regions. The lower part, however, where many more inputs have occurred, five neurons discretise the input space into five smaller subspaces.

learning results in a more *fine-grained* discretisation in those areas of the input space where most input have occurred in the past.

In this way, competitive learning can be used in applications where data has to be compressed such as telecommunication or storage. However, competitive learning has also be used in combination with *supervised* learning methods, and be applied to function approximation problems or classification problems. We will describe two examples: the "counterpropagation" method and the "learning vector quantization".

Counterpropagation

In a large number of applications, networks that perform vector quantisation are combined with another type of network in order to perform function approximation. An example of such a

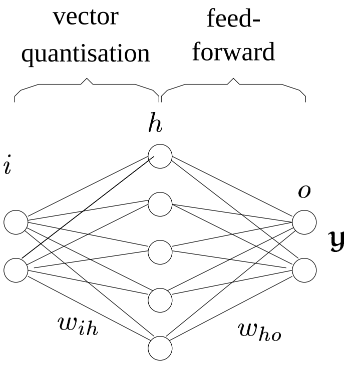


Figure 6.6: A network combining a vector quantisation layer with a 1-layer feed-forward neural network. This network can be used to approximate functions from $\Re^2$ to $\Re^2$, the input space $\Re^2$ is discretised in 5 disjoint subspaces.

network is given in figure 6.6. This network can approximate a function

$$f : \Re^n \rightarrow \Re^m$$

by associating with each neuron o a function value $[w_{1o}, w_{2o}, \ldots, w_{mo}]^T$ which is somehow representative for the function values $f(\mathbf{x})$ of inputs $\mathbf{x}$ represented by o. This way of approximating a function effectively implements a 'look-up table': an input $\mathbf{x}$ is assigned to a table entry k with $\forall o \neq k$: $\|\mathbf{x} - \mathbf{w}_k\| \leq \|\mathbf{x} - \mathbf{w}_o\|$, and the function value $[w_{1k}, w_{2k}, \ldots, w_{mk}]^T$ in this table entry is taken as an approximation of $f(\mathbf{x})$. A well-known example of such a network is the *Counterpropagation* network (Hecht-Nielsen, 1988).

Depending on the application, one can choose to perform the vector quantisation before learning the function approximation, or one can choose to learn the quantisation and the approximation layer simultaneously. As an example of the latter, the network presented in figure 6.6 can be supervisedly trained in the following way:

1. present the network with both input $\mathbf{x}$ and function value $\mathbf{d} = f(\mathbf{x})$;

2. perform the *unsupervised* quantisation step. For each weight vector, calculate the distance from its weight vector to the input pattern and find winner k. Update the weights w_{ih} with equation (6.6);

3. perform the *supervised* approximation step:

$$w_{ko}(t+1) = w_{ko}(t) + \gamma(d_o - w_{ko}(t)). \tag{6.13}$$

   This is simply the δ-rule with $y_o = \sum_h y_h w_{ho} = w_{ko}$ when k is the winning neuron and the desired output is given by $\mathbf{d} = f(\mathbf{x})$.

If we define a function $g(\mathbf{x}, k)$ as :

$$g(\mathbf{x}, k) = \begin{cases} 1 & \text{if } k \text{ is winner} \\ 0 & \text{otherwise} \end{cases} \tag{6.14}$$

it can be shown that this learning procedure converges to

$$w_{ho} = \int_{\Re^n} y_o g(\mathbf{x}, h) \quad d\mathbf{x}. \tag{6.15}$$

I.e., each table entry converges to the mean function value over all inputs in the subspace represented by that table entry. As we have seen before, the quantisation scheme tracks the input probability density function, which results in a better approximation of the function in those areas where input is most likely to appear.

Not all functions are represented accurately by this combination of quantisation and approximation layers. E.g., a simple identity or combinations of sines and cosines are much better approximated by multilayer back-propagation networks if the activation functions are chosen appropriately. However, if we expect our input to be (a subspace of) a high dimensional input space $\Re^n$ and we expect our function f to be discontinuous at numerous points, the combination of quantisation and approximation is not uncommon and probably very efficient. Of course this combination extends itself much further than the presented combination of the presented single layer competitive learning network and the single layer feed-forward network. The latter could be replaced by a reinforcement learning procedure (see chapter 7). The quantisation layer can be replaced by various other quantisation schemes, such as Kohonen networks (see section 6.2) or octree methods (Jansen, Smagt, & Groen, 1994). In fact, various modern statistical function approximation methods (CART, MARS (Breiman, Friedman, Olshen, & Stone, 1984; Friedman, 1991)) are based on this very idea, extended with the possibility to have the approximation layer influence the quantisation layer (e.g., to obtain a better or locally more fine-grained quantisation). Recent research (Rosen, Goodwin, & Vidal, 1992) also investigates in this direction.

Learning Vector Quantisation

It is an unpleasant habit in neural network literature, to also cover *Learning Vector Quantisation* (LVQ) methods in chapters on unsupervised clustering. Granted that these methods also perform a clustering or quantisation task and use similar learning rules, they are trained *supervisedly* and perform *discriminant analysis* rather than unsupervised clustering. These networks attempt to define 'decision boundaries' in the input space, given a large set of exemplary decisions (the training set); each decision could, e.g., be a correct class label.

A rather large number of slightly different LVQ methods is appearing in recent literature. They are all based on the following basic algorithm:

1. with each output neuron o, a class label (or decision of some other kind) y_o is associated;
2. a learning sample consists of input vector $\mathbf{x}^p$ together with its correct class label y_o^p;
3. using distance measures between weight vectors $\mathbf{w}_o$ and input vector $\mathbf{x}^p$, not only the winner k_1 is determined, but also the second best k_2:

$$\|\mathbf{x}^p - \mathbf{w}_{k_1}\| < \|\mathbf{x}^p - \mathbf{w}_{k_2}\| < \|\mathbf{x}^p - \mathbf{w}_i\| \qquad \forall o \neq k_1, k_2;$$

4. the labels $y_{k_1}^p$, $y_{k_2}^p$ are compared with d^p. The weight update rule given in equation (6.6) is used *selectively* based on this comparison.

An example of the last step is given by the LVQ2 algorithm by Kohonen (Kohonen, 1977), using the following strategy:

- if $y_{k_1}^p \neq d^p$ and $d^p = y_{k_2}^p$;
- and $\|\mathbf{x}^p - \mathbf{w}_{k_2}\| - \|\mathbf{x}^p - \mathbf{w}_{k_1}\| < \epsilon$;
- then $\mathbf{w}_{k_2}(t+1) = \mathbf{w}_{k_2} + \gamma(\mathbf{x} - \mathbf{w}_{k_2}(t))$
- and $\mathbf{w}_{k_1}(t+1) = \mathbf{w}_{k_1}(t) - \gamma(\mathbf{x} - \mathbf{w}_{k_1}(t))$

I.e., $\mathbf{w}_{k_2}$ with the correct label is moved *towards* the input vector, while $\mathbf{w}_{k_1}$ with the incorrect label is moved *away* from it.

The new LVQ algorithms that are emerging all use different implementations of these different steps, e.g., how to define class labels y_o, how many 'next-best' winners are to be determined, how to adapt the number of output neurons i and how to selectively use the weight update rule.

6.2 Kohonen network

The Kohonen network (Kohonen, 1982, 1984) can be seen as an extension to the competitive learning network, although this is chronologically incorrect. Also, the Kohonen network has a different set of applications.

In the Kohonen network, the output units in S are *ordered* in some fashion, often in a two-dimensional grid or array, although this is application-dependent. The ordering, which is chosen by the user[1], determines which output neurons are neighbours.

Now, when learning patterns are presented to the network, the weights to the output units are thus adapted such that the order present in the input space $\Re^N$ is preserved in the output, i.e., the neurons in S. This means that learning patterns which are near to each other in the input space (where 'near' is determined by the distance measure used in finding the winning unit)

[1] Of course, variants have been designed which automatically generate the structure of the network (Martinetz & Schulten, 1991; Fritzke, 1991).

must be mapped on output units which are also near to each other, i.e., the same or neighbouring units. Thus, if inputs are uniformly distributed in $\Re^N$ and the order must be preserved, the dimensionality of S must be at least N. The mapping, which represents a discretisation of the input space, is said to be *topology preserving*. However, if the inputs are restricted to a subspace of $\Re^N$, a Kohonen network can be used of lower dimensionality. For example: data on a two-dimensional manifold in a high dimensional input space can be mapped onto a two-dimensional Kohonen network, which can for example be used for visualisation of the data.

Usually, the learning patterns are random samples from $\Re^N$. At time t, a sample $\mathbf{x}(t)$ is generated and presented to the network. Using the same formulas as in section 6.1, the winning unit k is determined. Next, the weights to this winning unit *as well as its neighbours* are adapted using the learning rule

$$\mathbf{w}_o(t+1) = \mathbf{w}_o(t) + \gamma g(o,k)\left(\mathbf{x}(t) - \mathbf{w}_o(t)\right) \quad \forall o \in S. \tag{6.16}$$

Here, $g(o,k)$ is a decreasing function of the grid-distance between units o and k, such that $g(k,k) = 1$. For example, for $g()$ a Gaussian function can be used, such that (in one dimension!) $g(o,k) = \exp\left(-(o-k)^2\right)$ (see figure 6.7). Due to this collective learning scheme, input signals

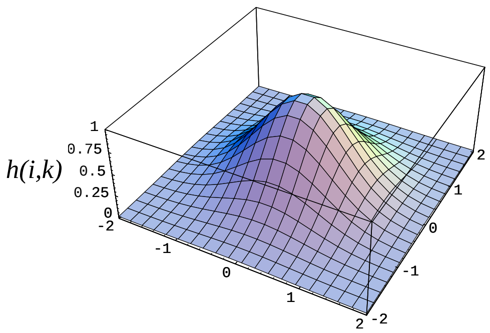


Figure 6.7: Gaussian neuron distance function $g()$. In this case, $g()$ is shown for a two-dimensional grid because it looks nice.

which are near to each other will be mapped on neighbouring neurons. Thus the topology inherently present in the input signals will be preserved in the mapping, such as depicted in figure 6.8.

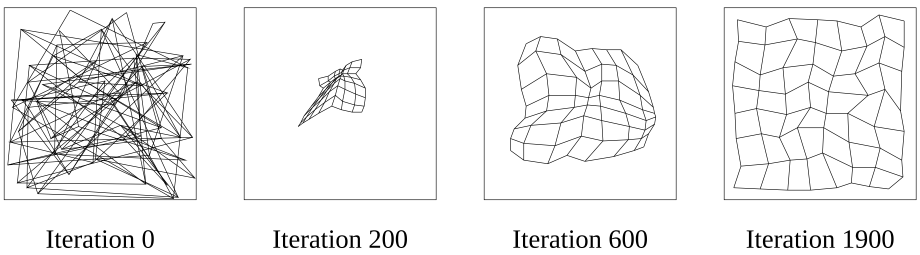


Figure 6.8: A topology-conserving map converging. The weight vectors of a network with two inputs and 8×8 output neurons arranged in a planar grid are shown. A line in each figure connects weight $w_{i,(o_1,o_2)}$ with weights $w_{i,(o_1+1,o_2)}$ and $w_{i,(i_1,i_2+1)}$. The leftmost figure shows the initial weights; the rightmost when the map is almost completely formed.

If the intrinsic dimensionality of S is less than N, the neurons in the network are 'folded' in the input space, such as depicted in figure 6.9.

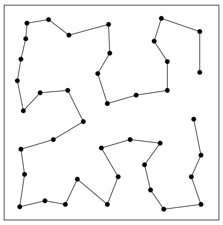


Figure 6.9: The mapping of a two-dimensional input space on a one-dimensional Kohonen network.

The topology-conserving quality of this network has many counterparts in biological brains. The brain is organised in many places so that aspects of the sensory environment are represented in the form of two-dimensional maps. For example, in the visual system, there are several topographic mappings of visual space onto the surface of the visual cortex. There are organised mappings of the body surface onto the cortex in both motor and somatosensory areas, and tonotopic mappings of frequency in the auditory cortex. The use of topographic representations, where some important aspect of a sensory modality is related to the physical locations of the cells on a surface, is so common that it obviously serves an important information processing function.

It does not come as a surprise, therefore, that already many applications have been devised of the Kohonen topology-conserving maps. Kohonen himself has successfully used the network for phoneme-recognition (Kohonen, Makisara, & Saramaki, 1984). Also, the network has been used to merge sensory data from different kinds of sensors, such as auditory and visual, 'looking' at the same scene (Gielen, Krommenhoek, & Gisbergen, 1991). Yet another application is in robotics, such as shown in section 8.1.1.

To explain the plausibility of a similar structure in biological networks, Kohonen remarks that the lateral inhibition between the neurons could be obtained via efferent connections between those neurons. In one dimension, those connection strengths form a 'Mexican hat' (see figure 6.10).

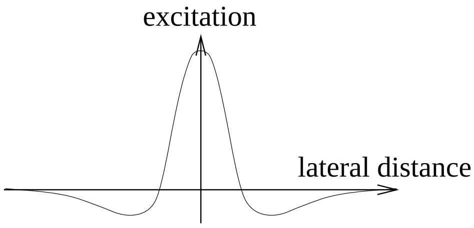


Figure 6.10: Mexican hat. Lateral interaction around the winning neuron as a function of distance: excitation to nearby neurons, inhibition to farther off neurons.

6.3 Principal component networks

6.3.1 Introduction

The networks presented in the previous sections can be seen as (nonlinear) vector transformations which map an input vector to a number of binary output elements or neurons. The weights are adjusted in such a way that they could be considered as prototype vectors (vectorial means) for the input patterns for which the competing neuron wins.

The self-organising transform described in this section rotates the input space in such a way that the values of the output neurons are as uncorrelated as possible and the energy or variances of the patterns is mainly concentrated in a few output neurons. An example is shown

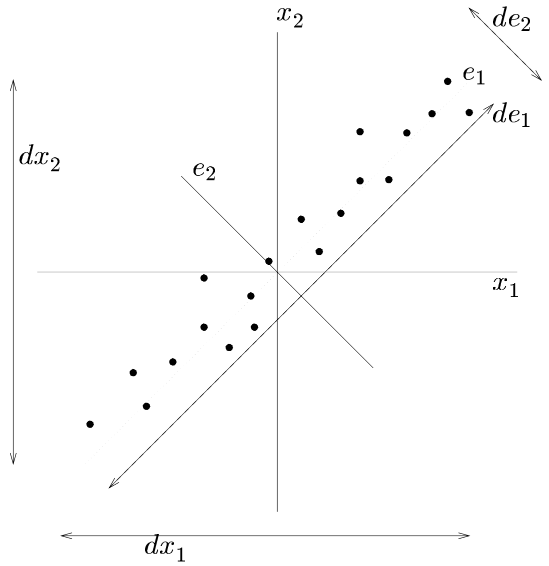


Figure 6.11: Distribution of input samples.

in figure 6.11. The two dimensional samples (x_1, x_2) are plotted in the figure. It can be easily seen that x_1 and x_2 are related, such that if we know x_1 we can make a reasonable prediction of x_2 and vice versa since the points are centered around the line $x_1 = x_2$. If we rotate the axes over $\pi/4$ we get the (e_1, e_2) axis as plotted in the figure. Here the conditional prediction has no use because the points have uncorrelated coordinates. Another property of this rotation is that the variance or energy of the transformed patterns is maximised on a lower dimension. This can be intuitively verified by comparing the spreads (d_{x_1}, d_{x_2}) and (d_{e_1}, d_{e_2}) in the figures. After the rotation, the variance of the samples is large along the e_1 axis and small along the e_2 axis.

This transform is very closely related to the eigenvector transformation known from image processing where the image has to be coded or transformed to a lower dimension and reconstructed again by another transform as well as possible (see section 9.3.2).

The next section describes a learning rule which acts as a Hebbian learning rule, but which scales the vector length to unity. In the subsequent section we will see that a linear neuron with a normalised Hebbian learning rule acts as such a transform, extending the theory in the last section to multi-dimensional outputs.

6.3.2 Normalised Hebbian rule

The model considered here consists of one linear(!) neuron with input weights $\mathbf{w}$. The output $y_o(t)$ of this neuron is given by the usual inner product of its weight $\mathbf{w}$ and the input vector $\mathbf{x}$:

$$y_o(.t) = \mathbf{w}(t)^T \mathbf{x}(t) \tag{6.17}$$

As seen in the previous sections, all models are based on a kind of Hebbian learning. However, the basic Hebbian rule would make the weights grow uninhibitedly if there were correlation in the input patterns. This can be overcome by normalising the weight vector to a fixed length, typically **1**, which leads to the following learning rule

$$\mathbf{w}(t+1) = \frac{\mathbf{w}(t) + \gamma y(t)\mathbf{x}(t)}{L\left(\mathbf{w}(t) + \gamma y(t)\mathbf{x}(t)\right)} \tag{6.18}$$

where $L(\cdot)$ indicates an operator which returns the vector length, and γ is a small learning parameter. Compare this learning rule with the normalised learning rule of competitive learning. There the delta rule was normalised, here the standard Hebb rule is.

Now the operator which computes the vector length, the norm of the vector, can be approximated by a Taylor expansion around $\gamma = 0$:

$$L\left(\mathbf{w}(t) + \gamma y(t)\mathbf{x}(t)\right) = 1 + \gamma \left.\frac{\partial L}{\partial \gamma}\right|_{\gamma=0} + O(\gamma^2). \tag{6.19}$$

When we substitute this expression for the vector length in equation (6.18), it resolves for small γ to[2]

$$\mathbf{w}(t+1) = \left(\mathbf{w}(t) + \gamma y(t)\mathbf{x}(t)\right)\left(1 - \gamma \left.\frac{\partial L}{\partial \gamma}\right|_{\gamma=0} + O(\gamma^2)\right). \tag{6.20}$$

Since $\frac{\delta L}{\delta \gamma}|_{\gamma=0} = y(t)^2$, discarding the higher order terms of γ leads to

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \gamma y(t)\left(\mathbf{x}(t) - y(t)\mathbf{w}(t)\right) \tag{6.21}$$

which is called the 'Oja learning rule' (Oja, 1982). This learning rule thus modifies the weight in the usual Hebbian sense, the first product terms *is* the Hebb rule $y_o(t)\mathbf{x}(t)$, but normalises its weight vector directly by the second product term $-y_o(t)y_o(t)\mathbf{w}(t)$. What exactly does this learning rule do with the weight vector?

6.3.3 Principal component extractor

Remember probability theory? Consider an N-dimensional signal $\mathbf{x}(t)$ with

- mean $\boldsymbol{\mu} = E(\mathbf{x}(t))$;
- correlation matrix $\mathbf{R} = E((\mathbf{x}(t) - \boldsymbol{\mu})(\mathbf{x}(t) - \boldsymbol{\mu})^T)$.

In the following we assume the signal mean to be zero, so $\boldsymbol{\mu} = \mathbf{0}$.

From equation (6.21) we see that the expectation of the weights for the Oja learning rule equals

$$E(\mathbf{w}(t+1)|\mathbf{w}(t)) = \mathbf{w}(t) + \gamma\left(\mathbf{R}\mathbf{w}(t) - \left(\mathbf{w}(t)^T\mathbf{R}\mathbf{w}(t)\right)\mathbf{w}(t)\right) \tag{6.22}$$

which has a continuous counterpart

$$\frac{d}{dt}\mathbf{w}(t) = \mathbf{R}\mathbf{w}(t) - \left(\mathbf{w}(t)^T\mathbf{R}\mathbf{w}(t)\right)\mathbf{w}(t). \tag{6.23}$$

Theorem 1 *Let the eigenvectors $\mathbf{e}_i$ of $\mathbf{R}$ be ordered with descending associated eigenvalues λ_i such that $\lambda_1 > \lambda_2 > \ldots > \lambda_N$. With equation (6.23) the weights $\mathbf{w}(t)$ will converge to $\pm\mathbf{e}_1$.*

Proof 1 *Since the eigenvectors of $\mathbf{R}$ span the N-dimensional space, the weight vector can be decomposed as*

$$\mathbf{w}(t) = \sum_i^N \beta_i(t)\mathbf{e}_i. \tag{6.24}$$

Substituting this in the differential equation and concluding the theorem is left as an exercise.

---

[2]Remembering that $1/(1 + a\gamma) = 1 - a\gamma + O(\gamma^2)$.

6.3.4 More eigenvectors

In the previous section it was shown that a single neuron's weight converges to the eigenvector of the correlation matrix with maximum eigenvalue, i.e., the weight of the neuron is directed in the direction of highest energy or variance of the input patterns. Here we tackle the question of how to find the remaining eigenvectors of the correlation matrix given the first found eigenvector.

Consider the signal $\mathbf{x}$ which can be decomposed into the basis of eigenvectors $\mathbf{e}_i$ of its correlation matrix $\mathbf{R}$,

$$\mathbf{x} = \sum_i^N \alpha_i \mathbf{e}_i \tag{6.25}$$

If we now subtract the component in the direction of $\mathbf{e}_1$, the direction in which the signal has the most energy, from the signal $\mathbf{x}$

$$\tilde{\mathbf{x}} = \mathbf{x} - \alpha_1 \mathbf{e}_1 \tag{6.26}$$

we are sure that when we again decompose $\tilde{\mathbf{x}}$ into the eigenvector basis, the coefficient $\alpha_1 = 0$, simply because we just subtracted it. We call $\tilde{\mathbf{x}}$ the *deflation* of $\mathbf{x}$.

If now a second neuron is taught on this signal $\tilde{\mathbf{x}}$, then its weights will lie in the direction of the remaining eigenvector with the highest eigenvalue. Since the deflation removed the component in the direction of the first eigenvector, the weight will converge to the remaining eigenvector with maximum eigenvalue. In the previous section we ordered the eigenvalues in magnitude, so according to this definition in the limit we will find $\mathbf{e}_2$. We can continue this strategy and find all the N eigenvectors belonging to the signal $\mathbf{x}$.

