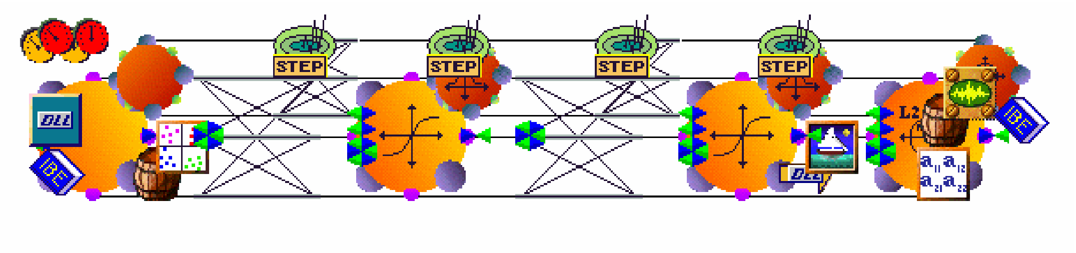
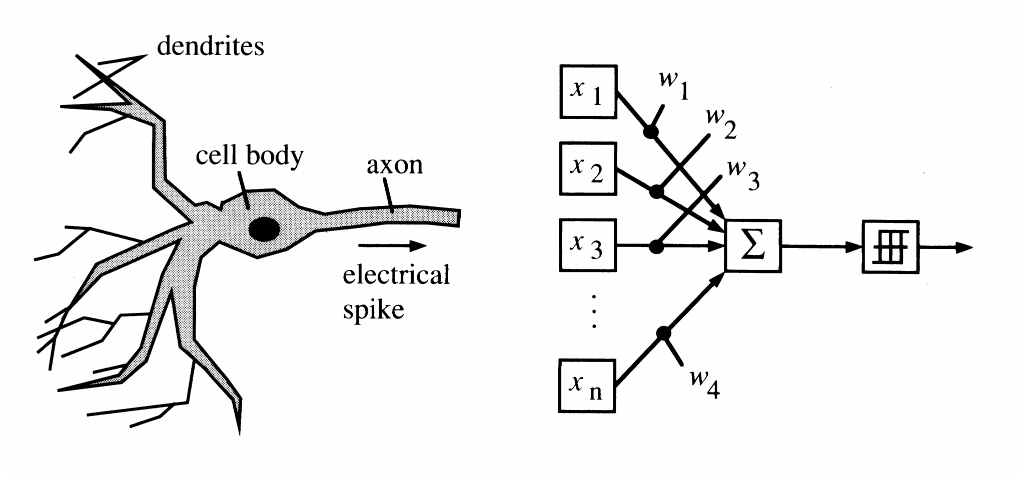
NEURAL NETWORKS

M. Hajek





2005

Table of contents

1 Introduction

1.1 What is a neural network?

Work on artificial neural networks, commonly referred to as *neural networks*, has been motivated right from its inception by the recognition that the brain computes in an entirely different way from the conventional digital computer. The struggle to understand the brain owes much to the pioneering work of Ramón y Cajál (1911), who introduced the idea of neurons as structural constituents of the brain. Typically, neurons are five to six orders of magnitude slower than silicon logic gates; events in a silicon chip happen in the nanosecond (10^{-9} s) range, whereas neural events happen in the millisecond (10^{-3} s) range. However, the brain makes up for the relatively slow rate of operation of a neuron by having a truly staggering number of neurons (nerve cells) with massive interconnections between them. It is estimated that there must be on the order of 10 billion neurons in the human cortex, and 60 trillion synapses or connections. The net result is that the brain is an enormously efficient structure. Specifically, the energetic efficiency of the brain is approximately 10^{-16} joules (J) per operation, whereas the corresponding value for the best computers (in 1994) is about 10^{-6} joules per operation.

Table 1-1: Parameters of biological and artificial neural networks.

	Biological NN	Artificial NN
Number of neurons	10^{10}	‘000
Number of synapses	60×10^{12}	‘000
Speed [s/op]	10^{-3}	10^{-9}
Energy [J/op]	10^{-16}	10^{-6}

The brain is a highly *complex*, *nonlinear*, and *parallel* information-processing system. It has the capability of organizing neurons so as to perform certain computations (e.g. pattern recognition, perception, and motor control) many times faster than the fastest digital computer. Consider, for example, human vision, which is an information-processing task. It is the function of the visual system to provide a representation of the environment around us and, more important, to supply the information we need to interact with the environment. To be specific, the brain routinely accomplishes perceptual recognition tasks (e.g. recognizing a familiar face embedded in an unfamiliar scene) in something of the order of 100-200 ms.

For another example, consider the sonar of a bat. Sonar is an active echo-location system. In addition to providing information about how far away a target (e.g. a flying insect) is bat sonar conveys information about the relative velocity of the target, the size of the target, the size of various features of the target, and the azimuth and elevation of the target. The complex neural computations needed to extract all this information from the target echo occur within a brain the size of a plum.

How, then, does a human brain or the brain of a bat do it? At birth, a brain has great structure and the ability to build up its own rules through what we usually refer to as *experience*. Indeed, experience is built up over the years, with the most dramatic development (i.e. hardwiring) of the human brain taking place in the first two years from birth; but the development continues well beyond that stage. During this early stage of development, about one million synapses are formed per second.

Synapses are elementary structural and functional units that mediate the interactions between neurons. The most common kind of synapse is a chemical synapse, which operates as follows. A presynaptic process liberates a transmitter substance that diffuses across the synaptic junction between neurons and then acts on a postsynaptic process. Thus a synapse converts a presynaptic electrical signal into a chemical signal and then back into a postsynaptic electrical signal. In traditional descriptions of neural organization, it is assumed that a synapse is a simple connection that can impose excitation or inhibition on the receptive neuron.

A developing neuron is synonymous with a plastic brain: *Plasticity* permits the developing nervous system to adapt to its surrounding environment. In an adult brain, plasticity may be accounted for by two mechanisms: the creation of new synaptic connections between neurons, and the modification of existing synapses.

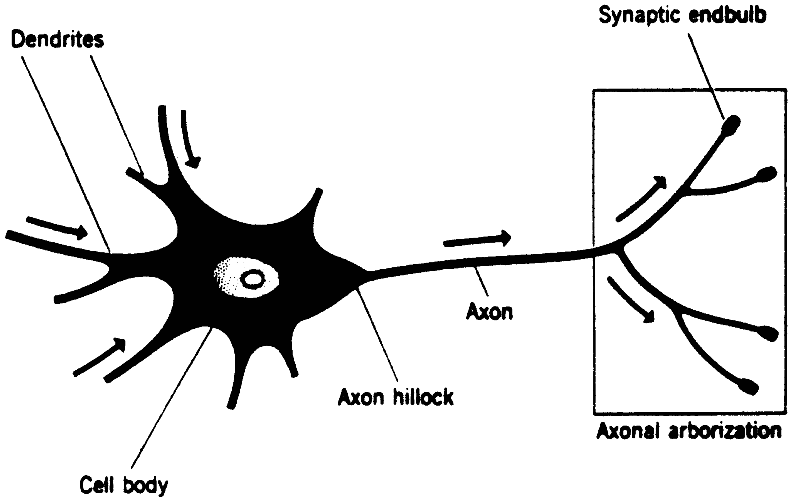


Figure 1-1: Biological neuron.

Axons act as transmission lines, and dendrites represent receptive zones. Neurons come in a wide variety of shapes and sizes in different parts of the brain. A pyramidal cell can receive 10,000 or more synaptic contacts and it can project onto thousands of target cells.

Just as plasticity appears to be essential to the functioning of neurons as information processing units in the human brain, so it is with neural networks made up of artificial neurons. *In its most general form, a neural network is a machine that is designed to model the way in which the brain performs a particular task or function of interest; the network is usually implemented using electronic components or simulated in software on a digital computer.* Our interest will be confined largely to neural networks that perform useful computations through a process of *learning*. To achieve good performance, neural networks employ a massive interconnection of simple computing cells referred to as *neurons* or *processing units*. We may thus offer the following definition of a neural network viewed as an *adaptive machine*:

A neural network is a massively parallel distributed processor that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:

1. *Knowledge is acquired by the network through a learning process.*
2. *Interneuron connection strengths known as synaptic weights are used to store the knowledge.*

The procedure used to perform the learning process is called a *learning algorithm*.

Neural networks are also referred to as neurocomputers, connectionist networks, parallel distributed processors, etc.

1.2 Benefits of neural networks

It is apparent from the above discussion that a neural network derives its computing power through, first, its *massively parallel distributed structure* and, second, its *ability to learn* and, therefore, *generalize*. Generalization refers to the neural network producing reasonable outputs for inputs not encountered during training (learning). These two information-processing capabilities make it possible for neural networks to solve complex (large-scale) problems that are currently intractable. In practice, however, neural networks cannot provide the solution working by themselves alone. Rather, they need to be integrated into a consistent system engineering approach. Specifically, a complex problem of interest is decomposed into a number of relatively simple tasks, and neural networks are assigned a subset of the tasks (e.g. pattern recognition, associative memory, control) that match their inherent capabilities. It is important to recognize, however, that we have a long way to go (if ever) before we can build a computer architecture that mimics a human brain. The use of neural networks offers the following useful properties and capabilities:

1. Nonlinearity. A neuron is basically a nonlinear device. Consequently, a neural network, made up of an interconnection of neurons, is itself nonlinear. Moreover, the nonlinearity is of a special kind in the sense that it is distributed throughout the network.
2. Input-output mapping. A popular paradigm of learning called *supervised learning* involves the modification of the synaptic weights of a neural network by applying a set of training samples. Each sample consists of a unique input signal and the corresponding desired response. The network is presented a sample picked at random from the set, and the synaptic weights (free parameters) of the network are modified so as to minimize the difference between the desired response and the actual response of the network produced by the input signal in accordance with an appropriate criterion. The training of the network is repeated for many samples in the set until the network reaches a steady state, where there are no further significant changes in the synaptic weights. The previously applied training samples may be reapplied during the training session, usually in a different order. Thus the network learns from the samples by constructing an input-output mapping for the problem at hand.
3. Adaptivity. Neural networks have a built-in capability to adapt their synaptic weights to changes in the surrounding environment. In particular, a neural network trained to operate in a specific environment can be easily retrained to deal with minor changes in the operating environmental conditions. Moreover, when it is operating in a nonstationary environment a neural network can be designed to change its synaptic weights in real time.

The natural architecture of a neural network for pattern classification, signal processing, and control applications, coupled with the adaptive capability of the network, makes it an ideal tool for use in adaptive pattern classification, adaptive signal processing, and adaptive control.

4. Contextual information. Knowledge is represented by the very structure and activation state of a neural network. Every neuron in the network is potentially affected by the global activity of all other neurons in the network. Consequently, contextual information is dealt with naturally by a neural network.
5. Fault tolerance. A neural network, implemented in hardware form, has the potential to be inherently fault tolerant in the sense that its *performance is degraded gracefully* under adverse operating. For example, if a neuron or its connecting links are damaged, recall of a stored pattern is impaired in quality. However, owing to the distributed nature of information in the network, the damage has to be extensive before the overall response of the network is degraded seriously. Thus, in principle, a neural network exhibits a graceful degradation in performance rather than catastrophic failure.
6. VLSI implementability. The massively parallel nature of a neural network makes it potentially fast for the computation of certain tasks. This same feature makes a neural network ideally suited for implementation using very-large-scale-integrated (VLS1) technology.
7. Uniformity of analysis and design. Basically, neural networks enjoy universality as information processors. We say this in the sense that the same notation is used in all the domains involving the application of neural networks. This feature manifests itself in different ways:
    a) Neurons, in one form or another, represent an ingredient *common* to all neural networks.
    b) This commonality makes it possible to *share* theories and learning algorithms in different applications of neural networks.
    c) Modular networks can be built through a *seamless integration of modules*.
8. Neurobiological analogy. The design of a neural network is motivated by analogy with the brain, which is a living proof that fault-tolerant parallel processing is not only physically possible but also fast and powerful. *Neurobiologists* look to (artificial) neural networks as a research tool for the interpretation of neurobiological phenomena. On the other hand, *engineers* look to neurobiology for new ideas to solve problems more complex than those based on conventional hard-wired design techniques. The neurobiological analogy is also useful in another important way: It provides a hope and belief that physical understanding of neurobiological structures could influence the art of electronics and thus VLSI.

1.3 Recommended Literature

1.3.1 Books

Freeman J.A., Skapura D.M.: *Neural networks - Algorithms, applications, and programming techniques*, Addison-Wesley, Reading, MA 1991
A good book for beginning programmers who want to learn how to write NN programs while avoiding any understanding of what NNs do or why they do it.

Hertz J., Krogh A., Palmer R.G.: *Introduction to the theory of neural computation Addison-Wesley*, Redwood City, CA 1991

This is an excellent classic work on neural nets from the perspective of physics covering a wide variety of networks. It provides a good balance of model development, computational algorithms, and applications.

Zurada J.M.: *Introduction to artificial neural systems*, West Publishing Company, St. Paul 1992
Cohesive and comprehensive book on neural nets as an engineering-oriented introduction, but also as a research foundation. Thorough exposition of fundamentals, theory and applications. Training and recall algorithms appear in boxes showing steps of algorithms, thus making programming of learning paradigms easy. Many illustrations and intuitive examples.

Freeman J.A.: *Simulating neural networks with Mathematica*, Addison-Wesley, Reading, MA 1994
Helps the reader make his own NNs. The Mathematica code for the programs in the book is also available through the internet.

Haykin S.: *Neural networks – A comprehensive Foundation* (2nd ed.), Prentice Hall, Upper Saddle River, 1999
The second edition is much better than the first (1994). It covers more topics, is easier to understand, and has better examples.

Principe J.C., Euliano N.R., Lefebvre W.C.: *Neural and adaptive systems – Fundamentals through simulations*, Wiley, New York, 2000
Includes a CD with the network simulation environment NeuroSolutions and many examples.

Duda R.O., Hart P.E., Stork D.G.: *Pattern classification*, Wiley, New York, 2001
This is a defining book for the field of pattern recognition with many problems and computer exercises.

Russell S., Norvig P.: *Artificial intelligence – A modern approach* (2nd ed.), Prentice Hall, Upper Saddle River, 2003
The book is a major step forward not only for the teaching of AI but also for the unified view of the field of AI. There are important insights in almost every chapter even for experts in the field.

1.3.2 Web sites

ftp://ftp.sas.com/pub/neural/FAQ.html
Copyright 1997, 1998, 1999, 2000, 2001, 2002 by Warren S. Sarle, Cary, NC, USA
http://www.statsoft.com/textbook/stneunet.html
Copyright StatSoft, Inc., 1984-2003
http://www.nd.com/nnreference.htm
Copyright © 2002 NeuroDimensions, Inc.
http://www.calresco.org/links.htm
Page Version 4.83 April 2005
http://bubl.ac.uk/link/n/neuralnetworks.htm
Page last updated: 17 March 2003

1.4 Brief history

1943	McCulloch, Pitts	Neuron
1948	Wiener	Cybernetics
1949	Hebb	Learning
1954	Minsky	Neural network
1956	Taylor	Associative memory
1958	Rosenblatt	Perceptron
1960	Widrow, Hoff	Adaline
1969	Minsky, Papert	Perceptron limits
	Lower research activity	
1980	Grossberg	ART
1982	Hopfield	Energy function
1982	Kohonen	SOM
1986	Rumelhart	Back-propagation
1988	Broomhead	RBF
1989	Mead	VLSI and NN

1.5 Models of a neuron

A *neuron* is an information-processing unit that is fundamental to the operation of a neural network. We may identify three basic elements of the neuron model:

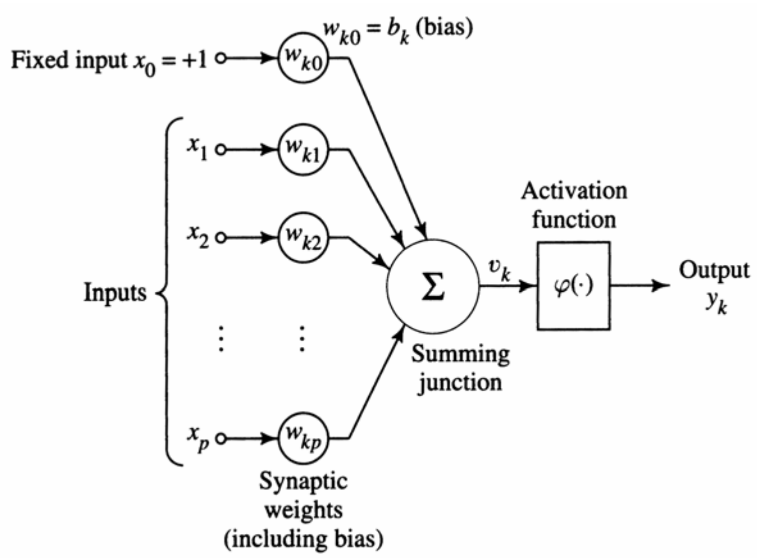


Figure 1-2: Nonlinear model of a neuron.

1. A set of *synapses*, each of which is characterized by a *weight* or *strength* of its own. Specifically, a signal x_j at the input of synapse j connected to neuron k is multiplied by the synaptic weight w_{kj}. It is important to make a note of the manner in which the subscripts of the synaptic weight w_{kj} are written. The first subscript refers to the neuron in question and the second subscript refers to the input end of the synapse to which the weight refers.

The weight w_{kj} is positive if the associated synapse is excitatory; it is negative if the synapse is inhibitory.

2. An *adder* for summing the input signals, weighted by the respective synapses of the neuron.
3. An *activation function* for limiting the amplitude of the output of a neuron. The activation function is also referred to in the literature as a *squashing function* in that it squashes (limits) the permissible amplitude range of the output signal to some finite value. Typically, the normalized amplitude range of the output of a neuron is written as the closed unit interval [0, 1] or alternatively [-1, 1].

The model of a neuron also includes an externally applied bias (threshold) $w_{k0} = b_k$ that has the effect of lowering or increasing the net input of the activation function.

In mathematical terms, we may describe a neuron *k* by writing the following pair of equations:

$$v_k = \sum_{j=0}^{p} w_{kj} x_j \tag{1.1}$$

$$y_k = \varphi(v_k) \tag{1.2}$$

or in a matrix form

$$v_k = \begin{bmatrix} w_{k0} & w_{k1} & \cdots & w_{kp} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_p \end{bmatrix} = w_k^T x \tag{1.3}$$

1.6 Types of activation functions

1.6.1 Threshold activation function (McCulloch–Pitts model)

$$\varphi(v) = \begin{Bmatrix} 1, & v \geq 0 \\ 0, & v < 0 \end{Bmatrix} \tag{1.4}$$

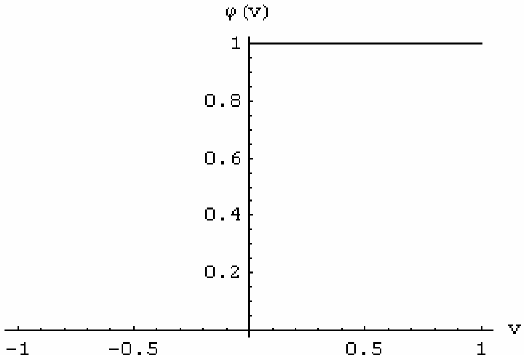


Figure 1-3: Threshold activation function.

In this model, the output of a neuron takes on the value of 1 if the total internal activity level of that neuron is nonnegative and 0 otherwise. This statement describes the all-or-none property of the McCulloch–Pitts model.

The McCulloch–Pitts model of a neuron is sometimes used with the output either -1 or +1:

$$\varphi(v) = \begin{Bmatrix} 1, & v \geq 0 \\ -1, & v < 0 \end{Bmatrix} \tag{1.5}$$

1.6.2 Piecewise-linear activation function

$$\varphi(v) = \begin{Bmatrix} 1, & v \geq \frac{1}{2} \\ v + \frac{1}{2}, & -\frac{1}{2} < v < \frac{1}{2} \\ 0, & v \leq -\frac{1}{2} \end{Bmatrix} \tag{1.6}$$

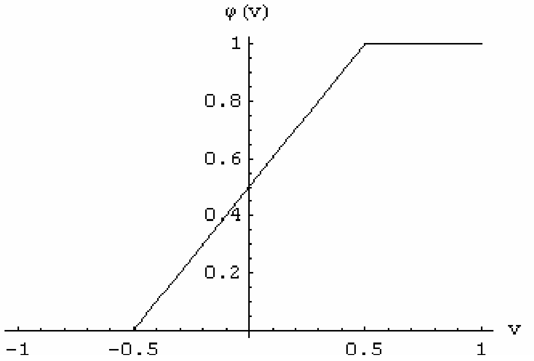


Figure 1-4: Piecewise-linear activation function

The amplification factor inside the linear region is assumed to be unity. The following two situations may be viewed as special forms of the piecewise linear function:

1. A *linear combiner* arises if the linear region of operation in maintained without running into saturation.
2. The piecewise-linear function reduces to a threshold function if the amplification factor of the linear region in made infinitely large.

1.6.3 Sigmoid (logistic) activation function

$$\varphi(v) = \frac{1}{1 + \exp(-v)} \tag{1.7}$$

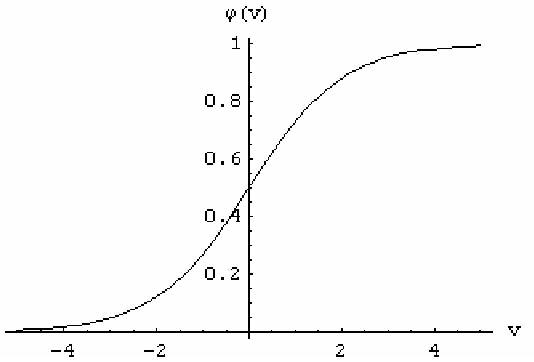


Figure 1-5: Sigmoid (logistic) activation function.

The sigmoid function is the most common form of activation function used in the construction of artificial neural networks. Whereas a threshold function assumes the value of 0 or 1, a sigmoid function assumes a continuous range of values form 0 and 1. Note also that the sigmoid function is differentiable, which is an important feature of neural network theory.

The derivative of the sigmoid function has a nice property, which makes is easy to calculate:

$$\frac{\partial\varphi(v)}{\partial v}=\varphi(v)(1-\varphi(v)) \qquad (1.8)$$

1.6.4 Hyperbolic tangent function

$$\varphi(v)=\tanh(\frac{v}{2})=\frac{1-\exp(-v)}{1+\exp(-v)} \qquad (1.9)$$

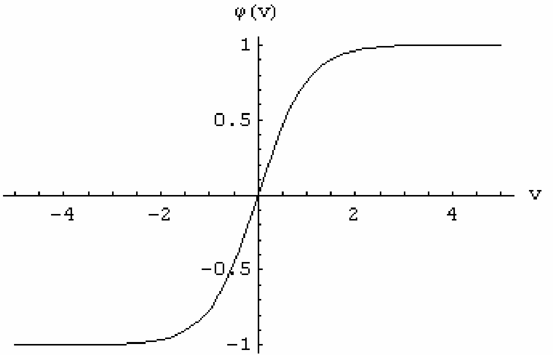


Figure 1-6: Hyperbolic tangent activation function.

The hyperbolic tangent function can be easily expressed in terms of the logistic function: (2 × logistic function – 1). Its derivative is also easy to calculate:

$$\frac{\partial\varphi(v)}{\partial v}=\frac{1}{2}(1+\varphi(v))(1-\varphi(v)) \qquad (1.10)$$

1.6.5 Softmax activation function

One approach toward approximating probabilities is to choose the output neuron nonlinearity to be exponential rather than sigmoidal and for each pattern to normalize the outputs to sum to 1. Let c be the number of output neurons. Each output is generated by the following activation function:

$$y_i = \frac{e^{v_i}}{\sum_{i=1}^{c} e^{v_i}}, \quad i = 1, 2, \ldots, c \tag{1.11}$$

It is clear that $\sum_{i=1}^{c} y_i = 1$. The softmax activation function is a smoothed version of a *winner-take-all* nonlinearity in which the maximum output is transformed to 1, and all others reduced to 0.

The use of softmax is appropriate when the network is to be used for estimating probabilities. For example, each output y_i can represent a probability that the corresponding neural network input belongs to class C_i.

1.7 Multilayer feedforward network

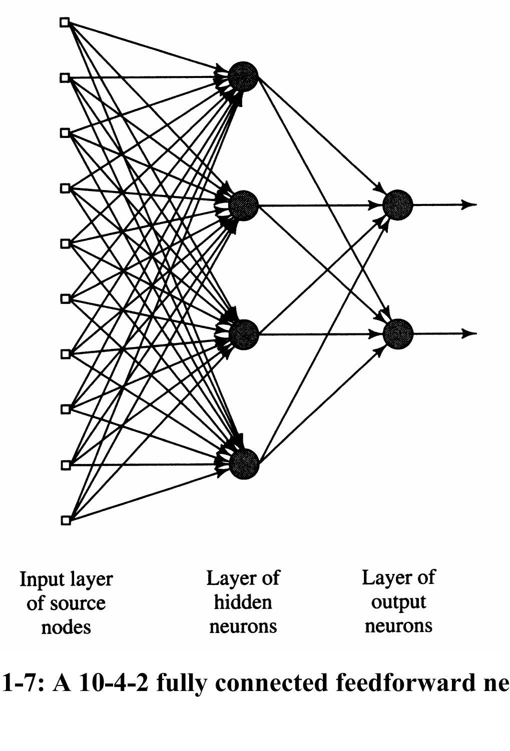


Figure 1-7: A 10-4-2 fully connected feedforward network.

The source nodes in the *input layer* of the network supply respective elements of the activation pattern (input vector), which constitute the input signals applied to the neurons (computation nodes) in the second layer (i.e. the first *hidden layer*). The output signals of the second layer are used as inputs to the third layer, and so on for the rest of the network. Typically, the neurons in each layer of the network have as their inputs the output signals of the preceding layer only. The set of output signals of the neurons in the *output layer* of the network constitutes the overall response of the network to the activation pattern supplied by the source nodes in the input layer. For brevity, the network shown above is referred to as a 10-4-2 network in that it has 10 source nodes, 4 hidden neurons, and 2 output neurons. As another example, a feedforward network with p source nodes, h_1 neurons in the first hidden layer, h_2 neurons in the second layer, and q neurons in the output layer is referred to as a p-h_1-h_2-q network.

A neural network is said to be *fully connected* if every node in each layer of the network is connected to every other node in the adjacent forward layer. If, however, some of the communication links (synaptic connections) are missing from the network, we say that the network is *partially connected.*

1.8 Problems

1. Draw a diagram of the logistic function $\varphi(v)=\dfrac{1}{1+e^{-av}}$ for various values of the parameter a.

2. A logistic (sigmoid) activation function is defined by $\varphi(v)=\dfrac{1}{1+e^{-v}}$. Show that the derivative of $\varphi(v)$ with respect to v is $\dfrac{d\varphi}{dv}=\varphi(v)(1-\varphi(v))$. What is the value of this derivative at the origin?

3. A tanh activation function is defined by $\varphi(v)=\dfrac{1-e^{-av}}{1+e^{-av}}=\tanh\left(\dfrac{av}{2}\right)$. Show that the derivative of $\varphi(v)$ with respect to v is given by $\dfrac{d\varphi}{dv}=\dfrac{a}{2}\left[1-\varphi^2(v)\right]$. What is the value of this derivative at the origin? Suppose that the slope parameter a is made infinitely large. What is the resulting form of $\varphi(v)$?

4. Another sigmoid function is the algebraic sigmoid $\varphi(v)=\dfrac{v}{\sqrt{1+v^2}}$. Draw a graph of this function and show that $\dfrac{d\varphi}{dv}=\dfrac{\varphi^3(v)}{v^3}$. Compare this graph with the graph of the hyperbolic tangent function (1.9).

5. A neuron has an activation function $\varphi(v)$ defined by the logistic function $\varphi(v)=\dfrac{1}{1+e^{-av}}$, where the slope parameter a is available for adjustment. Let $x_1, x_2, \ldots, x_m$ denote the input signals applied to the source nodes of the neuron. For convenience of presentation, we would like to absorb the slope parameter a in the sum of signals v by writing $\varphi(v)=\dfrac{1}{1+e^{-v}}$. How would you modify the inputs $x_1, x_2, \ldots, x_m$ to produce the same output as before?

6. A neuron *j* receives inputs from four other neurons whose output activity levels are -0.8, 0.2, 0.4, and -0.9. The respective synaptic weights of neuron *j* are 0.5, 0.3, 1.0, and -0.6. Assume that the bias applied to the neuron is zero. Calculate the output of neuron *j* for the following three situations:

   a) The neuron is linear.
   b) The neuron is represented by a McCulloch-Pitts model.
   c) The neuron has the logistic (sigmoid) activation function.

7. A fully connected feedforward network has 10 source nodes, 2 hidden layers, one with 4 neurons and the other with 3 neurons, and a single output neuron. Construct an architectural graph of this network.

8. Consider a multilayer feedforward network, all the neurons of which operate in their linear regions. Justify the statement that such a network is equivalent to a single layer feedforward network (i.e. a network with no hidden layer).

9. Figure 1-8 shows the signal-flow graph of a recurrent network made up of two neurons with self-feedback. Write the coupled system of two first-order nonlinear difference equations that describe the operation of the system. Implement the model e.g. in Mathematica and investigate the impact of synaptic connections on the stability of the recurrent network.

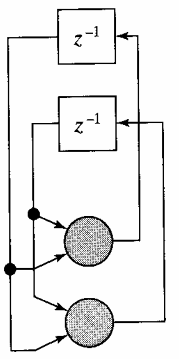


Figure 1-8: Recurrent network.

10. Construct a fully recurrent network with 5 neurons, but with no self-feedback.

11. A recurrent network has 3 source nodes, 2 hidden neurons, and 4 output neurons. Construct an architectural graph that describes such a network.

2 Learning process

Among the many interesting properties of a neural network is the ability of the network to *learn* from its environment and to improve its performance through learning. A neural network learns about its environment through an iterative process of adjustments applied to its synaptic weights and thresholds.

We define learning in the context of neural networks as follows:

Learning is a process by which the free parameters of a neural network are adapted through a continuing process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place.

This definition of the learning process implies the following sequence of events:

1. The neural network is stimulated by an environment.
2. The neural network undergoes changes as a result of this stimulation.
3. The neural network responds in a new way to the environment, because of the changes that have occurred in its internal structure.

Let $w_{kj}(n)$ denote the value of the synaptic weight w_{kj} at time n. At time n an *adjustment* $\Delta w_{kj}(n)$ is applied to the synaptic weight $w_{kj}(n)$, yielding the updated value

$$w_{kj}(n+1) = w_{kj}(n) + \Delta w_{kj}(n) \qquad (2.1)$$

A prescribed set of well-defined rules for the solution of a learning problem is called a *learning algorithm*. As one would expect, there is no unique learning algorithm for the design of neural networks. Rather, we have a "kit of tools" represented by a diverse variety of learning algorithms, each of which offers advantages of its own. Basically, learning algorithms differ from each other in the way in which the adjustment Δw_{kj} to the synaptic weight w_{kj} is formulated.

2.1 Error-correction learning

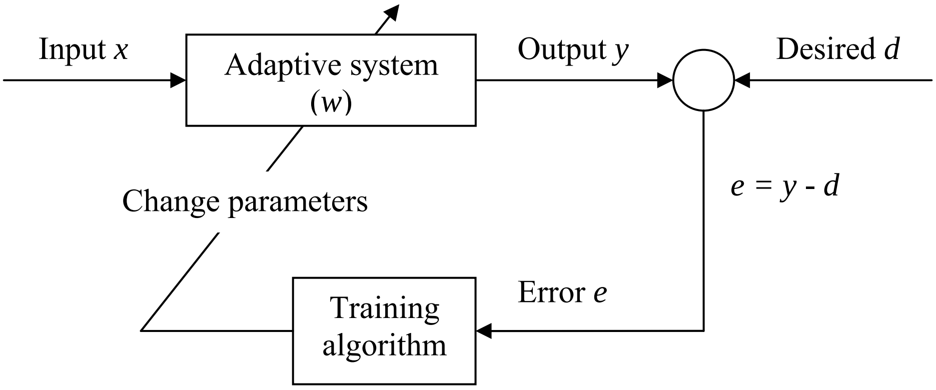


Figure 2-1: Error-correction learning diagram.

Let $d_k(n)$ denote some desired response or target response for neuron k at time n. Let the corresponding value of the actual response (output) of this neuron be denoted by $y_k(n)$. The response $y_k(n)$ is produced by a stimulus applied to the input of the network in which neuron k is embedded. The input vector and desired response $d_k(n)$ constitute a particular sample presented to the network at time n.

Typically, the actual response $y_k(n)$ of neuron k is different from the desired response $d_k(n)$. Hence, we may define an error signal

$$e_k(n) = y_k(n) - d_k(n) \tag{2.2}$$

The ultimate purpose of error-correction learning is to minimize a *cost function* based on the error signal $e_k(n)$.

A criterion commonly used for the cost function is the instantaneous value of the mean-square-error criterion

$$J(n) = \frac{1}{2}\sum_k e_k^2(n) \tag{2.3}$$

The network is then optimized by minimizing $J(n)$ with respect to the synaptic weights of the network. Thus, according to the *error-correction learning rule* (or *delta rule*), the synaptic weight adjustment is given by

$$\Delta w_{kj} = \eta e_k(n) x_j(n) \tag{2.4}$$

2.2 Hebbian learning

Hebb’s postulate of learning is the oldest and most famous of all learning rules:

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e. synchronously), then the strength of that synapse is selectively increased.
2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

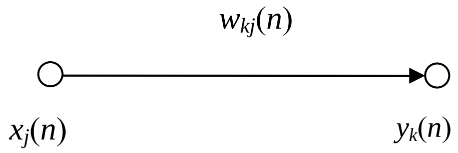


Figure 2-2: Synaptic connection.

To formulate Hebb's postulate of learning in mathematical terms, consider a synaptic weight w_{kj} with presynaptic and postsynaptic activities denoted by x_j and y_k, respectively. According to Hebb's postulate, the adjustment applied to the synaptic weight w_{kj} at time n is

$$\Delta w_{kj}(n) = F(y_k(n), x_j(n)) \tag{2.5}$$

As a special case we may use the activity *product rule*

$$\Delta w_{kj}(n) = \eta y_k(n) x_j(n) \tag{2.6}$$

where η is a positive constant that determines the *rate of learning*. This rule clearly emphasizes the correlational nature of a Hebbian synapse.

From this representation we see that the repeated application of the input signal x_j leads to an exponential growth that finally drives the synaptic weight w_{kj} into saturation.

$$\begin{aligned} w_{kj}(n+1) &= w_{kj}(n) + \eta y_k(n) x_j(n) \\ &= w_{kj}(n)(1+\eta x_j^2(n)) \end{aligned} \tag{2.7}$$

$$\begin{aligned} w_{kj}(n+2) &= w_{kj}(n+1) + \eta y_k(n+1) x_j(n+1) \\ &= w_{kj}(n)(1+\eta x_j^2(n))(1+\eta x_j^2(n+1)) \end{aligned} \tag{2.8}$$

If x_j stays constant then

$$w_{kj}(n+N) = w_{kj}(n)(1+\eta x_j^2)^N \tag{2.9}$$

To avoid such a situation from arising, we need to impose a limit on the growth of synaptic weights. One method for doing this is to introduce a nonlinear *forgetting factor* into the formula for the synaptic adjustment $\Delta w_{kj}(n)$. Specifically, we redefine $\Delta w_{kj}(n)$ as a *generalized activity product rule*:

$$\begin{aligned}\Delta w_{kj}(n) &= \eta y_k(n)x_j(n) - \alpha y_k(n)w_{kj}(n) \\ &= \alpha y_k(n)[cx_j(n) - w_{kj}(n)]\end{aligned} \qquad (2.10)$$

Where $c = \eta/\alpha$. If the weight $w_{kj}(n)$ increases to the point where

$$cx_j(n) - w_{kj}(n) = 0 \qquad (2.11)$$

a *balance point* is reached and the weight update stops.

2.3 Supervised learning

An essential ingredient of supervised is the availability of an external teacher, which is able to provide the neural network with a *desired* or *target response*. The network parameters are adjusted under the combined influence of the training vector and the error signal. This adjustment is carried out iteratively in a step-by-step fashion with the aim of eventually making the neural network emulate the teacher. This form of supervised learning is in fact an error-correction learning, which was already described.

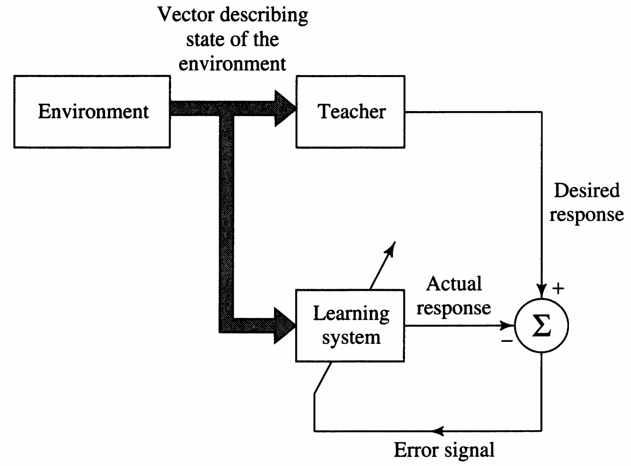


Figure 2-3: Supervised learning.

2.4 Unsupervised learning

In *unsupervised* or *self-organized* learning there is no external teacher to oversee the learning process. In other words, there are no specific samples of the function to be learned by the network. Rather, provision is made for a *task-independent measure* of the quality of representation that the network is required to learn and the free parameters of the network are optimized with respect to that measure. Once the network has become tuned to the statistical regularities of the input data, it develops the ability to form internal representations for encoding features of the input and thereby creates new classes automatically.

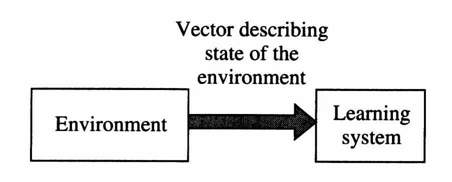


Figure 2-4: Unsupervised learning.

2.5 Learning tasks

In previous sections we have discussed different learning paradigms. In this section, we describe some basic learning tasks. The choice of a particular learning algorithm is influenced by the learning task that a neural network is required to perform.

2.5.1 Pattern recognition

Pattern recognition is formally defined as the process whereby a received pattern/signal is assigned to one of a prescribed number of classes (categories). A neural network performs pattern recognition by first undergoing a training session, during which the network is repeatedly presented a set of input patterns along with the category to which each particular pattern belongs. Later, a new pattern is presented to the network that has not been seen before, but which belongs to the same population of patterns used to train the network. The network is able to identify the class of that particular pattern because of the information it has extracted from the training data. Pattern recognition performed by a neural network is statistical in nature, with the patterns being represented by points in a multidimensional decision space. The decision space is divided into regions, each one of which is associated with a class. The decision boundaries are determined by the training process. The construction of these boundaries is made statistical by the inherent variability that exists within and between classes.

In generic terms, pattern-recognition machines using neural networks may take one of two forms:

1. The machine is split into two parts, an unsupervised network for feature extraction and a supervised network for classification, as shown in Figure 2-5a. In conceptual terms, a pattern is represented by a set of m observables, which may be viewed as a point x in an m-dimensional observation (data) space. Feature extraction is described by a transformation that maps the point x into an intermediate point y in a q-dimensional *feature space* with $q < m$, as indicated in Figure 2-5b. This transformation may be viewed as one of dimensionality reduction (i.e. data compression), the use of which is justified on the grounds that it simplifies the task of classification. The classification is itself described as a transformation that maps the intermediate point y into one of the classes in an r-dimensional decision space, where r is the number of classes to be distinguished.
2. The machine is designed as a single multilayer feedforward network using a supervised learning algorithm. In this second approach, the task of feature extraction is performed by the computational units in the hidden layer(s) of the network.

Which of these two approaches is adopted in practice depends on the application of interest.

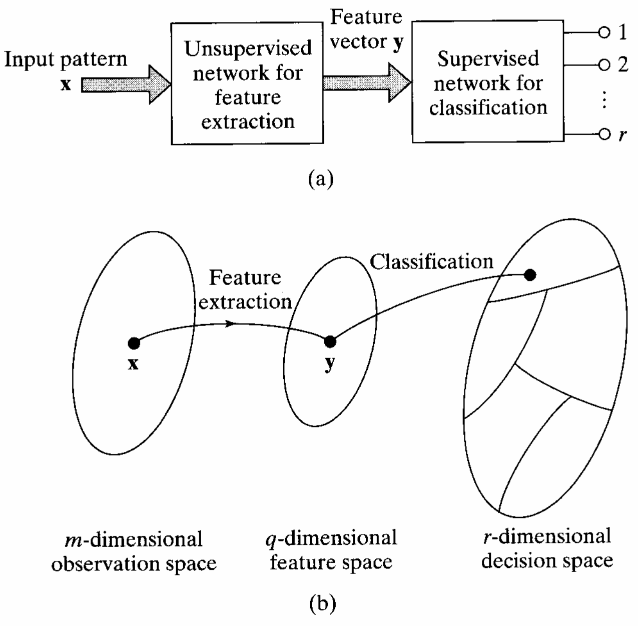

Input pattern
x
Unsupervised network for feature extraction
Feature vector y
Supervised network for classification
1
2
⋮
r
(a)
Feature extraction
Classification
x
y
m-dimensional observation space
q-dimensional feature space
r-dimensional decision space
(b)

Figure 2-5: Classical approach to pattern classification.

2.5.2 Function approximation

Consider a nonlinear input-output mapping described by the functional relationship

$$d = f(x) \tag{2.12}$$

where the vector x is the input and the vector d is the output. The vector-valued function $f(\cdot)$ is assumed to be unknown. To make up for the lack of knowledge about the function $f(\cdot)$, we are given the set of labelled examples:

$$T = \{(x_i, d_i)\}_{i=1}^{N} \tag{2.13}$$

The requirement is to design a neural network that approximates the unknown function $f(\cdot)$ such that the function $F(\cdot)$ describing the input-output mapping actually realized by the network is close enough to $f(\cdot)$ in a Euclidean sense over all inputs, as shown by

$$\|F(x) - f(x)\| < \varepsilon \quad \text{for all } x \tag{2.14}$$

where ε is a small positive number. Provided that the size N of the training set is large enough and the network is equipped with an adequate number of free parameters, then the approximation error ε can be made small enough for the task.

The approximation problem described here is a perfect candidate for supervised learning with x_i playing the role of input vector and d_i serving the role of desired response. We may turn this issue around and view supervised learning as an approximation problem.

2.6 Problems

1. An input signal of unit amplitude is applied repeatedly to a synaptic connection whose initial value is also unity. Calculate the variation in the synaptic weight with time using the following two rules:

   a) Simple form of Hebb's rule described in (2.6), assuming that the learning coefficient $\eta = 0.1$.
   b) Modified form of Hebb's rule described in (2.10), assuming that $\eta = 0.1$ and $c = 0.1$.

2. The Hebbian mechanism described by (2.6) involves the use of positive feedback. Justify the validity of this statement.

3. The delta learning rule (error-correction learning) $\Delta w_{kj}(n) = \eta\, e_k(n) x_j(n)$ and Hebb's learning rule $\Delta w_{kj}(n) = \eta\, y_k(n) x_j(n)$ represent two different methods of learning. Explain how these two learning rules differ from each other.

3 Perceptron

The *perceptron* is the simplest form of a neural network used for the classification of a special type of patterns, which are *linearly separable*. It consists of a single McCulloch-Pitts neuron with adjustable synaptic weights and bias (threshold). Rosenblatt (1958) proved that if the patterns (vectors) used to train the perceptron are drawn from linearly separable classes, then the perceptron algorithm converges and positions the decision surface in the form of a hyperplane between the classes. The proof of convergence of the algorithm is known as the *perceptron convergence theorem.*

The single-layer perceptron shown has a single neuron. Such a perceptron is limited to performing pattern classification with only two classes.

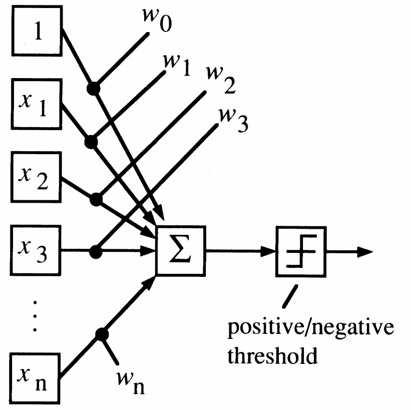


Figure 3-1: Perceptron.

$$v(x) = \sum_{i=0}^{n} w_i x_i = w^T x \tag{3.1}$$

$$y = \begin{cases} 1, & v \geq 0 \\ 0, & v < 0 \end{cases} \tag{3.2}$$

Equation $v(x) = 0$ defines a boundary between the region where the perceptron fires and the region where it outputs zero. This boundary is a line (decision line, decision hyperplane), which must be appropriately located during the process of learning. The perceptron can distinguish between empty and full patterns if and only if they are linearly separable.

Training set T is a set of pairs $[x^i, d^i]$, where the desired value d^i equals either 1 or 0 and $x^i = \begin{bmatrix} x_0^i & x_1^i & \cdots & x_n^i \end{bmatrix}^T$:

$$T = \begin{bmatrix} \begin{bmatrix} x^1 & d^1 \end{bmatrix} \\ \begin{bmatrix} x^2 & d^2 \end{bmatrix} \\ \begin{bmatrix} x^N & d^N \end{bmatrix} \end{bmatrix} \tag{3.3}$$

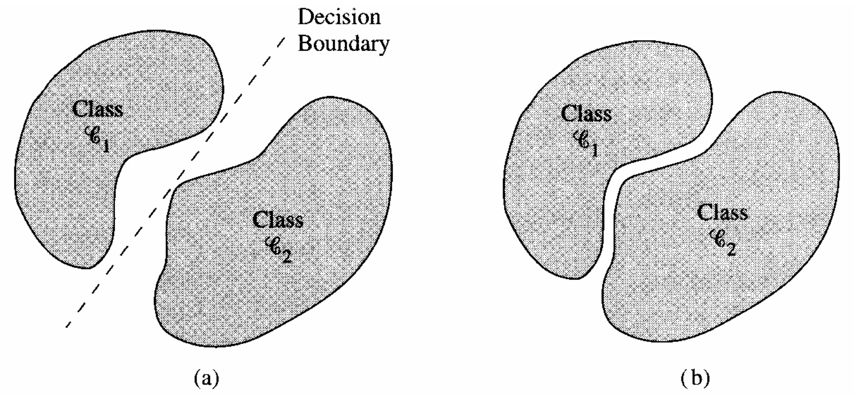


Figure 3-2: Linearly separable patterns (a) and nonlinearly separable patterns (b).

The training instance (sample) [x^m, d^m] is *misclassified* if the perceptron output y^m does not produce d^m.

An appropriate measure of misclassification is this criterion:

$$J = \sum_{i=1}^{N} (y^i - d^i) v(x^i) = \sum_{i=1}^{N} (y^i - d^i) w^T x^i \tag{3.4}$$

Let us discuss individual terms in the criterion J.

1. If the training sample is correctly classified then $(y^i - d^i) = 0$.
2. If $d^m = 1$ and $y^m = 0$ then $(y^m - d^m) = -1$ and $(y^m - d^m)v(x^m) > 0$. (It follows from (3.2) that the output $y^m = 0$ only if $v(x^m) < 0$.)
3. If $d^m = 0$ and $y^m = 1$ then $(y^m - d^m) = 1$ and $(y^m - d^m)v(x^m) \geq 0$. (It follows from (3.2) that the output $y^m = 1$ only if $v(x^m) > 0$.)

We see from the above, that the criterion J grows if training samples are misclassified and $J = 0$ if all samples are classified correctly.

As we have already seen, the equation $v(x) = 0$ defines the decision boundary. If x^1 and x^2 are both on the decision surface, then

$$w^T x^1 = w^T x^2 \tag{3.5}$$

or

$$w^T (x^1 - x^2) = 0 \tag{3.6}$$

and this shows that w is normal to any vector lying in the decision surface. The discriminant function $v(x)$ gives an algebraic measure of the distance from x to the hyperplane (decision surface). Perhaps the easiest way to see it is to express x as

$$x = x^p + r \frac{w}{\|w\|} \tag{3.7}$$

where x^p is the normal projection of x onto hyperplane H and r is the desired algebraic distance – positive if x is on the positive side and negative if x is on the negative side. Then, because $v(x^p) = w^T x^p = 0$,

$$\begin{aligned} v(x) &= w^T x \\ &= w^T (x^p + r\frac{w}{\|w\|}) \\ &= r\frac{w^T w}{\|w\|} = r\|w\| \end{aligned} \tag{3.8}$$

or

$$r = \frac{v(x)}{\|w\|} \tag{3.9}$$

In other words, $v(x)$ is proportional to the distance of x from the decision surface as shown in Figure 3-3.

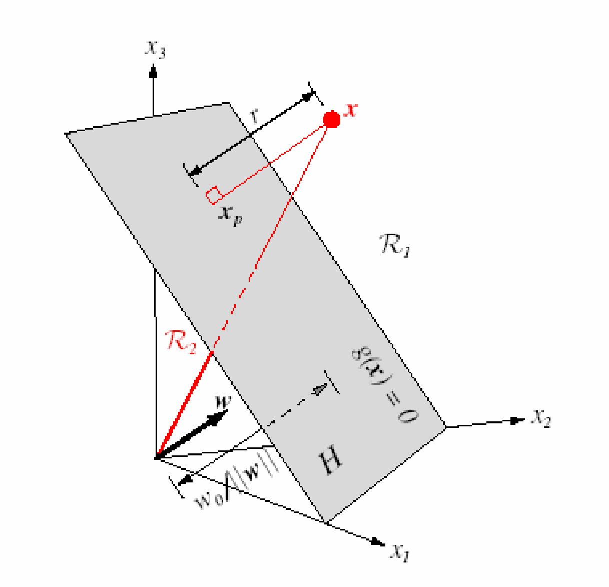


Figure 3-3: The linear decision boundary *H* separates the feature space into two half spaces *R*$_1$ and *R*$_2$.

A *gradient descent method* will be used to minimize the criterion *J*. Let *w*(*k*) be the *k*-th iteration of the weight vector *w*. The gradient descent method is based on the formula

$$w(k+1) = w(k) - \eta grad(J(w(k)) \tag{3.10}$$

$$grad(J) = \frac{\partial J}{\partial w} = \sum_{i=1}^{N} (y^i - d^i)x^i \tag{3.11}$$

The learning coefficient η controls the size of a step *against* the direction of the gradient (because of a minus sign). If η is too small learning is slow; if too large the process of the criterion minimization can be oscillatory. The optimum value of the learning coefficient η is usually found experimentally. If η is kept constant, we speak about a *fixed-increment learning algorithm*.

3.1 Batch learning

1. Initialize the weights *w* to zero or to (small) random values.
2. For each training sample *i* = 1,2,…,*N* from the training set *T*:
    a) Input the training sample to the perceptron.
    b) Calculate $(y^i - d^i)x^i$ and keep summing it.
    c) Evaluate $(y^i - d^i)w^T x^i$ and keep summing it.

    The results of these operations are *J* and grad(*J*).
3. Update the weights:

$$w(k+1) = w(k) - \eta\, grad(J(w(k)) \tag{3.12}$$

Keep repeating steps 2-3; stop when *J* = 0 (all samples are perfectly classified) or when *J* does not decrease any more (samples are not linearly separable). If samples are not linearly separable this algorithm finds a separation line, which separates them "as well as possible" w.r.t. the criterion used.

During the batch training, the whole training set is processed and *then* the weights are updated.

3.2 Sample-by-sample learning

1. Initialize the weights *w* to zero or to (small) random values.
2. For each training sample *i* = 1,2,…,*N* from the training set *T*:
    a) Input the training sample to the perceptron.
    b) Calculate$(y^i - d^i)x^i$. This term is called an *instantaneous value of the gradient*.
    c) Evaluate the following criterion from the whole training set:

$$J = \sum_{i=1}^{N} (y^i - d^i)w^T x^i \tag{3.13}$$

    d) Update the weights:

$$w(k+1) = w(k) - \eta(y^i - d^i)x^i \tag{3.14}$$

Keep repeating step 2; stop when *J* = 0 (all samples are perfectly classified) or when *J* does not decrease any more (samples are not linearly separable). If the samples are not linearly separable this algorithm finds a separation line, which separates them "as well as possible" w.r.t. the criterion used.

During the sample-by-sample training, the weights are updated *immediately* when an individual training sample is presented to the perceptron.

3.3 Momentum learning

Momentum learning is an improvement to the straight gradient-descent search in the sense that a memory term (the past increment to the weight) is used to *speed up and stabilize convergence*. In momentum learning, the equation to update the weights becomes

$$w(k+1) = w(k) - \eta\, grad(J(w(k)) + m(w(k) - w(k-1)) \qquad (3.15)$$

where η is the learning coefficient and *m* is the momentum coefficient. This is called momentum learning due to the form of the last term, which resembles the momentum in mechanics. Note that the weights are changed proportionally to how much they were updated in the last iteration. Thus if the search is going down the hill and finds a flat region, the weights are still changed, not because of the gradient (which is practically zero in a flat spot), but because of the rate of change in the weights. Likewise, in a narrow valley, where the gradient tends to bounce back and forth between hillsides, the momentum stabilizes the search because it tends to make the weights follow a smoother path.

The Figure 3-4 below illustrates the advantage of momentum learning. Imagine a ball (weight vector position) rolling down a hill (performance surface). If the ball reaches a small flat part of the hill, it will continue past this local minimum because of its momentum. A ball without momentum, however, will get stuck in this valley. Momentum learning is a robust method to speed up learning, and is generally recommended.

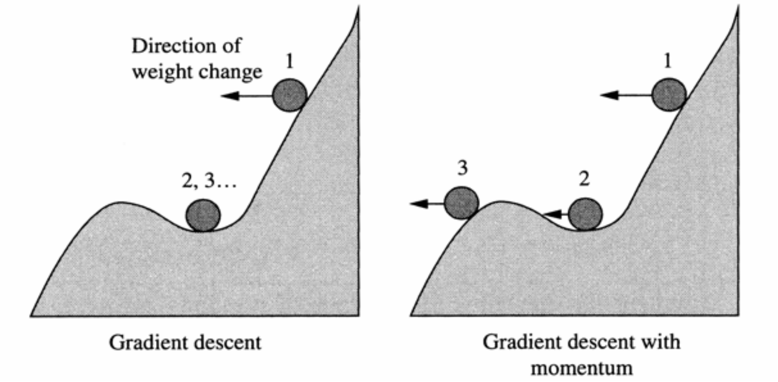


Figure 3-4: Why momentum learning helps.

3.4 Simulation results

3.4.1 Batch training

A Mathematica notebook `Perceptron.nb` allows the user to study the behaviour of various modifications of the perceptron training (learning) algorithm.

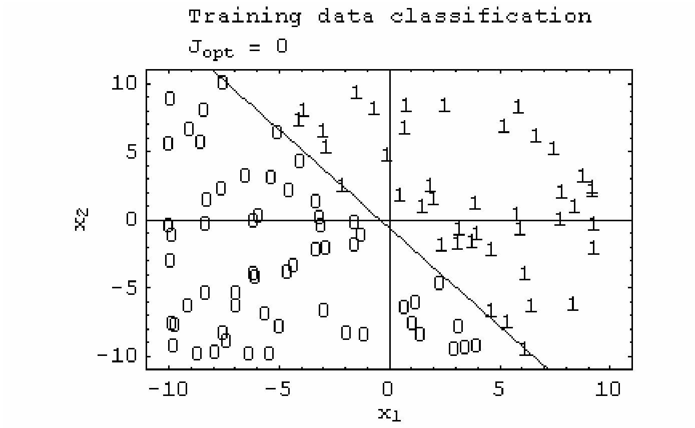


Figure 3-5: Two classes with a separation line.

We generated 100 training samples form two linearly separable clusters labelled “1” and “0”. The separation line is not unique; its equation is $x_2 = ax_1 + b$.

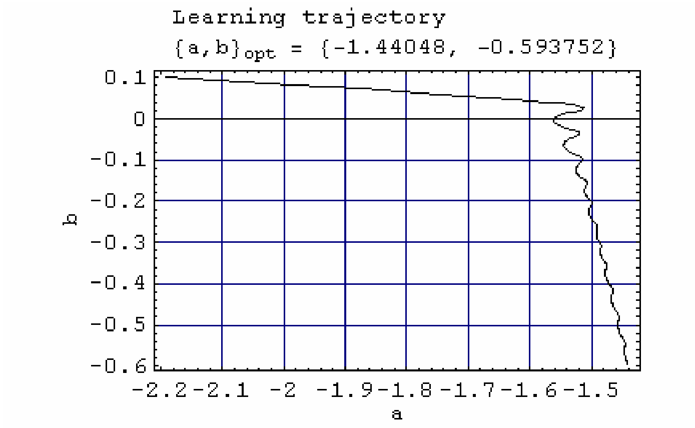


Figure 3-6: Learning path in a-b plane; m=1, batch training.

The Figure 3-6 shows the learning trajectory in a-b plane. Training parameters: batch, η = 0.2, m = 1.

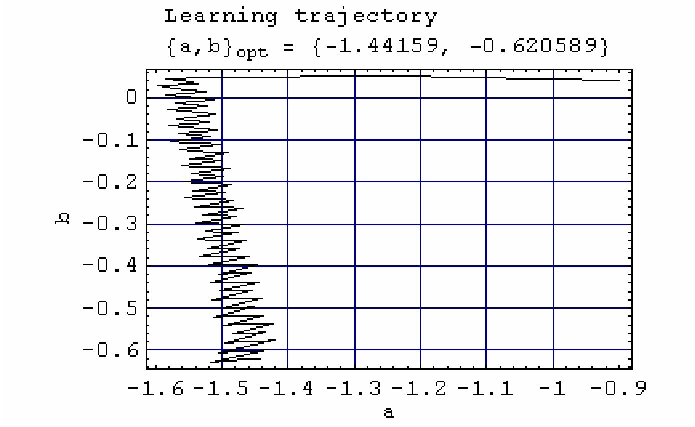


Figure 3-7: Learning path in a-b plane; m=0, batch training.

The Figure 3-7 shows the same case as Figure 3-6 but without the momentum term. The training process is less smooth (is rattling). Training parameters: batch, η = 0.2, m = 0.