We can write the deflation in neural network terms if we see that

$$y_o = \mathbf{w}^T \mathbf{x} = \mathbf{e}_1^T \sum_i^N \alpha_i \mathbf{e}_i = \alpha_i \tag{6.27}$$

since

$$\mathbf{w} = \mathbf{e}_1. \tag{6.28}$$

So that the deflated vector $\tilde{\mathbf{x}}$ equals

$$\tilde{\mathbf{x}} = \mathbf{x} - y_o \mathbf{w}. \tag{6.29}$$

The term subtracted from the input vector can be interpreted as a kind of a back-projection or expectation. Compare this to ART described in the next section.

6.4 Adaptive resonance theory

The last unsupervised learning network we discuss differs from the previous networks in that it is recurrent; as with networks in the next chapter, the data is not only fed forward but also back from output to input units.

6.4.1 Background: Adaptive resonance theory

In 1976, Grossberg (Grossberg, 1976) introduced a model for explaining biological phenomena. The model has three crucial properties:

1. a *normalisation* of the total network activity. Biological systems are usually very adaptive to large changes in their environment. For example, the human eye can adapt itself to large variations in light intensities;

2. *contrast enhancement* of input patterns. The awareness of subtle differences in input patterns can mean a lot in terms of survival. Distinguishing a hiding panther from a resting one makes all the difference in the world. The mechanism used here is contrast enhancement;

3. *short-term memory (STM) storage* of the contrast-enhanced pattern. Before the input pattern can be decoded, it must be stored in the short-term memory. The long-term memory (LTM) implements an arousal mechanism (i.e., the classification), whereas the STM is used to cause gradual changes in the LTM.

The system consists of two layers, $F1$ and $F2$, which are connected to each other via the LTM (see figure 6.12). The input pattern is received at $F1$, whereas classification takes place in $F2$. As mentioned before, the input is not directly classified. First a characterisation takes place

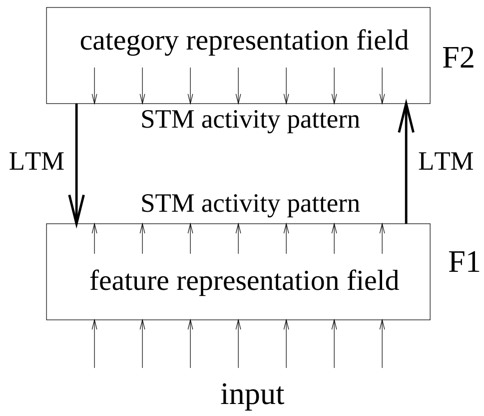


Figure 6.12: The ART architecture.

by means of extracting features, giving rise to activation in the feature representation field. The expectations, residing in the LTM connections, translate the input pattern to a categorisation in the category representation field. The classification is compared to the expectation of the network, which resides in the LTM weights from $F2$ to $F1$. If there is a match, the expectations are strengthened, otherwise the classification is rejected.

6.4.2 ART1: The simplified neural network model

The ART1 simplified model consists of two layers of binary neurons (with values 1 and 0), called $F1$ (the comparison layer) and $F2$ (the recognition layer) (see figure 6.13). Each neuron in $F1$ is connected to all neurons in $F2$ via the continuous-valued forward long term memory (LTM) W^f, and vice versa via the binary-valued backward LTM W^b. The other modules are gain 1 and 2 ($G1$ and $G2$), and a reset module.

Each neuron in the comparison layer receives three inputs: a component of the input pattern, a component of the feedback pattern, and a gain $G1$. A neuron outputs a 1 if and only if at least three of these inputs are high: the 'two-thirds rule.'

The neurons in the recognition layer each compute the inner product of their incoming (continuous-valued) weights and the pattern sent over these connections. The winning neuron then inhibits all the other neurons via lateral inhibition.

Gain 2 is the logical 'or' of all the elements in the input pattern $\mathbf{x}$.

Gain 1 equals gain 2, except when the feedback pattern from $F2$ contains any 1; then it is forced to zero.

Finally, the reset signal is sent to the active neuron in $F2$ if the input vector $\mathbf{x}$ and the output of $F1$ differ by more than some vigilance level.

Operation

The network starts by clamping the input at $F1$. Because the output of $F2$ is zero, $G1$ and $G2$ are both on and the output of $F1$ matches its input.

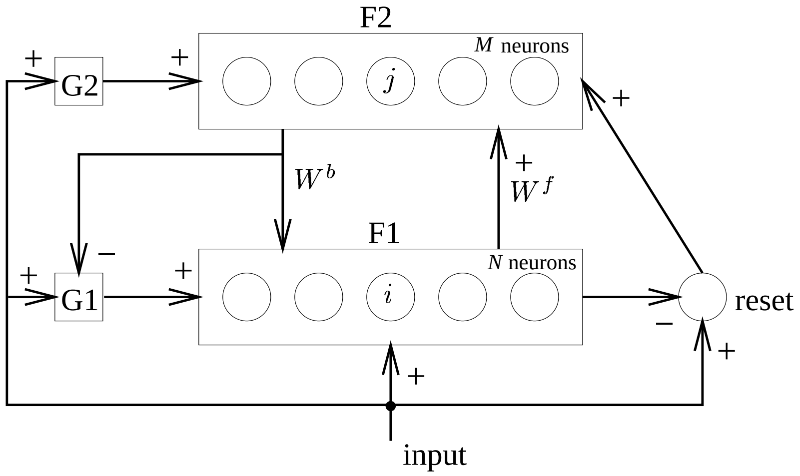


Figure 6.13: The ART1 neural network.

The pattern is sent to $F2$, and in $F2$ one neuron becomes active. This signal is then sent back over the backward LTM, which reproduces a binary pattern at $F1$. Gain 1 is inhibited, and only the neurons in $F1$ which receive a 'one' from both $\mathbf{x}$ and $F2$ remain active.

If there is a substantial mismatch between the two patterns, the reset signal will inhibit the neuron in $F2$ and the process is repeated.

Instead of following Carpenter and Grossberg's description of the system using differential equations, we use the notation employed by Lippmann (Lippmann, 1987):

1. Initialisation:

$$w_{ji}{}^b(0) = 1$$
$$w_{ij}{}^f(0) = \frac{1}{1+N}$$

   where N is the number of neurons in $F1$, M the number of neurons in $F2$, $0 \le i < N$, and $0 \le j < M$. Also, choose the vigilance threshold ρ, $0 \le \rho \le 1$;

2. Apply the new input pattern $\mathbf{x}$;

3. compute the activation values $\mathbf{y}'$ of the neurons in $F2$:

$$y_i{}' = \sum_{j=1}^{N} w_{ij}{}^f(t)\, x_i; \tag{6.30}$$

4. select the winning neuron k $(0 \le k < M)$;

5. vigilance test: if

$$\frac{\mathbf{w}_k{}^b(t)\cdot\mathbf{x}}{\mathbf{x}\cdot\mathbf{x}} > \rho, \tag{6.31}$$

   where $\cdot$ denotes inner product, go to step 7, else go to step 6. Note that $\mathbf{w}_k{}^b \cdot \mathbf{x}$ essentially is the inner product $\mathbf{x}^* \cdot \mathbf{x}$, which will be large if $\mathbf{x}^*$ and $\mathbf{x}$ are near to each other;

6. neuron k is disabled from further activity. Go to step 3;

7. Set for all l, $0 \le l < N$:

$$w_{kl}{}^b(t+1) = w_{kl}{}^b(t)\, x_l,$$
$$w_{lk}{}^f(t+1) = \frac{w_{kl}{}^b(t)\, x_l}{\frac{1}{2} + \sum_{i=1}^{N} w_{ki}{}^b(t)\, x_i};$$

8. re-enable all neurons in $F2$ and go to step 2.

Figure 6.14 shows exemplar behaviour of the network.

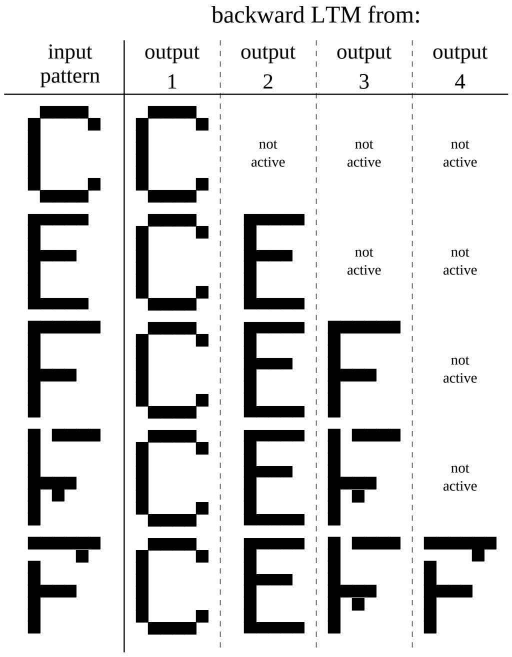


Figure 6.14: An example of the behaviour of the Carpenter Grossberg network for letter patterns. The binary input patterns on the left were applied sequentially. On the right the stored patterns (i.e., the weights of W^b for the first four output units) are shown.

6.4.3 ART1: The original model

In later work, Carpenter and Grossberg (Carpenter & Grossberg, 1987a, 1987b) present several neural network models to incorporate parts of the complete theory. We will only discuss the first model, ART1.

The network incorporates a follow-the-leader clustering algorithm (Hartigan, 1975). This algorithm tries to fit each new input pattern in an existing class. If no matching class can be found, i.e., the distance between the new pattern and all existing classes exceeds some threshold, a new class is created containing the new pattern.

The novelty in this approach is that the network is able to adapt to new incoming patterns, while the previous memory is not corrupted. In most neural networks, such as the back-propagation network, all patterns must be taught sequentially; the teaching of a new pattern might corrupt the weights for all previously learned patterns. By changing the structure of the network rather than the weights, ART1 overcomes this problem.

Normalisation

We will refer to a cell in $F1$ or $F2$ with k.

Each cell k in $F1$ or $F2$ receives an input s_k and respond with an activation level y_k.

In order to introduce normalisation in the model, we set $I = \sum s_k$ and let the relative input intensity $\Theta_k = s_k I^{-1}$.

So we have a model in which the change of the response y_k of an input at a certain cell k

- depends inhibitorily on all other inputs and the sensitivity of the cell, i.e., the surroundings of each cell have a negative influence on the cell $-y_k \sum_{l \neq k} s_l$;

- has an excitatory response as far as the input at the cell is concerned $+Bs_k$;
- has an inhibitory response for normalisation $-y_k s_k$;
- has a decay $-Ay_k$.

Here, A and B are constants. The differential equation for the neurons in $F1$ and $F2$ now is

$$\frac{dy_k}{dt} = -Ay_k + (B - y_k)s_k - y_k \sum_{l \neq k} s_l, \tag{6.32}$$

with $0 \leq y_k(0) \leq B$ because the inhibitory effect of an input can never exceed the excitatory input.

At equilibrium, when $dy_k/dt = 0$, and with $I = \sum s_k$ we have that

$$y_k(A + I) = Bs_k. \tag{6.33}$$

Because of the definition of $\Theta_k = s_k I^{-1}$ we get

$$y_k = \Theta_k \frac{BI}{A + I}. \tag{6.34}$$

Therefore, at equilibrium y_k is proportional to Θ_k, and, since

$$\frac{BI}{A + I} \leq B, \tag{6.35}$$

the total activity $y^{\text{total}} = \sum y_k$ never exceeds B: it is *normalised*.

Contrast enhancement

In order to make $F2$ react better on differences in neuron values in $F1$ (or vice versa), *contrast enhancement* is applied: the contrasts between the neuronal values in a layer are amplified. We can show that eq. (6.32) does not suffice anymore. In order to enhance the contrasts, we chop off all the equal fractions (uniform parts) in $F1$ or $F2$. This can be done by adding an extra inhibitory input proportional to the inputs from the other cells with a factor C:

$$\frac{dy_k}{dt} = -Ay_k + (B - y_k)s_k - (y_k + C) \sum_{l \neq k} s_l. \tag{6.36}$$

At equilibrium, when we set $B = (n - 1)C$ where n is the number of neurons, we have

$$y_k = \frac{nCI}{A + I} \left(\Theta_k - \frac{1}{n} \right). \tag{6.37}$$

Now, when an input in which all the s_k are equal is given, then all the y_k are zero: the effect of C is enhancing differences. If we set $B \leq (n - 1)C$ or $C/(B + C) \geq 1/n$, then more of the input shall be chopped off.

Discussion

The description of ART1 continues by defining the differential equations for the LTM. Instead of following Carpenter and Grossberg's description, we will revert to the simplified model as presented by Lippmann (Lippmann, 1987).

7 Reinforcement learning

In the previous chapters a number of supervised training methods have been described in which the weight adjustments are calculated using a set of 'learning samples', existing of input and desired output values. However, not always such a set of learning examples is available. Often the only information is a scalar evaluation r which indicates how well the neural network is performing. Reinforcement learning involves two subproblems. The first is that the 'reinforcement' signal r is often delayed since it is a result of network outputs in the past. This *temporal credit assignment problem* is solved by learning a 'critic' network which represents a cost function J predicting future reinforcement. The second problem is to find a learning procedure which adapts the weights of the neural network such that a mapping is established which minimizes J. The two problems are discussed in the next paragraphs, respectively. Figure 7.1 shows a reinforcement-learning network interacting with a system.

7.1 The critic

The first problem is how to construct a critic which is able to evaluate system performance. If the objective of the network is to minimize a direct measurable quantity r, performance feedback is straightforward and a critic is not required. On the other hand, how is *current* behavior to be evaluated if the objective concerns *future* system performance. The performance may for instance be measured by the cumulative or future error. Most reinforcement learning methods (such as Barto, Sutton and Anderson (Barto, Sutton, & Anderson, 1983)) use the temporal difference (TD) algorithm (Sutton, 1988) to train the critic.

Suppose the immediate cost of the system at time step k are measured by $r(\mathbf{x}_k, \mathbf{u}_k, k)$, as a function of system states $\mathbf{x}_k$ and control actions (network outputs) $\mathbf{u}_k$. The immediate measure r is often called the *external* reinforcement signal in contrast to the *internal* reinforcement signal in figure 7.1. Define the performance measure $J(\mathbf{x}_k, \mathbf{u}_k, k)$ of the system as a discounted

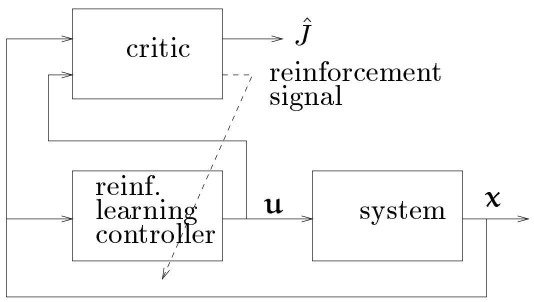


Figure 7.1: Reinforcement learning scheme.

cumulative of future cost. The task of the critic is to predict the performance measure:

$$J(\mathbf{x}_k, \mathbf{u}_k, k) = \sum_{i=k}^{\infty} \gamma^{i-k} r(\mathbf{x}_i, \mathbf{u}_i, i) \tag{7.1}$$

in which $\gamma \in [0, 1]$ is a discount factor (usually ≈ 0.95).

The relation between two successive prediction can easily be derived:

$$J(\mathbf{x}_k, \mathbf{u}_k, k) = r(\mathbf{x}_k, \mathbf{u}_k, k) + \gamma J(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}, k+1). \tag{7.2}$$

If the network is correctly trained, the relation between two successive network outputs $\hat{J}$ should be:

$$\hat{J}(\mathbf{x}_k, \mathbf{u}_k, k) = r(\mathbf{x}_k, \mathbf{u}_k, k) + \gamma \hat{J}(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}, k+1). \tag{7.3}$$

If the network is not correctly trained, the *temporal difference* $\delta(k)$ between two successive predictions is used to adapt the critic network:

$$\delta(k) = \left[r(\mathbf{x}_k, \mathbf{u}_k, k) + \gamma \hat{J}(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}, k+1) \right] - \hat{J}(\mathbf{x}_k, \mathbf{u}_k, k). \tag{7.4}$$

A learning rule for the weights of the critic network $\mathbf{w}_c(k)$, based on minimizing $\delta^2(k)$ can be derived:

$$\Delta \mathbf{w}_c(k) = -\alpha \varepsilon(k) \frac{\partial \hat{J}(\mathbf{x}_k, \mathbf{u}_k, k)}{\partial \mathbf{w}_c(k)} \tag{7.5}$$

in which α is the learning rate.

7.2 The controller network

If the critic is capable of providing an immediate evaluation of performance, the controller network can be adapted such that the optimal relation between system states and control actions is found. Three approaches are distinguished:

1. In case of a finite set of actions U, all actions may virtually be executed. The action which decreases the performance criterion most is selected:

   $$\mathbf{u}_k = \min_{u \in U} \hat{J}(\mathbf{x}_k, \mathbf{u}_k, k) \tag{7.6}$$

   The RL-method with this 'controller' is called Q-learning (Watkins & Dayan, 1992). The method approximates dynamic programming which will be discussed in the next section.

2. If the performance measure $J(\mathbf{x}_k, \mathbf{u}_k, k)$ is accurately predicted, then the gradient with respect to the controller command u_k can be calculated, assuming that the critic network is differentiable. If the measure is to be minimized, the weights of the controller $\mathbf{w}_r$ are adjusted in the direction of the negative gradient:

   $$\Delta \mathbf{w}_r(k) = -\beta \frac{\partial \hat{J}(\mathbf{x}_k, \mathbf{u}_k, k)}{\partial \mathbf{u}(k)} \frac{\partial \mathbf{u}(k)}{\partial \mathbf{w}_r(k)} \tag{7.7}$$

   with β being the learning rate. Werbos (Werbos, 1992) has discussed some of these gradient based algorithms in detail. Sofge and White (Sofge & White, 1992) applied one of the gradient based methods to optimize a manufacturing process.

3. A direct approach to adapt the controller is to use the difference between the predicted and the 'true' performance measure as expressed in equation 7.3. Suppose that the performance measure is to be minimized. Control actions that result in negative differences, i.e. the true performance is better than was expected, then the controller has to be 'rewarded'. On the other hand, in case of a positive difference, then the control action has to be 'penalized'. The idea is to explore the set of possible actions during learning and incorporate the beneficial ones into the controller. Learning in this way is related to trial-and-error learning studied by psychologists in which behavior is selected according to its consequences.

   Generally, the algorithms select *probabilistically* actions from a set of possible actions and update action probabilities on basis of the evaluation feedback. Most of the algorithms are based on a look-up table representation of the mapping from system states to actions (Barto et al., 1983). Each table entry has to learn which control action is best when that entry is accessed. It may be also possible to use a parametric mapping from systems states to action probabilities. Gullapalli (Gullapalli, 1990) adapted the weights of a single layer network. In the next section the approach of Barto et. al. is described.

7.3 Barto's approach: the ASE-ACE combination

Barto, Sutton and Anderson (Barto et al., 1983) have formulated 'reinforcement learning' as a learning strategy which does not need a set of examples provided by a 'teacher.' The system described by Barto explores the space of alternative input-output mappings and uses an evaluative feedback (reinforcement signal) on the consequences of the control signal (network output) on the environment. It has been shown that such reinforcement learning algorithms are implementing an on-line, incremental approximation to the dynamic programming method for optimal control, and are also called 'heuristic' dynamic programming (Werbos, 1990).

The basic building blocks in the Barto network are an *Associative Search Element* (ASE) which uses a stochastic method to determine the correct relation between input and output and an *Adaptive Critic Element* (ACE) which learns to give a correct prediction of future reward or punishment (Figure 7.2). The external reinforcement signal r can be generated by a special sensor (for example a collision sensor of a mobile robot) or be derived from the state vector. For example, in control applications, where the state $\mathbf{s}$ of a system should remain in a certain part A of the control space, reinforcement is given by:

$$r = \begin{cases} 0 & \text{if } \mathbf{s} \in A, \\ -1 & \text{otherwise.} \end{cases} \tag{7.8}$$

7.3.1 Associative search

In its most elementary form the ASE gives a binary output value $y_o(t) \in \{0, 1\}$ as a stochastic function of an input vector. The total input of the ASE is, similar to the neuron presented in chapter 2, the weighted sum of the inputs, with the exception that the bias input in this case is a stochastic variable $\mathcal{N}$ with mean zero normal distribution:

$$s(t) = \sum_{j=1}^{N} w_{Sj} x_j(t) + \mathcal{N}_j. \tag{7.9}$$

The activation function $\mathcal{F}$ is a threshold such that

$$y_o(t) = y(t) = \begin{cases} 1 & \text{if } s(t) > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{7.10}$$

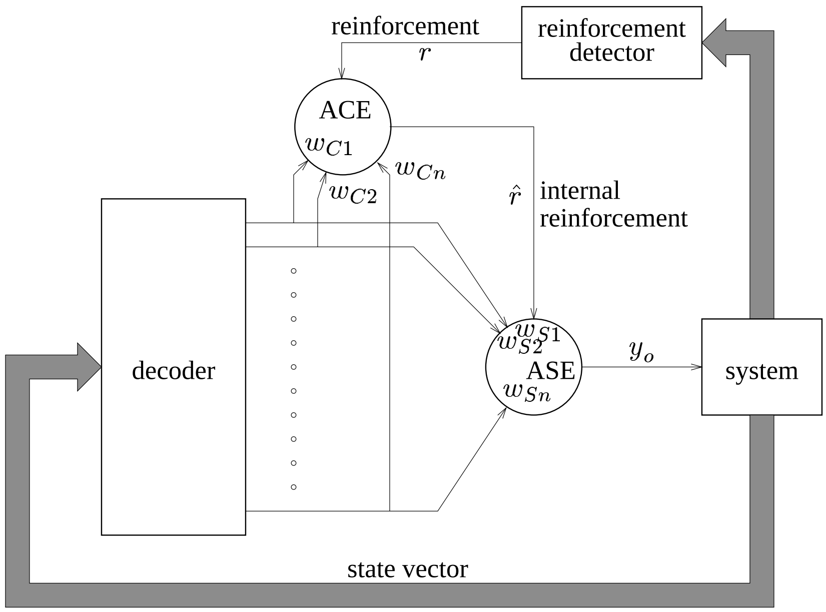


Figure 7.2: Architecture of a reinforcement learning scheme with critic element

For updating the weights, a Hebbian type of learning rule is used. However, the update is weighted with the reinforcement signal $r(t)$ and an 'eligibility' e_j is defined instead of the product $y_o(t)x_j(t)$ of input and output:

$$w_{Sj}(t+1) = w_{Sj}(t) + \alpha r(t) e_j(t) \tag{7.11}$$

where α is a learning factor. The eligibility e_j is given by

$$e_j(t+1) = \delta e_j(t) + (1-\delta) y_o(t) x_j(t) \tag{7.12}$$

with δ the decay rate of the eligibility. The eligibility is a sort of 'memory;' e_j is high if the signals from the input state unit j and the output unit are correlated over some time.

Using $r(t)$ in expression (7.11) has the disadvantage that learning only finds place when there is an external reinforcement signal. Instead of $r(t)$, usually a continuous *internal* reinforcement signal $\hat{r}(t)$ given by the ACE, is used.

Barto and Anandan (Barto & Anandan, 1985) proved convergence for the case of a *single* binary output unit and a set of *linearly independent* patterns $\mathbf{x}^p$. In control applications, the input vector is the (n-dimensional) state vector $\mathbf{s}$ of the system. In order to obtain a linear independent set of patterns $\mathbf{x}^p$, often a 'decoder' is used, which divides the range of each of the input variables s_i in a number of intervals. The aim is to divide the input (state) space in a number of disjunct subspaces (or 'boxes' as called by Barto). The input vector can therefore only be in one subspace at a time. The decoder converts the input vector into a binary valued vector $\mathbf{x}$, with only one element equal to one, indicating which subspace is currently visited. It has been shown (Kröse & Dam, 1992) that instead of a-priori quantisation of the input space, a self-organising quantisation, based on methods described in this chapter, results in a better performance.

7.3.2 Adaptive critic

The Adaptive Critic Element (ACE, or 'evaluation network') is basically the same as described in section 7.1. An error signal is derived from the temporal difference of two successive predictions (in this case denoted by p!) and is used for training the ACE:

$$\hat{r}(t) = r(t) + \gamma p(t) - p(t-1). \tag{7.13}$$

$p(t)$ is implemented as a series of 'weights' w_{Cj} to the ACE such that

$$p(t) = w_{Ck} \tag{7.14}$$

if the system is in state k at time t, denoted by $x_k = 1$. The function is learned by adjusting the w_{Cj}'s according to a 'delta-rule' with an error signal δ given by $\hat{r}(t)$:

$$\Delta w_{Cj}(t) = \beta \hat{r}(t) h_j(t). \tag{7.15}$$

β is the learning parameter and $h_j(t)$ indicates the 'trace' of neuron x_j:

$$h_j(t) = \lambda h_j(t-1) + (1-\lambda) x_j(t-1). \tag{7.16}$$

This trace is a low-pass filter or momentum, through which the credit assigned to state j increases while state j is active and decays exponentially after the activity of j has expired.

If $\hat{r}(t)$ is positive, the action u of the system has resulted in a higher evaluation value, whereas a negative $\hat{r}(t)$ indicates a deterioration of the system. $\hat{r}(t)$ can be considered as an *internal reinforcement* signal.

7.3.3 The cart-pole system

An example of such a system is the cart-pole balancing system (see figure 7.3). Here, a dynamics controller must control the cart in such a way that the pole always stands up straight. The controller applies a 'left' or 'right' force F of fixed magnitude to the cart, which may change direction at discrete time intervals. The model has four state variables:

x the position of the cart on the track,

θ the angle of the pole with the vertical,

$\dot{x}$ the cart velocity, and

$\dot{\theta}$ the angle velocity of the pole.

Furthermore, a set of parameters specify the pole length and mass, cart mass, coefficients of friction between the cart and the track and at the hinge between the pole and the cart, the control force magnitude, and the force due to gravity. The state space is partitioned on the basis of the following quantisation thresholds:

1. x: $\pm 0.8, \pm 2.4$m,
2. θ: $0, \pm 1, \pm 6, \pm 12°$,
3. $\dot{x}$: $\pm 0.5, \pm\infty$ m/s,
4. $\dot{\theta}$: $\pm 50, \pm\infty \circ$/s.

This yields $3 \times 6 \times 3 \times 3 = 162$ regions corresponding to all of the combinations of the intervals. The decoder output is a 162-dimensional vector. A negative reinforcement signal is provided when the state vector gets out of the admissible range: when $x > 2.4$, $x < -2.4$, $\theta > 12°$ or $\theta < -12°$. The system has proved to solve the problem in about 75 learning steps.

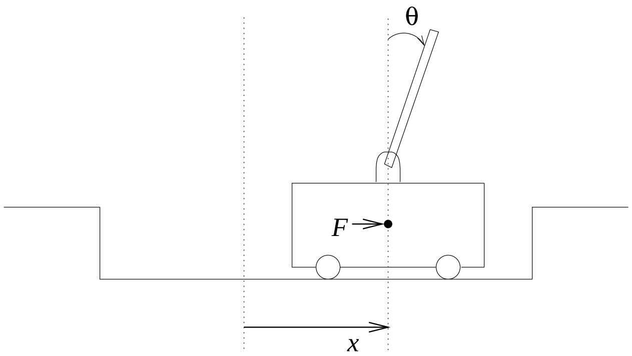


Figure 7.3: The cart-pole system.

7.4 Reinforcement learning versus optimal control

The objective of optimal control is generate control actions in order to optimize a predefined performance measure. One technique to find such a sequence of control actions which define an optimal control policy is Dynamic Programming (DP). The method is based on the principle of optimality, formulated by Bellman (Bellman, 1957): *Whatever the initial system state, if the first control action is contained in an optimal control policy, then the remaining control actions must constitute an optimal control policy for the problem with as initial system state the state remaining from the first control action.* The 'Bellman equations' follow directly from the principle of optimality. Solving the equations backwards in time is called dynamic programming.

Assume that a performance measure $J(\mathbf{x}_k, \mathbf{u}_k, k) \;=\; \sum_{i=k}^{N} \; r(\mathbf{x}_i, \mathbf{u}_i, i)$ with r being the immediate costs, is to be minimized. The minimum costs J_{min} of cost J can be derived by the Bellman equations of DP. The equations for the discrete case are (White & Jordan, 1992):

$$J_{min}(\mathbf{x}_k, \mathbf{u}_k, k) = \min_{u \in U} \left[J_{min}(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}, k+1) + r(\mathbf{x}_k, \mathbf{u}_k, k) \right], \tag{7.17}$$

$$J_{min}(\mathbf{x}_N) = r(\mathbf{x}_N). \tag{7.18}$$

The strategy for finding the optimal control actions is solving equation (7.17) and (7.18) from which $\mathbf{u}_k$ can be derived. This can be achieved backwards, starting at state $\mathbf{x}_N$. The requirements are a bounded N, and a model which is assumed to be an exact representation of the system and the environment. The model has to provide the relation between successive system states resulting from system dynamics, control actions and disturbances. In practice, a solution can be derived only for a small N and simple systems. In order to deal with large or infinity N, the performance measure could be defined as a *discounted* sum of future costs as expressed by equation 7.2.

Reinforcement learning provides a solution for the problem stated above without the use of a model of the system and environment. RL is therefore often called an 'heuristic' dynamic programming technique (Barto, Sutton, & Watkins, 1990),(Sutton, Barto, & Wilson, 1992),(Werbos, 1992). The most directly related RL-technique to DP is Q-learning (Watkins & Dayan, 1992). The basic idea in Q-learning is to estimate a function, Q, of states and actions, where Q is the *minimum* discounted sum of future costs $J_{min}(\mathbf{x}_k, \mathbf{u}_k, k)$ (the name 'Q-learning' comes from Watkins' notation). For convenience, the notation with J is continued here:

$$\hat{J}(\mathbf{x}_k, \mathbf{u}_k, k) \;=\; \gamma J_{min}(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}, k+1) \;+\; r(\mathbf{x}_k, \mathbf{u}_k, k) \tag{7.19}$$

The optimal control rule can be expressed in terms of $\hat{J}$ by noting that an optimal control action for state x_k is any action u_k that minimizes $\hat{J}$ according to equation 7.6.

The estimate of minimum cost $\hat{J}$ is updated at time step $k+1$ according equation 7.5 . The temporal difference $\varepsilon(k)$ between the 'true' and expected performance is again used:

$$\varepsilon(k) = \left[\gamma \min_{u \in U} \hat{J}(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}, k+1) \; + \; r(\mathbf{x}_k, \mathbf{u}_k, k)\right] \; - \; \hat{J}(\mathbf{x}_k, \mathbf{u}_k, k)$$

Watkins has shown that the function converges under some pre-specified conditions to the true optimal Bellmann equation (Watkins & Dayan, 1992): (1) the critic is implemented as a look-up table; (2) the learning parameter α must converge to zero; (3) all actions continue to be tried from all states.

Part III

APPLICATIONS

8 Robot Control

An important area of application of neural networks is in the field of robotics. Usually, these networks are designed to direct a manipulator, which is the most important form of the industrial robot, to grasp objects, based on sensor data. Another applications include the steering and path-planning of autonomous robot vehicles.

In robotics, the major task involves making movements dependent on sensor data. There are four, related, problems to be distinguished (Craig, 1989):

Forward kinematics. Kinematics is the science of motion which treats motion without regard to the forces which cause it. Within this science one studies the position, velocity, acceleration, and all higher order derivatives of the position variables. A very basic problem in the study of mechanical manipulation is that of *forward kinematics*. This is the static geometrical problem of computing the position and orientation of the end-effector ('hand') of the manipulator. Specifically, given a set of joint angles, the forward kinematic problem is to compute the position and orientation of the tool frame relative to the base frame (see figure 8.1).

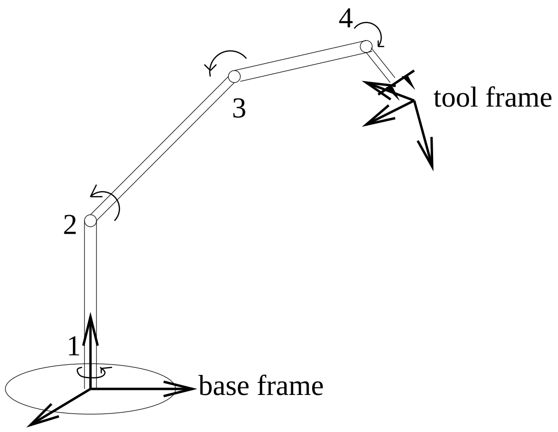


Figure 8.1: An exemplar robot manipulator.

Inverse kinematics. This problem is posed as follows: given the position and orientation of the end-effector of the manipulator, calculate all possible sets of joint angles which could be used to attain this given position and orientation. This is a fundamental problem in the practical use of manipulators.

The inverse kinematic problem is not as simple as the forward one. Because the kinematic equations are nonlinear, their solution is not always easy or even possible in a closed form. Also, the questions of existence of a solution, and of multiple solutions, arise.

Solving this problem is a least requirement for most robot control systems.

Dynamics. Dynamics is a field of study devoted to studying the *forces* required to cause motion. In order to accelerate a manipulator from rest, glide at a constant end-effector velocity, and finally decelerate to a stop, a complex set of torque functions must be applied by the joint actuators. In dynamics not only the geometrical properties (kinematics) are used, but also the physical properties of the robot are taken into account. Take for instance the weight (inertia) of the robotarm, which determines the force required to change the motion of the arm. The dynamics introduces two extra problems to the kinematic problems.

1. The robot arm has a 'memory'. Its responds to a control signal depends also on its history (e.g. previous positions, speed, acceleration).
2. If a robot grabs an object then the dynamics change but the kinematics don't. This is because the weight of the object has to be added to the weight of the arm (that's why robot arms are so heavy, making the relative weight change very small).

Trajectory generation. To move a manipulator from here to there in a smooth, controlled fashion each joint must be moved via a smooth function of time. Exactly how to compute these motion functions is the problem of *trajectory generation*.

In the first section of this chapter we will discuss the problems associated with the positioning of the end-effector (in effect, representing the inverse kinematics in combination with sensory transformation). Section 8.2 discusses a network for controlling the dynamics of a robot arm. Finally, section 8.3 describes neural networks for mobile robot control.

8.1 End-effector positioning

The final goal in robot manipulator control is often the positioning of the hand or end-effector in order to be able to, e.g., pick up an object. With the accurate robot arm that are manufactured, this task is often relatively simple, involving the following steps:

1. determine the target coordinates relative to the base of the robot. Typically, when this position is not always the same, this is done with a number of fixed cameras or other sensors which observe the work scene, from the image frame determine the position of the object in that frame, and perform a pre-determined coordinate transformation;
2. with a precise model of the robot (supplied by the manufacturer), calculate the joint angles to reach the target (i.e., the inverse kinematics). This is a relatively simple problem;
3. move the arm (dynamics control) and close the gripper.

The arm motion in point 3 is discussed in section 8.2. Gripper control is not a trivial matter at all, but we will not focus on that.

Involvement of neural networks. So if these parts are relatively simple to solve with a high accuracy, why involve neural networks? The reason is the applicability of robots. When 'traditional' methods are used to control a robot arm, accurate models of the sensors and manipulators (in some cases with unknown parameters which have to be estimated from the system's behaviour; yet still with accurate models as starting point) are required and the system must be calibrated. Also, systems which suffer from wear-and-tear (and which mechanical systems don't?) need frequent recalibration or parameter determination. Finally, the development of more complex (adaptive!) control methods allows the design and use of more flexible (i.e., less rigid) robot systems, both on the sensory and motory side.

8.1.1 Camera–robot coordination is function approximation

The system we focus on in this section is a work floor observed by a fixed cameras, and a robot arm. The visual system must identify the target as well as determine the visual position of the end-effector.

The target position $\mathbf{x}^{\text{target}}$ together with the visual position of the hand $\mathbf{x}^{\text{hand}}$ are input to the neural controller $\mathcal{N}(\cdot)$. This controller then generates a joint position $\boldsymbol{\theta}$ for the robot:

$$\boldsymbol{\theta} = \mathcal{N}(\mathbf{x}^{\text{target}}, \mathbf{x}^{\text{hand}}). \tag{8.1}$$

We can compare the neurally generated $\boldsymbol{\theta}$ with the optimal $\boldsymbol{\theta}_0$ generated by a fictitious perfect controller $\mathcal{R}(\cdot)$:

$$\boldsymbol{\theta}_0 = \mathcal{R}(\mathbf{x}^{\text{target}}, \mathbf{x}^{\text{hand}}). \tag{8.2}$$

The task of learning is to make the $\mathcal{N}$ generate an output 'close enough' to $\boldsymbol{\theta}_0$.

There are two problems associated with teaching $\mathcal{N}(\cdot)$:

1. generating learning samples which are in accordance with eq. (8.2). This is not trivial, since in useful applications $\mathcal{R}(\cdot)$ is an unknown function. Instead, a form of self-supervised or unsupervised learning is required. Some examples to solve this problem are given below;

2. constructing the mapping $\mathcal{N}(\cdot)$ from the available learning samples. When the (usually randomly drawn) learning samples are available, a neural network uses these samples to represent the whole input space over which the robot is active. This is evidently a form of interpolation, but has the problem that the input space is of a high dimensionality, and the samples are randomly distributed.

We will discuss three fundamentally different approaches to neural networks for robot end-effector positioning. In each of these approaches, a solution will be found for both the learning sample generation and the function representation.

Approach 1: Feed-forward networks

When using a feed-forward system for controlling the manipulator, a self-supervised learning system must be used.

One such a system has been reported by Psaltis, Sideris and Yamamura (Psaltis, Sideris, & Yamamura, 1988). Here, the network, which is constrained to two-dimensional positioning of the robot arm, learns by experimentation. Three methods are proposed:

1. *Indirect learning.*
   In indirect learning, a Cartesian target point $\mathbf{x}$ in world coordinates is generated, e.g., by a two cameras looking at an object. This target point is fed into the network, which generates an angle vector $\boldsymbol{\theta}$. The manipulator moves to position $\boldsymbol{\theta}$, and the cameras determine the new position $\mathbf{x}'$ of the end-effector in world coordinates. This $\mathbf{x}'$ again is input to the network, resulting in $\boldsymbol{\theta}'$. The network is then trained on the error $\epsilon_1 = \boldsymbol{\theta} - \boldsymbol{\theta}'$ (see figure 8.2).

   However, minimisation of ϵ_1 does not guarantee minimisation of the overall error $\epsilon = \mathbf{x} - \mathbf{x}'$. For example, the network often settles at a 'solution' that maps all $\mathbf{x}$'s to a single $\boldsymbol{\theta}$ (i.e., the mapping $\mathbb{I}$).

2. *General learning.*
   The method is basically very much like supervised learning, but here the plant input $\boldsymbol{\theta}$ must be provided by the user. Thus the network can directly minimise $|\boldsymbol{\theta} - \boldsymbol{\theta}'|$. The success of this method depends on the interpolation capabilities of the network. Correct choice of $\boldsymbol{\theta}$ may pose a problem.

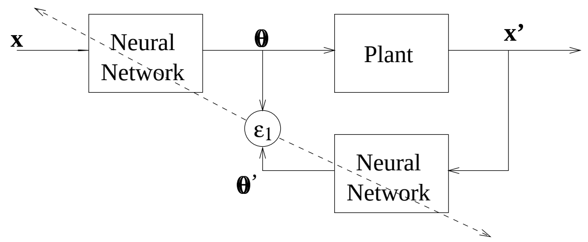


Figure 8.2: Indirect learning system for robotics. In each cycle, the network is used in two different places: first in the forward step, then for feeding back the error.

3. *Specialised learning.*

   Keep in mind that the goal of the training of the network is to minimise the error at the output of the plant: $\epsilon = \mathbf{x} - \mathbf{x}'$. We can also train the network by 'backpropagating' this error trough the plant (compare this with the backpropagation of the error in Chapter 4). This method requires knowledge of the Jacobian matrix of the plant. A Jacobian matrix of a multidimensional function F is a matrix of partial derivatives of F, i.e., the multidimensional form of the derivative. For example, if we have $Y = F(X)$, i.e.,

$$\begin{aligned} y_1 &= f_1(x_1, x_2, \ldots, x_n), \\ y_2 &= f_2(x_1, x_2, \ldots, x_n), \\ &\cdot \\ &\cdot \\ &\cdot \\ y_m &= f_m(x_1, x_2, \ldots, x_n) \end{aligned}$$

   then

$$\begin{aligned} \delta y_1 &= \frac{\partial f_1}{\partial x_1}\delta x_1 + \frac{\partial f_1}{\partial x_2}\delta x_2 + \ldots + \frac{\partial f_1}{\partial x_n}\delta x_n, \\ \delta y_2 &= \frac{\partial f_2}{\partial x_1}\delta x_1 + \frac{\partial f_2}{\partial x_2}\delta x_2 + \ldots + \frac{\partial f_2}{\partial x_n}\delta x_n, \\ &\cdot \\ &\cdot \\ &\cdot \\ \delta y_m &= \frac{\partial f_m}{\partial x_1}\delta x_1 + \frac{\partial f_m}{\partial x_2}\delta x_2 + \ldots + \frac{\partial f_m}{\partial x_n}\delta x_n \end{aligned}$$

   or

$$\delta Y = \frac{\partial F}{\partial X}\delta X. \tag{8.3}$$

   Eq. (8.3) is also written as

$$\delta Y = J(X)\delta X \tag{8.4}$$

   where J is the Jacobian matrix of F. So, the Jacobian matrix can be used to calculate the change in the function when its parameters change.

   Now, in this case we have

$$J_{ij} = \left[\frac{\partial P_i}{\partial \theta_j}\right] \tag{8.5}$$

   where $P_i(\boldsymbol{\theta})$ the ith element of the plant output for input $\boldsymbol{\theta}$. The learning rule applied here regards the plant as an additional and unmodifiable layer in the neural network. The

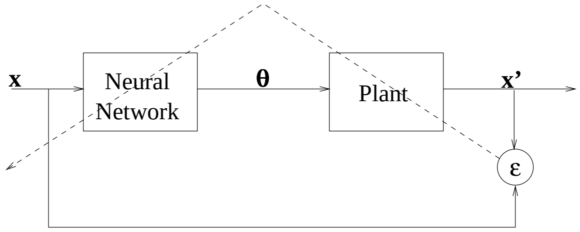


Figure 8.3: The system used for specialised learning.

total error $\epsilon = \mathbf{x} - \mathbf{x}'$ is propagated back through the plant by calculating the δ_j as in eq. (4.14):

$$\delta_j = \mathcal{F}'(s_j) \sum_i \delta_i \frac{\partial P_i(\boldsymbol{\theta})}{\partial \theta_j},$$
$$\delta_i = x_i - x'_i,$$

where i iterates over the outputs of the plant. When the plant is an unknown function, $\frac{\partial P_i(\boldsymbol{\theta})}{\partial \theta_j}$ can be approximated by

$$\frac{\partial P_i(\boldsymbol{\theta})}{\partial \theta_j} \approx \frac{P_i(\boldsymbol{\theta} + h\theta_j \mathbf{e}_j) - P_i(\boldsymbol{\theta})}{h} \tag{8.6}$$

where $\mathbf{e}_j$ is used to change the scalar θ_j into a vector. This approximate derivative can be measured by slightly changing the input to the plant and measuring the changes in the output.