3.4.2 *Sample-by-sample training*

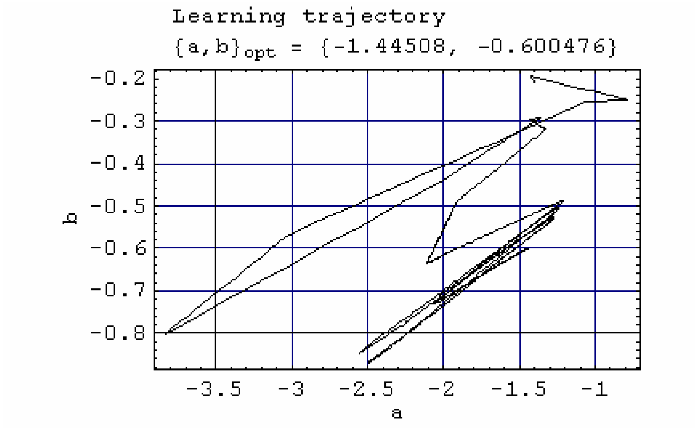


Figure 3-8: Learning path in a-b plane; m=1, sample-by-sample training.

The Figure 3-8 shows learning trajectory in a-b plane. Training parameters: sample-by-sample, $\eta = 0.2$, m = 1.

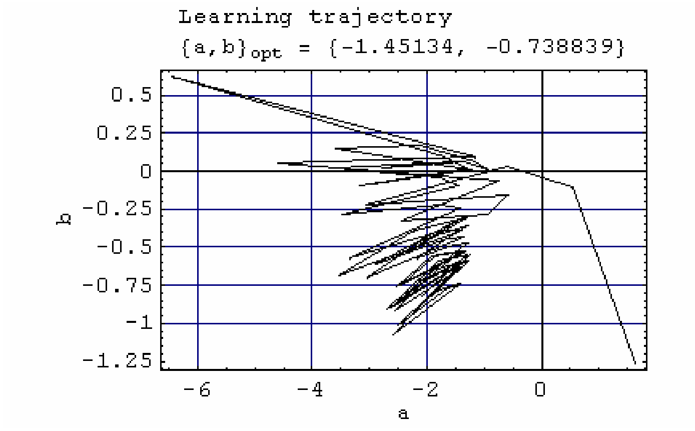


Figure 3-9: Learning path in a-b plane; m=0, sample-by-sample training.

The Figure 3-9 shows the same case as Figure 3-8 but without the momentum term. The training process is less smooth (is rattling). Training parameters: sample-by-sample, $\eta = 0.2$, m = 0.

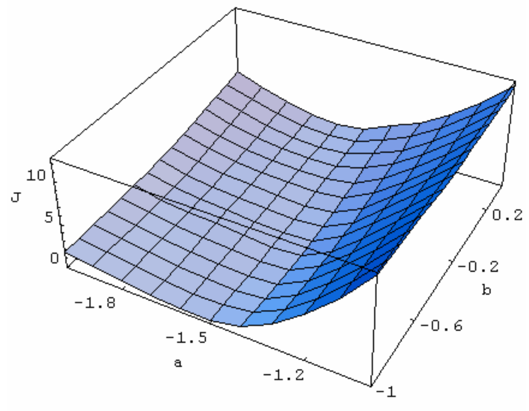


Figure 3-10: The criterion surface for the training set shown in Figure 3-5.

The separation line that separates both classes does not have to be unique. The criterion surface may form a plateau where the criterion $J = 0$. In this case, such a plateau is situated around $(a, b) = (-1.44, -0.60)$.

Study the above diagrams and notice the impact of the training parameters (η and m) on the learning process.

3.5 Perceptron networks

Several perceptrons can be combined to compute more complex functions. Since the perceptrons in Figure 3-11 are independent of one another, they can be trained separately in the same way as discussed in previous sections.

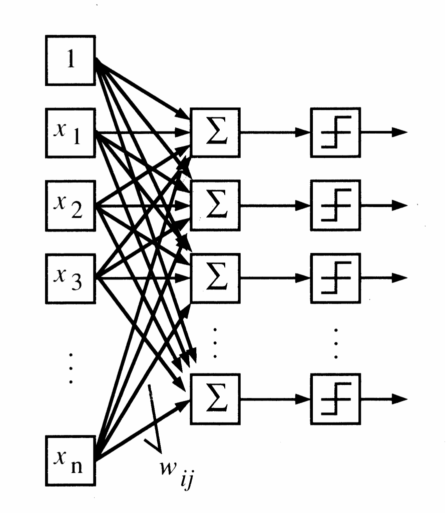


Figure 3-11: A perceptron with many inputs and many outputs.

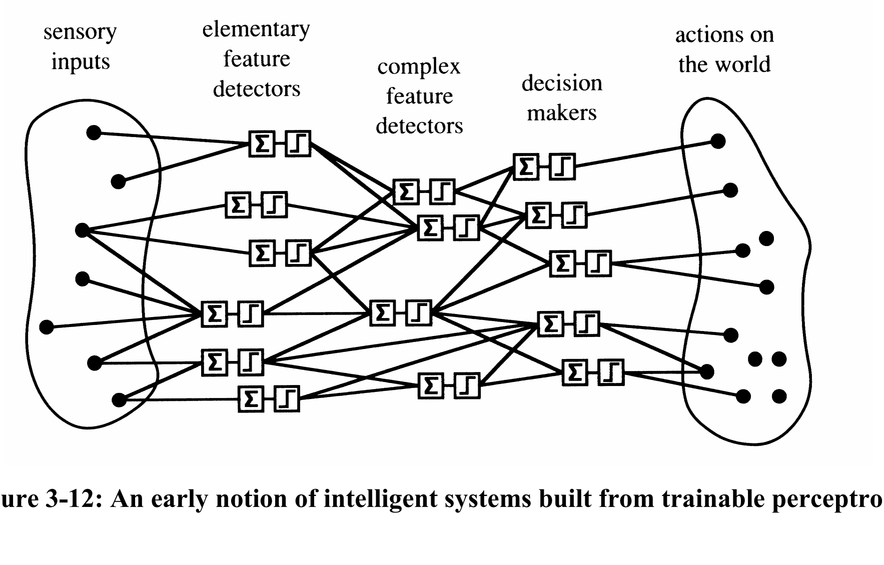
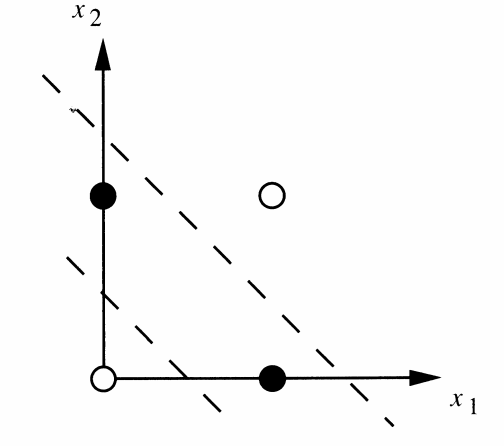


Figure 3-12: An early notion of intelligent systems built from trainable perceptrons.

At the time perceptrons were invented (around 1960), many people speculated that intelligent systems could be constructed out of perceptrons (see Figure 3-12). It turned out, however, that it is impossible to develop a convenient learning algorithm. When McCulloch-Pitts neurons were replaced by neurons with a differentiable activation function (1986), a back-propagation learning algorithm was invented and the architecture shown in Figure 3-12 became practical.

An example of a task that cannot be solved with a perceptron is the exclusive-or (XOR) problem shown in Figure 3-13. No single line can separate the "1" outputs from "0" outputs. If we could draw an elliptical decision surface, we could encircle the two “1” outputs in the XOR space. However, perceptrons are incapable of modelling such surfaces. Another idea is to employ two separate line-drawing stages. We could draw one line to isolate the point (1, 1) and then another line to divide the remaining three points into two categories. Using this idea, we can construct a multilayer perceptron to solve the problem. Such a device is shown in Figure 3-14.



x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3-13: A classification problem (XOR) that is not linearly separable.

Note how the output of the first perceptron serves as one of the inputs to the second perceptron, with a large, negatively weighted connection. The first perceptron sees the input (1, 1) and sends a massive inhibitory pulse to the second perceptron, causing that unit to output 0 regardless of its other inputs. If either of the inputs is 0, the second perceptron gets no inhibition from the first perceptron, and it outputs 1 if either of the inputs is 1.

The use of multilayer perceptrons, then, solves our knowledge representation problem. However, the convergence theorem does not extend to multilayer perceptrons and the synaptic connections cannot be learned.

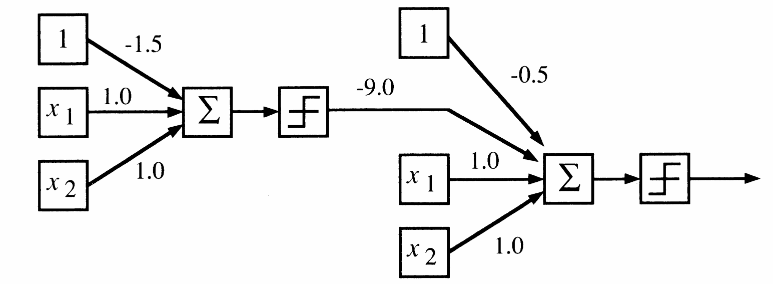


Figure 3-14: A multilayer perceptron that solver the XOR problem.

3.6 Problems

1. Consider the following two-dimensional patterns belonging to two classes *A* and *B*:

A:	(7, 5), (6, 3), (2, 4)
B:	(6, 0), (-1, -2), (3, -3)

Design a perceptron that will separate the patterns.

2. Consider two two-dimensional Gaussian-distributed classes C_1 and C_2 that have a common variance equal to 1. Their mean values are: $\mu_1 = (-5, -5)$ and $\mu_2 = (5, 5)$. These two classes are essentially linearly separable. Design a perceptron that separates these two classes.
Hint: Generate a training set with samples from C_1 and C_2 and train the perceptron.

3. Consider two two-dimensional Gaussian-distributed classes C_1 and C_2 that have a common variances equal to $\sigma_1 = 1$ and $\sigma_2 = 2$, respectively. Their mean values are: $\mu_1 = (0,0)$ and $\mu_2 = (2,0)$. These two classes overlap and are not linearly separable. Design a perceptron that separates these two classes as well as possible.
Hint: Generate a training set with samples from C_1 and C_2 and train the perceptron.

4. Design a perceptron that will be able to classify samples into three classes as shown in Figure 3-15.

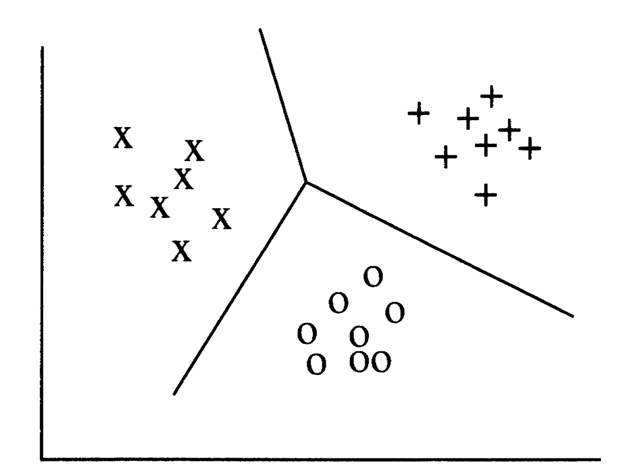


Figure 3-15: Three classes in a plane.

4 Back-propagation networks

Multilayer perceptrons have been applied successfully to solve some difficult diverse problems by training them in a supervised manner with a highly popular algorithm known as the *error back-propagation algorithm*. This algorithm is based on the *error-correction learning rule*.

Basically, the error back-propagation process consists of two passes through the different layers of the network: a *forward pass* and a *backward pass*. In the forward pass, activity pattern (input vector) is applied to the sensory nodes of the network, and its effect propagates through the network, layer by layer. Finally, a set of outputs is produced as the actual response of the network. During the forward pass the synaptic weights of network are all fixed. During the backward pass, on the other hand, the synaptic weights are all adjusted in accordance with the error-correction rule. Specifically, the actual response of the network is subtracted from a desired (target) response to produce an *error signal*. This error signal is then propagated backward through the network, against direction of synaptic connections - hence the name "error back-propagation". The synaptic weights are adjusted so as to make the actual response of the network move closer the desired response. The error back-propagation algorithm is also referred to in literature as the *back-propagation algorithm*, or simply *back-prop*.

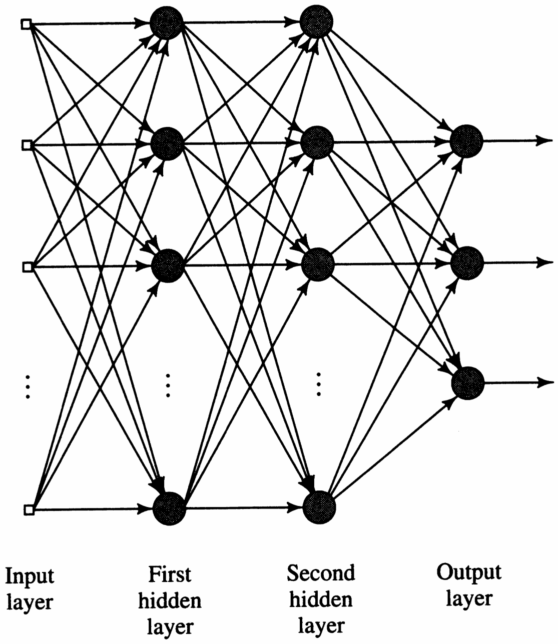


Figure 4-1: Diagram of a feed-forward back-propagation neural network.

The feed-forward back-propagation neural network in Figure 4-1 is fully connected, which means that a neuron in any layer is connected to all neurons in the previous layer. Signal flow through the network progresses in a forward direction, from left to right and on a layer-by-layer basis.

4.1 Forward pass

In the *forward pass* the synaptic weights remain unaltered and the signals are computed on a neuron-by-neuron basis. We will restrict ourselves to networks with one hidden layer only. Let us use the following notation:

$$x = \begin{bmatrix} x_0 & | & x_1 & \cdots & x_{n^x} \end{bmatrix}^T = \begin{bmatrix} x_0 & | & \bar{x}^T \end{bmatrix}^T$$ input vector with n^x+1 elements

$$h = \begin{bmatrix} h_0 & | & h_1 & \cdots & h_{n^h} \end{bmatrix}^T = \begin{bmatrix} h_0 & | & \bar{h}^T \end{bmatrix}^T$$ hidden layer with n^h+1 elements

$$y = \begin{bmatrix} y_1 & y_2 & \cdots & y_{n^y} \end{bmatrix}^T$$ output layer with n^y elements

$$u = \begin{bmatrix} u_1 & u_2 & \cdots & u_{n^h} \end{bmatrix}^T$$ sum of signals after synapses w^1

$$v = \begin{bmatrix} v_1 & v_2 & \cdots & v_{n^y} \end{bmatrix}^T$$ sum of signals after synapses w^2

$$w^1 = \left[\begin{array}{c|ccc} w^1_{10} & w^1_{11} & \cdots & w^1_{1n^x} \\ w^1_{20} & w^1_{21} & \cdots & w^1_{2n^x} \\ \vdots & \vdots & \cdots & \vdots \\ w^1_{n^h0} & w^1_{n^h1} & \cdots & w^1_{n^hn^x} \end{array}\right] = \begin{bmatrix} w^1_0 & | & \bar{w}^1 \end{bmatrix}$$ weight matrix n^h×(n^x+1)

$$w^2 = \left[\begin{array}{c|ccc} w^2_{10} & w^2_{11} & \cdots & w^2_{1n^h} \\ w^2_{20} & w^2_{21} & \cdots & w^2_{2n^h} \\ \vdots & \vdots & \cdots & \vdots \\ w^2_{n^y0} & w^2_{n^y1} & \cdots & w^2_{n^yn^h} \end{array}\right] = \begin{bmatrix} w^2_0 & | & \bar{w}^2 \end{bmatrix}$$ weight matrix n^y×(n^h+1)

The first neurons in the input and hidden layers (with index 0) are *bias neurons* and their outputs are equal to 1.

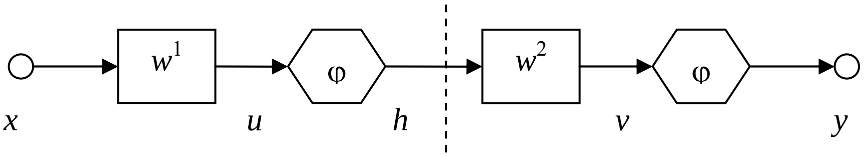


Figure 4-2: Signals in a network with one hidden layer; bias neurons are not shown.

The following equations describe the relation between the input x and output y:

$$\begin{aligned} u &= w^1 x \\ \bar{h} &= \varphi(u) \\ v &= w^2 h \\ y &= \varphi(v) \end{aligned} \qquad (4.1)$$

The input-output mapping is a nested composition of nonlinearities $\varphi(\sum(\varphi\sum(x)))$ where the number of function compositions is given by the number of network layers. The resulting map is very flexible and powerful but it is also hard to analyze.

4.2 Back-propagation

The error signal at the output of neuron *i* at iteration *n* (i.e. after presentation of the *n*-th training pattern) is

$$e_i(n) = y_i(n) - d_i(n) \tag{4.2}$$

We define the *instantaneous criterion I*(*n*) as the sum of squared errors over all output neurons:

$$I(n) = \frac{1}{2}\sum_{i=1}^{n^y} e_i^2(n) = \frac{1}{2} e(n)^T e(n) \tag{4.3}$$

The *overall criterion J* expressed as the average squared error is obtained by summing *I*(*n*) over all *n* and normalizing with respect to the training set size N:

$$J = \frac{1}{N}\sum_{n=1}^{N} I(n) = \frac{1}{2N}\sum_{n=1}^{N} e(n)^T e(n) \tag{4.4}$$

Both criteria are functions of synaptic weights of the network. The objective of the learning process is to adjust the synaptic weights so as to minimize *J*.

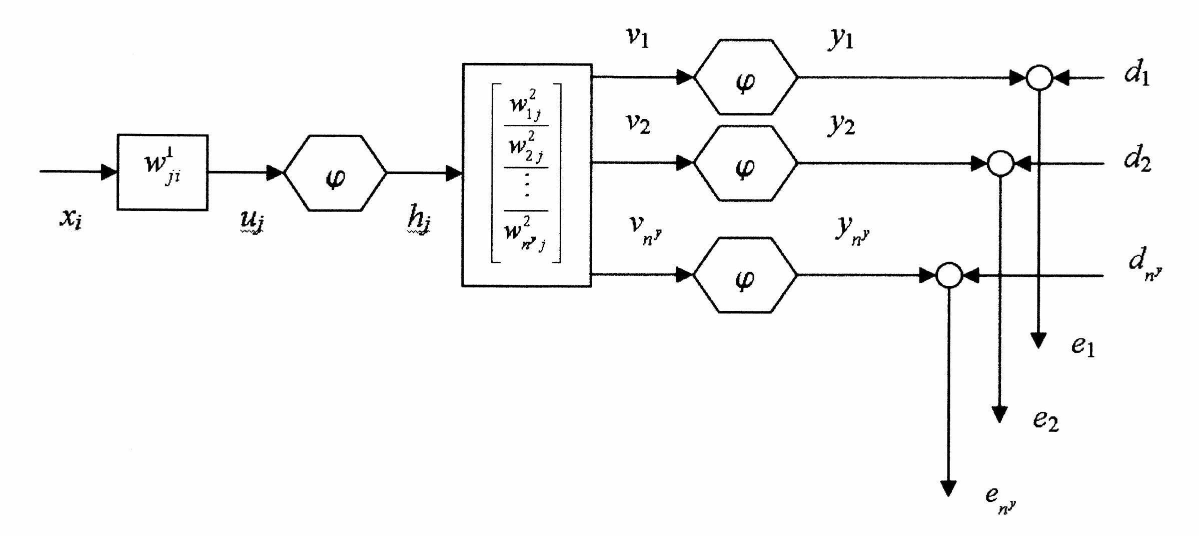


Figure 4-3: Signal propagation from the input x_i through the network.

We use a simple method of training in which the weights are updated on a sample-by-sample basis. The adjustments of weights are made in accordance with the respective errors computed for *each* sample presented to the network.

*4.2.1 Update of weight matrix w^2^*

The gradient $\partial I(n) / \partial w_{ij}^2(n)$ represents a sensitivity factor determining the direction of search in weight space for synaptic weight w_{ij}^2. According to the chain rule, we may express this gradient as follows:

$$\frac{\partial I(n)}{\partial w_{ij}^2(n)} = \frac{\partial I(n)}{\partial e_i(n)} \frac{\partial e_i(n)}{\partial y_i(n)} \frac{\partial y_i(n)}{\partial v_i(n)} \frac{\partial v_i(n)}{\partial w_{ij}^2(n)} \tag{4.5}$$

Individual partial derivatives are:

$$\begin{aligned} \partial I(n) / \partial e_i(n) &= e_i(n) \\ \partial e_i(n) / \partial y_i(n) &= 1 \\ \partial y_i(n) / \partial v_i(n) &= \varphi'(v_i(n)) \\ \partial v_i(n) / \partial w_{ij}^2(n) &= h_j(n) \end{aligned} \tag{4.6}$$

The use of these derivatives in (4.5) yields

$$\frac{\partial I(n)}{\partial w_{ij}^2(n)} = e_i(n)\varphi'(v_i(n))h_j(n) \tag{4.7}$$

The correction $\Delta w_{ij}^2(n)$ is defined by the *delta rule*

$$\Delta w_{ij}^2(n) = -\eta \frac{\partial I(n)}{\partial w_{ij}^2(n)} = -\eta \delta_i^2(n) h_j(n) \tag{4.8}$$

where the *local gradient*

$$\delta_i^2(n) = e_i(n)\varphi'(v_i(n)) = \frac{\partial I(n)}{\partial v_i(n)} \tag{4.9}$$

The update of weight matrix elements is

$$w_{ij}^2(n+1) = w_{ij}^2(n) + \Delta w_{ij}^2(n) \tag{4.10}$$

4.2.2 Update of weight matrix w^{1}

The gradient $\partial I(n)/\partial w_{ji}^{1}(n)$ represents a sensitivity factor determining the direction of search in weight space for synaptic weight w_{ji}^{1}. According to the chain rule, we may express this gradient as follows:

$$\frac{\partial I(n)}{\partial w_{ji}^{1}(n)}=\frac{\partial I(n)}{\partial h_j(n)}\frac{\partial h_j(n)}{\partial u_j(n)}\frac{\partial u_j(n)}{\partial w_{ji}^{1}(n)} \tag{4.11}$$

The most complicated term in (4.11) is the partial derivative $\partial I(n)/\partial h_j(n)$:

$$\frac{\partial I(n)}{\partial h_j(n)}=\frac{\partial(\frac{1}{2}\sum_{k=1}^{n^y}e_k^2(n))}{\partial h_j(n)}=\sum_{k=1}^{n^y}e_k(n)\frac{\partial e_k(n)}{\partial y_k(n)}\frac{\partial y_k(n)}{\partial v_k(n)}\frac{\partial v_k(n)}{\partial h_j(n)} \tag{4.12}$$

Substituting from (4.1), (4.6), and (4.9) into (4.12) we get

$$\frac{\partial I(n)}{\partial h_j(n)}=\sum_{k=1}^{n^y}\delta_k^2(n)w_{kj}^2(n) \tag{4.13}$$

Evaluation of remaining partial derivatives in (4.11) is straightforward:

$$\begin{aligned}\partial h_j(n)/\partial u_j(n)&=\varphi'(u_j(n))\\ \partial u_j(n)/\partial w_{ji}^{1}(n)&=x_i(n)\end{aligned} \tag{4.14}$$

The use of derivatives (4.13) and (4.14) in (4.11) yields

$$\frac{\partial I(n)}{\partial w_{ji}^{1}(n)}=\sum_{k=1}^{n^y}\delta_k^2(n)w_{kj}^2(n)\varphi'(u_j(n))x_i(n) \tag{4.15}$$

The correction $\Delta w_{ji}^{1}(n)$ is defined by the *delta rule*

$$\Delta w_{ji}^{1}(n)=-\eta\frac{\partial I(n)}{\partial w_{ji}^{1}(n)}=-\eta\delta_j^1(n)x_i(n) \tag{4.16}$$

where the *local gradient*

$$\delta_j^1(n)=\sum_{k=1}^{n^y}\delta_k^2(n)w_{kj}^2(n)\,\varphi'(u_j(n))=\frac{\partial I(n)}{\partial u_j(n)} \tag{4.17}$$

The update of weight matrix elements is

$$w_{ji}^{1}(n+1)=w_{ji}^{1}(n)+\Delta w_{ji}^{1}(n) \tag{4.18}$$

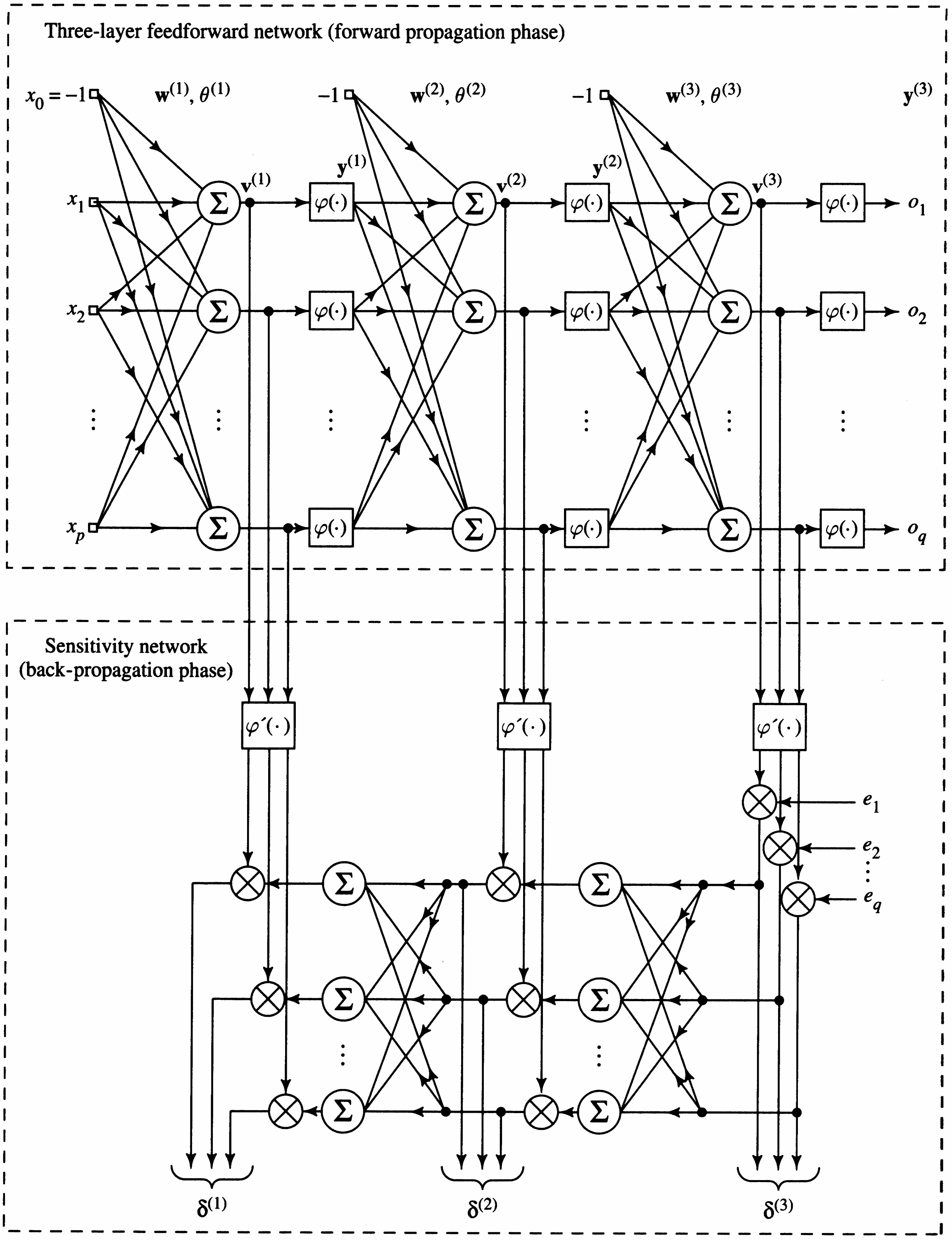


Figure 4-4: Diagram of a feedforward network and associated back-propagating error signals (sensitivity network). $\Delta w^{(1)} = -\eta_1 \delta^{(1)} x, \quad \Delta w^{(2)} = -\eta_2 \delta^{(2)} y^{(1)}, \quad \Delta w^{(3)} = -\eta_3 \delta^{(3)} y^{(2)}, \ \eta_1 > \eta_2 > \eta_3$

The *sensitivity network* has similar architecture to its feedforward counterpart and uses the same weights. The errors e_i are propagated backwards and are multiplied with the derivatives of activation functions. Hence, the activation functions must be differentiable; that is why

McCulloch-Pitts neurons cannot be used. Notice that the sensitivity network uses signals, which are available from the forward pass.

4.3 Two passes of computation

In the application of the back-propagation algorithm, two distinct passes of computation may be distinguished. The first pass is referred to as the forward pass, and the second one as the backward pass.

In the *forward pass* the synaptic weights remain unaltered and the function signals of the network are computed on a neuron-by-neuron basis according to (4.1). Thus the forward phase of computation begins at the first hidden layer by presenting it with the input vector, and terminates at the output layer by computing the error signal for each neuron of this layer.