A somewhat similar approach is taken in (Kröse, Korst, & Groen, 1990) and (Smagt & Kröse, 1991). Again a two-layer feed-forward network is trained with back-propagation. However, instead of calculating a desired output vector the input vector which should have invoked the current output vector is reconstructed, and back-propagation is applied to this new input vector and the existing output vector.

The configuration used consists of a monocular manipulator which has to grasp objects. Due to the fact that the camera is situated in the hand of the robot, the task is to move the hand such that the object is in the centre of the image and has some predetermined size (in a later article, a biologically inspired system is proposed (Smagt, Kröse, & Groen, 1992) in which the visual flow-field is used to account for the monocularity of the system, such that the dimensions of the object need not to be known anymore to the system).

One step towards the target consists of the following operations:

1. measure the distance from the current position to the target position in camera domain, $\mathbf{x}$;
2. use this distance, together with the current state $\boldsymbol{\theta}$ of the robot, as input for the neural network. The network then generates a joint displacement vector $\Delta\boldsymbol{\theta}$;
3. send $\Delta\boldsymbol{\theta}$ to the manipulator;
4. again measure the distance from the current position to the target position in camera domain, $\mathbf{x}'$;
5. calculate the move made by the manipulator in visual domain, $\mathbf{x} - {}^{t+1}_{t}R\mathbf{x}'$, where ${}^{t+1}_{t}R$ is the rotation matrix of the second camera image with respect to the first camera image;

6. teach the learning pair $(\mathbf{x} - {}_t^{t+1}R\mathbf{x}', \theta; \Delta\theta)$ to the network.

This system has shown to learn correct behaviour in only tens of iterations, and to be very adaptive to changes in the sensor or manipulator (Smagt & Kröse, 1991; Smagt, Groen, & Kröse, 1993).

By using a feed-forward network, the available learning samples are approximated by a single, smooth function consisting of a summation of sigmoid functions. As mentioned in section 4, a feed-forward network with one layer of sigmoid units is capable of representing practically any function. But how are the optimal weights determined in finite time to obtain this optimal representation? Experiments have shown that, although a reasonable representation can be obtained in a short period of time, an accurate representation of the function that governs the learning samples is often not feasible or extremely difficult (Jansen et al., 1994). The reason for this is the global character of the approximation obtained with a feed-forward network with sigmoid units: every weight in the network has a *global* effect on the final approximation that is obtained.

Building local representations is the obvious way out: every part of the network is responsible for a small subspace of the total input space. Thus accuracy is obtained locally (Keep It Small & Simple). This is typically obtained with a Kohonen network.

Approach 2: Topology conserving maps

Ritter, Martinetz, and Schulten (Ritter, Martinetz, & Schulten, 1989) describe the use of a Kohonen-like network for robot control. We will only describe the kinematics part, since it is the most interesting and straightforward.

The system described by Ritter *et al.* consists of a robot manipulator with three degrees of freedom (orientation of the end-effector is not included) which has to grab objects in 3D-space. The system is observed by two fixed cameras which output their (x, y) coordinates of the object and the end effector (see figure 8.4).

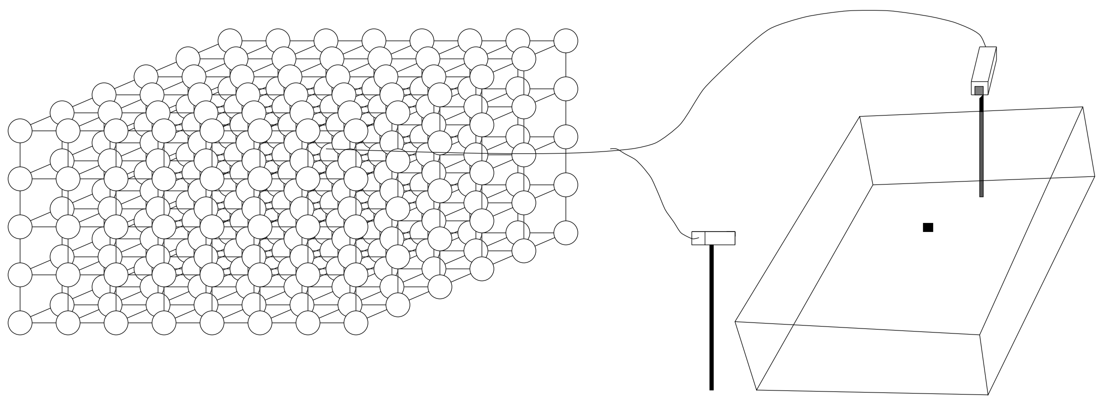


Figure 8.4: A Kohonen network merging the output of two cameras.

Each run consists of two movements. In the gross move, the observed location of the object $\mathbf{x}$ (a four-component vector) is input to the network. As with the Kohonen network, the neuron k with highest activation value is selected as winner, because its weight vector $\mathbf{w}_k$ is nearest to $\mathbf{x}$. The neurons, which are arranged in a 3-dimensional lattice, correspond in a $1-1$ fashion with subregions of the 3D workspace of the robot, i.e., the neuronal lattice is a *discrete representation* of the workspace. With each neuron a vector θ and Jacobian matrix A are associated. During gross move θ_k is fed to the robot which makes its move, resulting in retinal coordinates $\mathbf{x}_g$ of the end-effector. To correct for the discretisation of the working space, an additional move is made which is dependent of the distance between the neuron and the object in space $\mathbf{w}_k - \mathbf{x}$; this small displacement in Cartesian space is translated to an angle change using the Jacobian A_k:

$$\theta^{\text{final}} = \theta_k + A_k(\mathbf{x} - \mathbf{w}_k) \tag{8.7}$$

which is a first-order Taylor expansion of θ^{final}. The final retinal coordinates of the end-effector after this fine move are in $\mathbf{x}_f$.

Learning proceeds as follows: when an improved estimate $(\theta, A)^*$ has been found, the following adaptations are made *for all neurons* j:

$$\begin{aligned} \mathbf{w}_j{}^{\text{new}} &= \mathbf{w}_j{}^{\text{old}} + \gamma(t)\, g_{jk}(t) \left(\mathbf{x} - \mathbf{w}_j{}^{\text{old}}\right), \\ (\theta, A)_j^{\text{new}} &= (\theta, A)_j^{\text{old}} + \gamma'(t)\, g'_{jk}(t) \left((\theta, A)_j^* - (\theta, A)_j^{\text{old}}\right). \end{aligned}$$

If $g_{jk}(t) = g'_{jk}(t) = \delta_{jk}$, this is similar to perceptron learning. Here, as with the Kohonen learning rule, a distance function is used such that $g_{jk}(t)$ and $g'_{jk}(t)$ are Gaussians depending on the distance between neurons j and k with a maximum at $j = k$ (cf. eq. (6.6)).

An improved estimate $(\theta, A)^*$ is obtained as follows.

$$\theta^* = \theta_k + A_k(\mathbf{x} - \mathbf{x}_f), \tag{8.8}$$

$$\begin{aligned} A^* &= A_k + A_k(\mathbf{x} - \mathbf{w}_k - \mathbf{x}_f + \mathbf{x}_g) \times \frac{(\mathbf{x}_f - \mathbf{x}_g)^T}{\|\mathbf{x}_f - \mathbf{x}_g\|^2} \\ &= A_k + (\Delta\theta - A_k \Delta\mathbf{x}) \frac{\Delta\mathbf{x}^T}{\|\Delta\mathbf{x}\|^2}. \end{aligned} \tag{8.9}$$

In eq. (8.8), the final error $\mathbf{x} - \mathbf{x}_f$ in Cartesian space is translated to an error in joint space via multiplication by A_k. This error is then added to θ_k to constitute the improved estimate θ^* (steepest descent minimisation of error).

In eq. (8.9), $\Delta\mathbf{x} = \mathbf{x}_f - \mathbf{x}_g$, i.e., the change in retinal coordinates of the end-effector due to the fine movement, and $\Delta\theta = A_k(\mathbf{x} - \mathbf{w}_k)$, i.e., the related joint angles during fine movement. Thus eq. (8.9) can be recognised as an error-correction rule of the Widrow-Hoff type for Jacobians A.

It appears that after 6,000 iterations the system approaches correct behaviour, and that after 30,000 learning steps no noteworthy deviation is present.

8.2 Robot arm dynamics

While end-effector positioning via sensor–robot coordination is an important problem to solve, the robot itself will not move without dynamic control of its limbs.

Again, accurate control with non-adaptive controllers is possible only when accurate models of the robot are available, and the robot is not too susceptible to wear-and-tear. This requirement has led to the current-day robots that are used in many factories. But the application of neural networks in this field changes these requirements.

One of the first neural networks which succeeded in doing dynamic control of a robot arm was presented by Kawato, Furukawa, and Suzuki (Kawato, Furukawa, & Suzuki, 1987). They describe a neural network which generates motor commands from a desired trajectory in joint angles. Their system does not include the trajectory generation or the transformation of visual coordinates to body coordinates.

The network is extremely simple. In fact, the system is a feed-forward network, but by carefully choosing the basis functions, the network can be restricted to one learning layer such that finding the optimal is a trivial task. In this case, the basis functions are thus chosen that the function that is approximated is a linear combination of those basis functions. This approach is similar to that presented in section 4.5.

Dynamics model. The manipulator used consists of three joints as the manipulator in figure 8.1 without wrist joint. The desired trajectory $\boldsymbol{\theta}_d(t)$, which is generated by another subsystem, is fed into the inverse-dynamics model (figure 8.5). The error between $\boldsymbol{\theta}_d(t)$ and $\boldsymbol{\theta}(t)$ is fed into the neural model.

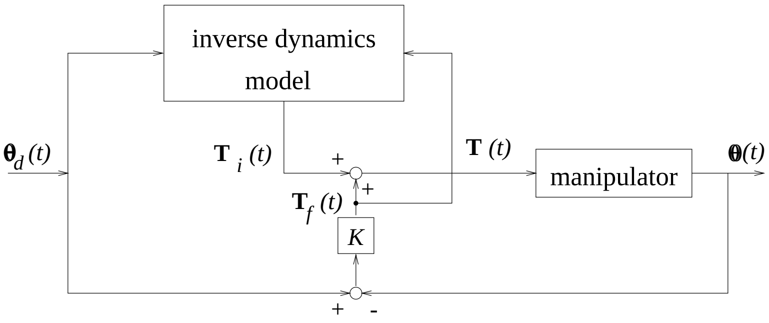


Figure 8.5: The neural model proposed by Kawato *et al.*

The neural model, which is shown in figure 8.6, consists of three perceptrons, each one feeding in one joint of the manipulator. The desired trajectory $\boldsymbol{\theta}_d = (\theta_{d1}, \theta_{d2}, \theta_{d3})$ is fed into 13 nonlinear subsystems. The resulting signals are weighted and summed, such that

$$T_{ik}(t) = \sum_{l=1}^{13} w_{lk} x_{lk}, \qquad (k = 1, 2, 3), \tag{8.10}$$

with

$$\begin{aligned} x_{l1} &= f_l(\theta_{d1}(t), \theta_{d2}(t), \theta_{d3}(t)), \\ x_{l2} = x_{l3} &= g_l(\theta_{d1}(t), \theta_{d2}(t), \theta_{d3}(t)), \end{aligned}$$

and f_l and g_l as in table 8.1.

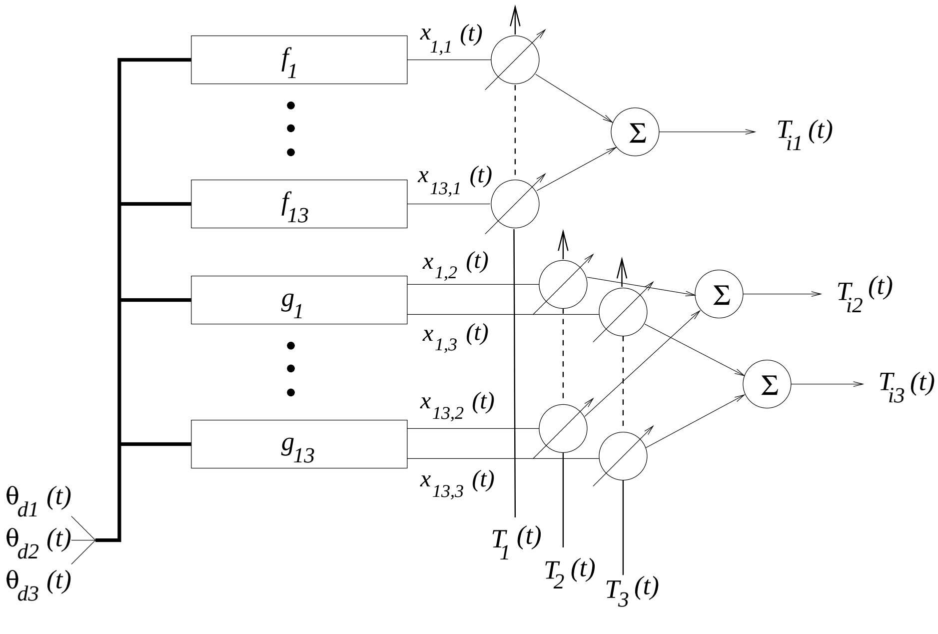


Figure 8.6: The neural network used by Kawato *et al.* There are three neurons, one per joint in the robot arm. Each neuron feeds from thirteen nonlinear subsystems. The upper neuron is connected to the rotary base joint (cf. joint 1 in figure 8.1), the other two neurons to joints 2 and 3.

l	$f_l(\theta_1,\theta_2,\theta_3)$	$g_l(\theta_1,\theta_2,\theta_3)$
1	$\ddot\theta_1$	$\ddot\theta_2$
2	$\ddot\theta_1 \sin^2\theta_2$	$\ddot\theta_3$
3	$\ddot\theta_1 \cos^2\theta_2$	$\ddot\theta_2 \cos\theta_3$
4	$\ddot\theta_1 \sin^2(\theta_2+\theta_3)$	$\ddot\theta_3 \cos\theta_3$
5	$\ddot\theta_1 \cos^2(\theta_2+\theta_3)$	$\dot\theta_1^2 \sin\theta_2\cos\theta_2$
6	$\ddot\theta_1 \sin\theta_2 \sin(\theta_2+\theta_3)$	$\dot\theta_1^2 \sin(\theta_2+\theta_3)\cos(\theta_2+\theta_3)$
7	$\dot\theta_1\dot\theta_2 \sin\theta_2\cos\theta_2$	$\dot\theta_1^2 \sin\theta_2\cos(\theta_2+\theta_3)$
8	$\dot\theta_1\dot\theta_2 \sin(\theta_2+\theta_3)\cos(\theta_2+\theta_3)$	$\dot\theta_1^2 \cos\theta_2\sin(\theta_2+\theta_3)$
9	$\dot\theta_1\dot\theta_2 \sin\theta_2\cos(\theta_2+\theta_3)$	$\dot\theta_2^2 \sin\theta_3$
10	$\dot\theta_1\dot\theta_2 \cos\theta_2\sin(\theta_2+\theta_3)$	$\dot\theta_3^2 \sin\theta_3$
11	$\dot\theta_1\dot\theta_3 \sin(\theta_2+\theta_3)\cos(\theta_2+\theta_3)$	$\dot\theta_2\dot\theta_3 \sin\theta_3$
12	$\dot\theta_1\dot\theta_3 \sin\theta_2\cos(\theta_2+\theta_3)$	$\dot\theta_2$
13	$\dot\theta_1$	$\dot\theta_3$

Table 8.1: Nonlinear transformations used in the Kawato model.

The feedback torque $\mathsf{T}_f(t)$ in figure 8.5 consists of

$$T_{fk}(t) = K_{pk}(\theta_{dk}(t) - \theta_k(t)) + K_{vk}\frac{d\theta_k(t)}{dt}, \qquad (k = 1,2,3),$$
$$K_{vk} = 0 \quad \text{unless } |\theta_k(t) - \theta_{dk}(\text{objective point})| < \varepsilon.$$

The feedback gains K_p and K_v were computed as $(517.2, 746.0, 191.4)^T$ and $(16.2, 37.2, 8.4)^T$.

Next, the weights adapt using the delta rule

$$\gamma\frac{dw_{ik}}{dt} = x_{ik}T_1 = x_{ik}(T_{fk} - T_{ik}), \qquad (k = 1,2,3). \tag{8.11}$$

A desired move pattern is shown in figure 8.7. After 20 minutes of learning the feedback torques are nearly zero such that the system has successfully learned the transformation. Although the applied patterns are very dedicated, training with a repetitive pattern $\sin(\omega_k t)$, with $\omega_1 : \omega_2 : \omega_3 = 1 : \sqrt{2} : \sqrt{3}$ is also successful.

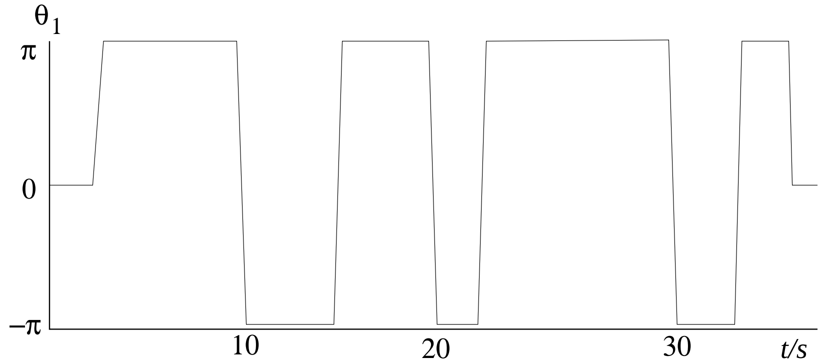


Figure 8.7: The desired joint pattern for joints 1. Joints 2 and 3 have similar time patterns.

The usefulness of neural algorithms is demonstrated by the fact that novel robot architectures, which no longer need a very rigid structure to simplify the controller, are now constructed. For example, several groups (Katayama & Kawato, 1992; Hesselroth, Sarkar, Smagt, & Schulten, 1994) report on work with a pneumatic musculo-skeletal robot arm, with rubber actuators replacing the DC motors. The very complex dynamics and environmental temperature dependency of this arm make the use of non-adaptive algorithms impossible, where neural networks succeed.

8.3 Mobile robots

In the previous sections some applications of neural networks on robot arms were discussed. In this section we focus on mobile robots. Basically, the control of a robot arm and the control of a mobile robot is very similar: the (hierarchical) controller first plans a path, the path is transformed from Cartesian (world) domain to the joint or wheel domain using the inverse kinematics of the system and finally a dynamic controller takes care of the mapping from set-points in this domain to actuator signals. However, in practice the problems with mobile robots occur more with path-planning and navigation than with the dynamics of the system. Two examples will be given.

8.3.1 Model based navigation

Jorgensen (Jorgensen, 1987) describes a neural approach for path-planning. Robot path-planning techniques can be divided into two categories. The first, called *local planning* relies on information available from the current 'viewpoint' of the robot. This planning is important, since it is able to deal with fast changes in the environment. Unfortunately, by itself local data is generally not adequate since occlusion in the line of sight can cause the robot to wander into dead end corridors or choose non-optimal routes of travel. The second situation is called *global path-planning*, in which case the system uses global knowledge from a topographic map previously stored into memory. Although global planning permits optimal paths to be generated, it has its weakness. Missing knowledge or incorrectly selected maps can invalidate a global path to an extent that it becomes useless. A possible third, 'anticipatory' planning combined both strategies: the local information is constantly used to give a best guess what the global environment may contain.

Jorgensen investigates two issues associated with neural network applications in unstructured or changing environments. First, can neural networks be used in conjunction with direct sensor readings to associatively approximate global terrain features not observable from a single robot perspective. Secondly, is a neural network fast enough to be useful in path relaxation planning, where the robot is required to optimise motion and situation sensitive constraints.

For the first problem, the system had to store a number of possible sensor maps of the environment. The robot was positioned in eight positions in each room and 180° sonar scans were made from each position. Based on these data, for each room a map was made. To be able to represent these maps in a neural network, the map was divided into 32×32 grid elements, which could be projected onto the 32×32 nodes neural network. The maps of the different rooms were 'stored' in a Hopfield type of network. In the operational phase, the robot wanders around, and enters an unknown room. It makes one scan with the sonar, which provides a partial representation of the room map (see figure 8.8). This pattern is clamped onto the network, which will regenerate the best fitting pattern. With this information a global path-planner can be used. The results which are presented in the paper are not very encouraging. With a network of 32×32 neurons, the total number of weights is 1024 squared, which costs more than 1 Mbyte of storage if only one byte per weight is used. Also the speed of the recall is low: Jorgensen mentions a recall time of more than two and a half hour on an IBM AT, which is used on board of the robot.

Also the use of a simulated annealing paradigm for path planning is not proving to be an effective approach. The large number of settling trials (> 1000) is far too slow for real time, when the same functions could be better served by the use of a potential field approach or distance transform.