The *backward pass*, on the other hand, starts at the output layer by passing the error signals leftward through the sensitivity network, layer by layer, and recursively computing the δ (i.e. the local gradient) for each neuron. This recursive process permits the synaptic weights of the network to undergo changes in accordance with the delta rules (4.8) and (4.16). For a neuron located in the output layer, the δ is simply equal to the error signal of that neuron multiplied by the first derivative of its nonlinearity. Given the δ's for the neurons of the output layer, we next use (4.17) to compute the δ's for all the neurons in the hidden layer and, therefore, the changes to the weights of all connections feeding into it. The recursive computation is continued, layer by layer by propagating the changes to all synaptic weights made.

The computation of the δ for each neuron of the multilayer perceptron requires knowledge of the derivative of the activation function φ associated with that neuron. For this derivative to exist, we require the function φ to be differentiable. An example of a continuously differentiable nonlinear activation function commonly used in multilayer perceptrons is the sigmoidal nonlinearity (1.7) or hyperbolic tangent (1.9). According to those nonlinearities, the amplitude of the output lies inside the range [0, 1] or [-1, 1], respectively.

Note from (1.8) $\varphi'(v)$ attains its maximum value when $\varphi = 0.5$, and its minimum value (zero) at $\varphi \to 0$, or $\varphi \to 1$. Since the amount change in a synaptic weight of the network is proportional to the derivative $\varphi'(v)$, it follows that for a sigmoidal activation function the synaptic weights are changed the most for those neurons in the network for which the function signals are in midrange. It is this feature of back-propagation learning that contributes to its stability.

4.4 Stopping criteria

The back-propagation algorithm cannot, in general, be shown to converge, nor are there well-defined criteria for stopping its operation. Rather, there are some reasonable criteria, each with its own practical merit, which may be used to terminate the weight adjustments. To formulate such a criterion, the logical thing to do is to think in terms of the unique properties of a local or global minimum of the error surface.

1. The back-propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold. The drawback of this convergence criterion is that, for successful trials, learning times may be long.

2. The back-propagation algorithm is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small. Typically, the rate of change in the average squared error is considered to be small enough if it lies in the range of 0.1 to 1 percent per epoch.

3. The back-propagation algorithm is terminated when the weight updates are sufficiently small.

4. Another useful criterion for convergence is as follows. After each learning iteration, the network is tested for its generalization performance. The learning process is stopped when the generalization performance is adequate, or when it is apparent that the generalization performance has peaked.

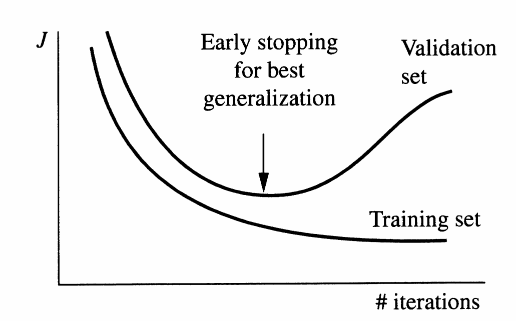


Figure 4-5: Validation for early stopping.

5. The simplest way to stop the training is to limit the number of iterations to a predetermined value. This stopping criterion is frequently used mainly when a new problem is solved and nothing is known about the shape and properties of the error surface.

4.5 Momentum learning

The back-propagation algorithm minimizes the criterion by the method of steepest descent. The smaller we make the learning rate parameter η, the smaller will the changes to the synaptic weights in the network be one iteration to the next and the smoother will be the trajectory in weight space. This improvement, however, is attained at the cost of a slower rate of learning. If, on the hand, we make the learning parameter η too large so as to speed up the rate of learning the learning process may become unstable (oscillatory). A simple method of increasing the rate of learning and yet avoiding the danger of instability is to modify the delta rule by including a momentum term:

$$\Delta w(n+1) = m\,\Delta w(n) - \eta \frac{\partial I(n)}{\partial w(n)} \qquad (4.19)$$

This equation with the *momentum coefficient m* is called the *generalized delta rule*.

To show the effect of the momentum *m* on the learning speed let us suppose that the gradient $\partial I / \partial w$ is constant for several consecutive steps. By applying (4.19) several times we get

$$\begin{aligned}
\Delta w(n+2) &= m^2 \Delta w(n) - (m+1)\eta\,(\partial I / \partial w) \\
\Delta w(n+3) &= m^3 \Delta w(n) - (m^2+m+1)\eta\,(\partial I / \partial w) \\
&\ \vdots \\
\Delta w(n+k) &= m^k \Delta w(n) - (m^{k-1}+m^{k-2}+\cdots+m+1)\eta\,(\partial I / \partial w) \\
&= m^k \Delta w(n) - \frac{1+m^k}{1-m}\eta \frac{\partial I}{\partial w}
\end{aligned} \qquad (4.20)$$

If the number of steps *k* is large and |*m*| < 1 then

$$\lim_{x\to\infty} \Delta w(n+k) = -\frac{\eta}{1-m}\frac{\partial I}{\partial w} = -\bar{\eta}\frac{\partial I}{\partial w} \qquad (4.21)$$

We see from (4.21) that the momentum effectively increases the learning coefficient η to $\bar{\eta}$.

The momentum term represents only a minor modification to the delta rule and yet it has highly beneficial effect on learning. It also helps to prevent the learning process from terminating in a shallow local minimum on the error surface.

4.6 Rate of learning

As we already mentioned in the previous section, for fast convergence to the neighbourhood of the minimum, a large step size is desired. However, the solution with a large step size suffers from *rattling*, i.e. the iterative process continues to wander around a neighbourhood of the minimum without ever stabilizing as shown in Figure 4-6.

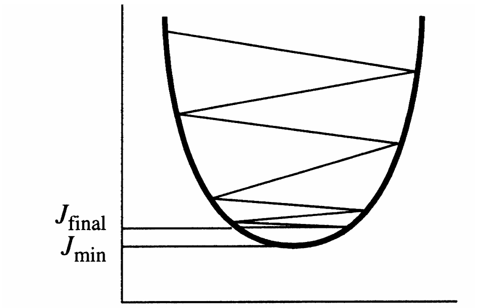


Figure 4-6: Rattling in the neighbourhood of the minimum.

One attractive solution is to use a large learning rate in the beginning of training and then decrease the learning rate to obtain good accuracy on the final weight values. This is called *learning rate scheduling*. This simple idea can be implemented with a variable step size controlled by

$$\eta(n+1) = \eta(n) - \beta \qquad (4.22)$$

where $\eta(0) = \eta_0$ is the initial step size, and β is a small constant. Note that the step size is being linearly decreased at each iteration. If we have control of the number of iterations, we can start with a large step size and decrease it to practically zero toward the end of training. The value of β needs to be experimentally determined. Many neural network packages activate the decay (4.22) after user defined number of iterations until a certain minimum value of η is reached.

All neurons in the multilayer perceptron should desirably learn at the same rate. Typically, the last layers tend to have larger local gradients than the layers at the front end of the network. Hence, the learning parameter η and momentum *m* should be assigned a smaller values in the last layers than the front-end layers. Neurons with many inputs should have a smaller learning-rate parameters than neurons with few inputs.

4.7 Pattern and batch modes of training

In a practical application of the back-propagation algorithm, learning results from the many presentations of a training set to the network. One complete presentation of the entire training set during the learning process is called an *epoch*. The learning process is maintained on an epoch-by-epoch basis until the synaptic weights stabilize and the average squared error over the entire training set converges to some minimum value. It is good practice to *randomize the order of presentation of training examples* from one epoch to the next. This randomization tends to make the search in weight space stochastic over the learning cycles, thus avoiding the possibility of limit cycles in the evolution of the synaptic weight vectors.

For a given training set, back-propagation learning may thus proceed in one of two basic ways:

1. In the *pattern mode* of back-propagation learning, weight updating is performed immediately after the presentation of each training example. This is the very mode of operation for which the derivation of the back-propagation algorithm presented here applies.
2. In the *batch mode* of back-propagation learning, weight updating is performed after the presentation of all the training examples that constitute an epoch.

The pattern mode of training is generally preferred over the batch mode, because it requires less local storage for each synaptic connection. Moreover, given that the patterns are presented to the network in a random manner, the use of sample-by-sample updating of weights makes the search in weight space stochastic in nature, which, in turn, makes it less likely for the back-propagation algorithm to be trapped in a local minimum. On the other hand, the use of batch mode of training provides a more accurate estimate of the gradient vector. The relative effectiveness of the two training modes depends in the problem at hand.

4.8 Weight initialization

The first step in back-propagation learning is, of course, to initialize the network. A good choice for the initial values of the free parameters (i.e. adjustable synaptic weights) of the network can be of tremendous help in a successful network design. In cases where prior information is available, it may be better to use the prior information to guess the initial values of the free parameters. But how do we initialize the network if no prior information is available? The customary practice is to set all the free parameters of the network to random numbers that are uniformly distributed, e.g. inside the interval [-0.3, 0.3]. The initial values should not be too small because δ's are multiplied by weights (see Figure 4-4) and small weight values will cause the error gradients to be very small.

The wrong choice of initial weights (e.g. too large weights) can lead to a phenomenon known as *premature saturation*. This phenomenon refers to a situation where the instantaneous sum of squared errors $I(n)$ remains almost constant for some period of time during the learning process. Such a phenomenon cannot be considered as a local minimum, because the squared error continues to decrease after this period is finished. (The premature saturation phenomenon corresponds to a saddle point in the error surface.)

When a training pattern is applied to the input layer of a multilayer perceptron, the output values of the network are calculated through a sequence of forward computations that involves inner products and sigmoidal transformations. This is followed by a sequence of backward computations that involves the calculation of error signals and pertinent slope of the sigmoidal activation function, and culminates in synaptic weight adjustments. Suppose that, for a particular training pattern, the net internal activity level of an output neuron (computation node) is computed to have a large magnitude. Then, assuming that the sigmoidal activation function of the neuron has the limiting values -1 and +1, we find that the corresponding slope of the activation function for that neuron will be very small, and the output value for the neuron will be close to -1 or +1. In such a situation, we say that the neuron is in *saturation*. The neurons that are in saturation do not learn any more. If this happens the best way is to initialize the synaptic connections again, possibly with smaller values, and start training afresh.

4.9 Generalization

A network is said to *generalize* well when the input-output relationship computed by the network is correct (or nearly so) for input/output patterns (test data) never used in creating or training the network. Here, of course, it is assumed that test data are drawn from the same population used to generate the training data.

The learning process may be viewed as a curve-fitting problem. The network itself may be considered simply as a nonlinear input-output mapping. Such a viewpoint then permits us to look on generalization not as a mystical property of neural networks but rather simply as the effect of a good nonlinear interpolation.

Figure 4-7a illustrates how generalization may occur in a hypothetical network. The nonlinear input-output mapping represented by the curve depicted in this figure is computed by the network as a result of learning the points labelled as training data. The point marked on the curve as generalization is thus seen to be simply as the result of interpolation performed by the network.

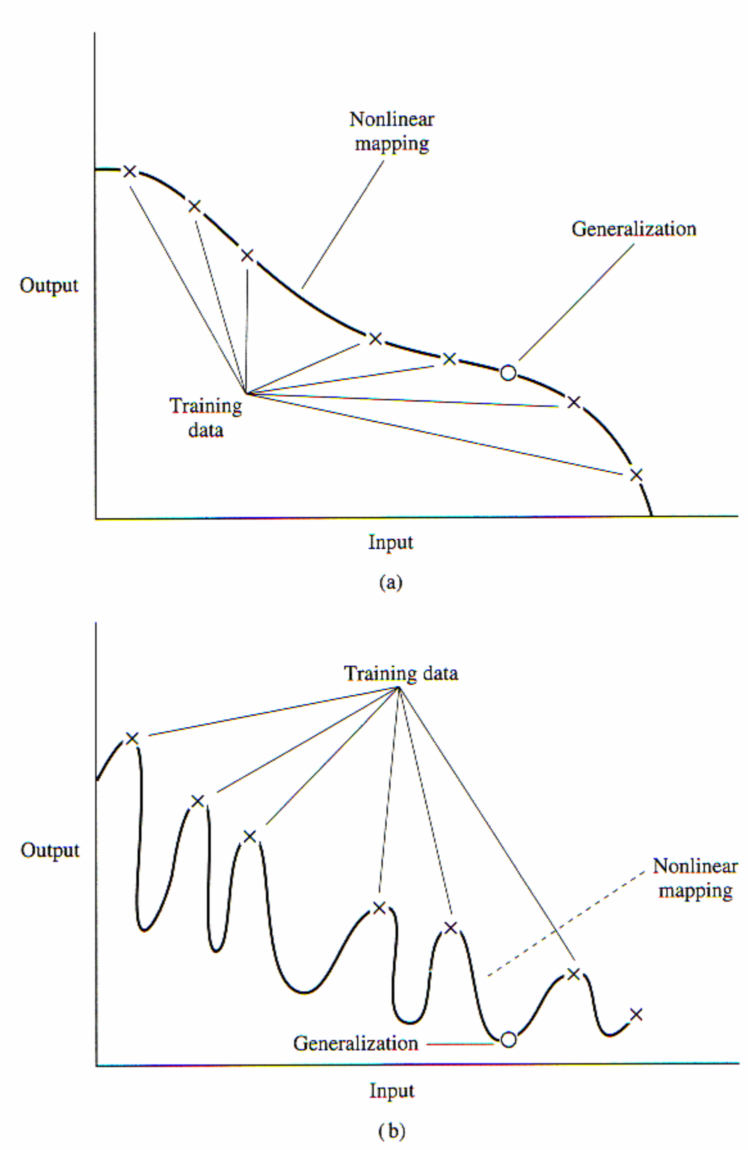


Figure 4-7: (a) Properly fitted data (good generalization). (b) Overfitted data (poor generalization).

When, however, a neural network learns too many specific input-output relations (i.e. it is *over-trained*), the network may memorize the training data and, therefore, be less able to generalize between similar input-output patterns. Such a network typically uses more hidden neurons than is actually necessary, with the result that its plasticity is too high and unintended curves in the problem space are stored in the synaptic weights. An example of how poor generalization due to memorization in a neural network may occur is illustrated in Figure 4-7b for the same data depicted in Figure 4-7a. Memorization is essentially a look-up table, which implies that the input-output mapping computed by the neural network is not smooth. The smoothness of input-output mapping is intimately related to such model-selection criteria as the *Occam razor*, the essence of which is to select the simplest function in the absence of any prior knowledge to the contrary. In the context of our present discussion, the simplest network means the smoothest function that approximates the mapping for a given error criterion.

4.10 Training set size

Generalization is influenced by three factors: the size and efficiency of the training set, the architecture of the network, and the physical complexity of the problem at hand. Clearly, we have no control over the latter. In the context of the other two factors, we may view the issue of generalization from two different perspectives:

1. The architecture of the network is fixed (hopefully in accordance with the physical complexity of the underlying problem), and the issue to be resolved is that of determining the size of the training set needed for a good generalization to occur.
2. The size of the training set is fixed, and the issue of interest is that of determining the best architecture of network for achieving good generalization.

Both of these viewpoints are valid in their own individual ways; the former viewpoint will be discussed here.

Neural networks require a lot of data for appropriate training, because there are no a priori assumptions about the data. The number of training patterns *N* required to classify test examples with an error ε is

$$N > \frac{W}{\varepsilon} \tag{4.23}$$

where $W = (n^x + 1)n^h + (n^h + 1)n^y$ is the number of weights in the network and ε is the fraction of errors permitted on test. The equation (4.23) shows that the number of required training patterns increases linearly with the number of synaptic connections. A rule of thumb states that $N \approx 10W$.

In this rule of thumb it is assumed that the training data is representative of all the conditions encountered in the test set. The main focus when creating the training set should be to collect data that covers the full known operating conditions of the problem we want to model. If the training set does not contain data from some areas of pattern space, the machine classification in those areas will be based on extrapolation. This may or may not correspond to the true classification boundary (desired output), so we should always choose samples for the training set that cover as much of the input space as possible.

We should remark that in (4.23) the practical limiting quantity is the number of training patterns. Most of the time we need to compromise the size of the network to achieve appropriate training. A reasonable approach to reduce the number of weights in the network is to sparsely connect the input layer to the first hidden layer (which normally contains the largest number of weights). Another possibility is to use feature extraction (i.e. a pre-processor) that decreases the input space dimensionality, thus reducing the number of weights in the network.

4.11 Network size

To solve real-world problems with neural networks, we usually require the use of highly structured networks of a rather large size. A practical issue that arises in this context is minimizing the size of the network and yet maintaining good performance. A network with minimum size is less likely to learn the noise in the training data, and may thus generalize better to new data. We may achieve this design objective in one of two ways:

1. *Network growing*, in which case we start with a small network and add a new neuron or a new layer of hidden neurons only when we are unable to meet the design specification. The cascade-correlation learning architecture is an example of the network-growing approach.
2. *Network pruning*, in which case we start with a large network with an adequate performance for the problem at hand, and then prune it by weakening or eliminating certain synaptic weights in a selective and orderly fashion. For example, the synaptic connections that are much smaller than others can be eliminated.

4.11.1 Complexity regularization

A neural network should be sufficiently large to solve the problem, but not larger. If the network is too large it will not generalize well and it will tend to memorize the training data set. An intermediate network is the best choice. It means that the design procedure should include a criterion for selecting the *model complexity* (i.e. the number of neurons and, hence, synaptic connections). Specifically, the learning objective is to find a weight vector that minimizes the total risk

$$R(w) = J(w) + \lambda\, J_c(w) \tag{4.24}$$

The first term is the standard *performance measure* used before (see (4.4)), the second term $J_c(w)$ is the complexity penalty, which depends on the network and its evaluation extends over all the synaptic connections in the network. The *regularization parameter* λ represents the relative importance of the complexity-penalty term with respect to the performance-measure term. When λ is zero, the back-propagation learning process is unconstrained, with the network being completely determined from the training examples. When λ is made infinitely large, on the other hand, the implication is that the constraint imposed by the complexity penalty is by itself sufficient to specify the network, which is another way of saying that the training examples are unreliable.

In the *weight-decay* procedure, the complexity regularization is defined as the squared norm of the weight vector *w* in the network

$$J_c(w) = \|w\|^2 = \sum w_i^2 \tag{4.25}$$

This complexity regularization term forces some of the synaptic weights to take on the values close to zero. Such weights are called *excess weights* and can be eliminated.

Many neural network packages use a *genetic algorithm* to optimize the network architecture. A complexity regularization term can then be expressed as above in (4.25), or simply as the number of hidden neurons n^h , or as a combination of both.

4.12 Training, testing, and validation sets

The essence of back-propagation learning is to encode an input-output relation, represented by a set of examples $\{x, d\}$, with a back-propagation network well trained in the sense that it learns enough about the past to generalize to the future. We may train the same network several times and get different synaptic connections for each training run. We may also design several networks with different architectures. Our task is to choose, within a set of candidate model structures, the best-one according to a certain criterion.

A standard tool in statistics, known as *cross-validation*, provides a guiding principle. First, the available data set is randomly partitioned into a *training set* and *testing set*. The training set is further partitioned into two subsets:

1. A subset used for *training the network.*
2. A subset used for *evaluation of the performance* of the network (i.e. validation). The validation subset is typically 10 to 20 percent of the training set.

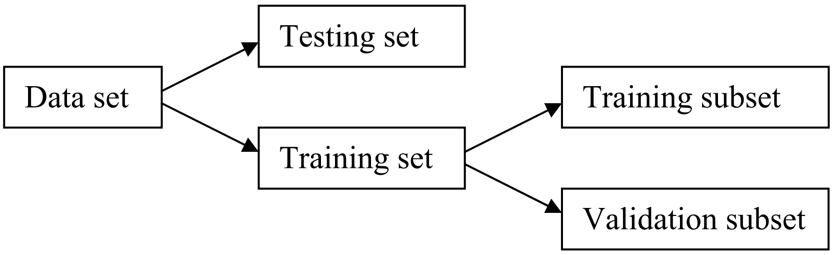


Figure 4-8: Splitting the data into subsets.

The motivation here is to *validate* the model on a data set different from the one used for parameter estimation. In this way, we may use the testing set to assess the performance of various candidate model structures, and thereby choose the best one. The particular model with the best-performing parameter values is then trained on the full training set, and the generalization performance of the resulting network is measured on the test set.

4.13 Approximation of functions

A back-propagation neural network may be viewed as a practical vehicle for performing a *nonlinear input-output mapping* of a general nature. It has been proved that a single hidden layer network is sufficient to uniformly approximate any continuous function. This finding is called a *universal approximation theorem*, which is stated below and is directly applicable to back-propagation networks:

<u>Universal approximation theorem</u> *Let* $\varphi(\cdot)$ *be a non-constant, bounded, and monotone-increasing continuous function. Let* I_p *denote the p-dimensional unit hypecube* $[0, 1]^p$. *The space of continuous functions on* I_p *is denoted by* $C(I_p)$. *Then, given any function* $f \in C(I_p)$ *and* $\varepsilon > 0$, *there exist an integer M and sets of real constants* α_i, θ_i, *and* w_{ij}, *where* $i = 1,\ldots,M$ *and* $j = 1,\ldots,p$ *such that we may define*

$$F(x_1,\ldots,x_p) = \sum_{i=1}^{M} \alpha_i \varphi(\sum_{j=1}^{p} w_{ij} x_j - \theta_i) \tag{4.26}$$

as an approximate realization of the function f(·); that is

$$\left| F(x_1,\ldots,x_p) - f(x_1,\ldots,x_p) \right| < \varepsilon \tag{4.27}$$

for all $\{x_1,\ldots,x_p\} \in I_p$.

The universal approximation theorem is an *existence theorem* in the sense that it provides the mathematical justification for the approximation of an arbitrary continuous function. In effect, the theorem states that a single hidden layer is sufficient for a network to compute a uniform ε approximation to a given training set represented by the set of inputs $x_1,\ldots,x_p$ and a desired (target) output $f(x_1,\ldots,x_p)$. However, the theorem does not say that a single hidden layer is optimum in the sense of learning time or ease of implementation. It also says nothing about the number of hidden neurons needed.

The universal approximation theorem is important from a theoretical viewpoint, because it provides the necessary mathematical tool for the viability of feedforward networks with a single hidden layer as a class of approximate solutions. Without such a theorem, we could conceivably be searching for a solution that cannot exist. However, the theorem is only an existence proof; it does not tell us how to construct the network to do the approximation.

Nevertheless, the universal approximation theorem has limited practical value. The theorem assumes that the continuous function to be approximated is given and that a hidden layer of unlimited size is available for the approximation. Both of these assumptions are violated in most practical applications.

The problem with back-propagation networks using a single hidden layer is that the neurons therein tend to interact with each other globally. In complex situations this interaction makes it difficult to improve the approximation at one point without worsening it at some other point. On the other hand, with two hidden layers the approximation (curve-fitting) process becomes sometimes more manageable.

In case of function approximation, the output neurons have usually a linear activation function. If a squashing (sigmoid or hyperbolic tangent) function is used the desired output must be carefully scaled so that it fits well into the linear region of the squashing activation function.

4.14 Examples

4.14.1 Classification problem – interlocked spirals

The data generation program for this problem is available in `Spiral.nb` Mathematica notebook.

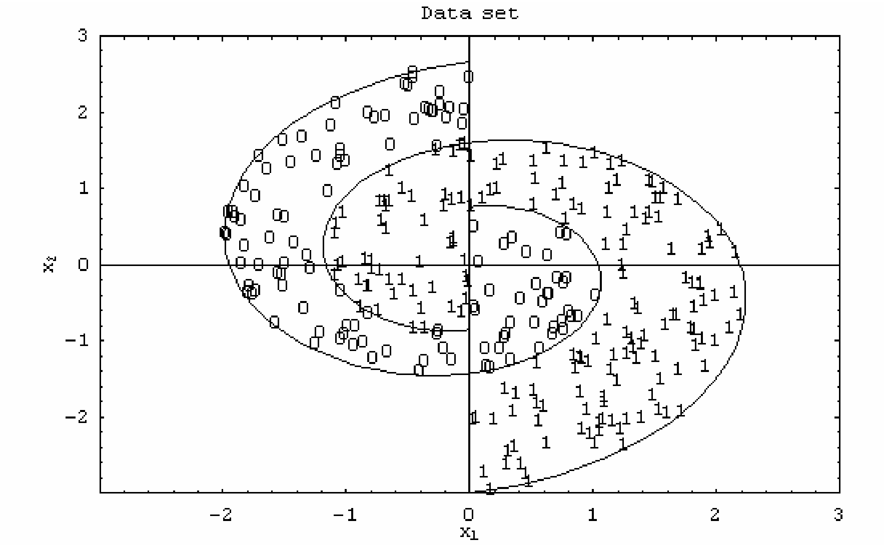


Figure 4-9: Distribution of classes C$_1$ = "0" and C$_2$ = "1".

A back-propagation neural network was trained on $N = 1195$ samples uniformly distributed in the region shown in Figure 4-9. The data was split into training (80%) and validation (20%) subsets. The NeuroSolutions program was used to generate and train the network. The network configuration was 2-10-1 (2 inputs, 10 hidden neurons, 1 output) and all neurons had a tangent hyperbolic activation function. The total number of weights was (2+1)×10 + (10+1)×1 = 41. The training subset was large enough: it had 956 samples, which is 23 times more than the number of weights.

A back-propagation neural network output y is never strictly equal to 0 or 1 but can take any value in between. The output was post-processed according to the formula

$$\bar{y} = \begin{cases} 1, & y \geq 0.5 \\ 0, & y < 0.5 \end{cases} \tag{4.28}$$

The network was tested on 1176 samples generated separately from the training set (out-of-sample testing). The following *confusion matrix* summarizes the classification results:

Table 4-1: Confusion matrix. Classes C$_1$ and C$_2$ contain samples labelled "0" and "1", respectively.

		Predicted classification		Total true
		C$_1$	C$_2$	
Desired classification	C$_1$	$n_{11} = 473$	$n_{12} = 4$	$row_1 = 477$
	C$_2$	$n_{21} = 9$	$n_{22} = 690$	$row_2 = 699$
Total predicted		$col_1 = 482$	$col_2 = 694$	$N = 1176$

The confusion matrix says that $n_{11} = N(\tilde{C}_1 | C_1) = 473$ samples belonging to class C$_1$ were classified correctly as $\tilde{C}_1$. Similarly, $n_{22} = N(\tilde{C}_2 | C_2) = 690$ samples belonging to class C$_2$ were classified correctly as $\tilde{C}_2$. However, $n_{12} = N(\tilde{C}_2 | C_1) = 4$ samples belonging to class C$_1$ were classified incorrectly as $\tilde{C}_2$. Similarly, $n_{21} = N(\tilde{C}_1 | C_2) = 9$ samples belonging to class

C₂ were classified in incorrectly as $\tilde{C}_1$. The confusion matrix is an excellent way of quantifying the accuracy of a classifier.

There are other classification criteria that can be calculated from the confusion matrix: *classification error*, *sensitivity*, and *specificity*.

$$\begin{aligned}\text{classification error} &= \frac{\text{sum of off-diagonal elements}}{\text{total number of samples}}100 \\ &= \frac{N - \sum_i n_{ii}}{N}100\,\% \end{aligned} \tag{4.29}$$

$$\begin{aligned}\text{sensitivity}(C_i) &= \frac{\text{samples classified correctly as } \tilde{C}_i \text{ and members of } C_i}{\text{total number of samples in } C_i}100 \\ &= \frac{n_{ii}}{row_i}100\,\% \end{aligned} \tag{4.30}$$

$$\begin{aligned}\text{specificity}(C_i) &= \frac{\text{samples not classified as } \tilde{C}_i \text{ and not members of } C_i}{\text{total number of samples not in } C_i}100 \\ &= \frac{N - (row_i + col_i - n_{ii})}{N - row_i}100\,\% \end{aligned} \tag{4.31}$$

N is the total number of samples, row_i is the sum of elements in row i (total number of samples in C_i), col_i is the sum of elements in column i (total number of samples classified as $\tilde{C}_i$), and $N = \sum_i row_i = \sum_i col_i$.

Sensitivity is the ability to detect “true positive” matches. The most sensitive classification finds all true matches but might have lots of false positives.

Specificity is the ability to reject “false positive” matches. The most specific classification will return only true matches but might have lots of false negatives.

In our example, the above classification criteria have the following values:

$$\begin{aligned}\text{classification error} &= \frac{13}{1176} = 1.11\% \\ \text{sensitivity}(C_1) &= \frac{473}{477} = 99.16\% \\ \text{specificity}(C_1) &= \frac{1176-(482+477-473)}{699} = 98.71\% \\ \text{sensitivity}(C_2) &= \frac{690}{699} = 98.71\% \\ \text{specificity}(C_2) &= \frac{1176-(694+699-690)}{477} = 99.16\% \end{aligned}$$

The neural network has learnt the separation boundary well as shown in Figure 4-10. Samples denoted "o" correspond to the neural network output $y \in [0.35, 0.65]$, i.e. the output is close to the decision level of 0.5. Samples denoted "M" are misclassified – they are on a "wrong" side of the separation boundary. However, all of them are misclassified "only slightly".

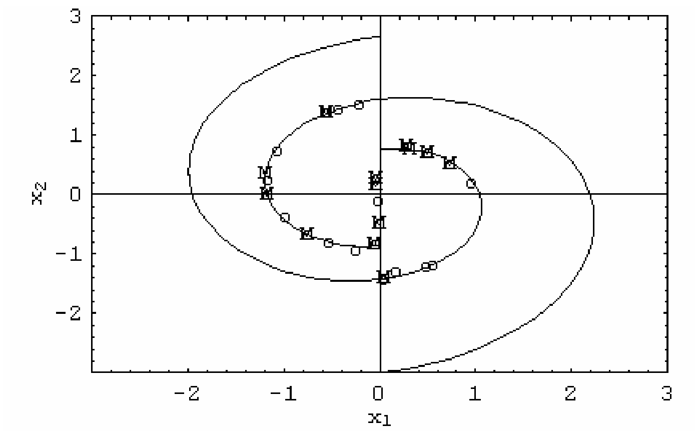
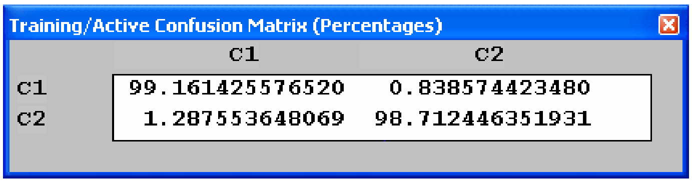


Figure 4-10: Classification boundary learnt by the neural network.

4.14.2 Classification statistics

NeuroSolutions generates a confusion matrix shown in Figure 4-11. This confusion matrix is best explained by putting it into the same format as the confusion matrix in Table 4-1.



Training/Active Confusion Matrix (Percentages)

	c1	c2
c1	99.161425576520	0.838574423480
c2	1.287553648069	98.712446351931

Figure 4-11: Confusion matrix for the "interlocked spiral" problem generated by NeuroSolutions.

Table 4-2: Confusion table with percentages.

		Predicted classification		Total true
		C_1	C_2	
Desired classification	C_1	$p_{11} = 99.161$	$p_{12} = 0.839$	$pC_1 = 40.561$
	C_2	$p_{21} = 1.288$	$p_{22} = 98.712$	$pC_2 = 59.439$
Total predicted		$pcol_1 = 40.986$	$pcol_2 = 59.014$	100

The confusion matrix says that $p_{11} = (n_{11} / row_1)100\%$ of samples belonging to class C_1 were classified correctly as $\tilde{C}_1$. Similarly, $p_{22} = (n_{22} / row_2)100\%$ of samples belonging to class C_2 were classified correctly as $\tilde{C}_2$. However, $p_{12} = (n_{12} / row_1)100\%$ of samples belonging to class C_1 were classified incorrectly as $\tilde{C}_2$. Similarly, $p_{21} = (n_{21} / row_2)100\%$ of samples belonging to class C_2 were classified in incorrectly as $\tilde{C}_1$. Percentage $pC_1 = (row_1 / N)100\%$ refers to the fraction of samples belonging to class C_1 and $pC_2 = (row_2 / N)100\%$ is the fraction of samples belonging to class C2. Similarly, $pcol_1 = (col_1 / N)100\%$ is the fraction of samples classified as $\tilde{C}_1$ and $pcol_2 = (col_2 / N)100\%$ is the fraction of samples classified as $\tilde{C}_2$. Clearly, $\sum_i pC_i = \sum_i pcol_i = 100$.

Let us suppose that we know the confusion matrix shown in Figure 4-11 and percentages pC_1 and pC_2. The statistics (4.29) - (4.31) can be easily calculated from the given percentages:

$$\text{classification error} = \left[100 - \sum_i \frac{p_{ii}\, pC_i}{100} \right] \% \qquad (4.32)$$

$$\text{sensitivity}(C_i) = p_{ii}\ \% \qquad (4.33)$$

$$\text{specificity}(C_i) = \frac{10^4 - pC_i(100 - p_{ii}) - \sum_k p_{ki}\, pC_k}{100 - pC_i} \% \qquad (4.34)$$

4.14.3 Classification problem – overlapping classes

The data generation program for this problem is available in `Overlaps.nb` Mathematica notebook. Explore the notebook to find more details about the optimum Bayesian classifier and misclassification probabilities calculation.

The objective is to distinguish between two classes of overlapping two-dimensional, Gaussian-distributed patterns labelled 1 and 2. The classes C_1 and C_2 consist of points with coordinates (x_1, x_2); their distribution is characterized by the following conditional probability density function:

$$f(x \mid C_i) = \frac{1}{2\pi\sigma_i^2} \exp\left(-\frac{1}{2\sigma_i^2} \left\| x - \mu_i \right\|^2\right), \quad i = 1,2 \qquad (4.35)$$

where

mean vector	$\mu_1 = [0, 0]^T$
mean vector	$\mu_2 = [2, 0]^T$
standard deviation	$\sigma_1 = 1$
standard deviation	$\sigma_2 = 2$

The two classes are assumed to be equally likely. Their probability density functions are shown in Figure 4-12.

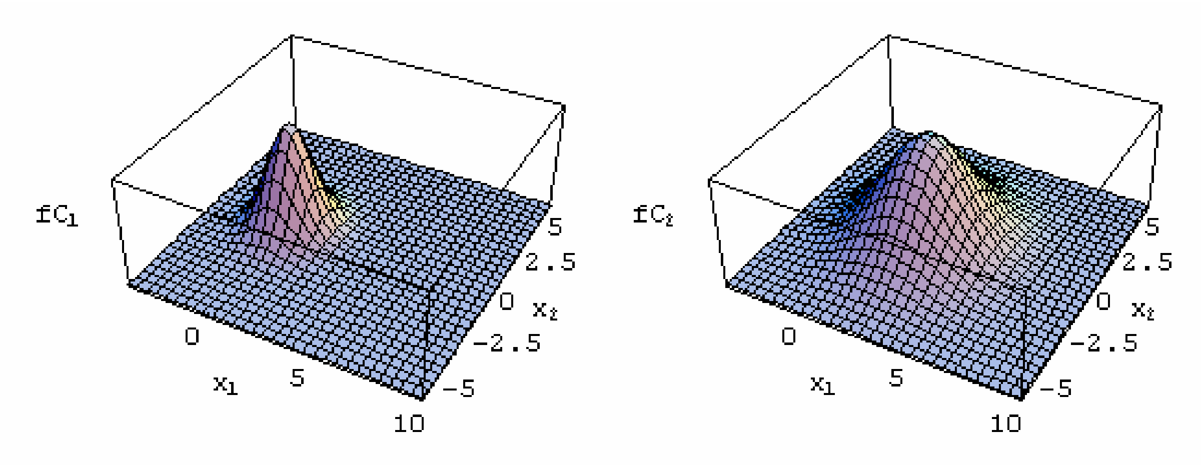


Figure 4-12: The probability density function of class C_1 (left) and class C_2 (right).

A back-propagation neural network was trained on 999 samples; 502 samples were from class C_1 and 497 from class C_2. The data are shown in Figure 4-13 along with the optimal Bayesian decision boundary. The data was split into training (80%) and validation (20%) subsets. NeuroSolutions simulation package was used to generate and train the network. The network configuration was 2-5-2 (2 inputs, 5 hidden neurons, 2 outputs). The hidden neurons had tangent hyperbolic activation function. A softmax activation function was used in the output layer. Notice that we used two outputs in this case. The output (1,0) stands for class C_1 and output (0,1) represents class C_2. The total number of weights was $(2+1)\times 5 + (5+1)\times 2 = 27$. The training subset was large enough: it had 799 samples, which is 29 times more than the number of weights.

Outputs (y_1, y_2) of a back-propagation neural network with softmax activation function at the output layer represent probabilities with which the sample belongs to class C_1 and C_2, respectively. The final decision will be based on a winner-take-all principle: the sample belongs to the category with the highest probability.

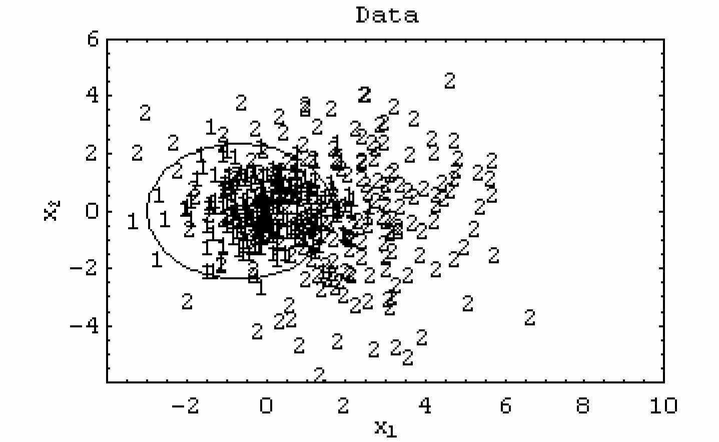


Figure 4-13: Training data normally distributed according to probability density functions (4.35). A circle shows the optimal Bayesian decision boundary.

The network was tested on 999 samples generated separately from the training set (out-of-sample testing); 498 samples were from class C_1 and 501 from class C_2. The Figure 4-14 shows samples that are close to the decision boundary. We can see that the neural network produces results that are close to the optimal Bayesian classificator. Figure 4-15 shows misclassified samples. In theory, samples inside the blue circle are classified as elements of class C_1 and samples outside the blue circle are classified as elements of class C_2. As the classes are overlapping misclassification is unavoidable. The other samples labelled “1” and “2” were classified correctly.

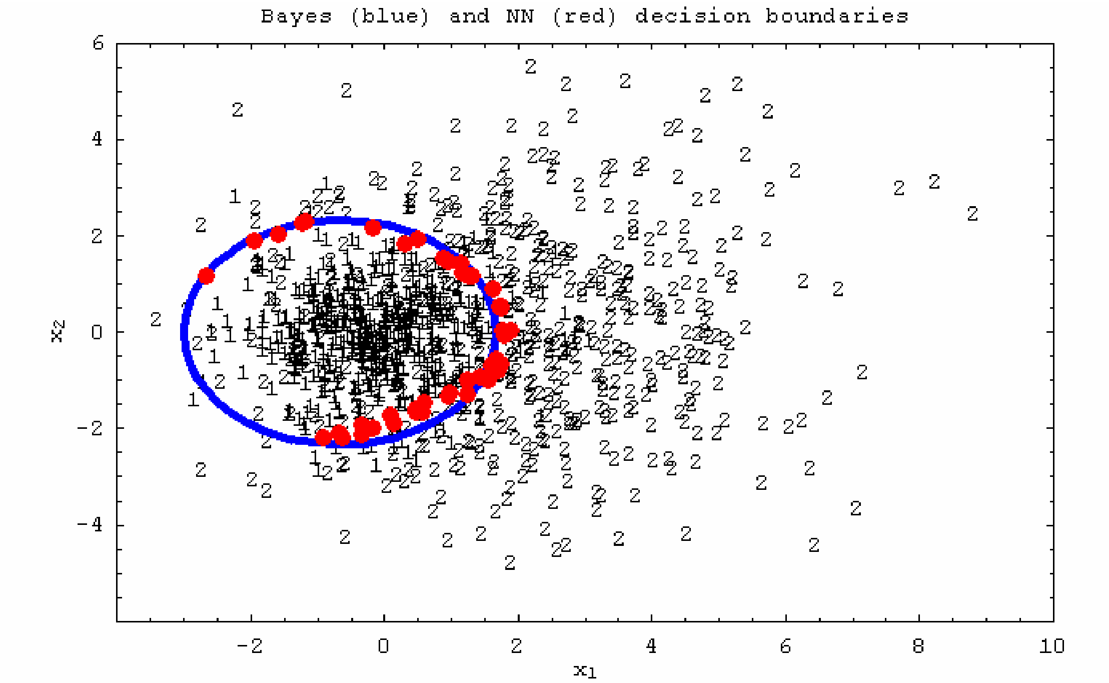


Figure 4-14: Samples corresponding to NN output 0.5±0.05 (red points).

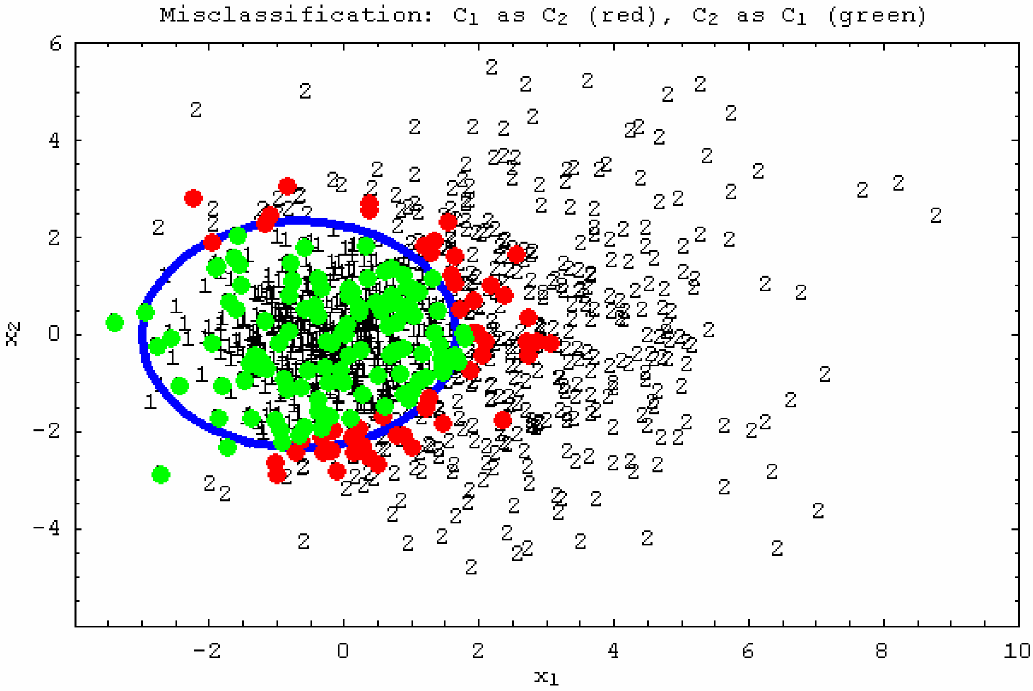


Figure 4-15: Misclassified samples.

Comparison of neural network performance with the optimum Bayesian classifier is shown in Table 4-3. It seems that the neural network performs better than the optimum Bayesian classifier, which is impossible. The network was trained and tested on a limited number of samples that happened to be "easier" to classify.

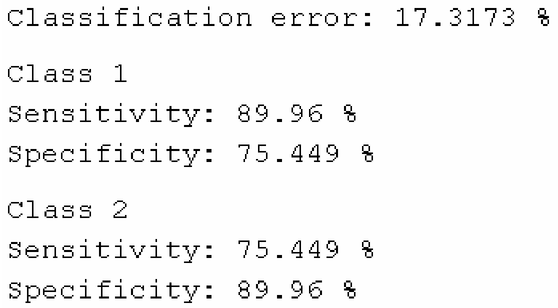
Table 4-3: Comparison of Bayesian and NN classifiers performance.

Bayes/**NN** – misclassification [%]	
C_1 classified as C_2	C_2 classified as C_2
10.56/**10.04**	26.42/**24.55**

Table 4-4: Confusion matrix and relative class sizes.

		Predicted classification		Total true
		C_1	C_2	
Desired classification	C_1	p_{11} = 89.960	p_{12} = 10.040	pC_1 = 49.850
	C_2	p_{21} = 24.551	p_{22} = 75.449	pC_2 = 50.150

A confusion matrix in Table 4-4 can be used to calculate classification statistics by using equations (4.32) - (4.34):



```
Classification error: 17.3173 %

Class 1
Sensitivity: 89.96 %
Specificity: 75.449 %

Class 2
Sensitivity: 75.449 %
Specificity: 89.96 %
```

The above statistics say the following:

1. 17.3% of samples are misclassified. As there were 999 samples in the testing set, about 172 are misclassified.
2. Sensitivity: If a sample belongs to C_1 there is an 89.96% chance it will classified as $\tilde{C}_1$. There will be about 10% of "false positives", i.e. a sample belongs to C_1 but is classified differently.
3. Specificity: If a sample does not belong to C_1 there is an 75.45% chance it will classified as not $\tilde{C}_1$ (i.e. $\tilde{C}_2$ in our case). There will be about 25% of "false negatives", i.e. a sample does not belong to C_1 but is classified as $\tilde{C}_1$ (being in C_1).

4.14.4 *Function approximation*

The data generation program for this problem is available in `funcApprox.nb` Mathematica notebook.

The objective is to fit data generated by the function

$$f(x) = 27x^4 - 60x^3 + 39x^2 - 6x \qquad (4.36)$$

Uniformly distributed noise from the interval [-0.1, 0.1] was added to the function values. The training data set consists of 250 samples and is shown in Figure 4-16. The data was equally split into training and validation subsets.

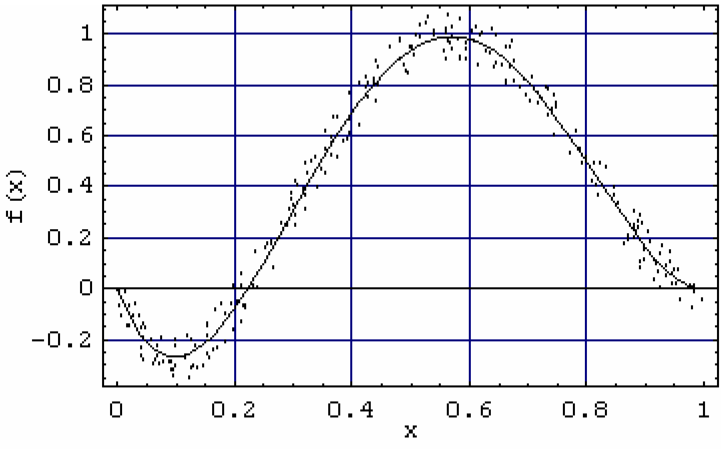


Figure 4-16: Training data along with the function f(x).

Neuro-Genetic Optimizer (NGO) generated a 1-5Lo-1Li neural network, i.e. the network with 1 input, 5 hidden neurons with sigmoid (logistic) activation functions and 1 output neuron with a linear activation function. (Output neurons are usually linear in case of function approximation.)

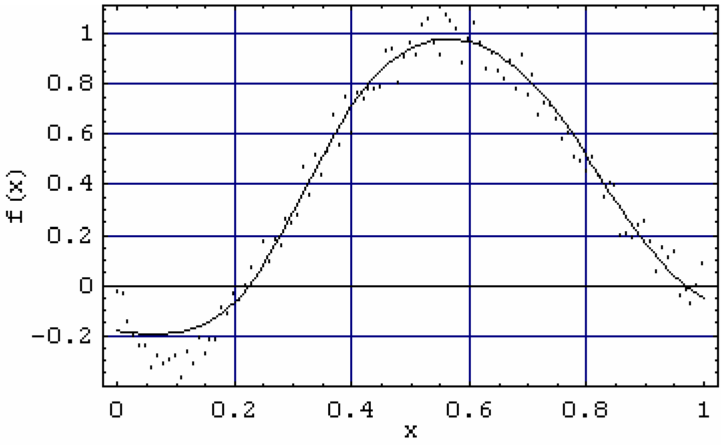


Figure 4-17: Testing data (dots) and neural network output (solid line). The coefficient of determination $R^2 = 0.973$.

The trained neural network was tested on 100 data generated in the same way as the training data. The data were, however, different form the training ones. The quality of data approximation is shown in Figure 4-17. We can see that the approximation is generally good. If we want to improve the fitting for small values of *x* we would have to locate more training samples into that region.

A goodness of fit can be measured by the *coefficient of determination* denoted by R^2. The coefficient is calculated from the formula

$$R^2 = \frac{SST - SSE}{SST} \tag{4.37}$$

The *total sum of squares*

$$SST = \sum_{i=1}^{N} (d_i - \bar{d})^2 \tag{4.38}$$

represents the error present if there is no regression equation fitted at all. The term *SSE* is the *sum of squares of regression residuals*:

$$SSE = \sum_{i=1}^{N} (y_i - d_i)^2 \tag{4.39}$$

In our particular case, the coefficient of determination $R^2 = 0.973$, which means, that 97.3% of the data variation can be explained by the neural network model.

Several neural networks can be compared by using an *adjusted coefficient of determination* R^2_{adjust}, which is calculated according to the following formula:

$$R^2_{adjust} = 1 - \frac{n-1}{n-p}(1 - R^2) \tag{4.40}$$

where *n* is the number of samples and *p* is the number of adjustable parameters (synaptic connections).

4.15 Problems

1. Use a back-propagation neural network to solve the XOR problem shown in Figure 3-13.

2. Train a back-propagation neural network to learn the following functions:

   a) $f(x) = 1/x, \quad 1 \le x \le 100$
   b) $f(x) - \log_{10} x, \quad 1 \le x \le 10$
   c) $f(x) = \exp(-x), \quad 1 \le x \le 10$
   d) $f(x) = \sin x, \quad 0 \le x \le \pi/2$

For each function do the following:

a) Set up two sets of data, one for network training, and the other for testing.
b) Use the training data for the network training.
c) Evaluate the accuracy of the network by using the test data.

Use a single hidden layer but with a variable number of hidden neurons. Investigate how the network performance is affected by varying size of the hidden layer.

3. Consider the *encoding problem* in which a set of orthogonal input patterns are mapped with a set of orthogonal output patterns through a small set of hidden neurons. Figure 4-18 shows the basic architecture for solving this problem. Essentially, the problem is to learn an encoding of a *p*-bit pattern into a $\log_2(p)$-bit pattern, and then learn to decode this representation into the output pattern. Construct the mapping generated by a back-propagation for the case of identity mapping.

Input pattern	Output pattern
1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0	0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0	0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0	0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0	0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0	0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0	0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1

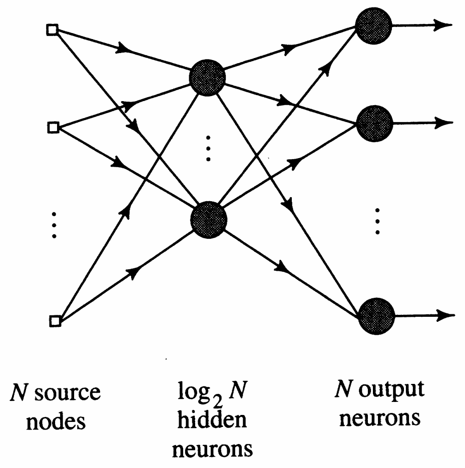


Figure 4-18: Encoding problem; training patterns and the network architecture.

Hints: (a) You may consider using the same data file both for inputs and desired outputs. (b) As there are not enough data samples (only eight) it may be useful to repeat the same data several times (e.g. 10 times) in the data file. (c) Try to figure out how the data are represented by three hidden neurons ($\log_2 8 = 3$).

4. The data presented in Table 4-5 show the weights of eye lenses of wild Australian rabbits as a function of age. Using the back-propagation algorithm, design a multilayer perceptron that provides a nonlinear least squares approximation to this data set. Use the trained perceptron to predict the eye lenses weights for rabbits 100, 200, 300, …, 800 days old.

Table 4-5: Weights [mg] of eye lenses of wild Australian rabbits as a function of age [days].

Age	Weight	Age	Weight	Age	Weight	Age	Weight
15	21.66	75	94.6	218	174.18	338	203.23
15	22.75	82	92.5	218	173.03	347	188.38
15	22.3	85	105	219	173.54	354	189.7
18	31.25	91	101.7	224	178.86	357	195.31
28	44.79	91	102.9	225	177.68	375	202.63
29	40.55	97	110	227	173.73	394	224.82
37	50.25	98	104.3	232	159.98	513	203.3
37	46.88	125	134.9	232	161.29	535	209.7
44	52.03	142	130.68	237	187.07	554	233.9
50	63.47	142	140.58	246	176.13	591	234.7
50	61.13	147	155.3	258	183.4	648	244.3
60	81	147	152.2	276	186.26	660	231
61	73.09	150	144.5	285	189.66	705	242.4
64	79.09	159	142.15	300	186.09	723	230.77
65	79.51	165	139.81	301	186.7	756	242.57
65	65.31	183	153.22	305	186.8	768	232.12
72	71.9	192	145.72	312	195.1	860	246.7
75	86.1	195	161.1	317	216.41		

Hints: (a) Make sure that the network generalizes well. (b) Experiment with the number of hidden neurons and eventually select a reasonable simple network. (c) Present your results in a nice graphical format.

5. The data presented in the file `Iris.xls` and shown in Table 4-6 contain 150 records. Each record consists of NN inputs (sepal length, sepal width, petal length, and petal width) and desired outputs. The outputs represent three classes of Iris plants (Setosa, Versicolor, and Virginica). Design a multilayer perceptron that classifies the Iris plants. Explore several NNs and eventually select one of them as the final solution. You should justify you selection in terms of NN complexity and performance.

6. NeuroSolutions contain the following learning algorithms that can be used to train a back-propagation network: Step, Momentum, Quickprop, Delta-bar-delta (DBD), and Conjugate gradient. Investigate the relative performance of the available learning algorithms on any of the above problems 1-5.

Table 4-6: Iris classification data.

SepalL	SepalW	PetalL	PetalW	Setosa	Versicolor	Virginica	SepalL	SepalW	PetalL	PetalW	Setosa	Versicolor	Virginica
5.1	3.5	1.4	0.2	1	0	0	6.6	3	4.4	1.4	0	1	0
4.9	3	1.4	0.2	1	0	0	6.8	2.8	4.8	1.4	0	1	0
4.7	3.2	1.3	0.2	1	0	0	6.7	3	5	1.7	0	1	0
4.6	3.1	1.5	0.2	1	0	0	6	2.9	4.5	1.5	0	1	0
5	3.6	1.4	0.2	1	0	0	5.7	2.6	3.5	1	0	1	0
5.4	3.9	1.7	0.4	1	0	0	5.5	2.4	3.8	1.1	0	1	0
4.6	3.4	1.4	0.3	1	0	0	5.5	2.4	3.7	1	0	1	0
5	3.4	1.5	0.2	1	0	0	5.8	2.7	3.9	1.2	0	1	0
4.4	2.9	1.4	0.2	1	0	0	6	2.7	5.1	1.6	0	1	0
4.9	3.1	1.5	0.1	1	0	0	5.4	3	4.5	1.5	0	1	0
5.4	3.7	1.5	0.2	1	0	0	6	3.4	4.5	1.6	0	1	0
4.8	3.4	1.6	0.2	1	0	0	6.7	3.1	4.7	1.5	0	1	0
4.8	3	1.4	0.1	1	0	0	6.3	2.3	4.4	1.3	0	1	0
4.3	3	1.1	0.1	1	0	0	5.6	3	4.1	1.3	0	1	0
5.8	4	1.2	0.2	1	0	0	5.5	2.5	4	1.3	0	1	0
5.7	4.4	1.5	0.4	1	0	0	5.5	2.6	4.4	1.2	0	1	0
5.4	3.9	1.3	0.4	1	0	0	6.1	3	4.6	1.4	0	1	0
5.1	3.5	1.4	0.3	1	0	0	5.8	2.6	4	1.2	0	1	0
5.7	3.8	1.7	0.3	1	0	0	5	2.3	3.3	1	0	1	0
5.1	3.8	1.5	0.3	1	0	0	5.6	2.7	4.2	1.3	0	1	0
5.4	3.4	1.7	0.2	1	0	0	5.7	3	4.2	1.2	0	1	0
5.1	3.7	1.5	0.4	1	0	0	5.7	2.9	4.2	1.3	0	1	0
4.6	3.6	1	0.2	1	0	0	6.2	2.9	4.3	1.3	0	1	0
5.1	3.3	1.7	0.5	1	0	0	5.1	2.5	3	1.1	0	1	0
4.8	3.4	1.9	0.2	1	0	0	5.7	2.8	4.1	1.3	0	1	0
5	3	1.6	0.2	1	0	0	6.3	3.3	6	2.5	0	0	1
5	3.4	1.6	0.4	1	0	0	5.8	2.7	5.1	1.9	0	0	1
5.2	3.5	1.5	0.2	1	0	0	7.1	3	5.9	2.1	0	0	1
5.2	3.4	1.4	0.2	1	0	0	6.3	2.9	5.6	1.8	0	0	1
4.7	3.2	1.6	0.2	1	0	0	6.5	3	5.8	2.2	0	0	1
4.8	3.1	1.6	0.2	1	0	0	7.6	3	6.6	2.1	0	0	1
5.4	3.4	1.5	0.4	1	0	0	4.9	2.5	4.5	1.7	0	0	1
5.2	4.1	1.5	0.1	1	0	0	7.3	2.9	6.3	1.8	0	0	1
5.5	4.2	1.4	0.2	1	0	0	6.7	2.5	5.8	1.8	0	0	1
4.9	3.1	1.5	0.1	1	0	0	7.2	3.6	6.1	2.5	0	0	1
5	3.2	1.2	0.2	1	0	0	6.5	3.2	5.1	2	0	0	1
5.5	3.5	1.3	0.2	1	0	0	6.4	2.7	5.3	1.9	0	0	1
4.9	3.1	1.5	0.1	1	0	0	6.8	3	5.5	2.1	0	0	1
4.4	3	1.3	0.2	1	0	0	5.7	2.5	5	2	0	0	1
5.1	3.4	1.5	0.2	1	0	0	5.8	2.8	5.1	2.4	0	0	1
5	3.5	1.3	0.3	1	0	0	6.4	3.2	5.3	2.3	0	0	1
4.5	2.3	1.3	0.3	1	0	0	6.5	3	5.5	1.8	0	0	1
4.4	3.2	1.3	0.2	1	0	0	7.7	3.8	6.7	2.2	0	0	1
5	3.5	1.6	0.6	1	0	0	7.7	2.6	6.9	2.3	0	0	1
5.1	3.8	1.9	0.4	1	0	0	6	2.2	5	1.5	0	0	1
4.8	3	1.4	0.3	1	0	0	6.9	3.2	5.7	2.3	0	0	1
5.1	3.8	1.6	0.2	1	0	0	5.6	2.8	4.9	2	0	0	1
4.6	3.2	1.4	0.2	1	0	0	7.7	2.8	6.7	2	0	0	1
5.3	3.7	1.5	0.2	1	0	0	6.3	2.7	4.9	1.8	0	0	1
5	3.3	1.4	0.2	1	0	0	6.7	3.3	5.7	2.1	0	0	1
7	3.2	4.7	1.4	0	1	0	7.2	3.2	6	1.8	0	0	1
6.4	3.2	4.5	1.5	0	1	0	6.2	2.8	4.8	1.8	0	0	1
6.9	3.1	4.9	1.5	0	1	0	6.1	3	4.9	1.8	0	0	1
5.5	2.3	4	1.3	0	1	0	6.4	2.8	5.6	2.1	0	0	1
6.5	2.8	4.6	1.5	0	1	0	7.2	3	5.8	1.6	0	0	1
5.7	2.8	4.5	1.3	0	1	0	7.4	2.8	6.1	1.9	0	0	1
6.3	3.3	4.7	1.6	0	1	0	7.9	3.8	6.4	2	0	0	1
4.9	2.4	3.3	1	0	1	0	6.4	2.8	5.6	2.2	0	0	1
6.6	2.9	4.6	1.3	0	1	0	6.3	2.8	5.1	1.5	0	0	1
5.2	2.7	3.9	1.4	0	1	0	6.1	2.6	5.6	1.4	0	0	1
5	2	3.5	1	0	1	0	7.7	3	6.1	2.3	0	0	1
5.9	3	4.2	1.5	0	1	0	6.3	3.4	5.6	2.4	0	0	1
6	2.2	4	1	0	1	0	6.4	3.1	5.5	1.8	0	0	1
6.1	2.9	4.7	1.4	0	1	0	6	3	4.8	1.8	0	0	1
5.6	2.9	3.6	1.3	0	1	0	6.9	3.1	5.4	2.1	0	0	1
6.7	3.1	4.4	1.4	0	1	0	6.7	3.1	5.6	2.4	0	0	1
5.6	3	4.5	1.5	0	1	0	6.9	3.1	5.1	2.3	0	0	1
5.8	2.7	4.1	1	0	1	0	5.8	2.7	5.1	1.9	0	0	1
6.2	2.2	4.5	1.5	0	1	0	6.8	3.2	5.9	2.3	0	0	1
5.6	2.5	3.9	1.1	0	1	0	6.7	3.3	5.7	2.5	0	0	1
5.9	3.2	4.8	1.8	0	1	0	6.7	3	5.2	2.3	0	0	1
6.1	2.8	4	1.3	0	1	0	6.3	2.5	5	1.9	0	0	1
6.3	2.5	4.9	1.5	0	1	0	6.5	3	5.2	2	0	0	1
6.1	2.8	4.7	1.2	0	1	0	6.2	3.4	5.4	2.3	0	0	1
6.4	2.9	4.3	1.3	0	1	0	5.9	3	5.1	1.8	0	0	1

5 The Hopfield network

The *Hopfield network* is a recurrent network that *stores information in a dynamically stable configuration*. The standard discrete version of the Hopfield network uses the McCulloch-Pitts model of a neuron (1.5) with the output either -1 or 1. The network operates in an unsupervised manner. It may be used as a *content-addressable memory* or as a platform for solving optimization problems of a combinatorial kind.