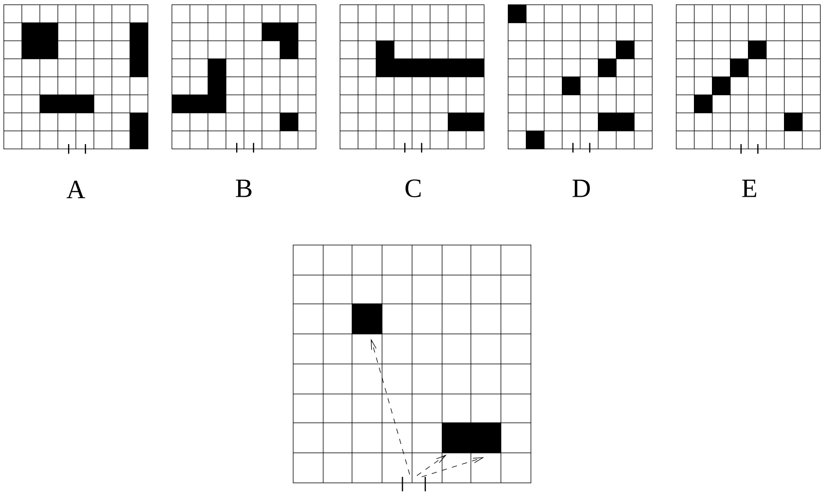


Figure 8.8: Schematic representation of the stored rooms, and the partial information which is available from a single sonar scan.

8.3.2 Sensor based control

Very similar to the sensor based control for the robot arm, as described in the previous sections, a mobile robot can be controlled directly using the sensor data. Such an application has been developed at Carnegy-Mellon by Touretzky and Pomerleau. The goal of their network is to drive a vehicle along a winding road. The network receives two type of sensor inputs from the sensory system. One is a 30×32 (see figure 8.9) pixel image from a camera mounted on the roof of the vehicle, where each pixel corresponds to an input unit of the network. The other input is an 8×32 pixel image from a laser range finder. The activation levels of units in the range finder's retina represent the distance to the corresponding objects.

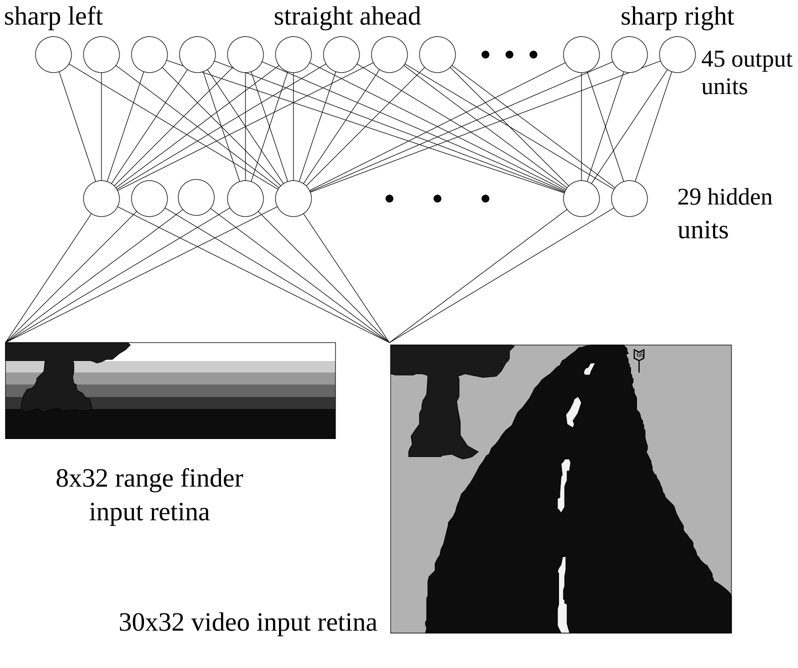


Figure 8.9: The structure of the network for the autonomous land vehicle.

The network was trained by presenting it samples with as inputs a wide variety of road images taken under different viewing angles and lighting conditions. 1,200 Images were presented, 40 times each while the weights were adjusted using the back-propagation principle. The authors claim that once the network is trained, the vehicle can accurately drive (at about 5 km/hour) along '... a path though a wooded area adjoining the Carnegie Mellon campus, under a variety of weather and lighting conditions.' The speed is nearly twice as high as a non-neural algorithm running on the same vehicle.

Although these results show that neural approaches can be possible solutions for the sensor based control problem, there still are serious shortcomings. In simulations in our own laboratory, we found that networks trained with examples which are provided by human operators are not always able to find a correct approximation of the human behaviour. This is the case if the human operator uses other information than the network's input to generate the steering signal. Also the learning of in particular back-propagation networks is dependent on the sequence of samples, and, for all supervised training methods, depends on the distribution of the training samples.

9 Vision

9.1 Introduction

In this chapter we illustrate some applications of neural networks which deal with visual information processing. In the neural literature we find roughly two types of problems: the modelling of biological vision systems and the use of artificial neural networks for machine vision. We will focus on the latter.

The primary goal of machine vision is to obtain information about the environment by processing data from one or multiple two-dimensional arrays of intensity values ('images'), which are projections of this environment on the system. This information can be of different nature:

- recognition: the classification of the input data in one of a number of possible classes;
- geometric information about the environment, which is important for autonomous systems;
- compression of the image for storage and transmission.

Often a distinction is made between low level (or early) vision, intermediate level vision and high level vision. Typical low-level operations include image filtering, isolated feature detection and consistency calculations. At a higher level segmentation can be carried out, as well as the calculation of invariants. The high level vision modules organise and control the flow of information from these modules and combine this information with high level knowledge for analysis.

Computer vision already has a long tradition of research, and many algorithms for image processing and pattern recognition have been developed. There appear to be two computational paradigms that are easily adapted to massive parallelism: local calculations and neighbourhood functions. Calculations that are strictly localised to one area of an image are obviously easy to compute in parallel. Examples are filters and edge detectors in early vision. A cascade of these local calculations can be implemented in a feed-forward network.

The first section describes feed-forward networks for vision. Section 9.3 shows how back-propagation can be used for image compression. In the same section, it is shown that the PCA neuron is ideally suited for image compression. Finally, sections 9.4 and 9.5 describe the cognitron for optical character recognition, and relaxation networks for calculating depth from stereo images.

9.2 Feed-forward types of networks

The early feed-forward networks as the perceptron and the adaline were essentially designed to be be visual pattern classifiers. In principle a multi-layer feed-forward network is able to learn to classify all possible input patterns correctly, but an enormous amount of connections is needed (for the perceptron, Minsky showed that many problems can only be solved if each hidden unit is connected to all inputs). The question is whether such systems can still be regarded as 'vision' systems. No use is made of the spatial relationships in the input patterns and the problem of classifying a set of 'real world' images is the same as the problem of classifying a set of artificial random dot patterns which are, according to Smeulders, no 'images.' For that reason, most successful neural vision applications combine self-organising techniques with a feed-forward architecture, such as for example the neocognitron (Fukushima, 1988), described in section 9.4. The neocognitron performs the mapping from input data to output data by a layered structure in which at each stage increasingly complex features are extracted. The lower layers extract local features such as a line at a particular orientation and the higher layers aim to extract more global features.

Also there is the problem of translation invariance: the system has to classify a pattern correctly independent of the location on the 'retina.' However, a standard feed-forward network considers an input pattern which is translated as a totally 'new' pattern. Several attempts have been described to overcome this problem, one of the more exotic ones by Widrow (Widrow, Winter, & Baxter, 1988) as a layered structure of adalines.

9.3 Self-organising networks for image compression

In image compression one wants to reduce the number of bits required to store or transmit an image. We can either require a perfect reconstruction of the original or we can accept a small deterioration of the image. The former is called a lossless coding and the latter a lossy coding. In this section we will consider lossy coding of images with neural networks.

The basic idea behind compression is that an n-dimensional stochastic vector $\mathbf{n}$, (part of) the image, is transformed into an m-dimensional stochastic vector

$$\mathbf{m} = \mathbf{T}\mathbf{n}. \tag{9.1}$$

After transmission or storage of this vector $\tilde{\mathbf{m}}$, a discrete version of $\mathbf{m}$, we can make a reconstruction of $\mathbf{n}$ by some sort of inverse transform $\tilde{\mathbf{T}}$ so that the reconstructed signal equals

$$\tilde{\mathbf{n}} = \tilde{\mathbf{T}}\tilde{\mathbf{n}}. \tag{9.2}$$

The error of the compression and reconstruction stage together can be given as

$$\epsilon = E\left[\|\mathbf{n} - \tilde{\mathbf{n}}\|\right]. \tag{9.3}$$

There is a trade-off between the dimensionality of $\mathbf{m}$ and the error ϵ. As one decreases the dimensionality of $\mathbf{m}$ the error increases and vice versa, i.e., a better compression leads to a higher deterioration of the image. The basic problem of compression is finding $\mathbf{T}$ and $\tilde{\mathbf{T}}$ such that the information in $\mathbf{m}$ is as compact as possible with acceptable error ϵ. The definition of acceptable depends on the application area.

The cautious reader has already concluded that dimension reduction is in itself not enough to obtain a compression of the data. The main importance is that some aspects of an image are more important for the reconstruction then others. For example, the mean grey level and generally the low frequency components of the image are very important, so we should code these features with high precision. Other, like high frequency components, are much less important so these can be coarse-coded. So, when we reduce the dimension of the data, we are actually trying to concentrate the information of the data in a few numbers (the low frequency components) which can be coded with precision, while throwing the rest away (the high frequency components). In this section we will consider coding an image of 256×256 pixels. It is a bit tedious to transform the whole image directly by the network. This requires a huge amount of neurons. Because the statistical description over parts of the image is supposed to be stationary, we can break the image into 1024 blocks of size 8×8, which is large enough to entail a local statistical description and small enough to be managed. These blocks can then be coded separately, stored or transmitted, where after a reconstruction of the whole image can be made based on these coded 8×8 blocks.

9.3.1 Back-propagation

The process above can be interpreted as a 2-layer neural network. The inputs to the network are the 8×8 patters and the desired outputs are the same 8×8 patterns as presented on the input units. This type of network is called an auto-associator.

After training with a gradient search method, minimising ϵ, the weights between the first and second layer can be seen as the coding matrix $\mathbf{T}$ and between the second and third as the reconstruction matrix $\tilde{\mathbf{T}}$.

If the number of hidden units is smaller then the number of input (output) units, a compression is obtained, in other words we are trying to squeeze the information through a smaller channel namely the hidden layer.

This network has been used for the recognition of human faces by Cottrell (Cottrell, Munro, & Zipser, 1987). He uses an input and output layer of 64×64 units (!) on which he presented the whole face at once. The hidden layer, which consisted of 64 units, was classified with another *network* by means of a delta rule. Is this complex network invariant to translations in the input?

9.3.2 Linear networks

It is known from statistics that the optimal transform from an n-dimensional to an m-dimensional stochastic vector, optimal in the sense that ϵ contains the lowest energy possible, equals the concatenation of the first m eigenvectors of the correlation matrix $\mathbf{R}$ of $\mathbf{N}$. So if $(\boldsymbol{e}_1, \boldsymbol{e}_2, .., \boldsymbol{e}_n)$ are the eigenvectors of $\mathbf{R}$, ordered in decreasing corresponding eigenvalue, the transformation matrix is given as $\mathbf{T} = [\boldsymbol{e}_1 \boldsymbol{e}_2 \ldots \boldsymbol{e}_2]^T$.

In section 6.3.1 a linear neuron with a normalised Hebbian learning rule was able to learn the eigenvectors of the correlation matrix of the input patterns. The definition of the optimal transform given above, suits exactly in the PCA network we have described.

So we end up with a $64 \times m \times 64$ network, where m is the desired number of hidden units which is coupled to the total error ϵ. Since the eigenvalues are ordered in decreasing values, which are the outputs of the hidden units, the hidden units are ordered in importance for the reconstruction.

Sanger (Sanger, 1989) used this implementation for image compression. The test image is shown in figure 9.1. It is 256×256 with 8 bits/pixel.

After training the image four times, thus generating 4×1024 learning patterns of size 8×8, the weights of the network converge into figure 9.2.

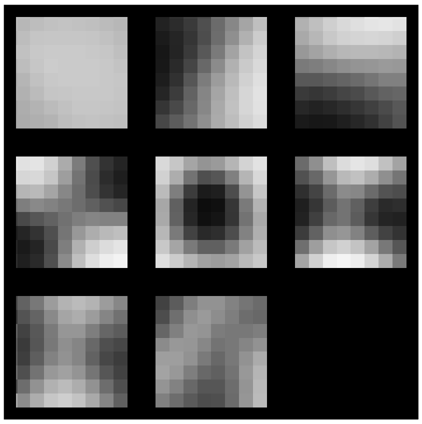
9.3.3 Principal components as features

If parts of the image are very characteristic for the scene, like corners, lines, shades etc., one speaks of features of the image. The extraction of features can make the image understanding task on a higher level much easer. If the image analysis is based on features it is very important that the features are tolerant of noise, distortion etc.

From an image compression viewpoint it would be smart to code these features with as little bits as possible, just because the definition of features was that they occur frequently in the image.

So one can ask oneself if the two described compression methods also extract features from the image. Indeed this is true and can most easily be seen in fig. 9.2. It might not be clear directly, but one can see that the weights are converged to:

Figure 9.1: Input image for the network. The image is divided into 8×8 blocks which are fed to the network.



ordering
of the neurons:

neuron 0	neuron 1	neuron 2
neuron 3	neuron 4	neuron 5
neuron 6	neuron 7	(unused)

Figure 9.2: Weights of the PCA network. The final weights of the network trained on the test image. For each neuron, an 8×8 rectangle is shown, in which the grey level of each of the elements represents the value of the weight. Dark indicates a large weight, light a small weight.

- neuron 0: the mean grey level;
- neuron 1 and neuron 2: the first order gradients of the image;
- neuron 3 … neuron 5: second orders derivates of the image.

The features extracted by the principal component network are the gradients of the image.

9.4 The cognitron and neocognitron

Yet another type of unsupervised learning is found in the *cognitron*, introduced by Fukushima as early as 1975 (Fukushima, 1975). This network, with primary applications in pattern recognition, was improved at a later stage to incorporate scale, rotation, and translation invariance resulting in the *neocognitron* (Fukushima, 1988), which we will not discuss here.

9.4.1 Description of the cells

Central in the cognitron is the type of neuron used. Whereas the Hebb synapse (unit k, say), which is used in the perceptron model, increases an incoming weight (w_{jk}) if and only if the incoming signal (y_j) is high and a control input is high, the synapse introduced by Fukushima increases (the absolute value of) its weight ($|w_{jk}|$) only if it has positive input y_j and a maximum activation value $y_k = \max(y_{k_1}, y_{k_2}, \ldots, y_{k_n})$, where $k_1, k_2, \ldots, k_n$ are all 'neighbours' of k. Note that this learning scheme is competitive and unsupervised, and the same type of neuron has, at a later stage, been used in the competitive learning network (section 6.1) as well as in other unsupervised networks.

Fukushima distinguishes between *excitatory* inputs and *inhibitory* inputs. The output of an excitatory cell u is given by[1]

$$u(k) = \mathcal{F}\left[\frac{1+e}{1+h} - 1\right] = \mathcal{F}\left[\frac{e-h}{1+h}\right], \tag{9.4}$$

where e is the excitatory input from u-cells and h the inhibitory input from v-cells. The activation function is

$$\mathcal{F}(x) = \begin{cases} x & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{9.5}$$

When the inhibitory input is small, i.e., $h \ll 1$, $u(k)$ can be approximated by $u(k) = e - h$, which agrees with the formula for a conventional linear threshold element (with a threshold of zero).

When both the excitatory and inhibitory inputs increase in proportion, i.e.,

$$e = \epsilon x, \qquad h = \eta x \tag{9.6}$$

(ϵ, η constants) and $\epsilon > \eta$, then eq. (9.4) can be transformed into

$$u(i) = \frac{(\epsilon - \eta)x}{1 + \eta x} = \frac{\epsilon - \eta}{2\eta}\Big(1 + \tanh(\tfrac{1}{2}\log \eta x)\Big) \tag{9.7}$$

i.e., a squashing function as in figure 2.2.

9.4.2 Structure of the cognitron

The basic structure of the cognitron is depicted in figure 9.3.

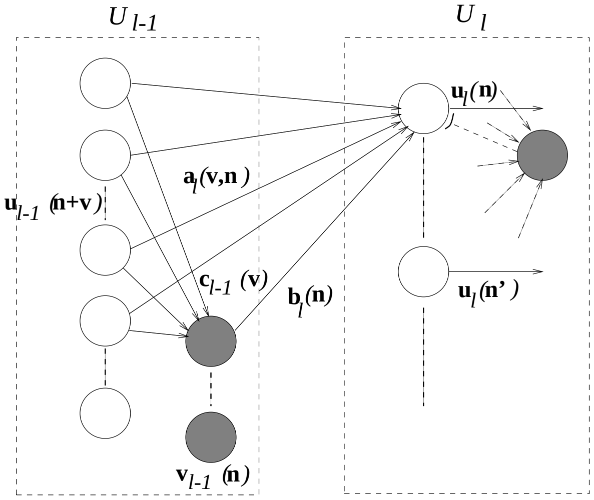


Figure 9.3: The basic structure of the cognitron.

The cognitron has a multi-layered structure. The l-th layer U_l consists of excitatory neurons $u_l(\mathbf{n})$ and inhibitory neurons $v_l(\mathbf{n})$, where $\mathbf{n} = (n_x, n_y)$ is a two-dimensional location of the cell.

[1]Here our notational system fails. We adhere to Fukushima's symbols.

A cell $u_l(\mathbf{n})$ receives inputs via modifiable connections $a_l(\boldsymbol{\nu}, \mathbf{n})$ from neurons $u_{l-1}(\mathbf{n}+\boldsymbol{\nu})$ and connections $b_l(\mathbf{n})$ from neurons $v_{l-1}(\mathbf{n})$, where $\boldsymbol{\nu}$ is in the connectable area (cf. area of attention) of the neuron. Furthermore, an inhibitory cell $v_{l-1}(\mathbf{n})$ receives inputs via fixed excitatory connections $c_{l-1}(\boldsymbol{\nu})$ from the neighbouring cells $u_{l-1}(\mathbf{n}+\boldsymbol{\nu})$, and yields an output equal to its weighted input:

$$v_{l-1}(\mathbf{n}) = \sum_{\boldsymbol{\nu}} c_{l-1}(\boldsymbol{\nu})\, u_{l-1}(\mathbf{n}+\boldsymbol{\nu}). \tag{9.8}$$

where $\sum_{\boldsymbol{\nu}} c_{l-1}(\boldsymbol{\nu}) = 1$ and are fixed.

It can be shown that the growing of the synapses (i.e., modification of the a and b weights) ensures that, if an excitatory neuron has a relatively large response, the excitatory synapses grow faster than the inhibitory synapses, and vice versa.

Receptive region

For each cell in the cascaded layers described above a connectable area must be established. A connection scheme as in figure 9.4 is used: a neuron in layer U_l connects to a small region in layer U_{l-1}.

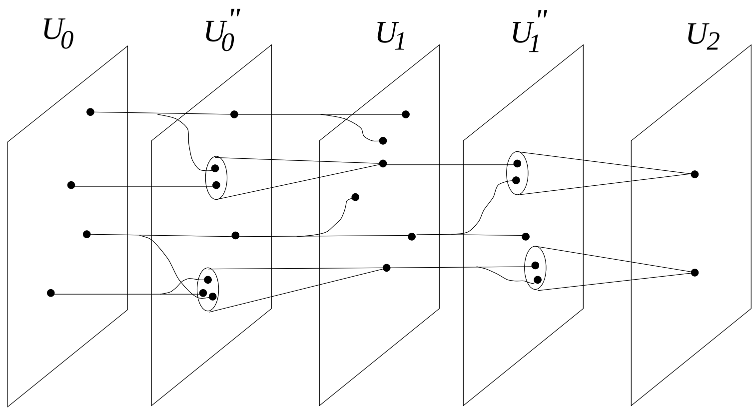


Figure 9.4: Cognitron receptive regions.

If the connection region of a neuron is constant in all layers, a too large number of layers is needed to cover the whole input layer. On the other hand, increasing the region in later layers results in so much overlap that the output neurons have near identical connectable areas and thus all react similarly. This again can be prevented by increasing the size of the vicinity area in which neurons compete, but then only one neuron in the output layer will react to some input stimulus. This is in contradiction with the behaviour of biological brains.

A solution is to distribute the connections probabilistically such that connections with a large deviation are less numerous.

9.4.3 Simulation results

In order to illustrate the working of the network, a simulation has been run with a four-layered network with 16×16 neurons in each layer. The network is trained with four learning patterns, consisting of a vertical, a horizontal, and two diagonal lines. Figure 9.5 shows the activation levels in the layers in the first two learning iterations.

After 20 learning iterations, the learning is halted and the activation values of the neurons in layer 4 are fed back to the input neurons; also, the maximum output neuron alone is fed back, and thus the input pattern is 'recognised' (see figure 9.6).