The Hopfield network is a single large layer of neurons. Every neuron connects to every other, usually in a symmetric way.

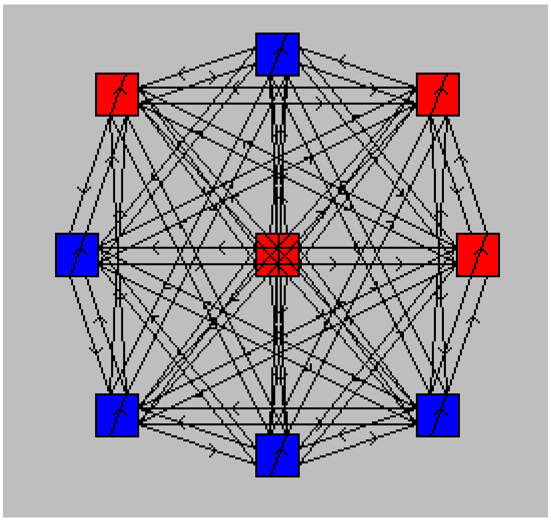


Figure 5-1: Architecture of a simple Hopfield network.

Information is stored in the Hopfield network by means of synaptic weights. The stored pattern is retrieved, given a reasonable subset of the information content of that pattern. Moreover, a content-addressable memory is *error-correcting* in the sense that it can override inconsistent information in the cues presented to it.

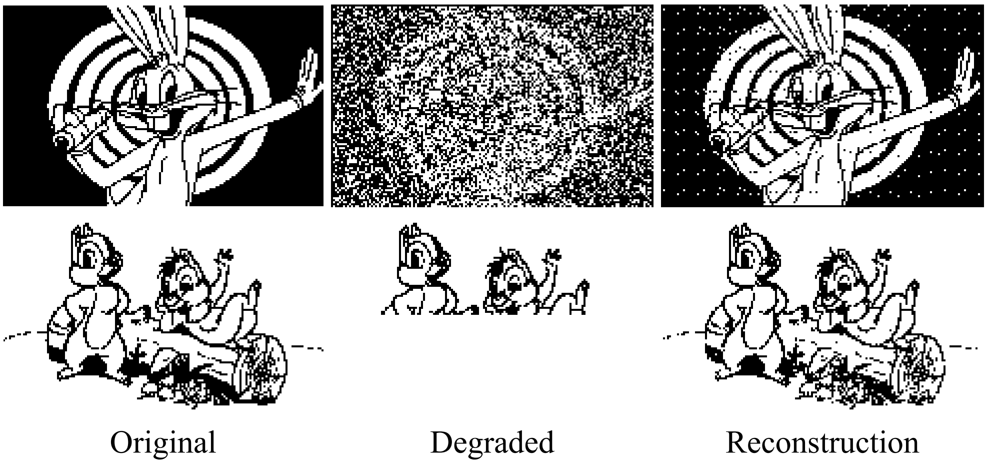


Figure 5-2: Hopfield network reconstructing degraded images from noisy (top) or partial (bottom) cues.

The Hopfield network is designed to store a number of patterns. It does this by creating an *energy surface* which has attractors representing each of the stored patterns. The noisy and partial cues are states of the system, which are close to the attractors. As a Hopfield network cycles it slides from the noisy pattern down the energy surface into the closest attractor representing the closest stored pattern.

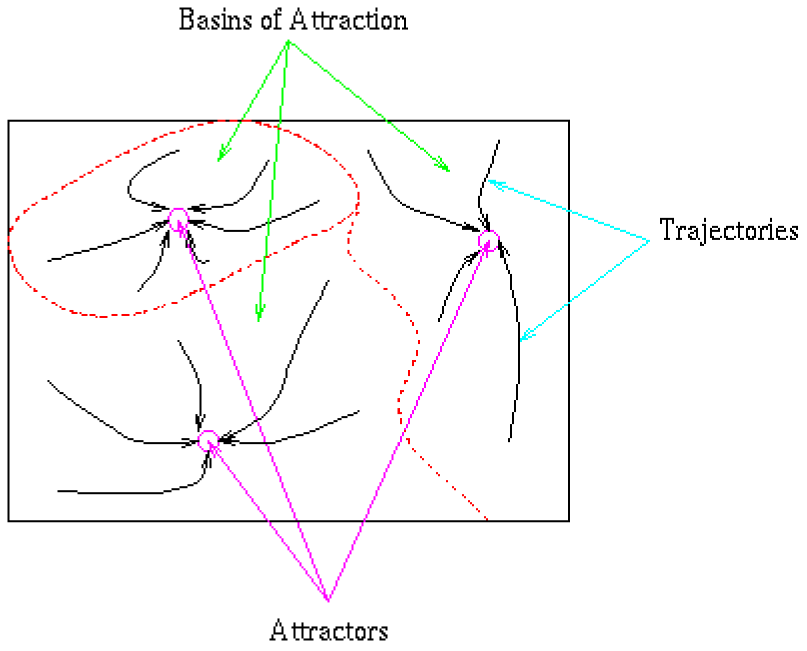


Figure 5-3: Attractors, basins of attraction and trajectories.

5.1 The storage of patterns

Suppose we wish to store a set of p-dimensional binary vectors d^1, … , d^N. We call these vectors *fundamental memories*, representing the patterns to be stored in the network. According to the Hebb's postulate of learning the synapse w_{ji} between neurons d_i and d_j should increase if both neurons are simultaneously activated. It means that if $d_i = d_j = \pm 1$ the synapse w_{ji} should increase, otherwise it should decrease. The simplest way how to implement the Hebb's postulate is to use the outer product of the fundamental memories:

$$w = \frac{1}{p}\sum_{i=1}^{N} d^i (d^i)^T - \frac{N}{p} I \tag{5.1}$$

where w is the $p \times p$ synaptic weight matrix of the network and I is the identity matrix. Notice that w is symmetric, i.e. $w_{ij} = w_{ji}$. The reason for using $1/p$ as the constant of proportionality is to simplify the mathematical description of information retrieval; it does not have to be used. If not used the elements of weight matrix w are integers. The weight matrix has zeroes on the main diagonal due to the term $(N/p)I$. It means that neurons do not have any self-feedback.

The learning rule (5.1) is a "one-shot" computation, no iterations are involved.

5.1.1 Example

Let us calculate the synaptic weight matrix that stores the following two 4-dimensional patterns:

$$d^1 = \begin{bmatrix} +1 \\ -1 \\ -1 \\ +1 \end{bmatrix}, \quad d^2 = \begin{bmatrix} +1 \\ +1 \\ -1 \\ +1 \end{bmatrix}, \quad p = 4, \quad N = 2$$

Outer products are:

$$d^1(d^1)^T = \begin{bmatrix} +1 \\ -1 \\ -1 \\ +1 \end{bmatrix} \begin{bmatrix} +1 & -1 & -1 & +1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}, \quad d^2(d^2)^T = \begin{bmatrix} 1 & 1 & -1 & 1 \\ 1 & 1 & -1 & 1 \\ -1 & -1 & 1 & -1 \\ 1 & 1 & -1 & 1 \end{bmatrix}$$

Finally, the weight matrix

$$w = \frac{1}{4} \begin{bmatrix} 2 & 0 & -2 & 2 \\ 0 & 2 & 0 & 0 \\ -2 & 0 & 2 & -2 \\ 2 & 0 & -2 & 2 \end{bmatrix} - \frac{2}{4} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -0.5 & 0.5 \\ 0 & 0 & 0 & 0 \\ -0.5 & 0 & 0 & -0.5 \\ 0.5 & 0 & -0.5 & 0 \end{bmatrix}$$

If the constant 1/*p* is not used the weight matrix will be

$$w = \begin{bmatrix} 0 & 0 & -2 & 2 \\ 0 & 0 & 0 & 0 \\ -2 & 0 & 0 & -2 \\ 2 & 0 & -2 & 0 \end{bmatrix}$$

5.2 The retrieval of patterns

The sum of postsynaptic potentials delivered to neuron *i* is

$$v_i = \sum_{j=1}^{p} w_{ij} x_j = w_i x \qquad (5.2)$$

where w_i is the *i*-th row of the weight matrix *w*. The sum of postsynaptic potentials delivered to all neurons can be simply expressed in a matrix form

$$v = wx \qquad (5.3)$$

The neuron *i* modifies its output according to the *dynamical rule*

$$x_i(t+1) = \begin{Bmatrix} +1, & v_i > 0 \\ x_i(t), & v_i = 0 \\ -1, & v_i < 0 \end{Bmatrix} \qquad (5.4)$$

where *t* is the iteration step.

During the retrieval phase, a *p*-dimensional vector *s*, called a *probe*, is imposed on the Hopfield network as its initial state, i.e. $x(0) = s$. The probe vector has elements equal to ±1 and represents an incomplete or noisy version of a fundamental memory of the network. Information retrieval then proceeds in accordance with the dynamical rule (5.4).

Each neuron *i* of the network randomly but at some fixed rate examines the net activation potential v_i applied to it. If, at that instant of time, the potential v_i is greater than zero, neuron *i* will switch its state to +1, or remain in that state if it is already there. Similarly, if the potential v_i is less than zero, neuron *i* will switch its state to -1, or remain in that state if it is already there. If v_i is exactly zero, neuron *i* is left in its previous state, regardless of whether it was on or off. The state updating from one iteration to the next is therefore deterministic, but the selection of a neuron to perform the updating is done randomly. The asynchronous (serial) updating procedure described here is continued until there are no further changes to report. The state vector *x* that is reached is called a *stable state* or a *fixed point* of the state space of the system. We will show later that the Hopfield network will always converge to a stable state when the retrieval operation is performed *asynchronously*.

It is also possible to update all neurons *synchronously* (so called *Little model*), i.e. to calculate the state $x(t+1)$ at the next iteration step and perform the updates afterwards for all neurons. Whereas the asynchronously updated Hopfield network always converges to a stable state, the Little model will always converge to a stable state or a limit cycle of length at most 2.

5.3 Summary of the Hopfield model

1. *Storage* (learning). A set of *p*-dimensional binary vectors $d^1, \ldots, d^N$ (fundamental memories) are stored in the synaptic weights of the network according to the formula (5.1).
2. *Initialization*. All neurons are initialized to their starting values by a probe vector *s*, i.e. $x(0) = s$.
3. *Iteration until convergence*. The outputs of all neurons are updated asynchronously (i.e. one at a time) according to the rule (5.4). The iteration is repeated until the state vector *x* remains unchanged.
4. *Output*. Let x_{fixed} denotes the stable state computed at the end of step 3. The x_{fixed} is the resulting output of the network.

5.4 Energy function

Consider a Hopfield network with symmetric weight matrix $w = w^T$. The diagonal elements w_{ii} of the matrix w are equal to zero. The network state at an iteration step t is $x(t)$. The *energy function* of the Hopfield network is defined by

$$E(t) = -\frac{1}{2} x(t)^T w\, x(t) = -\frac{1}{2} \sum_{i=1, i \neq j}^{p} \sum_{j=1}^{p} w_{ji} x_i(t) x_j(t) \qquad (5.5)$$

The energy change ΔE due to a change Δx_j in the state neuron j is given by

$$\Delta E(t) = -\Delta x_j(t) \sum_{i=1, i \neq j}^{p} w_{ji} x_i(t) \qquad (5.6)$$

According to (5.2)

$$v_j = \sum_{i=1}^{p} w_{ji} x_i = w_j x \qquad (5.7)$$

Therefore

$$\Delta E(t) = -\Delta x_j(t) v_j(t) \qquad (5.8)$$

Let us investigate the equation (5.8) in more detail having the dynamical rule (5.4) in mind:

1. The output of the neuron j either stays unchanged or changes from -1 to +1 ($\Delta x_j = 2$) if $v_j > 0$. It means that $\Delta E < 0$.
2. The output of the neuron j either stays unchanged or changes from +1 to -1 ($\Delta x_j = -2$) if $v_j < 0$. It means that $\Delta E < 0$.
3. The output of the neuron j does not change ($\Delta x_j = 0$) if $v_j = 0$. It means that $\Delta E = 0$.

The above analysis shows that whenever a neuron output (state variable) gets updated the energy function decreases. The magnitude of the decrease is equal to $2\left|v_j(t)\right|$, which is not an infinitesimally small value. The energy function cannot keep decreasing for ever because its smallest possible value is $-\frac{1}{2} \sum_{i=1, i \neq j}^{p} \sum_{j=1}^{p} \left|w_{ji}\right|$, as can be seen from (5.5).

The above observation indicates that the energy function $E(t)$ is a monotonically decreasing function with a lower limit. According to the Liapunov stability theory state changes will continue until a local minimum of the energy landscape is reached, at which point the network relaxation process (information retrieval) stops. The *energy landscape* describes the dependence of the energy function E on the network state for a specified set of synaptic weights.

The local minima of the energy landscape correspond to the attractors of the phase space (see Figure 5-3), which are the assigned fundamental memories of the network. To guarantee the retrieval of fundamental memory, two conditions must be satisfied:

1. The fundamental memories are all stable.
2. The stable patterns have sizeable basin of attraction.

Suppose that the retrieval algorithm is initiated with a probe representing a starting point in the energy landscape. Then, as the algorithm iterates its way to a solution, the starting point moves through the energy landscape toward a local minimum. When the local minimum is actually reached, the algorithm stays there, because everywhere else in the close vicinity is an uphill climb.

5.5 Spurious states

When a Hopfield network is used to store N patterns by means of the Hebb learning formula (5.1) for synaptic weights, the network is usually found to have *spurious attractors* (*spurious states*). Spurious states represent stable states of the network that are different from the fundamental memories of the network. How do spurious states arise?

First of all, we note that the energy function E (5.5) is symmetric in the sense that its value remains unchanged if the states of the neurons are reversed:

$$E(t) = -\frac{1}{2} x(t)^T w\, x(t) = -\frac{1}{2} (-x(t))^T w(-x(t)) \tag{5.9}$$

It means that the network also stores negative images of its fundamental memories.

Second, there is an attractor for every *mixture* of the stored patterns. A mixture state corresponds to a linear combination of an *odd* number of patterns:

$$d^{mixture} = \mathrm{sgn}(\sum \alpha_i d^i) \tag{5.10}$$

Third, for a large number N of fundamental memories, the energy landscape has local minima that are not correlated with any of these memories embedded in the network. Such spurious states are sometimes referred to as *spin-glass states*, by analogy with spin-glass models of statistical mechanics.

The absence of self-feedback (w_{ii} = 0) has a beneficial effect as far as spurious states are concerned. If $w_{ii} \neq 0$ for all i, additional stable spurious states might be produced in the neighbourhood of a desired attractor.

Unfortunately, the fundamental memories of a Hopfield network are not always stable. The probable instability of fundamental memories and the possible existence of spurious states, tend to decrease the efficiency of a Hopfield network as a content-addressable memory.

Let us investigate the probable existence of spurious states further. The matrix of synaptic weights w_{ji} from neuron i to neuron j defined in (5.1) is calculated over all fundamental memories. This equation is reproduced here for convenience of presentation:

$$w = \frac{1}{p}\sum_{i=1}^{N} d^i (d^i)^T - \frac{N}{p} I \tag{5.11}$$

The sum of postsynaptic potentials delivered to neurons after a probe s is presented is (see (5.3))

$$v = ws = \frac{1}{p}\sum_{i=1, i\neq j}^{N} d^i (d^i)^T s + \frac{1}{p} d^j (d^j)^T s - \frac{N}{p} Is \tag{5.12}$$

Let us suppose that the probe s equals to the fundamental memory d^j. Then (5.12) becomes

$$v = d^j + \frac{1}{p}\sum_{i=1, i\neq j}^{N} d^i (d^i)^T d^j - \frac{N}{p} I d^j = (1 - \frac{N}{p}) d^j + \frac{1}{p}\sum_{i=1, i\neq j}^{N} d^i (d^i)^T d^j \tag{5.13}$$

because $(d^j)^T s = (d^j)^T d^j = p$. The second term in the above equation can be viewed as a result of "crosstalk" between the elements of fundamental memory d^j under test and some other fundamental memories. This second term may be regarded as the "noise" component of v. The scalar product $(d^i)^T d^j = 0$ if the fundamental memory d^j is orthogonal to other fundamental memories; in that case the "noise" will be equal to zero and spurious states will not occur.

The existence of spurious states is more likely if more fundamental memories are stored in the network. The Hopfield network storage capacity is, therefore, limited. The following estimates for the maximum number of fundamental memories N_{max} have been derived (p is the number of neurons in the network):

1. *Most* of the fundamental memories are recalled perfectly with probability 99% if

$$N_{\max} < \frac{p}{2 \ln p} \tag{5.14}$$

2. *All* of the fundamental memories are recalled perfectly with probability 99% if

$$N_{\max} < \frac{p}{4 \ln p} \tag{5.15}$$

The above estimates refer to randomly generated fundamental memories. In practice, the fundamental memories will usually represent certain patterns and the storage capacity will depend on the degree of "crosstalk" among them and may be much smaller than N_{max} given in (5.14) and (5.15).

The Hebb's learning postulate (5.11) is not the only or the best way how to calculate the weight matrix *w*. There is an ongoing research on how to learn the weight matrix *w* to increase the capacity of the Hopfield network.

5.6 Computer experiment

Hopfield network has been implemented in Mathematica and is available in the file `Hopfield.nb`. Six files `stmpxx.jpg` with two-dimensional images are also provided.

Three images (fundamental memories) shown in Figure 5-4 were stored in a Hopfield network. The images consist of 32×32 pixels, which means that the number of neurons p = 1024. The weight matrix w has 1024 rows and columns, and zeroes on the main diagonal. The matrix is symmetric but no attempt was made to implement it in a storage-saving way. The constant of proportionality $1/p$ in (5.11) was not used so that all weights are integers, which considerably shortens computation time.

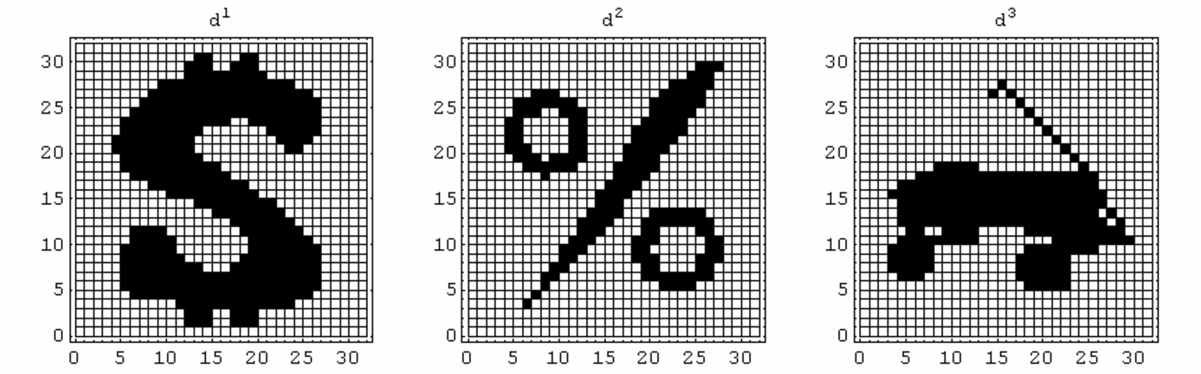


Figure 5-4: Three fundamental memories d^1, d^2, and d^3.

All fundamental memories and their negative images are represented by stable states. This can be easily tested by making the probe s equal to the fundamental memories or their negative images.

If the probe s is a random vector all stable states will be probably reached after sufficient number of attempts, i.e. after presenting different random probes many times. Steady states with very small basins of attraction may be difficult to reach.

The Hopfield network with the fundamental memories shown in Figure 5-4 developed a steady state (and its negative image), shown in Figure 5-5, which is a mixture of fundamental memories:

$$x^{mixture} = \text{sgn}(0.331d^1 + 0.475d^2 + 0.380d^3) \qquad (5.16)$$

If we attempt to store more fundamental memories we may find that some of them will not be stable (therefore will not be retrievable) and that mixtures and spurious states will be created. The theoretical capacity of a Hopfield network with 1024 neurons is 37 < N_{max} < 74 (see equations (5.14) and (5.15)) when storing random fundamental memories. This computer experiment with images indicates that the actual capacity is about 10 times smaller.

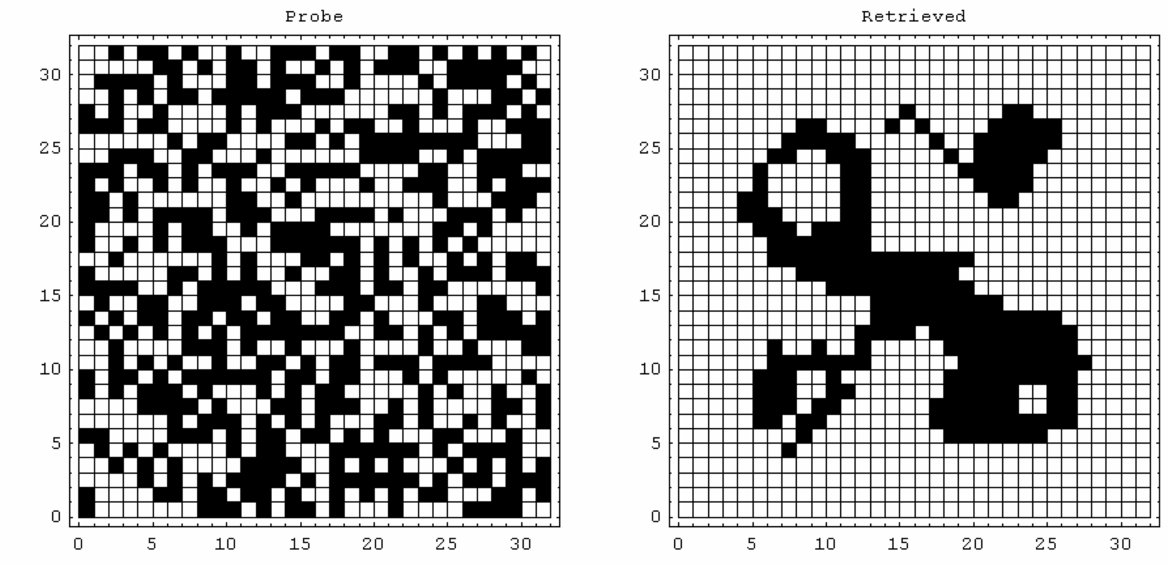


Figure 5-5: A mixture of fundamental memories created its own stable state.

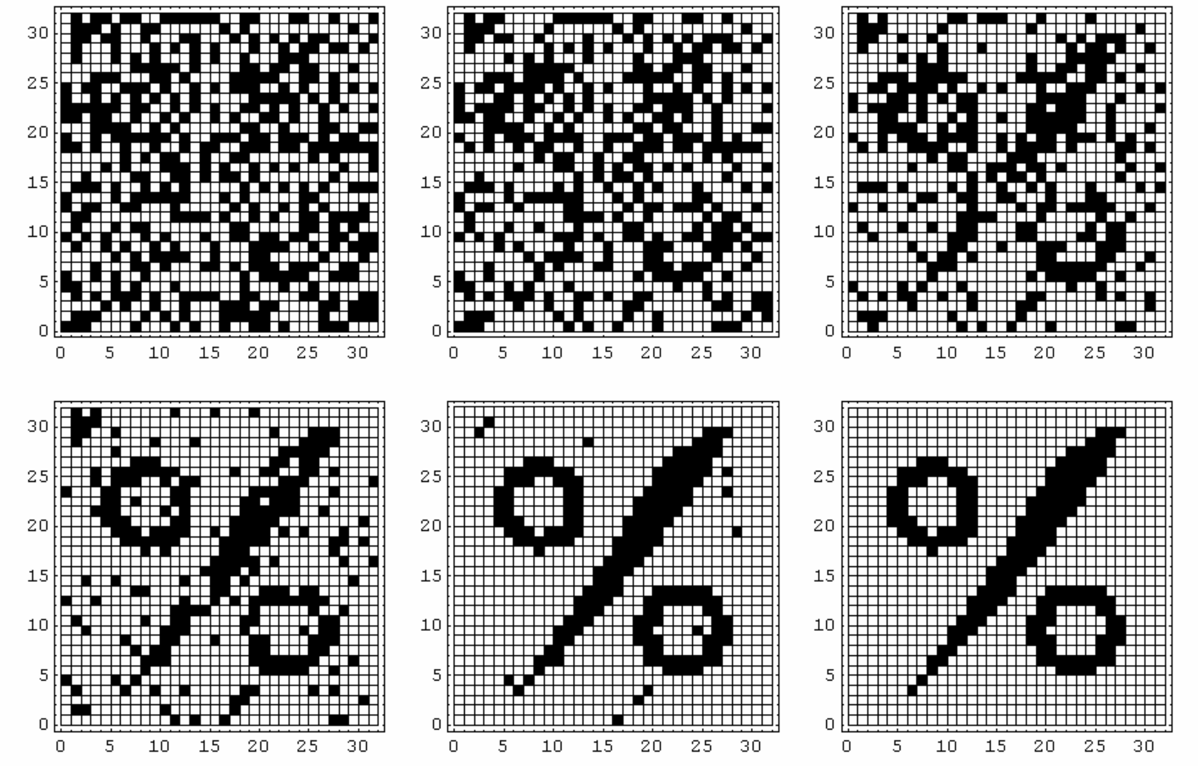


Figure 5-6: Gradual emergence of a fundamental memory during relaxation.

Six snapshots from an animated relaxation are shown in Figure 5-6 and Figure 5-7. The upper right snapshots in both figures are similar; small differences drive the relaxation process to different basins of attraction.

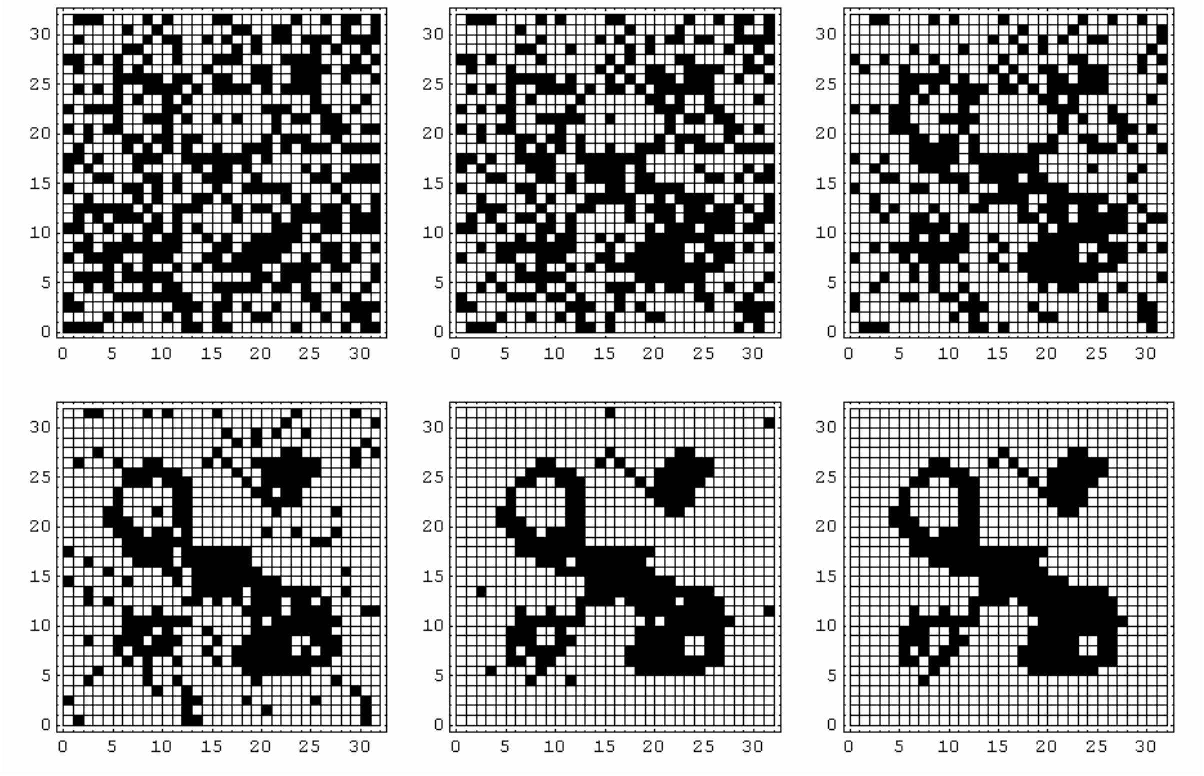


Figure 5-7: Gradual emergence of a mixture of fundamental memories during relaxation.

5.7 Combinatorial optimization problem

This class of optimization problems includes the *travelling salesman problem* (TSP). Given the positions of a specified number of cities, the problem is to find the shortest tour that starts and finishes at the same city. The TSP is thus simple to state but hard to solve exactly. Hopfield and Tank (*Biological Cybernetics*, **52** (1985), 141-152) demonstrated how a Hopfield network can be used to solve the TSP.

Let us consider 5 cities as shown in Figure 5-8. The cities should be visited in such a way that

1. Each city is visited only once
2. Only one city is visited at a time
3. The total distance travelled is minimized
4. The journey starts and ends in the same city

There are 5! = 120 possible routes. All of them can be easily generated and the shortest one then selected. This brutal force method is becoming quickly unfeasible with growing number of cities; in case of 10 cities the number of possible routes is 10! = 3,628,800. How can the TSP problem be converted into a Hopfield network?

Let *n* be the number of cities. Each city will be associated with an *n*-tuple. For example, the *n*-tuple for city A is

A: (0, 0, 1, 0, … , 0)

where "1" in the 3rd position means that the city A was the 3rd city visited. All *n*-tuples for 5 cities are shown in the Table 5-1:

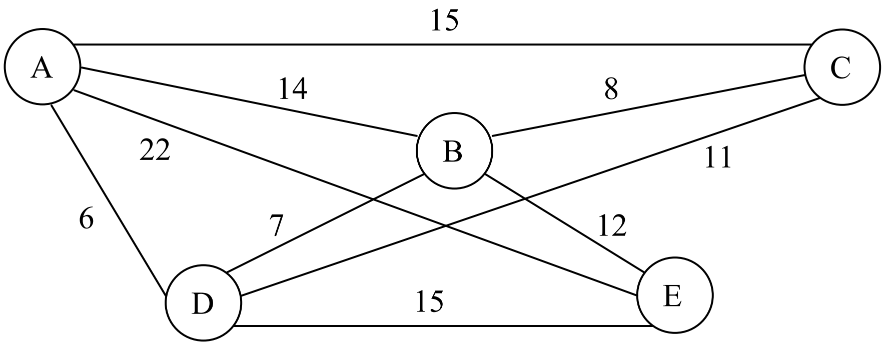


Figure 5-8: TSP with five cities.

Table 5-1: Tuples for individual cities. The cities are visited in this order: C-A-E-D-B.

City	Stop number					
	1	2	3	4	5	Tuple no.
A		1				1
B					1	2
C	1					3
D				1		4
E			1			5

Each cell inTable 5-1 is represented by one neuron and the whole Hopfield layer looks like this:

Tuple 1	Tuple 2	Tuple 3	Tuple 4	Tuple 5

5.7.1 Energy function for TSP

In section 5.1, synaptic connections (weight matrix) were calculated from the Hebb's postulate, and later, in section 5.4, an energy function was calculated from the weight matrix. We will now proceed in the opposite way: We will at first find a convenient energy function and then set up the weights accordingly.

The energy function must satisfy the following requirements:

1. Each city is visited only once. Hence, each row in Table 5-1 must contain only one "1".
2. Only one city is visited at a time. Hence, each column in Table 5-1 must contain only one "1".
3. All cities must be visited. It means, that Table 5-1 must contain *n* "1"s.
4. The total distance travelled must be as short as possible.

Let us use this notation:

$V_{x,i}$ output of the neuron corresponding to city x and stop i (x-th row and i-th column in Table 5-1)
$d_{x,y}$ distance between cities x and y

The energy function contains four terms

$$E = E_A + E_B + E_C + E_D \tag{5.17}$$

The individual components are defined below:

$$E_A = A\sum_{x=1}^{n}\left[\sum_{i=1}^{n}\sum_{j=1,\,j\neq i}^{n} V_{x,i}V_{x,j}\right] \tag{5.18}$$

The term in square brackets equals to zero if there is only one "1" in the row for city x.

$$E_B = B\sum_{i=1}^{n}\left[\sum_{x=1}^{n}\sum_{y=1,\,y\neq x}^{n} V_{x,i}V_{y,i}\right] \tag{5.19}$$

The term in square brackets equals to zero if there is only one "1" in the column for stop i.

$$E_C = C\left[\sum_{x=1}^{n}\sum_{i=1}^{n} V_{x,i} - n\right]^2 \tag{5.20}$$

This term in square brackets equals to zero if there are n "1"s in the.

$$E_D = \sum_{x=1}^{n}\sum_{y=1,\,y\neq x}^{n}\sum_{i=1}^{n} d_{x,y}V_{x,i}(V_{y,i+1} + V_{y,i-1}) \tag{5.21}$$

The last city visited is followed by the first one (the salesman returns back to the starting point) and vice versa. Therefore,

$$V_{y,n+1} = V_{y,1} \quad \text{and} \quad V_{y,0} = V_{y,n} \tag{5.22}$$

5.7.2 Weight matrix

The weight matrix will be constructed from the energy function derived in section 5.7.1. It is easier to specify what the network should *not* do if the energy function is to be minimized. The weight matrix will be, therefore, developed in terms of *inhibitions* between neurons.

Weights will be indexed with four indices and will have this meaning:

$w_{x,i,y,j}$ connection between neurons x-i and y-j

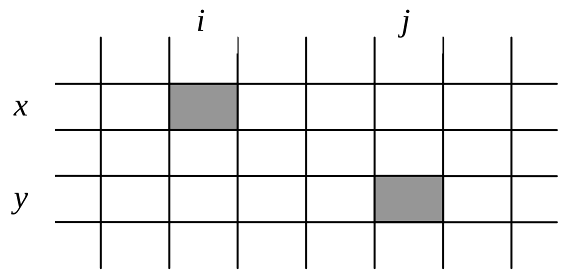


Figure 5-9: Weight $w_{x,i,y,j}$ represents the connection between neurons *x-i* and *y-j*.

The Kronecker δ_{ij} function will be used in forthcoming expressions:

$$\delta_{ij} = \begin{Bmatrix} 1, & i = j \\ 0, & i \neq j \end{Bmatrix} \tag{5.23}$$

Term E_A

If one unit in a row fires it should inhibit other units in that row. This is a kind of a winner-takes-all competition. The weight will be

$$w_{x,i,y,j} = -A\delta_{x,y}(1-\delta_{i,j}) \tag{5.24}$$

If $i \neq j$ and $x = y$ then $w_{x,j,y,j} = -A$. The neuron at position (x, i) will inhibit others but not itself.

Term E_B

If one unit in a column fires it should inhibit other units in that column. This is a kind of a winner-takes-all competition. The weight will be

$$w_{x,i,y,j} = -B\delta_{i,j}(1-\delta_{x,y}) \tag{5.25}$$

If $x \neq y$ and $i = j$ then $w_{x,j,y,j} = -B$. The neuron at position (x, i) will inhibit others but not itself.

Term E_C

This term involves all neurons so that it has a global character. All neurons will be inhibited equally:

$$w_{x,i,y,j} = -C \tag{5.26}$$

Term E_D

The weight between adjacent cities is inhibited in proportion to their distance:

$$w_{x,i,y,j} = -Dd_{x,y}(\delta_{j,i+1} + \delta_{j,i-1}) \tag{5.27}$$

All the above inhibitions (5.24) to (5.27) will be summed up for each weight. Note that the indices x, i, y, j run from 1 to n, where n is the number of cities.

Let us calculate three weights for the TSP problem with five cities shown in Table 5-2.

Table 5-2: Calculation of three weights for the TSP problem.

City	Stop number					
	1	2	3	4	5	Tuple no.
A		▤				1
B			▥		▥	2
C						3
D				▨		4
E		▤			▨	5

The weight between horizontally shaded cells:

$$w_{a,2,e,2} = -B - C \tag{5.28}$$

The weight between vertically shaded cells:

$$w_{b,3,b,5} = -A - C \tag{5.29}$$

The weight between diagonally shaded cells:

$$w_{d,4,e,5} = -C - Dd_{d,e} \tag{5.30}$$

The total number of weights is $5^4 = 625$.

5.7.3 An analog Hopfield network for TSP

The diagram of an analog processing element in shown in Figure 5-10. Many such processing elements are implemented on a microchip and work in parallel. Its behaviour is described by an ordinary differencial equation

$$\frac{dv_{x,i}}{dt} = -\frac{v_{x,i}}{\tau} + \sum_{x,i,y,j} w_{x,i,y,j} V_{y,j} \tag{5.31}$$

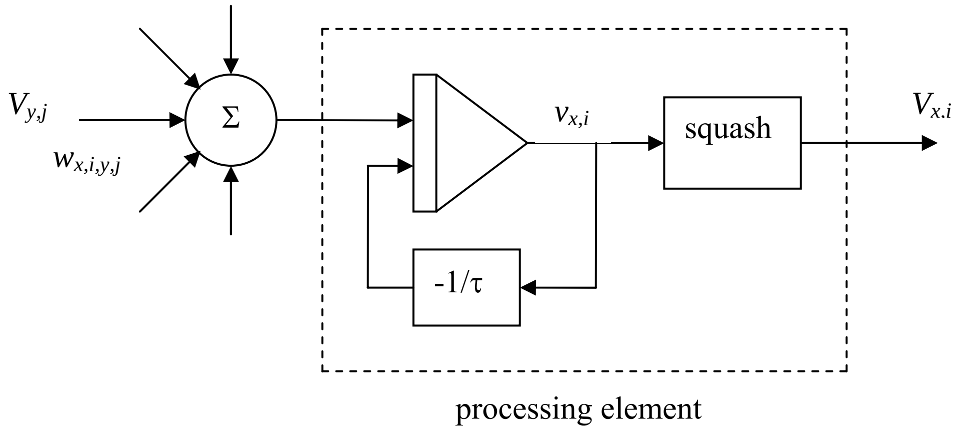


Figure 5-10: An analog processing element.

The squashing function is

$$V_{x,i} = \frac{1 + \tanh(\lambda v_{x,i})}{2} \tag{5.32}$$

The success of the analog Hopfield network is highly sensitive to its parameters. The following parameter values for a 10-city problem have been reported: $A = B = 500$, $C = 200$, $D = 500$, $\tau = 1$, $\lambda = 50$.

5.8 Problems

1. Consider a Hopfield network made up of five neurons, which is required to store the following three fundamental memories:

   $d^1 = [+1, +1, +1, +1, +1]^T$
   $d^2 = [+1, -1, -1, +1, -1]^T$
   $d^3 = [-1, +1, -1, +1, +1]^T$

   a) Evaluate the 5×5 synaptic weight matrix of the network.
   b) Investigate the retrieval performance of the network when it is presented with a noisy version of d^1 in which the second element is reversed in polarity.
   c) Calculate the energy E at each relaxation step.

2. Investigate the use of synchronous updating for the retrieval performance of the Hopfield network described in problem 5.1.

3. Show that the following are also fundamental memories of the Hopfield network described in problem 5.1:

   $d^1 = [-1, -1, -1, -1,-1]^T$
   $d^2 = [-1, +1, +1, -1, +1]^T$
   $d^3 = [+1, -1, +1, -1, -1]^T$

6 Self-organizing feature maps

Self-organizing feature maps represent a special class of artificial neural networks based on *competitive unsupervised learning*. The output neurons of the network compete among themselves to be activated (fired), with the result that only one output neuron, or one neuron per group, is on at any one time. The output neurons that win the competition are called *winner-takes-all neurons*. One way of inducing a winner-takes-all competition among the output neurons is to use lateral inhibitory connections (i.e. negative feedback paths) between them.

In a *self-organizing feature map*, the neurons are placed at the nodes of a lattice that is usually one- or two-dimensional; higher-dimensional maps are also possible but not as common. The neurons become selectively tuned to various input patterns or classes of input patterns in the course of a competitive learning process. The locations of the winning neurons tend to become ordered with respect to each other in such a way that a meaningful coordinate system for different input features is created over the lattice. A self-organizing feature map is therefore characterized by the formation of a topographic map of the input patterns, in which *the spatial locations (coordinates) of the neurons in the lattice correspond to intrinsic features of the input patterns* (Kohonen 1982-1990).

The Kohonen self-organizing map (SOM) performs a mapping from a continuous input space to a discrete output space, preserving the topological properties of the input. This means that points close to each other in the input space are mapped to the same neighbouring neurons in the output space.

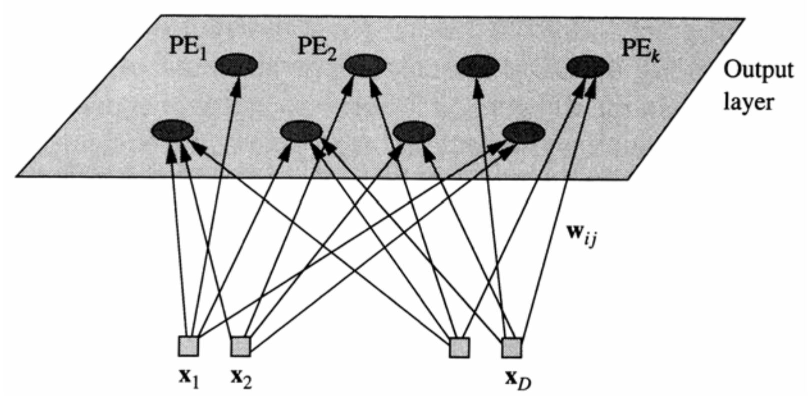


Figure 6-1: Architecture of a SOM with a 2-D output.

6.1 Activation bubbles

Let us consider a one-dimensional lattice of neurons as shown in Figure 6-2. There are forward connections x_i from the primary source of excitation, and those that are internal to the network by virtue of self-feedback and lateral feedback. These two types of local connections serve two different purposes. The weighted sum of the input signals at each neuron is designed to perform feature detection. Hence each neuron produces a selective response to a particular set of input signals. The feedback connections, on the other hand, produce excitatory or inhibitory effects, depending on the distance from the winning neuron.

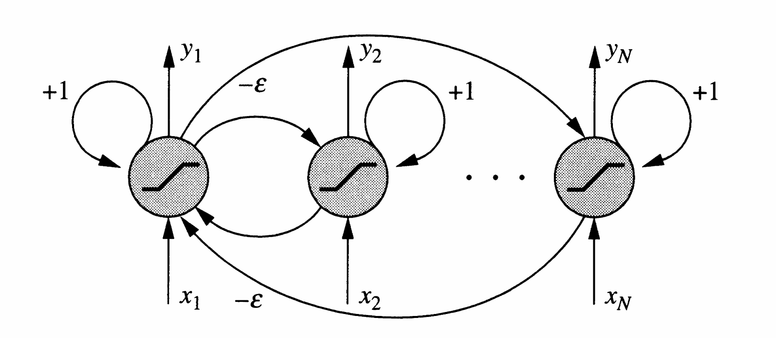


Figure 6-2: A layer of competing neurons.

The lateral feedback is usually described by a Mexican hat function shown in Figure 6-3. According to this figure, we may distinguish three distinct areas of lateral interaction between neurons: (1) a short-range lateral excitation area, (2) a penumbra of inhibitory action, and (3) an area of weaker excitation that surrounds the inhibitory penumbra.

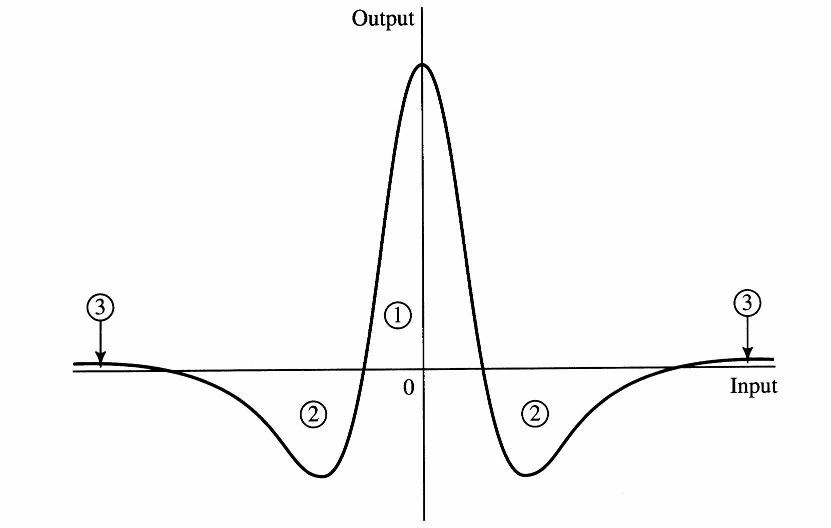


Figure 6-3: The Mexican hat function of lateral interconnections.

The neural network with lateral feedback exhibits two important characteristics. First, the network tends to concentrate its activity into local clusters, referred to as *activity bubbles*. Second, the locations of the activity bubbles are determined by the nature of the input signals.

Let x_1, x_2, … denote the input signals applied to the network. Let w_{j1}, w_{j2}, … denote the corresponding synaptic weights of neuron *j*. Let $c_{j,-K}$, … , $c_{j,-1}$, $c_{j,0}$, $c_{j,1}$, … $c_{j,K}$ denote the lateral feedback weights connected to neuron *j*, where K is the "radius" of the lateral interaction. Let y_1, y_2, … denote the output signals of the network. We may thus express the output signal (response) of neuron *j* as follows:

$$y_j = \varphi(\sum_i w_{ji} x_i + \sum_{k=-K}^{K} c_{jk} y_{j+k}) \tag{6.1}$$

where φ(·) is some nonlinear function that limits the value of y_j and ensures that $y_j \geq 0$. The first term in the argument of φ is a stimulus applied on neuron *j* by the input signals x_i.

The solution to the nonlinear equation (6.1) is found iteratively:

$$y_j(n+1) = \varphi\left(\sum_i w_{ji}x_i + \beta \sum_{k=-K}^{K} c_{jk} y_{j+k}(n)\right) \qquad (6.2)$$

where n denotes an iteration step. The parameter β controls the rate of convergence of the iteration process.

The equation (6.2) represents a feedback system. The system includes both positive and negative feedback, corresponding to the excitatory and inhibitory parts of the Mexican hat function, respectively. The limiting action of the nonlinear activation function $\varphi(\cdot)$ causes the spatial response $y_j(n)$ to stabilize in a certain fashion, dependent on the value assigned to β. If β is large enough, then in the final state corresponding to $n\to\infty$, the values of y_j tend to concentrate inside a spatially bounded activity bubble. The bubble is centred at a point where the initial response $y_j(0)$ due to the stimulus $\sum_i w_{ji}x_i$ is maximum.

6.2 Self-organizing feature-map algorithm

Kohonen simplifies the calculation of activity bubbles formation by introducing a computational trick: He introduces a topological neighbourhood of active neurons and removes lateral connections. Thus, instead of recursively computing the activity of each neuron according to equation (6.2), he simply finds the winning neuron and assumes the other neurons are also active if they belong to the neighbourhood or have their activities proportional to the Gaussian function evaluated at each neuron's distance from the winner.

The neurons are arranged into a two-dimensional lattice as shown in Figure 6-4.

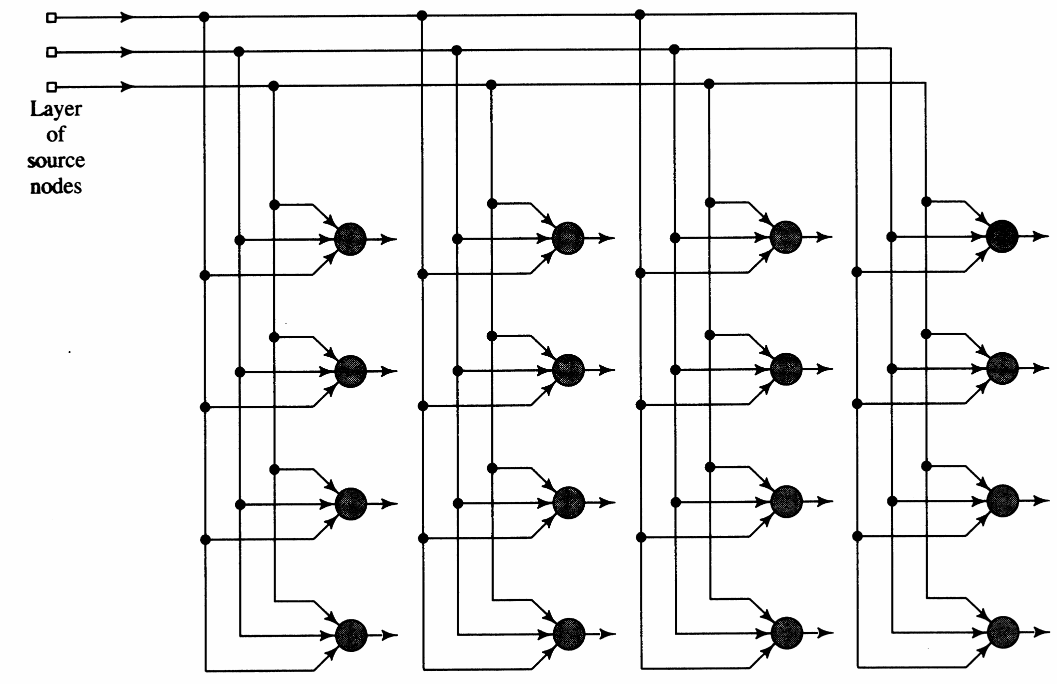


Figure 6-4: Two-dimensional lattice of *N* neurons.

The input vector *x* and the synaptic weight vector w_j of neuron *j* are

$$\begin{aligned} x &= \begin{bmatrix} x_1 & x_2 & \cdots & x_p \end{bmatrix}^T \\ w_j &= \begin{bmatrix} w_{j1} & w_{j2} & \cdots & w_{jp} \end{bmatrix}^T \end{aligned} \tag{6.3}$$

To find the best match of the input vector *x* with the synaptic weight vectors w_j, we simply compare the inner products $w_j^T x$ for *j* = 1, 2, … , *N* and select the largest. Note that $w_j^T x$ is identical with the stimulus in (6.2). Thus, by selecting the neuron with the largest inner product $w_j^T x$, we will have in effect determined the location where the activity bubble is to be formed.

In the formulation of an adaptive algorithm, it is convenient to *normalize* the weight vectors w_j. Then the best matching criterion is equivalent to the minimum Euclidean distance between vectors *x* and w_j. Let *i*(*x*) be the index of the neuron that best matches the input vector *x*. This index is determined by

$$i(x) = \arg\min_j \left\| x - w_j \right\| \tag{6.4}$$

The particular neuron *j* that satisfies this condition is called the *best-matching* or *winning neuron* for the input vector *x*. By using (6.4), a continuous input space is mapped onto a discrete set of neurons. Depending on the application of interest, the response of the network could be either the index *i*(*x*) of the winning neuron (i.e. its position in the lattice), or the synaptic weight vector w_j that is closest to the input vector in a Euclidean sense.

The topology of iterations in the self-organizing feature-map (SOFM) algorithm defines which neurons in the two-dimensional lattice are in fact neighbours. Let $\Lambda_{i(x)}(n)$ denote the *topological neighbourhood* of winning neuron *i*(*x*). The neighborhood $\Lambda_{i(x)}(n)$ is chosen to be a function of the discrete time *n*; hence we may also refer to $\Lambda_{i(x)}(n)$ as a *neighbourhood function*. Numerous simulations have shown that the best results in self-organization are obtained if the neighbourhood function $\Lambda_{i(x)}(n)$ is selected fairly wide in the beginning and then permitted to shrink with time *n*. This behaviour is equivalent to initially using a strong positive lateral feedback, and then enhancing the negative lateral feedback. The important point to note here is that the use of a neighbourhood function $\Lambda_{i(x)}(n)$ around the winning neuron *i*(*x*) provides a clever computational shortcut for emulating the formation of a localized response by lateral feedback (activity bubble).

6.2.1 Adaptive Process

For the network to be self-organizing, the synaptic weight vector w_j of neuron *j* is required to change in relation to the input vector *x*. The question is how to make the change. In Hebb's postulate of learning, a synaptic weight is increased with a simultaneous occurrence of presynaptic and postsynaptic activities. The use of such a rule is well suited for associative learning. A slightly modified formula (2.10) for Hebbian learning with a forgetting factor will be used.

Only the neurons inside the neighbourhood $\Lambda_{i(x)}(n)$ are active:

$$y_j = \begin{cases} 1, & \text{inside } \Lambda_{i(x)} \\ 0, & \text{outside } \Lambda_{i(x)} \end{cases} \tag{6.5}$$

If we assume that the coefficient $c = 1$ in (2.10), we may compute the updated value of the weight vector as follows:

$$w_j(n+1) = \begin{cases} w_j(n) + \eta(n)[x - w_j(n)], & j \in \Lambda_{i(x)}(n) \\ w_j(n), & j \notin \Lambda_{i(x)}(n) \end{cases} \tag{6.6}$$

The effect of the update equation (6.6) is to move the synaptic vector w_i of the winning neuron i toward the input vector x. The algorithm therefore leads to a topological ordering of the feature map in the input space in the sense that neurons that are adjacent in the lattice will tend to have similar synaptic weight vectors.

6.2.2 Summary of the SOFM algorithm

1. *Initialization*. Choose random values for the initial vectors $w_j(0)$. All initial weight vectors should be different.
2. *Input*. Present a sample vector x to the network.
3. *Similarity matching*. Find the best matching (winning) neuron $i(x)$ at time n using formula (6.4).
4. *Weight updating*. Adjust the synaptic weight vectors of all neurons, using the update formula (6.6). Both the learning parameter $\eta(n)$ and the neighbourhood function $\Lambda_{i(x)}(n)$ are varied dynamically during learning for best results.
5. *Continuation*. Repeat steps 2, 3, and 4 until no noticeable changes in the feature map are observed.

6.2.3 Selection of Parameters

The learning process involved in the computation of a feature map is stochastic in nature, which means that the accuracy of the map depends on the number of iterations of the SOFM algorithm. Moreover, the success of map formation is critically dependent on how the main parameters of the algorithm, namely, the learning-rate parameter $\eta(n)$ and the neighbourhood function $\Lambda_{i(x)}(n)$, are selected. Unfortunately, there is no theoretical basis for the selection of these parameters. They are usually determined by a process of trial and error. Nevertheless, the following observations provide a useful guide.

1. The learning-rate parameter $\eta(n)$ used to update the synaptic weight vector $w_j(n)$ should be time-varying. In particular, during the first 1000 iterations or so, $\eta(n)$ should begin with a value close to unity; thereafter, $\eta(n)$ should decrease gradually, but staying above 0.1. The exact form of variation of $\eta(n)$ with n is not critical. It is during this initial phase of the algorithm that the topological ordering of the weight vectors $w_j(n)$ takes place. This phase of the learning process is called the *ordering phase*. The remaining (relatively long) iterations of the algorithm are needed principally for the fine tuning of the computational map; this second

phase of the learning process is called the *convergence phase*. For good statistical accuracy, $\eta(n)$ should be maintained during the convergence phase at a small value (on the order of 0.01 or less) for a fairly long period of time, which is typically thousands of iterations.