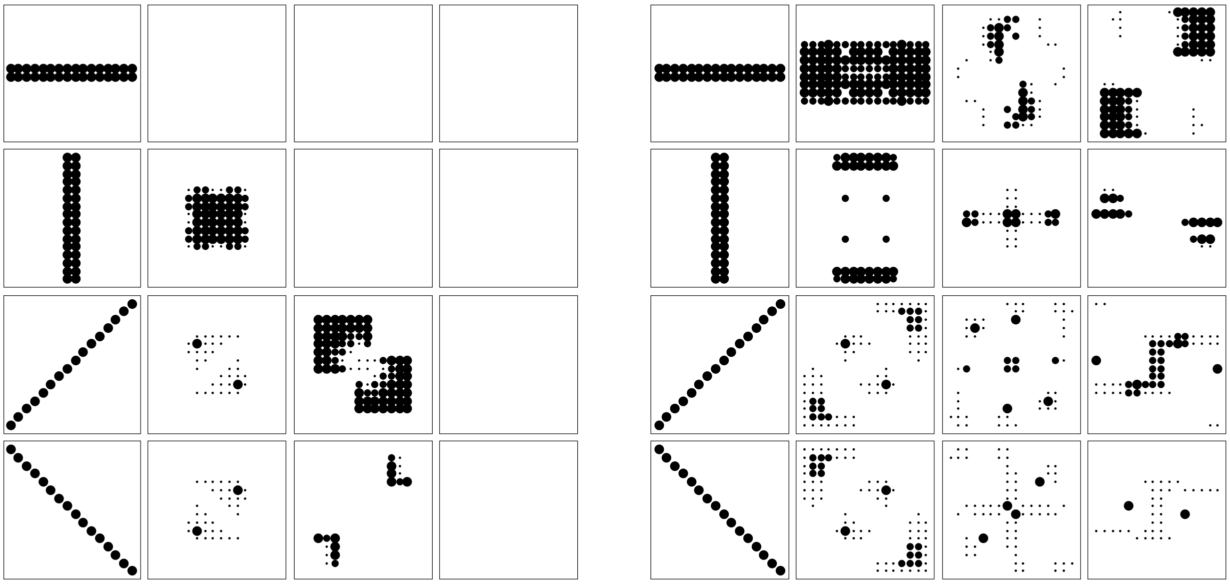


Figure 9.5: Two learning iterations in the cognitron.
Four learning patterns (one in each row) are shown in iteration 1 (a.) and 2 (b.). Each column in a. and b. shows one layer in the network. The activation level of each neuron is shown by a circle. A large circle means a high activation. In the first iteration (a.), a structure is already developing in the second layer of the network. In the second iteration, the second layer can distinguish between the four patterns.

9.5 Relaxation types of networks

As demonstrated by the Hopfield network, a relaxation process in a connectionist network can provide a powerful mechanism for solving some difficult optimisation problems. Many vision problems can be considered as optimisation problems, and are potential candidates for an implementation in a Hopfield-like network. A few examples that are found in the literature will be mentioned here.

9.5.1 Depth from stereo

By observing a scene with two cameras one can retrieve depth information out of the images by finding the pairs of pixels in the images that belong to the same point of the scene. The calculation of the depth is relatively easy; finding the correspondences is the main problem. One solution is to find features such as corners and edges and match those, reducing the computational complexity of the matching. Marr (Marr, 1982) showed that the correspondence problem can be solved correctly when taking into account the physical constraints underlying the process. Three matching criteria were defined:

- Compatibility: Two descriptive elements can only match if they arise from the same physical marking (corners can only match with corners, 'blobs' with 'blobs,' etc.);
- Uniqueness: Almost always a descriptive element from the left image corresponds to exactly one element in the right image and vice versa;
- Continuity: The disparity of the matches varies smoothly almost everywhere over the image.

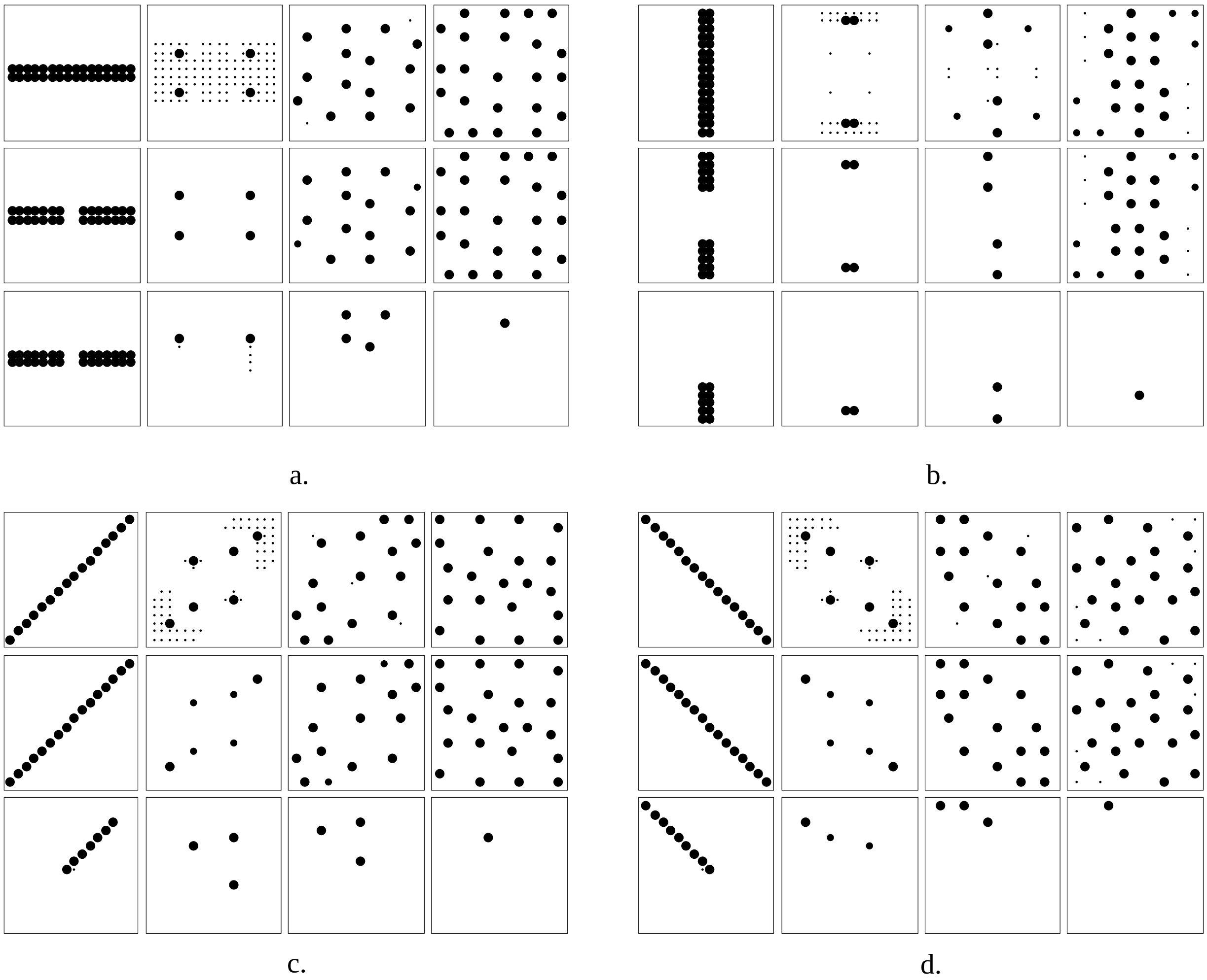


Figure 9.6: Feeding back activation values in the cognitron.

The four learning patterns are now successively applied to the network (row 1 of figures a–d). Next, the activation values of the neurons in layer 4 are fed back to the input (row 2 of figures a–d). Finally, all the neurons except the most active in layer 4 are set to 0, and the resulting activation values are again fed back (row 3 of figures a–d). After as little as 20 iterations, the network has shown to be rather robust.

Marr's 'cooperative' algorithm (also a 'non-cooperative' or local algorithm has been described (Marr, 1982)) is able to calculate the disparity map from which the depth can be reconstructed. This algorithm is some kind of neural network, consisting of neurons $N(x, y; d)$, where neuron $N(x, y; d)$ represents the hypothesis that pixel (x, y) in the left image corresponds with pixel $(x + d, y)$ in the right image. The update function is

$$N^{t+1}(x, y; d) = \sigma\left(\sum_{\substack{x',y';d' \in \\ S(x,y;d)}} N^t(x', y'; d') - \epsilon \sum_{\substack{x',y';d' \in \\ O(x,y;d)}} N^t(x', y'; d') + N^0(x, y; d)\right). \tag{9.9}$$

Here, ϵ is an inhibition constant, σ is a threshold function, $S(x, y; d)$ is the local excitatory neighbourhood, and $O(x, y; d)$ is the local inhibitory neighbourhood, which are chosen as follows:

$$S(x, y; d) = \{ (r, s, t) \mid (r = x \vee r = x - d) \wedge s = y \}, \tag{9.10}$$

$$O(x, y; d) = \{ (r, s, t) \mid d = t \wedge \| (r, s) - (x, y) \| \leq w \}. \tag{9.11}$$

The network is loaded with the cross correlation of the images at first: $N^0(x, y; d) = I_l(x, y)I_r(x + d, y)$, where I_l and I_r are the intensity matrices of the left and right image respectively. This network state represents all possible matches of pixels. Then the set of possible matches is reduced by recursive application of the update function until the state of the network is stable.

The algorithm converges in about ten iterations. Then the disparity of a pixel (x, y) is displayed by the firing neuron in the set $\{N(r, s; d) \mid r = x, s = y\}$. In each of these sets there should be exactly one neuron firing, but if the algorithm could not compute the exact disparity, for instance at hidden contours, there may be zero or more than one neurons firing.

9.5.2 Image restoration and image segmentation

The restoration of degraded images is a branch of digital picture processing closely related to image segmentation and boundary finding. An analysis of the major applications and procedures may be found in (Rosenfeld & Kak, 1982). An algorithm which is based on the minimisation of an energy function and can very well be parallelised is given by Geman and Geman (Geman & Geman, 1984). Their approach is based on stochastic modelling, in which image samples are considered to be generated by a random process that changes its statistical properties from region to region. The random process that that generates the image samples is a two-dimensional analogue of a Markov process, called a Markov random field. Image segmentation is then considered as a statistical estimation problem in which the system calculates the optimal estimate of the region boundaries for the input image. Simultaneously estimation of the region properties and boundary properties has to be performed, resulting in a set of nonlinear estimation equations that define the optimal estimate of the regions. The system must find the maximum a posteriori probability estimate of the image segmentation. Geman and Geman showed that the problem can be recast into the minimisation of an energy function, which, in turn, can be solved approximately by optimisation techniques such as simulated annealing. The interesting point is that simulated annealing can be implemented using a network with local connections, in which the network iterates into a global solution using these local operations.

9.5.3 Silicon retina

Mead and his co-workers (Mead, 1989) have developed an analogue VLSI vision preprocessing chip modelled after the retina. The design not only replicates many of the important functions of the first stages of retinal processing, but it does so by replicating in a detailed way both the structure and dynamics of the constituent biological units. The logarithmic compression from photon input to output signal is accomplished by analogue circuits, while similarly space and time averaging and temporal differentiation are accomplished by analogue processes and a resistive network (see section 11.2.1).

Part IV

IMPLEMENTATIONS

Implementation of neural networks can be divided into three categories:

- software simulation;
- (hardware) emulation[2];
- hardware implementation.

The distinction between the former two categories is not clear-cut. We will use the term *simulation* to describe software packages which can run on a variety of host machines (e.g., PYGMALION, the Rochester Connectionist Simulator, NeuralWare, Nestor, etc.). Implementation of neural networks on general-purpose multi-processor machines such as the Connection Machine, the Warp, transputers, etc., will be referred to as *emulation*. *Hardware implementation* will be reserved for neuro-chips and the like which are specifically designed to run neural networks.

To evaluate and provide a taxonomy of the neural network simulators and emulators discussed, we will use the descriptors of table 9.1 (cf. (DARPA, 1988)).

1. **Equation type**: many networks are defined by the type of equation describing their operation. For example, Grossberg's ART (cf. section 6.4) is described by the differential equation

$$\frac{dx_k}{dt} = -Ax_k + (B - x_k)I_k - x_k \sum_{j \neq k} I_j, \tag{9.12}$$

   in which $-Ax_k$ is a decay term, $+BI_k$ is an external input, $-x_k I_k$ is a normalisation term, and $-x_k \sum_{j \neq k} I_j$ is a neighbour shut-off term for competition. Although differential equations are very powerful, they require a high degree of flexibility in the software and hardware and are thus difficult to implement on special-purpose machines. Other types of equations are, e.g., difference equations as used in the description of Kohonen's topological maps (see section 6.2), and optimisation equations as used in back-propagation networks.
2. **Connection topology**: the design of most general purpose computers includes random access memory (RAM) such that each memory position can be accessed with uniform speed. Such designs always present a trade-off between size of memory and speed of access. The topology of neural networks can be matched in a hardware design with fast local interconnections instead of global access. Most networks are more or less local in their interconnections, and a global RAM is unnecessary.
3. **Processing schema**: although most artificial neural networks use a synchronous update, i.e., the output of the network depends on the previous state of the network, asynchronous update, in which components or blocks of components can be updated one by one, can be implemented much more efficiently. Also, continuous update is a possibility encountered in some implementations.
4. **Synaptic transmission mode**: most artificial neural networks have a transmission mode based on the neuronal activation values multiplied by synaptic weights. In these models, the propagation time from one neuron to another is neglected. On the other hand, biological neurons output a series of pulses in which the frequency determines the neuron output, such that propagation times are an essential part of the model. Currently, models arise which make use of temporal synaptic transmission (Murray, 1989; Tomlinson & Walker, 1990).

Table 9.1: A possible taxonomy.

The following chapters describe general-purpose hardware which can be used for neural network applications, and neuro-chips and other dedicated hardware.

[2] The term *emulation* (see, e.g., (Mallach, 1975) for a good introduction) in computer design means running one computer to execute instructions specific to another computer. It is often used to provide the user with a machine which is seemingly compatible with earlier models.

10 General Purpose Hardware

Parallel computers (Almasi & Gottlieb, 1989) can be divided into several categories. One important aspect is the *granularity* of the parallelism. Broadly speaking, the granularity ranges from *coarse-grain* parallelism, typically up to ten processors, to *fine-grain* parallelism, up to thousands or millions of processors.

Both fine-grain and coarse-grain parallelism is in use for emulation of neural networks. The former model, in which one or more processors can be used for each neuron, corresponds with table 9.1's type 2, whereas the second corresponds with type 1. We will discuss one model of both types of architectures: the (extremely) fine-grain Connection Machine and coarse-grain Systolic arrays, viz. the Warp computer. A more complete discussion should also include transputers which are very popular nowadays due to their very high performance/price ratio (Group, 1987; Board, 1989; Eckmiller, Hartmann, & Hauske, 1990). In this case, descriptor 1 of table 9.1 is most applicable.

Besides the granularity, the computers can be categorised by their operation. The most widely used categorisation is by Flynn (Flynn, 1972) (see table 10.1). It distinguishes two types of parallel computers: SIMD (Single Instruction, Multiple Data) and MIMD (Multiple Instruction, Multiple Data). The former type consists of a number of processors which execute the same instructions but on different data, whereas the latter has a separate program for each processor. Fine-grain computers are usually SIMD, while coarse grain computers tend to be MIMD (also in correspondence with table 9.1, entries 1 and 2).

		Number of Data Streams	
		single	multiple
Number of Instruction Streams	single	SISD (von Neumann)	SIMD (vector, array)
	multiple	MISD (pipeline?)	MIMD (multiple micros)

Table 10.1: Flynn's classification.

Table 10.2 shows a comparison of several types of hardware for neural network simulation. The speed entry, measured in interconnects per second, is an important measure which is often used to compare neural network simulators. It measures the number of multiply-and-add operations that can be performed per second. However, the comparison is not 100% honest: it does not always include the time needed to fetch the data on which the operations are to be performed, and may also ignore other functions required by some algorithms such as the computation of a sigmoid function. Also, the speed is of course dependent of the algorithm used.

	HARDWARE	WORD LENGTH	STORAGE (K Intcnts)	SPEED (K Int/s)	COST (K$)	SPEED / COST
WORKSTATIONS						
Micro/Mini	PC/AT	16	100	25	5	5.0
Computers	Sun 3	32	250	250	20	12.5
	VAX	32	100	100	300	0.33
	Symbolics	32	32,000	35	100	0.35
Attached	ANZA	8–32	500	45	10	4.5
Processors	$\Delta - 1$	32	1,000	10,000	15	667
	Transputer	16	2,000	3,000	4	750
Bus-oriented	Mark III, IV	16	1,000	500	75	6.7
	MX/1–16	16	50,000	120,000	300	400
MASSIVELY	CM–2 (64K)	32	64,000	13,000	2,000	6.5
PARALLEL	Warp (10)	32	320	17,000	300	56.7
	Warp (20)			32,000		
	Butterfly (64)	32	60,000	8,000	500	16
SUPER-COMPUTERS	Cray XMP	64	2,000	50,000	4,000	12.5

Table 10.2: Hardware machines for neural network simulation.
The authors are well aware that the mentioned computer architectures are archaic... current computer architectures are several orders of magnitute faster. For instance, current day Sun Sparc machines (e.g., an Ultra at 200 MHz) benchmark at almost 300,000 dhrystones per second, whereas the archaic Sun 3 benchmarks at about 3,800. Prices of both machines (then vs. now) are approximately the same. Go figure! Nevertheless, the table gives an insight of the performance of different types of architectures.

10.1 The Connection Machine

10.1.1 Architecture

One of the most outstanding fine-grain SIMD parallel machines is Daniel Hillis' Connection Machine (Hillis, 1985; Corporation, 1987), originally developed at MIT and later built at Thinking Machines Corporation. The original model, the CM–1, consists of 64K (65,536) one-bit processors, divided up into four units of 16K processors each. The units are connected via a cross-bar switch (the *nexus*) to up to four front-end computers (see figure 10.1). The large number of extremely simple processors make the machine a *data parallel* computer, and can be best envisaged as active memory.

Each processor chip contains 16 processors, a control unit, and a router. It is connected to a memory chip which contains 4K bits of memory per processor. Each processor consists of a one-bit ALU with three inputs and two outputs, and a set of registers. The control unit decodes incoming instructions broadcast by the front-end computers (which can be DEX VAXes or Symbolics Lisp machines). At any time, a processor may be either listening to the incoming instruction or not.

The router implements the communication algorithm: each router is connected to its nearest neighbours via a two-dimensional grid (the *NEWS* grid) for fast neighbour communication; also, the chips are connected via a Boolean 12-cube, i.e., chips i and j are connected if and only if $|i - j| = 2^k$ for some integer k. Thus at most 12 hops are needed to deliver a message. So there are 4,096 routers connected by 24,576 bidirectional wires.

By slicing the memory of a processor, the CM can also implement virtual processors.

The CM–2 differs from the CM–1 in that it has 64K bits instead of 4K bits memory per processor, and an improved I/O system.

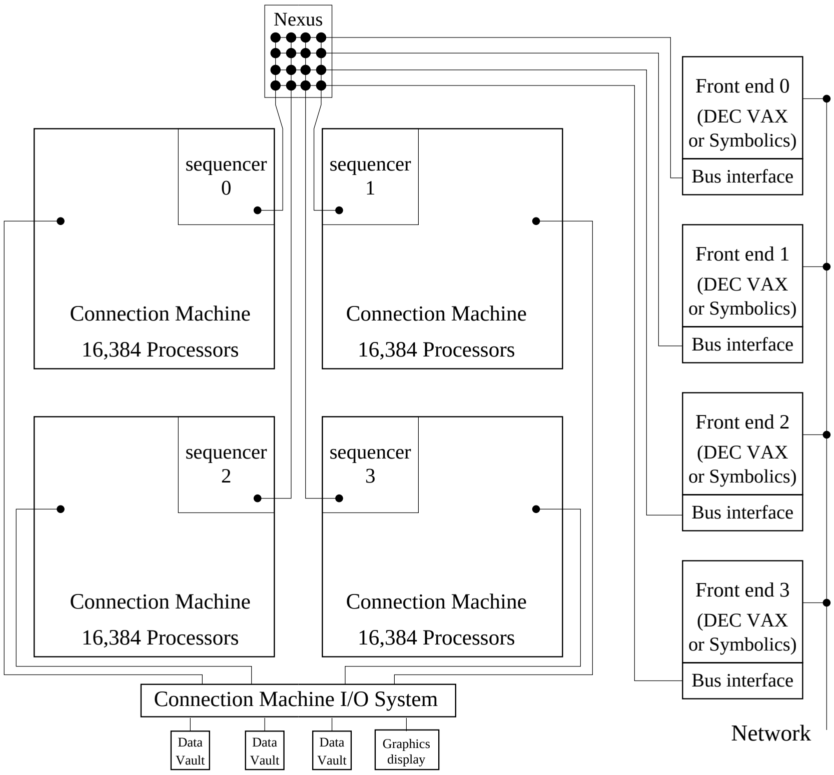


Figure 10.1: The Connection Machine system organisation.

10.1.2 Applicability to neural networks

There have been a few researchers trying to implement neural networks on the Connection Machine (Blelloch & Rosenberg, 1987; Singer, 1990). Even though the Connection Machine has a topology which matches the topology of most artificial neural networks very well, the *relatively* slow message passing system makes the machine not very useful as a general-purpose neural network simulator. It appears that the Connection Machine suffers from a dramatic decrease in throughput due to communication delays (Hummel, 1990). Furthermore, the cost/speed ratio (see table 10.2) is very bad compared to, e.g., a transputer board. As an effect, the Connection Machine is not widely used for neural network simulation.

One possible implementation is given in (Blelloch & Rosenberg, 1987). Here, a back-propagation network is implemented by allocating one processor per unit and one per outgoing weight and one per incoming weight. The processors are thus arranged that each processor for a unit is immediately followed by the processors for its outgoing weights and preceded by those for its incoming weights. The feed-forward step is performed by first clamping input units and next executing a *copy-scan* operation by moving those activation values to the next k processors (the outgoing weight processors). The weights then multiply themselves with the activation values and perform a *send* operation in which the resulting values are sent to the processors allocated for incoming weights. A *plus-scan* then sums these values to the next layer of units in the network. The feedback step is executed similarly. Both the feed-forward and feedback steps can be interleaved and pipelined such that no layer is ever idle. For example, for the feed-forward step, a new pattern $\mathbf{x}^p$ is clamped on the input layer while the next layer is computing on $\mathbf{x}^{p-1}$, etc.

To prevent inefficient use of processors, one weight could also be represented by one processor.

10.2 Systolic arrays

Systolic arrays (Kung & Leierson, 1979) take the advantage of laying out algorithms in two dimensions. The design favours compute-bound as opposed to I/O-bound operations. The name *systolic* is derived from the analogy of pumping blood through a heart and feeding data through a systolic array.

A typical use is depicted in figure 10.2. Here, two band matrices A and B are multiplied and added to C, resulting in an output $C + AB$. Essential in the design is the reuse of data elements, instead of referencing the memory each time the element is needed.

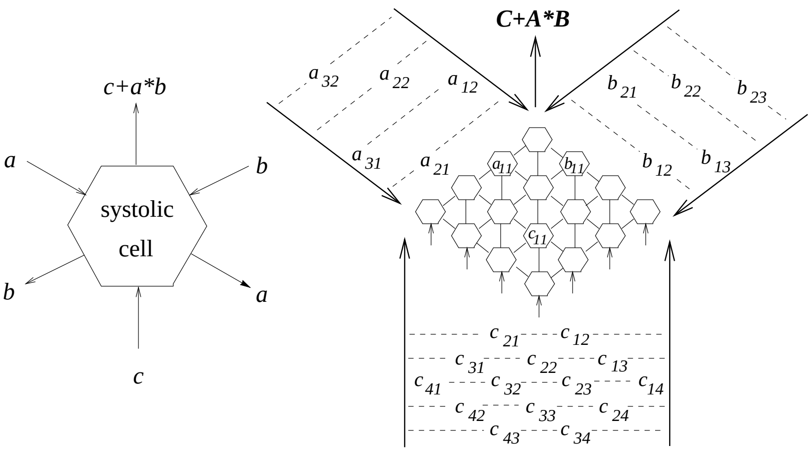


Figure 10.2: Typical use of a systolic array.

The Warp computer, developed at Carnegie Mellon University, has been used for simulating artificial neural networks (Pomerleau, Gusciora, Touretzky, & Kung, 1988) (see table 10.2). It is a system with ten or more programmable one-dimensional systolic arrays. Two data streams, one of which is bi-directional, flow through the processors (see figure 10.3). To implement a matrix product $W\mathbf{x} + \theta$, the W is not a stream as in figure 10.2 but stored in the memory of the processors.

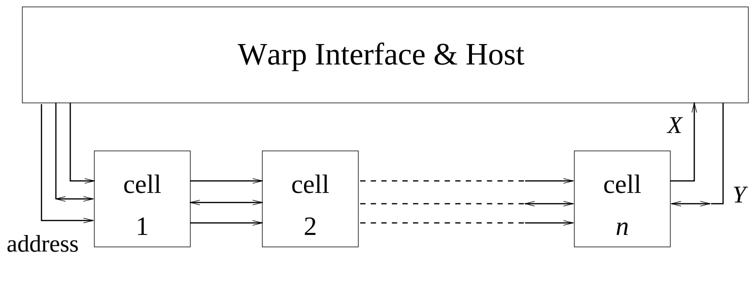


Figure 10.3: The Warp system architecture.

11 Dedicated Neuro-Hardware

Recently, many neuro-chips have been designed and built. Although many techniques, such as digital and analogue electronics, optical computers, chemical implementation, and bio-chips, are investigated for implementing neuro-computers, only digital and analogue electronics, and in a lesser degree optical implementations, are at present feasible techniques. We will therefore concentrate on such implementations.

11.1 General issues

11.1.1 Connectivity constraints

Connectivity within a chip

A major problem with neuro-chips always is the connectivity. A single integrated circuit is, in current-day technology, planar with limited possibility for cross-over connections. This poses a problem. Whereas connectivity to nearest neighbour can be implemented without problems, connectivity to the second nearest neighbour results in a cross-over of four which is already problematic. On the other hand, full connectivity between a *set* of input and output units can be easily attained when the input and output neurons are situated near two edges of the chip (see figure 11.1). Note that the number of neurons in the chip grows linearly with the size of the chip, whereas in the earlier layout, the dependence is quadratic.

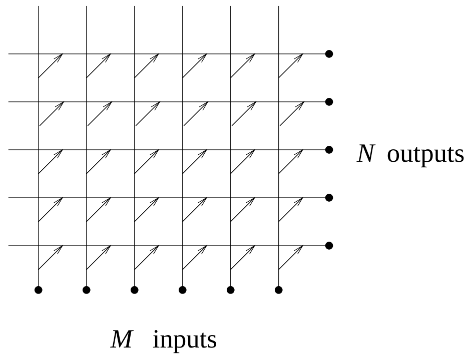


Figure 11.1: Connections between M input and N output neurons.

Connectivity between chips

To build large or layered ANN's, the neuro-chips have to be connected together. When only few neurons have to be connected together, or the chips can be placed in subsequent rows in feed-forward types of networks, this is no problem. But in other cases, when large numbers of neurons in one chip have to be connected to neurons in other chips, there are a number of problems:

- designing chip packages with a very large number of input or output leads;
- fan-out of chips: each chip can ordinarily only send signals two a small number of other chips. Amplifiers are needed, which are costly in power dissipation and chip area;
- wiring.

A possible solution would be using optical interconnections. In this case, an external light source would reflect light on one set of neurons, which would reflect part of this light using deformable mirror spatial light modulator technology on to another set of neurons. Also under development are three-dimensional integrated circuits.

11.1.2 Analogue vs. digital

Due to the similarity between artificial and biological neural networks, analogue hardware seems a good choice for implementing artificial neural networks, resulting in cheaper implementations which operate at higher speed. On the other hand, digital approaches offer far greater flexibility and, not to be neglected, arbitrarily high accuracy. Also, digital chips can be designed without the need of very advanced knowledge of the circuitry using CAD/CAM systems, whereas the design of analogue chips requires good theoretical knowledge of transistor physics as well as experience.

An advantage that analogue implementations have over digital neural networks is that they closely match the physical laws present in neural networks (table 9.1, point 1). First of all, weights in a neural network can be coded by one single analogue element (e.g., a resistor) where several digital elements are needed[1]. Secondly, very simple rules as Kirchoff's laws[2] can be used to carry out the addition of input signals. As another example, Boltzmann machines (section 5.3) can be easily implemented by amplifying the natural noise present in analogue devices.

11.1.3 Optics

As mentioned above, optics could be very well used to interconnect several (layers of) neurons. One can distinguish two approaches. One is to store weights in a planar transmissive or reflective device (e.g., a spatial light modulator) and use lenses and fixed holograms for interconnection. Figure 11.2 shows an implementation of optical matrix multiplication. When N is the linear size of the optical array divided by wavelength of the light used, the array has capacity for N^2 weights, so it can fully connect N neurons with N neurons (Fahrat, Psaltis, Prata, & Paek, 1985).

A second approach uses volume holographic correlators, offering connectivity between two areas of N^2 neurons for a total of N^4 connections[3]. A possible use of such volume holograms in an all-optical network would be to use the system for image completion (Abu-Mostafa & Psaltis, 1987). A number of images could be stored in the hologram. The input pattern is correlated with each of them, resulting in output patterns with a brightness varying with the

[1] On the other hand, the opposite can be found when considering the *size* of the element, especially when high accuracy is needed. However, once artificial neural networks have outgrown rules like back-propagation, high accuracy might not be needed.

[2] The Kirchoff laws state that for two resistors R_1 and R_2 (1) in series, the total resistance can be calculated using $R = R_1 + R_2$, and (2) in parallel, the total resistance can be found using $1/R = 1/R_1 + 1/R_2$ (Feynman, Leighton, & Sands, 1983).

[3] Well ... not exactly. Due to diffraction, the total number of independent connections that can be stored in an ideal medium is N^3, i.e., the volume of the hologram divided by the cube of the wavelength. So, in fact $N^{3/2}$ neurons can be connected with $N^{3/2}$ neurons.

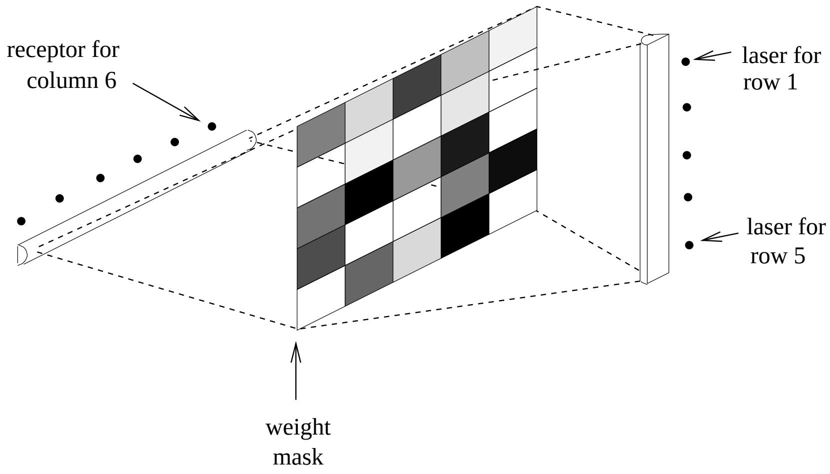


Figure 11.2: Optical implementation of matrix multiplication.

degree of correlation. The images are fed into a threshold device which will conduct the image with highest brightness better than others. This enhancement can be repeated for several loops.

11.1.4 Learning vs. non-learning

It is generally agreed that the major forte of neural networks is their ability to *learn*. Whereas a network with fixed, pre-computed, weight values could have its merit in industrial applications, on-line adaptivity remains a design goal for most neural systems.

With respect to learning, we can distinguish between the following levels:

1. **fixed weights**: the design of the network determines the weights. Examples are the retina and cochlea chips of Carver Mead's group discussed below (cf. a ROM (Read-Only Memory) in computer design);

2. **pre-programmed weights**: the weights in the network can be set only once, when the chip is installed. Many optical implementations fall in this category (cf. PROM (Programmable ROM));

3. **programmable weights**: the weights can be set more than once by an external device (cf. EPROM (Erasable PROM) or EEPROM (Electrically Erasable PROM));

4. **on-site adapting weights**: the learning mechanism is incorporated in the network (cf. RAM (Random Access Memory)).

11.2 Implementation examples

11.2.1 Carver Mead's silicon retina

The chips devised by Carver Mead's group at Caltech (Mead, 1989) are heavily inspired by biological neural networks. Mead attempts to build analogue neural chips which match biological neurons as closely as possible, including extremely low power consumption, fully analogue hardware, and operation in continuous time (table 9.1, point 3). One example of such a chip is the *Silicon Retina* (Mead & Mahowald, 1988).

Retinal structure

The off-center retinal structure can be described as follows. Light is transduced to electrical signals by photo-receptors which have a primary pathway through the *triad synapses* to the *bipolar cells*. The bipolar cells are connected to the *retinal ganglion* cells which are the output cells of the retina. The *horizontal cells*, which are also connected via the triad synapses to the photo-receptors, are situated directly below the photo-receptors and have synapses connected to the axons leading to the bipolar cells.

The system can be described in terms of the triad synapse's three elements:

1. the photo-receptor outputs the logarithm of the intensity of the light;
2. the horizontal cells form a network which averages the photo-receptor over space and time;
3. the output of the bipolar cell is proportional to the difference between the photo-receptor output and the horizontal cell output.

The photo-receptor

The photo-receptor circuit outputs a voltage which is proportional to the logarithm of the intensity of the incoming light. There are two important consequences:

1. several orders of magnitude of intensity can be handled in a moderate signal level range;
2. the voltage difference between two points is proportional to the contrast ratio of their illuminance.

The photo-receptor can be implemented using a photo-detector, two FET's[4] connected in series[5] and one transistor (see figure 11.3). The lowest photo-current is about $10^{-14}A$ or 10^5 photons

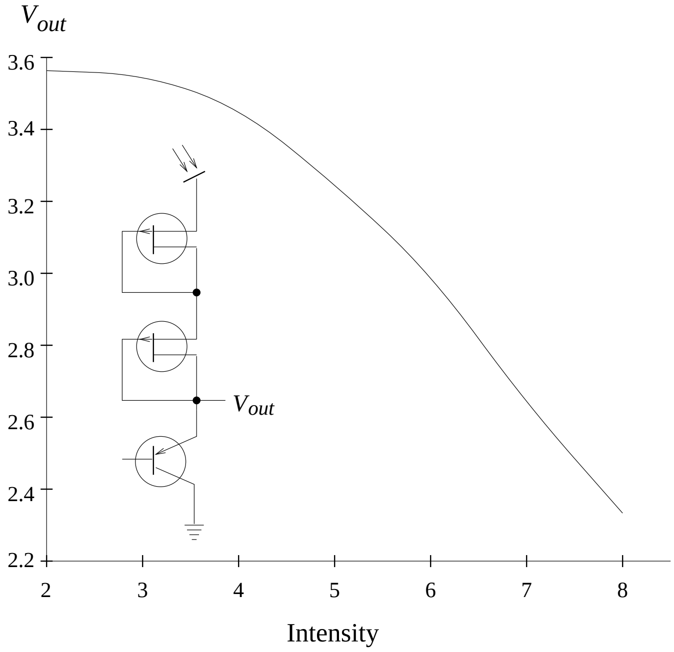


Figure 11.3: The photo-receptor used by Mead. To prevent current being drawn from the photo-receptor, the output is only connected to the gate of the transistor.

per second, corresponding with a moonlit scene.

[4]Field Effect Transistor

[5]A detailed description of the electronics involved is out of place here. However, we will provide figures where useful. See (Mead, 1989) for an in-depth study.

Horizontal resistive layer

Each photo-receptor is connected to its six neighbours via resistors forming a hexagonal array. The voltage at every node in the network is a spatially weighted average of the photo-receptor inputs, such that farther away inputs have less influence (see figure 11.4(a)).

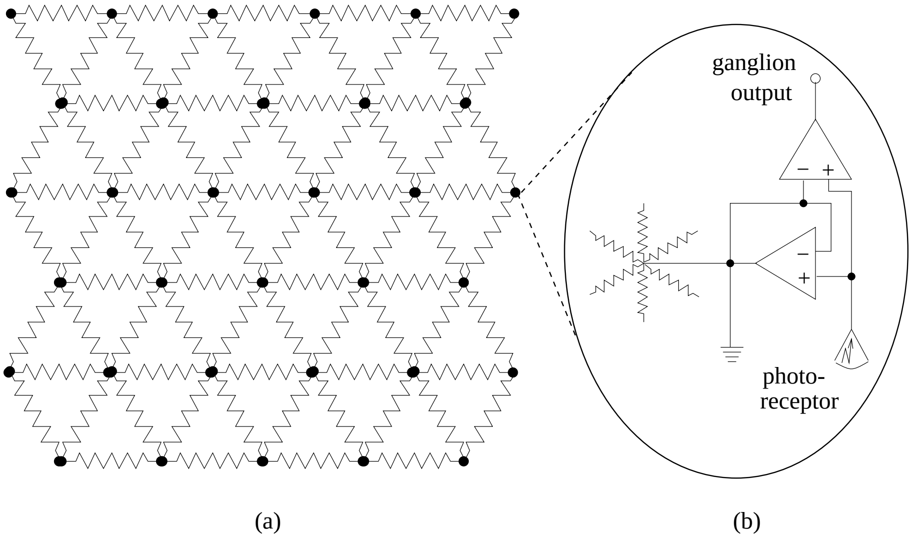


Figure 11.4: The resistive layer (a) and, enlarged, a single node (b).

Bipolar cell

The output of the bipolar cell is proportional to the difference between the photo-receptor output and the voltage of the horizontal resistive layer. The architecture is shown in figure 11.4(b). It consists of two elements: a wide-range amplifier which drives the resistive network towards the photo-receptor output, and an amplifier sensing the voltage difference between the photo-receptor output and the network potential.

Implementation

A chip was built containing 48×48 pixels. The output of every pixel can be accessed independently by providing the chip with the horizontal and vertical address of the pixel. The selectors can be run in two modes: static probe or serial access. In the first mode, a single row and column are addressed and the output of a single pixel is observed as a function of time. In the second mode, both vertical and horizontal shift registers are clocked to provide a serial scan of the processed image for display on a television display.

Performance

Several experiments show that the silicon retina performs similarly as biological retina (Mead & Mahowald, 1988). Similarities are shown between sensitivity for intensities; time responses for a single output when flashes of light are input; response to contrast edges.

11.2.2 LEP's LNeuro chip

A radically different approach is the LNeuro chip developed at the Laboratoires d'Electronique Philips (LEP) in France (Theeten, Duranton, Mauduit, & Sirat, 1990; Duranton & Sirat, 1989). Whereas most neuro-chips implement Hopfield networks (section 5.2) or, in some cases, Kohonen networks (section 6.2) (due to the fact that these networks have *local* learning rules), these digital neuro-chips can be configured to incorporate any learning rule and network topology.

Architecture

The LNeuro chip, depicted in figure 11.5, consists of an multiply-and-add or *relaxation* part, and a learning part. The LNeuro 1.0 has a parallelism of 16. The weights w_{ij} are 8 bits long in the relaxation phase. and 16 bit in the learning phase.

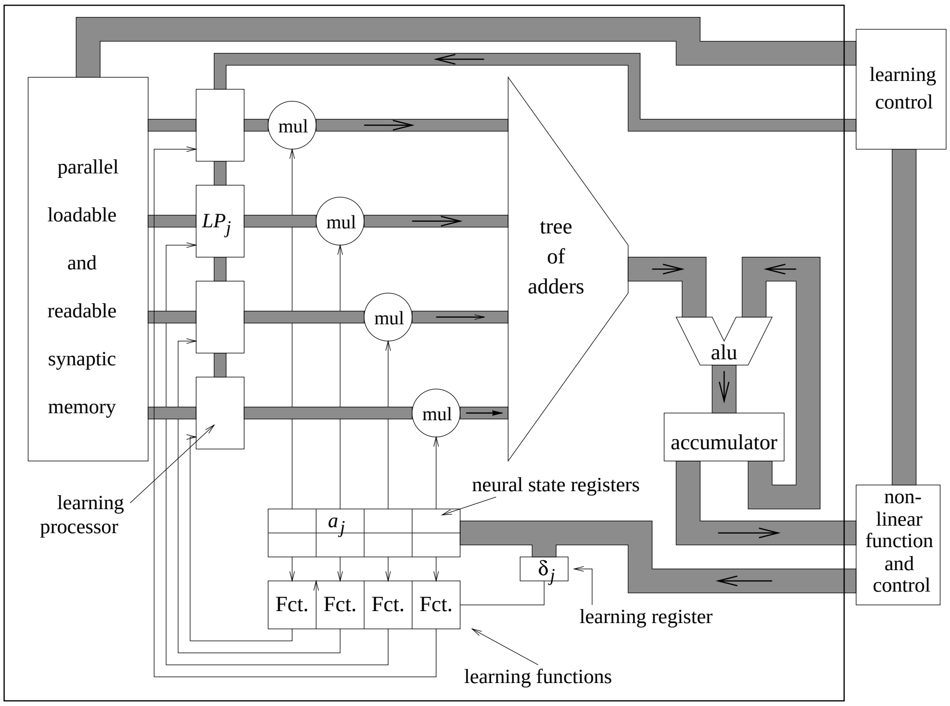


Figure 11.5: The LNeuro chip. For clarity, only four neurons are drawn.

Multiply-and-add

The multiply-and-add in fact performs a matrix multiplication

$$y_k(t+1) = \mathcal{F}\left(\sum_j w_{jk} y_j(t)\right). \tag{11.1}$$

The input activations y_k are kept in the *neural state registers*. For each neural state there are two registers. These can be used to implement synchronous or asynchronous update. In the former mode, the computed state of neurons wait in registers until all states are known; then the whole register is written into the register used for the calculations. In asynchronous mode, however, every new state is directly written into the register used for the next calculation.

The arithmetical logical unit (ALU) has an external input to allow for accumulation of external partial products. This can be used to construct larger, structured, or higher-precision networks.

The neural states (y_k) are coded in one to eight bits, whereas either eight or sixteen bits can be used for the weights which are kept in a RAM. In order to save silicon area, the multiplications $w_{jk} y_j$ are serialised over the bits of y_j, replacing N eight by eight bit parallel multipliers by N eight bit AND gates. The partial products are saved and added in the *tree of adders*.

The computation thus increases linearly with the number of neurons (instead of quadratic in simulation on serial machines).

The activation function is, for reasons of flexibility, kept off-chip. The results of the weighted sum calculation go off-chip serially (i.e., bit by bit), and the result must be written back to the neural state registers.

Finally, a column of latches is included to temporarily store memory values, such that during a multiply of the weight with several bits the memory can be freely accessed. These latches in fact take part in the learning mechanism described below.

Learning

The remaining parts in the chip are dedicated to the learning mechanism. The learning mechanism is designed to implement the Hebbian learning rule (Hebb, 1949)

$$w_{jk} \leftarrow w_{jk} + \delta_k y_j \tag{11.2}$$

where δ_k is a scalar which only depends on the output neuron k. To simplify the circuitry, eq. (11.2) is simplified to

$$w_{jk} \leftarrow w_{jk} + g(y_k, y_j)\delta_k \tag{11.3}$$

where $g(y_k, y_j)$ can have value -1, 0, or $+1$. In effect, eq. (11.3) either increments or decrements the w_{jk} with δ_k, or keeps w_{jk} unchanged. Thus eq. (11.2) can be simulated by executing eq. (11.3) several times over the same set of weights.

The weights $\mathbf{w}_k$ related to the output neuron k are all modified in parallel. A learning step proceeds as follows. Every *learning processor* (see figure 11.5) LP_j loads the weight w_{jk} from the synaptic memory, the δ_k from the *learning register*, and the neural state y_j. Next, they all modify their weights in parallel using eq. (11.3) and write the adapted weights back to the synaptic memory, also in parallel.

References

Abu-Mostafa, Y. A., & Psaltis, D. (1987). Optical neural computers. *Scientific American*, *?*, 88-95. [Cited on p. 116.]

Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, *9*(1), 147–169. [Cited on p. 54.]

Ahalt, S. C., Krishnamurthy, A. K., Chen, P., & Melton, D. (1990). Competititve learning algorithms for vector quantization. *Neural Networks*, *3*, 277-290. [Cited on p. 60.]

Almasi, G. S., & Gottlieb, A. (1989). *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company Inc. [Cited on p. 111.]

Almeida, L. B. (1987). A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *Proceedings of the First International Conference on Neural Networks* (Vol. 2, pp. 609–618). IEEE. [Cited on p. 50.]

Amit, D. J., Gutfreund, H., & Sompolinsky, H. (1986). Spin-glass models of neural networks. *Physical Review A*, *32*(2), 1007–1018. [Cited on p. 50.]

Anderson, J. A. (1977). Neural models with cognitive implications. In D. LaBerge & S. J. Samuels (Eds.), *Basic Processes in Reading Perception and Comprehension Models* (p. 27-90). Hillsdale, NJ: Erlbaum. [Cited on pp. 18, 50.]

Anderson, J. A., & Rosenfeld, E. (1988). *Neurocomputing: Foundations of Research*. Cambridge, MA: The MIT Press. [Cited on p. 9.]

Barto, A. G., & Anandan, P. (1985). Pattern-recognizing stochastic learning automata. *IEEE Transactions on Systems, Man and Cybernetics*, *15*, 360–375. [Cited on p. 78.]

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning problems. *IEEE Transactions on Systems, Man and Cybernetics*, *13*, 834-846. [Cited on pp. 75, 77.]

Barto, A. G., Sutton, R. S., & Watkins, C. (1990). Sequential decision problems and neural networks. In D. Touretsky (Ed.), *Advances in Neural Information Processing II*. DUNNO. [Cited on p. 80.]

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. [Cited on p. 80.]

Blelloch, G., & Rosenberg, C. R. (1987). Network learning on the Connection Machine. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 323–326). DUNNO. [Cited on p. 113.]

Board, J. A. B., Jr. (1989). *Transputer Research and Applications 2: Proceedings of the Second Conference on the North American Transputer Users Group*. Durham, NC: IOS Press. [Cited on p. 111.]

Boomgaard, R. van den, & Smeulders, A. (1989). Self learning image processing using a-priori knowledge of spatial patterns. In T. Kanade, F. C. A. Groen, & L. O. Hertzberger (Eds.), *Proceedings of the I.A.S. Conference* (p. 305-314). Elsevier Science Publishers. [Cited on p. 42.]

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and Regression Trees.* Wadsworth and Broks/Cole. [Cited on p. 63.]

Bruce, A. D., Canning, A., Forrest, B., Gardner, E., & Wallace, D. J. (1986). Learning and memory properties in fully connected networks. In J. S. Denker (Ed.), *AIP Conference Proceedings 151, Neural Networks for Computing* (pp. 65–70). DUNNO. [Cited on p. 52.]

Carpenter, G. A., & Grossberg, S. (1987a). A massively parallel architecture for a self-organizing neural pattern recognition machine. *Computer Vision, Graphics, and Image Processing*, *37*, 54–115. [Cited on pp. 57, 72.]

Carpenter, G. A., & Grossberg, S. (1987b). ART 2: Self-organization of stable category recognition codes for analog input patterns. *Applied Optics*, *26*(23), 4919-4930. [Cited on p. 72.]

Corporation, T. M. (1987). *Connection Machine Model CM–2 Technical Summary* (Tech. Rep. Nos. HA87–4). Thinking Machines Corporation. [Cited on p. 112.]

Cottrell iG.W., Munro, P., & Zipser, D. (1987). *Image compression by back-propagation: a demonstration of extensional programming* (Tech. Rep. No. TR 8702). USCD, Institute of Cognitive Sciences. [Cited on p. 99.]

Craig, J. J. (1989). *Introduction to Robotics.* Addison-Wesley Publishing Company. [Cited on p. 85.]

Cun, Y. L. (1985). Une procedure d'apprentissage pour reseau a seuil assymetrique. *Proceedings of Cognitiva*, *85*, 599–604. [Cited on p. 33.]

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, *2*(4), 303–314. [Cited on p. 33.]

DARPA. (1988). *DARPA Neural Network Study.* AFCEA International Press. [Cited on pp. 9, 109.]

Dastani, M. M. (1991). *Functie-Benadering met Feed-Forward Netwerken.* Unpublished master's thesis, Universiteit van Amsterdam, Faculteit Computer Systemen. [Cited on pp. 39, 40.]

Duranton, M., & Sirat, J. A. (1989). Learning on VLSI: A general purpose digital neurochip. In *Proceedings of the Third International Joint Conference on Neural Networks.* DUNNO. [Cited on p. 119.]

Eckmiller, R., Hartmann, G., & Hauske, G. (1990). *Parallel Processing in Neural Systems and Computers.* Elsevier Science Publishers B.V. [Cited on p. 111.]

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, *14*, 179–211. [Cited on p. 48.]

Fahrat, N. H., Psaltis, D., Prata, A., & Paek, E. (1985). Optical implementation of the Hopfield model. *Applied Optics*, *24*, 1469–1475. [Cited on p. 116.]

Feldman, J. A., & Ballard, D. H. (1982). Connectionist models and their properties. *Cognitive Science*, *6*, 205–254. [Cited on p. 16.]

Feynman, R. P., Leighton, R. B., & Sands, M. (1983). *The Feynman Lectures on Physics.* Reading (MA), Menlo Park (CA), London, Sidney, Manila: Addison-Wesley Publishing Company. [Cited on p. 116.]

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, *C–21*, 948-960. [Cited on p. 111.]

Friedman, J. H. (1991). Multivariate adaptive regression splines. *Annals of Statistics*, *19*, 1-141. [Cited on p. 63.]

Fritzke, B. (1991). Let it grow—self-organizing feature maps with problem dependent cell structure. In T. Kohonen, K. Mäkisara, O. Simula, & J. Kangas (Eds.), *Proceedings of the 1991 International Conference on Artificial Neural Networks* (pp. 403–408). North-Holland/Elsevier Science Publishers. [Cited on p. 64.]

Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, *20*, 121–136. [Cited on pp. 57, 100.]

Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Networks*, *1*, 119–130. [Cited on pp. 57, 98, 100.]

Funahashi, K.-I. (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks*, *2*(3), 193–192. [Cited on p. 33.]

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability.* New York: W. H. Freeman. [Cited on p. 53.]

Geman, S., & Geman, D. (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *6*, 721–724. [Cited on p. 105.]

Gielen, C., Krommenhoek, K., & Gisbergen, J. van. (1991). A procedure for self-organized sensor-fusion in topologically ordered maps. In T. Kanade, F. C. A. Groen, & L. O. Hertzberger (Eds.), *Proceedings of the Second International Conference on Autonomous Systems* (pp. 417–423). Elsevier Science Publishers. [Cited on p. 66.]

Gorman, R. P., & Sejnowski, T. J. (1988). Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, *1*(1), 75-89. [Cited on p. 45.]

Grossberg, S. (1976). Adaptive pattern classification and universal recoding I & II. *Biological Cybernetics*, *23*, 121–134, 187–202. [Cited on pp. 57, 69.]

Group, O. U. (1987). *Parallel Programming of Transputer Based Machines: Proceedings of the 7th Occam User Group Technical Meeting.* Grenoble, France: IOS Press. [Cited on p. 111.]

Gullapalli, V. (1990). A stochastic reinforcement learning algorithm for learning real-valued functions. *Neural Networks*, *3*, 671-692. [Cited on p. 77.]

Hartigan, J. A. (1975). *Clustering Algorithms.* New York: John Wiley & Sons. [Cited on p. 72.]

Hartman, E. J., Keeler, J. D., & Kowalski, J. M. (1990). Layered neural networks with Gaussian hidden units as universal approximations. *Neural Computation*, *2*(2), 210–215. [Cited on p. 33.]

Hebb, D. O. (1949). *The Organization of Behaviour.* New York: Wiley. [Cited on pp. 18, 121.]

Hecht-Nielsen, R. (1988). Counterpropagation networks. *Neural Networks*, *1*, 131-139. [Cited on p. 63.]

Hertz, J., Krogh, A., & Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation.* Addison Wesley. [Cited on p. 9.]

Hesselroth, T., Sarkar, K., Smagt, P. van der, & Schulten, K. (1994). Neural network control of a pneumatic robot arm. *IEEE Transactions on Systems, Man, and Cybernetics*, *24*(1), 28–38. [Cited on p. 93.]

Hestenes, M. R., & Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Nat. Bur. Standards J. Res.*, *49*, 409-436. [Cited on p. 41.]

Hillis, W. D. (1985). *The Connection Machine.* The MIT Press. [Cited on p. 112.]

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, *79*, 2554–2558. [Cited on pp. 18, 50.]

Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences*, *81*, 3088–3092. [Cited on p. 52.]

Hopfield, J. J., Feinstein, D. I., & Palmer, R. G. (1983). 'unlearning' has a stabilizing effect in collective memories. *Nature*, *304*, 159–159. [Cited on p. 52.]

Hopfield, J. J., & Tank, D. W. (1985). 'neural' computation of decisions in optimization problems. *Biological Cybernetics*, *52*, 141–152. [Cited on p. 53.]

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366. [Cited on p. 33.]

Hummel, R. A. (1990). *Personal Communication.* [Cited on p. 113.]

Jansen, A., Smagt, P. P. van der, & Groen, F. C. A. (1994). Nested networks for robot control. In A. F. Murray (Ed.), *Neural Network Applications.* Kluwer Academic Publishers. (In press) [Cited on pp. 63, 90.]

Jordan, M. I. (1986a). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society* (pp. 531–546). Hillsdale, NJ: Erlbaum. [Cited on p. 48.]

Jordan, M. I. (1986b). *Serial OPrder: A Parallel Distributed Processing Approach* (Tech. Rep. No. 8604). San Diego, La Jolla, CA: Institute for Cognitive Science, University of California. [Cited on p. 48.]

Jorgensen, C. C. (1987). Neural network representation of sensor graphs in autonomous robot path planning. In *IEEE First International Conference on Neural Networks* (Vol. IV, pp. 507–515). IEEE. [Cited on p. 94.]

Josin, G. (1988). Neural-space generalization of a topological transformation. *Biological Cybernetics*, *59*, 283–290. [Cited on p. 46.]

Katayama, M., & Kawato, M. (1992). *A Parallel-Hierarchical Neural Network Model for Motor Control of a Musculo-Skeletal System* (Tech. Rep. Nos. TR–A–0145). ATR Auditory and Visual Perception Research Laboratories. [Cited on p. 93.]

Kawato, M., Furukawa, K., & Suzuki, R. (1987). A hierarchical neural-network model for control and learning of voluntary movement. *Biological Cybernetics*, *57*, 169–185. [Cited on p. 91.]

Kohonen, T. (1977). *Associative Memory: A System-Theoretical Approach.* Springer-Verlag. [Cited on pp. 18, 50, 64.]

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, *43*, 59–69. [Cited on p. 64.]

Kohonen, T. (1984). *Self-Organization and Associative Memory.* Berlin: Springer-Verlag. [Cited on p. 64.]

Kohonen, T. (1995). *Self-Organizing Maps.* Springer. [Cited on p. 9.]

Kohonen, T., Makisara, M., & Saramaki, T. (1984). Phonotopic maps—insightful representation of phonological features for speech recognition. In *Proceedings of the 7th IEEE International Conference on Pattern Recognition.* DUNNO. [Cited on p. 66.]

Kröse, B. J. A., & Dam, J. W. M. van. (1992). Learning to avoid collisions: A reinforcement learning paradigm for mobile robot manipulation. In *Proceedings of IFAC/IFIP/IMACS International Symposium on Artificial Intelligence in Real-Time Control* (p. 295-300). Delft: IFAC, Laxenburg. [Cited on p. 78.]

Kröse, B. J. A., Korst, M. J. van der, & Groen, F. C. A. (1990). Learning strategies for a vision based neural controller for a robot arm. In O. Kaynak (Ed.), *IEEE International Workshop on Intelligent Motor Control* (pp. 199–203). IEEE. [Cited on p. 89.]

Kung, H. T., & Leierson, C. E. (1979). Systolic arrays (for VLSI). In *Sparse Matrix Proceedings, 1978.* Academic Press. (also in: *Algorithms for VLSI Processor Arrays*, C. Mead & L. Conway, eds., Addison-Wesley, 1980, 271-292) [Cited on p. 114.]

Lippmann, R. P. (1987). An introduction to computing with neural nets. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *2*(4), 4–22. [Cited on pp. 71, 73.]

Lippmann, R. P. (1989). Review of neural networks for speech recognition. *Neural Computation*, *1*, 1–38. [Cited on p. 58.]

Mallach, E. G. (1975). Emulator architecture. *Computer*, *8*, 24–32. [Cited on p. 109.]

Marr, D. (1982). *Vision.* San Francisco: W. H. Freeman. [Cited on pp. 103, 104.]

Martinetz, T., & Schulten, K. (1991). A "neural-gas" network learns topologies. In T. Kohonen, K. Mäkisara, O. Simula, & J. Kangas (Eds.), *Proceedings of the 1991 International Conference on Artificial Neural Networks* (pp. 397–402). North-Holland/Elsevier Science Publishers. [Cited on p. 64.]

McClelland, J. L., & Rumelhart, D. E. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* The MIT Press. [Cited on pp. 9, 15.]

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 115–133. [Cited on p. 13.]

Mead, C. (1989). *Analog VLSI and Neural Systems.* Reading, MA: Addison-Wesley. [Cited on pp. 105, 117, 118.]

Mead, C. A., & Mahowald, M. A. (1988). A silicon model of early visual processing. *Neural Networks*, *1*(1), 91–97. [Cited on pp. 117, 119.]

Mel, B. W. (1990). *Connectionist Robot Motion Planning.* San Diego, CA: Academic Press. [Cited on p. 16.]

Minsky, M., & Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry.* The MIT Press. [Cited on pp. 13, 26, 31, 33.]

Murray, A. F. (1989). Pulse arithmetic in VLSI neural networks. *IEEE Micro*, *9*(12), 64–74. [Cited on p. 109.]

Oja, E. (1982). A simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, *15*, 267–273. [Cited on p. 68.]

Parker, D. B. (1985). *Learning-Logic* (Tech. Rep. Nos. TR–47). Cambridge, MA: Massachusetts Institute of Technology, Center for Computational Research in Economics and Management Science. [Cited on p. 33.]

Pearlmutter, B. A. (1989). Learning state space trajectories in recurrent neural networks. *Neural Computation*, *1*(2), 263–269. [Cited on p. 50.]

Pearlmutter, B. A. (1990). *Dynamic Recurrent Neural Networks* (Tech. Rep. Nos. CMU–CS–90–196). Pittsburgh, PA 15213: School of Computer Science, Carnegie Mellon University. [Cited on pp. 17, 50.]

Pineda, F. (1987). Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, *19*(59), 2229–2232. [Cited on p. 50.]

Polak, E. (1971). *Computational Methods in Optimization.* New York: Academic Press. [Cited on p. 41.]

Pomerleau, D. A., Gusciora, G. L., Touretzky, D. S., & Kung, H. T. (1988). Neural network simulation at Warp speed: How we got 17 million connections per second. In *IEEE Second International Conference on Neural Networks* (Vol. II, p. 143-150). DUNNO. [Cited on p. 114.]

Powell, M. J. D. (1977). Restart procedures for the conjugate gradient method. *Mathematical Programming*, *12*, 241-254. [Cited on p. 41.]

Press, W. H., Flannery, B. P., Teukolsky, S. A., & Vetterling, W. T. (1986). *Numerical Recipes: The Art of Scientific Computing.* Cambridge: Cambridge University Press. [Cited on pp. 40, 41.]

Psaltis, D., Sideris, A., & Yamamura, A. A. (1988). A multilayer neural network controller. *IEEE Control Systems Magazine*, *8*(2), 17–21. [Cited on p. 87.]

Ritter, H. J., Martinetz, T. M., & Schulten, K. J. (1989). Topology-conserving maps for learning visuo-motor-coordination. *Neural Networks*, *2*, 159–168. [Cited on p. 90.]

Ritter, H. J., Martinetz, T. M., & Schulten, K. J. (1990). *Neuronale Netze.* Addison-Wesley Publishing Company. [Cited on p. 9.]

Rosen, B. E., Goodwin, J. M., & Vidal, J. J. (1992). Process control with adaptive range coding. *Biological Cybernetics*, *66*, 419-428. [Cited on p. 63.]

Rosenblatt, F. (1959). *Principles of Neurodynamics.* New York: Spartan Books. [Cited on pp. 23, 26.]

Rosenfeld, A., & Kak, A. C. (1982). *Digital Picture Processing.* New York: Academic Press. [Cited on p. 105.]

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*, 533–536. [Cited on p. 33.]

Rumelhart, D. E., & McClelland, J. L. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* The MIT Press. [Cited on pp. 9, 15.]

Rumelhart, D. E., & Zipser, D. (1985). Feature discovery by competitive learning. *Cognitive Science*, *9*, 75–112. [Cited on p. 57.]

Sanger, T. D. (1989). Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural Networks*, *2*, 459–473. [Cited on p. 99.]

Sejnowski, T. J., & Rosenberg, C. R. (1986). *NETtalk: A Parallel Network that Learns to Read Aloud* (Tech. Rep. Nos. JHU/EECS–86/01). The John Hopkins University Electrical Engineering and Computer Science Department. [Cited on p. 45.]

Silva, F. M., & Almeida, L. B. (1990). Speeding up backpropagation. In R. Eckmiller (Ed.), *Advanced Neural Computers* (pp. 151–160). North-Holland. [Cited on p. 42.]

Singer, A. (1990). *Implementations of Artificial Neural Networks on the Connection Machine* (Tech. Rep. Nos. RL90–2). Cambridge, MA: Thinking Machines Corporation. [Cited on p. 113.]

Smagt, P. van der, Groen, F., & Kröse, B. (1993). *Robot Hand-Eye Coordination Using Neural Networks* (Tech. Rep. Nos. CS–93–10). Department of Computer Systems, University of Amsterdam. (ftp'able from `archive.cis.ohio-state.edu`) [Cited on p. 90.]

Smagt, P. van der, Kröse, B. J. A., & Groen, F. C. A. (1992). Using time-to-contact to guide a robot manipulator. In *Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 177–182). IEEE. [Cited on p. 89.]

Smagt, P. P. van der, & Kröse, B. J. A. (1991). A real-time learning neural robot controller. In T. Kohonen, K. Mäkisara, O. Simula, & J. Kangas (Eds.), *Proceedings of the 1991 International Conference on Artificial Neural Networks* (pp. 351–356). North-Holland/Elsevier Science Publishers. [Cited on pp. 89, 90.]

Sofge, D., & White, D. (1992). Applied learning: optimal control for manufacturing. In D. Sofge & D. White (Eds.), *Handbook of Intelligent Control, Neural, Fuzzy, and Adaptive Approaches.* Van Nostrand Reinhold, New York. (In press) [Cited on p. 76.]

Stoer, J., & Bulirsch, R. (1980). *Introduction to Numerical Analysis.* New York–Heidelberg–Berlin: Springer-Verlag. [Cited on p. 41.]

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*, 9-44. [Cited on p. 75.]

Sutton, R. S., Barto, A., & Wilson, R. (1992). Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems*, *6*, 19-22. [Cited on p. 80.]

Theeten, J. B., Duranton, M., Mauduit, N., & Sirat, J. A. (1990). The LNeuro chip: A digital VLSI with on-chip learning mechanism. In *Proceedings of the International Conference on Neural Networks* (Vol. I, pp. 593–596). DUNNO. [Cited on p. 119.]

Tomlinson, M. S., Jr., & Walker, D. J. (1990). DNNA: A digital neural network architecture. In *Proceedings of the International Conference on Neural Networks* (Vol. II, pp. 589–592). DUNNO. [Cited on p. 109.]

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279-292. [Cited on pp. 76, 80, 81.]

Werbos, P. (1992). Approximate dynamic programming for real-time control and neural modelling. In D. Sofge & D. White (Eds.), *Handbook of Intelligent Control, Neural, Fuzzy, and Adaptive Approaches.* Van Nostrand Reinhold, New York. [Cited on pp. 76, 80.]

Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Unpublished doctoral dissertation, Harvard University. [Cited on p. 33.]

Werbos, P. W. (1990). A menu for designs of reinforcement learning over time. In W. T. M. III, R. S. Sutton, & P. J. Werbos (Eds.), *Neural Networks for Control.* MIT Press/Bradford. [Cited on p. 77.]

White, D., & Jordan, M. (1992). Optimal control: a foundation for intelligent control. In D. Sofge & D. White (Eds.), *Handbook of Intelligent Control, Neural, Fuzzy, and Adaptive Approaches.* Van Nostrand Reinhold, New York. [Cited on p. 80.]

Widrow, B., & Hoff, M. E. (1960). Adaptive switching circuits. In *1960 IRE WESCON Convention Record* (pp. 96–104). DUNNO. [Cited on pp. 23, 27.]

Widrow, B., Winter, R. G., & Baxter, R. A. (1988). Layered neural nets for pattern recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *36*(7), 1109-1117. [Cited on p. 98.]

Wilson, G. V., & Pawley, G. S. (1988). On the stability of the travelling salesman problem algorithm of Hopfield and tank. *Biological Cybernetics*, *58*, 63–70. [Cited on p. 53.]

Index

R

S

T

U