2. For topological ordering of the weight vectors w_j to take place, careful consideration has to be given to the neighbourhood function $\Lambda_{i(x)}(n)$. Generally, the function $\Lambda_{i(x)}(n)$ is taken to include neighbours in a square region around the winning neuron, as illustrated in Figure 6-5. For example, a "radius" of one includes the winning neuron plus the eight neighbours. However, the function $\Lambda_{i(x)}(n)$ may take other forms, such as hexagonal or even a continuous Gaussian shape. In any case, the neighbourhood function $\Lambda_{i(x)}(n)$ usually begins such that it includes all neurons in the network and then gradually shrinks with time. To be specific, during the initial phase of 1000 iterations or so, when topological ordering of the synaptic weight vectors takes place, the radius of $\Lambda_{i(x)}(n)$ is permitted to shrink linearly with time n to a small value of only a couple of neighbouring neurons. During the convergence phase, $\Lambda_{i(x)}(n)$ should contain only the nearest neighbours of winning neuron i, which may eventually be 1 or 0 neighbouring neurons.

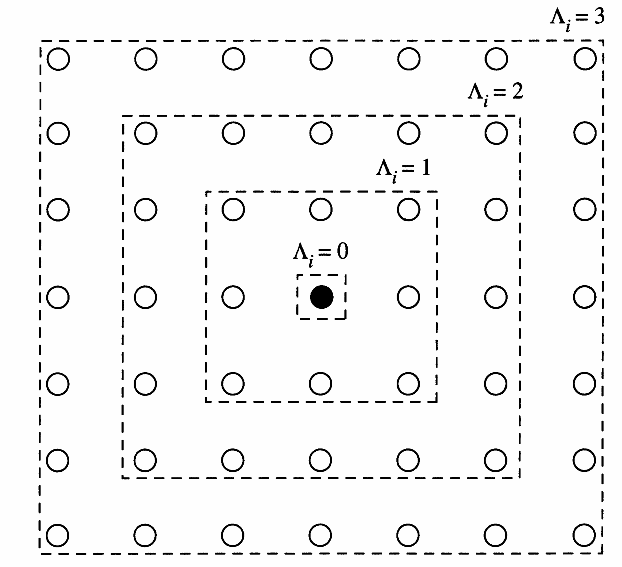


Figure 6-5: Square topological neighbourhood $\Lambda_{i(x)}(n)$ of varying size around winning neuron *i*, identified as a black circle.

6.2.4 Reformulation of the topological neighbourhood

In the traditional form of the SOFM algorithm, all the neurons located inside this topological neighbourhood $\Lambda_{i(x)}(n)$ fire at the same rate, and the interaction among those neurons is independent of their lateral distance from the winning neuron i. There is, however, evidence that a neuron that is firing tends to excite the neurons in its immediate neighbourhood more than those farther away from it. Let d_{ji} denote the lateral distance of neuron j from the winning neuron i, which is a Euclidean measure in the output space. Let π_{ji} denote the amplitude of the topological neighbourhood centred on the winning neuron i. A typical choice of π_{ji} is the Gaussian-type function

$$\pi_{ji} = \exp(-\frac{d_{ji}^2}{2\sigma^2}) \qquad (6.7)$$

where σ is the "effective width" of the topological neighbourhood.

We may now rewrite the weight update equation (6.6) for updating the synaptic vector w_j of neuron j at lateral distance d_{ji} from the winning neuron $i(x)$ as

$$w_j(n+1) = \begin{cases} w_j(n) + \eta(n)\pi_{ji(x)}(n)[x - w_j(n)], & j \in \Lambda_{i(x)}(n) \\ w_j(n), & j \notin \Lambda_{i(x)}(n) \end{cases} \tag{6.8}$$

As π_{ji} is almost zero for large distances from the winning neuron this equation can be further simplified:

$$w_j(n+1) = w_j(n) + \eta(n)\pi_{ji(x)}(n)[x - w_j(n)] \tag{6.9}$$

The equation (6.8) is preferred from computational efficiency point of view; the weights of more distant neurons do not have to be updated at all if the updates are extremely small. The rest of the SOFM algorithm is the same as in the summary in section 6.2.2.

In equation (6.9), the learning rate $\eta(n)$ and the neighbourhood function $\pi_{ji(x)}(n)$ are both dependent on time n. For the SOFM algorithm to converge, it is important that the width $\sigma(n)$ of the topological neighbourhood as well as the learning rate $\eta(n)$ be permitted to decrease slowly during the learning process. A popular choice for the dependence of $\sigma(n)$ and $\eta(n)$ on time n is the exponential decay described as

$$\sigma(n) = \sigma_0 \exp(-\frac{n}{\tau_1}) \tag{6.10}$$

and

$$\eta(n) = \eta_0 \exp(-\frac{n}{\tau_2}) \tag{6.11}$$

The constants σ_0 and η_0 are the values of $\sigma(n)$ and $\eta(n)$ at the initiation of the SOFM algorithm (i.e. at $n = 0$), and τ_1 and τ_2 are their *time constants*. Equations (6.10) and (6.11) are applied to the neighbourhood function and the learning rate parameter only during the ordering phase (i.e. the first thousand iterations or so); then small values are used for a very long time.

6.3 Examples

6.3.1 Classification problem – overlapping classes

This classification problem was described in section 4.14.3 and solved with a back-propagation network. The back-propagation network used a supervised training, which means that the class membership was known. We will attempt to classify the same data with a self-organizing map. The main difference is that only the input coordinates x_1 and x_2 will be used to train the network and the SOFM will have to find by itself the relations in the training data by using unsupervised learning. No information about class membership of individual samples and the number of classes is made available to the network.

The SOFM was generated and trained by Viscovery SOMine package. A two-dimensional map consists of 2000 neurons and is trained on 6000 samples. (The same samples were used in section 4.14.3 but they were not divided into training and validation subsets in this case.)

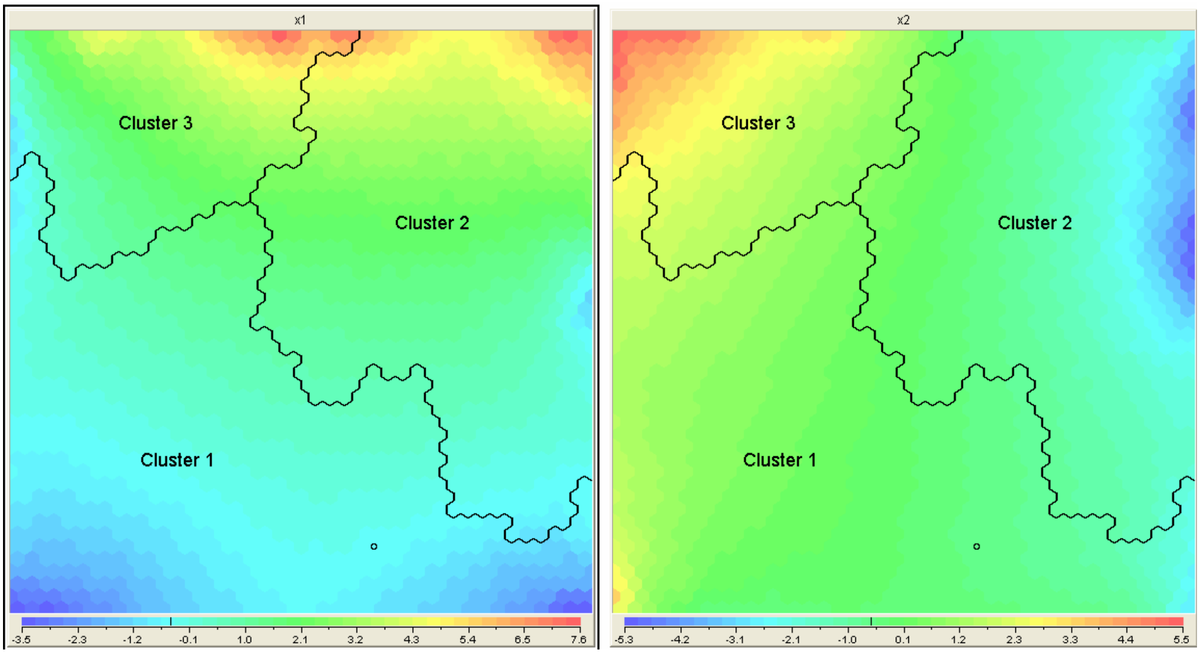


Figure 6-6: Three clusters shown with corresponding colour-coded values of input variables x_1 and x_2.

The meaning of clusters is not known at this point. The clusters must be "calibrated", i.e. their meaning will be determined by placing data into them. After doing so we found that cluster 1 corresponds to the data class C_1 and clusters 2 and 3 correspond to the data class C_2.

The SOFM was tested on 2000 samples generated separately from the training set. The following figures show misclassified testing samples.

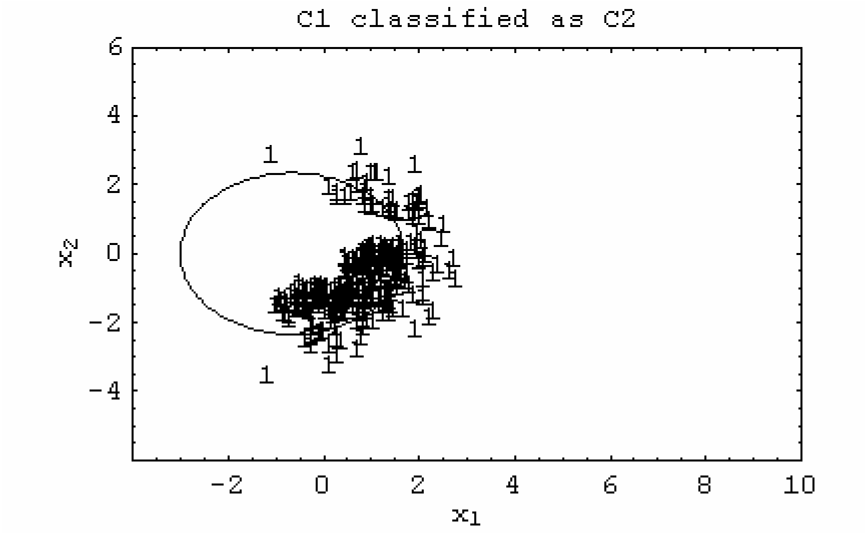


Figure 6-7: Elements of class C_1 misclassified as C_2 – 15.60%.

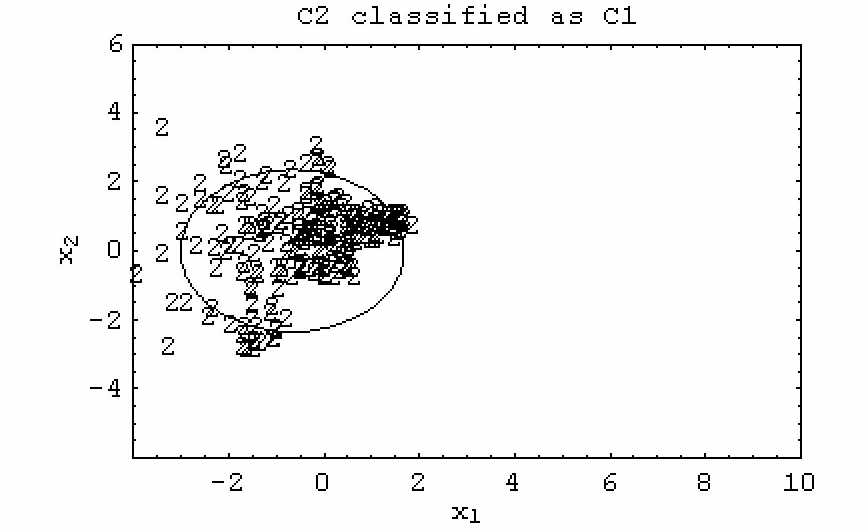


Figure 6-8: Elements of class C_2 misclassified as C_1 – 10.44%.

Bayesian classifier - misclassification	
Class 1	Class 2
10.56 %	26.42 %

SOFM - misclassification	
Class 1	Class 2
15.60 %	10.44 %

6.4 Problems

1. The data presented in the file `Iris.xls` and shown in Table 4-6 contain 150 records. Each record consists of NN inputs (sepal length, sepal width, petal length, and petal width) and desired outputs. The outputs represent three classes of Iris plants (Setosa, Versicolor, and Virginica). *The outputs will be ignored in this exercise*.

   Use a SOM to process the input data (i.e. sepal length, sepal width, petal length, and petal width) and produce a confusion table indicating how the SOM was able to classify the training samples. Note that the SOM does not know anything about the existence and the number of classes.

2. The data presented in the file `Divorce.xls` and shown in Table 6-1 contain most frequent causes of divorce in various US states. Present the data to a SOM and retrieve various sorts of information from the map.

Table 6-1: Causes of divorce in US.

	incompat	cruelty	desertn	non_supp	alcohol	felony	impotenc	insanity	separate
ALABAMA	1	1	1	1	1	1	1	1	1
ARIZONA	1	0	0	0	0	0	0	0	0
CALIFORNIA	1	0	0	0	0	0	0	1	0
CONNECTICUT	1	1	1	1	1	1	0	1	1
FLORIDA	1	0	0	0	0	0	0	1	0
HAWAII	1	0	0	0	0	0	0	0	1
ILLINOIS	0	1	1	0	1	1	1	0	0
IOWA	1	0	0	0	0	0	0	0	0
KENTUCKY	1	0	0	0	0	0	0	0	0
MAINE	1	1	1	1	1	0	1	1	0
MASSACHUSETTS	1	1	1	1	1	1	1	0	1
MINNESOTA	1	0	0	0	0	0	0	0	0
MISSOURI	1	0	0	0	0	0	0	0	0
NEBRASKA	1	0	0	0	0	0	0	0	0
NEW_HAMPSHIRE	1	1	1	1	1	1	1	0	0
NEW_MEXICO	1	1	1	0	0	0	0	0	0
NORTH_CAROLINA	0	0	0	0	0	0	1	1	1
OHIO	1	1	1	0	1	1	1	0	1
OREGON	1	0	0	0	0	0	0	0	0
RHODE_ISLAND	1	1	1	1	1	1	1	0	1
SOUTH_DAKOTA	0	1	1	1	1	1	0	0	0
TEXAS	1	1	1	0	0	1	0	1	1
VERMONT	0	1	1	1	0	1	0	1	1
WASHINGTON	1	0	0	0	0	0	0	0	1
WISCONSIN	1	0	0	0	0	0	0	0	1
ALASKA	1	1	1	0	1	1	1	1	0
ARKANSAS	0	1	1	1	1	1	1	1	1
COLORADO	1	0	0	0	0	0	0	0	0
DELAWARE	1	0	0	0	0	0	0	0	1
GEORGIA	1	1	1	0	1	1	1	1	0
IDAHO	1	1	1	1	1	1	0	1	1
INDIANA	1	0	0	0	0	1	1	1	0
KANSAS	1	1	1	0	1	1	1	1	0
LOUISIANA	0	0	0	0	0	1	0	0	1
MARYLAND	0	1	1	0	0	1	1	1	1
MICHIGAN	1	0	0	0	0	0	0	0	0
MISSISSIPPI	1	1	1	0	1	1	1	1	0
MONTANA	1	0	0	0	0	0	0	0	0
NEVADA	1	0	0	0	0	0	0	1	1
NEW_JERSEY	0	1	1	0	1	1	0	1	1
NEW_YORK	0	1	1	0	0	1	0	0	1
NORTH_DAKOTA	1	1	1	1	1	1	1	1	0
OKLAHOMA	1	1	1	1	1	1	1	1	0
PENNSYLVANIA	0	1	1	0	0	1	1	1	0
SOUTH_CAROLINA	0	1	1	0	1	0	0	0	1
TENNESSEE	1	1	1	1	1	1	1	0	0
UTAH	0	1	1	1	1	1	1	1	0
VIRGINIA	0	1	0	0	0	1	0	0	1
WEST_VIRGINIA	1	1	1	0	1	1	0	1	1
WYOMING	1	0	0	0	0	0	0	1	1

7 Temporal processing with neural networks

The back-propagation algorithm described in section 4 has established itself as the most popular method for the design of neural networks. However, a major limitation of the standard back-propagation algorithm is that it can only learn an input-output mapping that is *static*. Consequently, the multilayer perceptron so trained has a static structure that maps an input vector *x* onto an output vector *y*, as depicted in Figure 7-1. This form of static input-output mapping is well suited for pattern-recognition applications (e.g. optical character recognition), where both the input vector *x* and the output vector *y* represent spatial patterns that are independent of time.

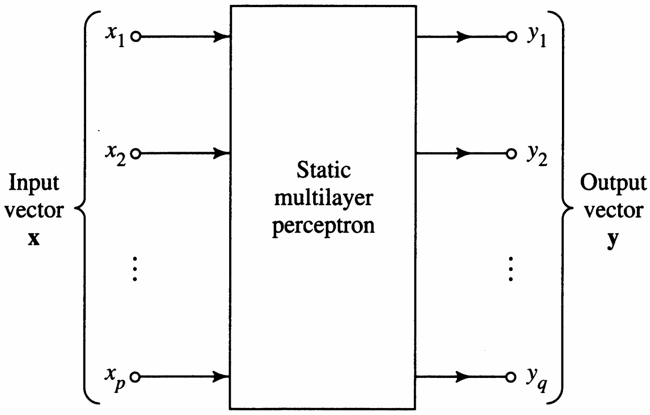


Figure 7-1: Static back-propagation network as a pattern classifier or function approximator.

The standard back-propagation algorithm may also be used to perform nonlinear prediction on a stationary time series. A time series is said to be *stationary* if its statistics do not change with time. In such a case we may also use a static back-propagation network, as shown in Figure 7-2, where the blocks labelled z^{-1} represent *unit delays*.

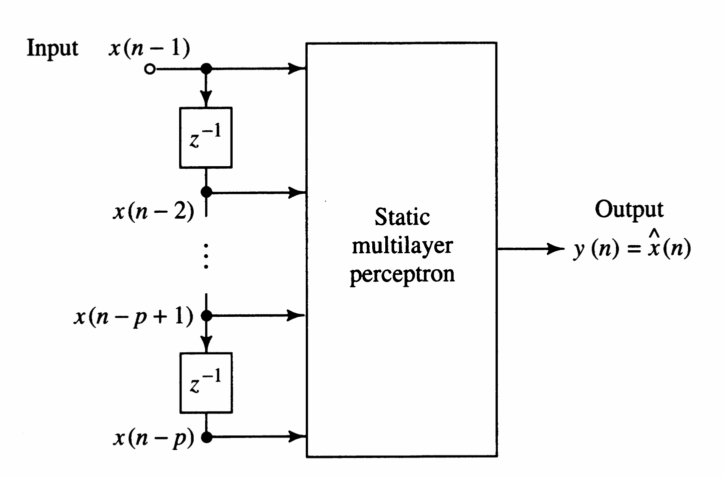


Figure 7-2: Static back-propagation network as a nonlinear predictor.

The input vector *x* is now defined in terms of the past samples *x*(*n*-1), … , *x*(*n* – *p*):

$$x = \left[x(n-1) \quad x(n-2) \quad \cdots \quad x(n-p)\right]^T \tag{7.1}$$

We refer to p as the *prediction order*. Thus the scalar output $y(n)$ produced in response to the input vector x equals to a *one-step prediction* $\hat{x}(n)$:

$$y(n) = \hat{x}(n) \tag{7.2}$$

The actual value $x(n)$ of the input signal represents the *desired response*.

We know that *time* is important in many of the cognitive tasks encountered in practice, such as vision, speech, signal processing, and motor control. The question is how to represent time. In particular, how can we extend the design of a back-propagation network so that it assumes a time-varying form and therefore will be able to deal with time-varying signals? The answer to this question is to allow time to be represented by the effect it has on signal processing. This means providing the mapping network *dynamic properties* that make it responsive to time-varying signals.

For a neural network to be dynamic, it must be given *memory*. One way in which this requirement can be accomplished is to introduce *time delays* into the synaptic structure of the network and to adjust their values during the learning phase. Another way in which a neural network can assume dynamic behaviour is to make it *recurrent*, that is, to build feedback into its design.

7.1 Spatio-temporal model of a neuron

The neural network analysis presented in previous chapters was based on the nonlinear model of a neuron shown in Figure 1-2. Limitation of this model is that it only accounts for the *spatial* behaviour of a neuron by incorporating a set of fixed synaptic weights. To extend the usefulness of this model for *temporal processing*, we need to modify it so as to account for the temporal nature of the input data. A general method of representing temporal behaviour is to model each synapse by a *linear, time-invariant filter*, as shown in Figure 7-3.

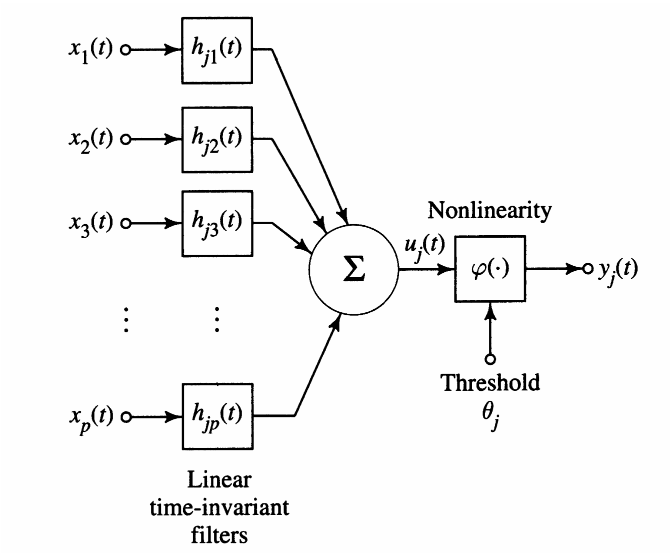


Figure 7-3: Dynamic model of a neuron using linear, time-invariant filters as synapses.

The temporal behaviour of synapse *i* belonging to neuron *j* may thus be described by an impulse response $h_{ji}(t)$ that is a function of continuous time *t*. By definition, the impulse response of a linear time-invariant filter is the response produced by a unit impulse (Dirac delta function) applied to the filter at time $t = 0$. Let $x_i(t)$ denote the stimulus applied to synapse *i*. The response of synapse *i* due to the input $x_i(t)$ is equal to the *convolution* of its impulse response $h_{ji}(t)$ with the input $x_i(t)$:

$$h_{ji}(t) * x_i(t) = \int_{-\infty}^{t} h_{ji}(\tau)x_i(t-\tau)d\tau = \int_{-\infty}^{t} x_i(\tau)h_{ji}(t-\tau)d\tau \tag{7.3}$$

Given a neuron *j* with *p* synapses, the net activation potential $v_j(t)$ is given by

$$\begin{aligned} v_j(t) = u_j(t) - \theta_j &= \sum_{i=1}^{p} h_{ji}(t) * x_i(t) - \theta_j \\ &= \sum_{i=1}^{p} \int_{-\infty}^{t} h_{ji}(\tau)x_i(t-\tau)d\tau - \theta_j \end{aligned} \tag{7.4}$$

Finally, $v_j(t)$ is passed through an activation function φ(•) to produce the overall output

$$y_j(t) = \varphi(v_j(t)) \tag{7.5}$$

Equations (7.4) and (7.5) describe the spatio-temporal behaviour of the neuron.

7.2 Finite duration impulse response (FIR) model

The synaptic filters shown in Figure 7-3 are continuous-time filters. Each synaptic filter is typically characterized as follows:

1. The synaptic filter is *causal*, which means that the synapse does not respond before the stimulus is applied to its input:

$$h_{ji}(t) = 0, \quad t < 0 \tag{7.6}$$

2. The synaptic filter has *finite memory*:

$$h_{ji}(t) = 0, \quad t > T \tag{7.7}$$

where *T* is the *memory span*, assumed to be the same for all synapses.

Equation (7.4) can be written as

$$v_j(t) = \sum_{i=1}^{p} \int_{0}^{T} h_{ji}(\tau)x_i(t-\tau)d\tau - \theta_j \tag{7.8}$$

It is convenient, from a computational viewpoint, to approximate the convolution integral in equation (7.8) by a *convolution sum.* We replace the continuous time *t* by a discrete time variable *n*

$$t = n\Delta t \tag{7.9}$$

where *n* is an integer and Δ*t* is the *sampling period.* Then we may approximate the equation (7.8) as

$$\begin{aligned} v_j(n\Delta t) &= \sum_{i=1}^{p}\sum_{l=0}^{M} h_{ji}(l\Delta t)x_i((n-l)\Delta t)\Delta t - \theta_j \\ &= \sum_{i=1}^{p}\sum_{l=0}^{M} w_{ji}(l\Delta t)x_i((n-l)\Delta t) - \theta_j \end{aligned} \tag{7.10}$$

where $M = \dfrac{T}{\Delta t}$ and $w_{ji}(l\Delta t) = h_{ji}(l\Delta t)\Delta t$. We may simplify the notation by omitting Δ*t*:

$$v_j(n) = \sum_{i=1}^{p}\sum_{l=0}^{M} w_{ji}(l)x_i(n-l) - \theta_j \tag{7.11}$$

The integer *n* is usually referred to as the *discrete-time variable.* The inner index *l* is the *temporal index* and the outer index *i* is the *spatial index.* Equation (7.11) allows us to implement the spatio-temporal model of a neuron as shown in Figure 7-4.

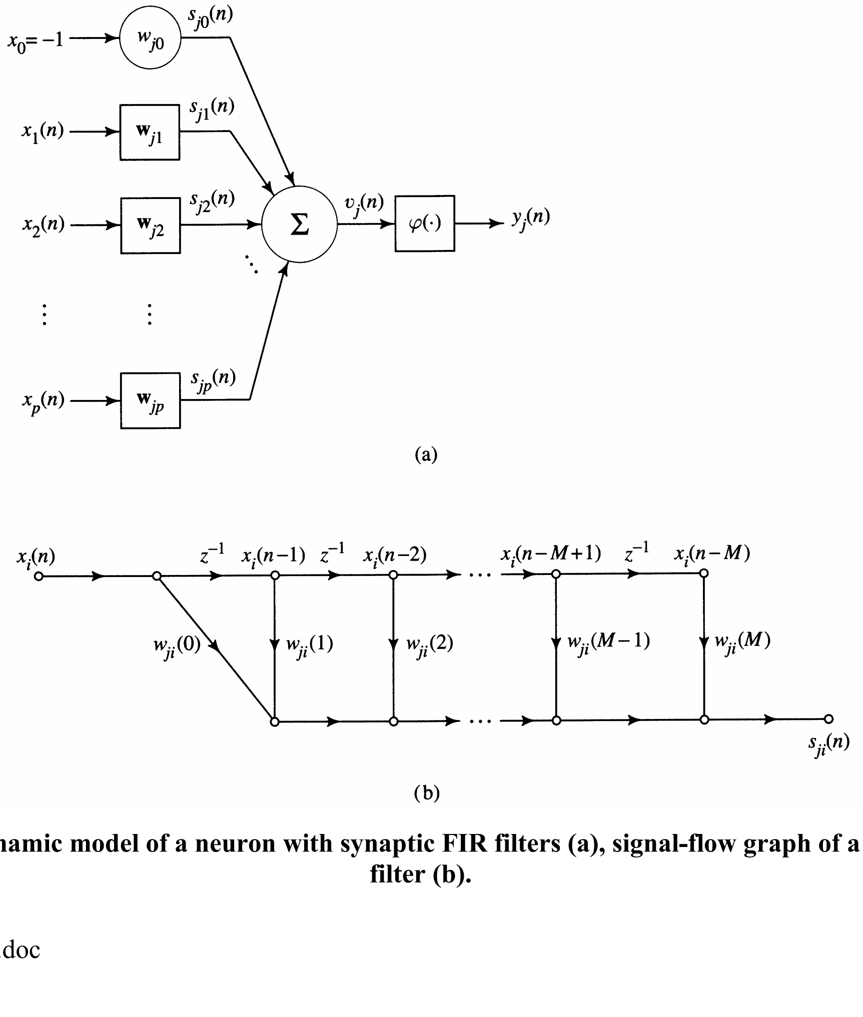


Figure 7-4: Dynamic model of a neuron with synaptic FIR filters (a), signal-flow graph of a synaptic FIR filter (b).

The weight vector $\mathbf{w}_{ji} = \left[w_{ji}(0) \quad w_{ji}(1) \quad \cdots \quad w_{ji}(M) \right]^T$ represents the synapse *i* belonging to neuron *j*. The weight w_{j0} represents the threshold θ_j. Figure 7-4a shows the *finite-duration impulse response (FIR) model* of a single neuron.

7.3 FIR back-propagation network

Consider a back-propagation network whose hidden and output neurons are all based on the FIR model shown in Figure 7-4. As shown in Figure 7-4b, the signal $s_{ji}(n)$ appearing at the output of the *i*th synapse of neuron *j* is given by a linear combination of delayed values of the intput signal $x_i(n)$ as seen from the *convolution sum*

$$s_{ji}(n) = \sum_{l=0}^{M} w_{ji}(l) x_i(n-l) \tag{7.12}$$

This equation may be rewritten in matrix form

$$s_{ji}(n) = \mathbf{w}_{ji}^T \, \mathbf{x}_i(n) \tag{7.13}$$

where

$$\begin{aligned} \mathbf{x}_i(n) &= \left[x_i(n) \quad x_i(n-1) \quad \cdots \quad x_i(n-M) \right]^T \\ \mathbf{w}_{ji} &= \left[w_{ji}(0) \quad w_{ji}(1) \quad \cdots \quad w_{ji}(M) \right]^T \end{aligned} \tag{7.14}$$

Summing the contributions of the complete set of *p* synapses describes the output $y_j(n)$ of neuron *j*:

$$v_j(n) = \sum_{i=1}^{p} s_{ji}(n) - s_{j0}(n) = \sum_{i=1}^{p} \mathbf{w}_{ji}^T \mathbf{x}_i(n) - w_{j0} \tag{7.15}$$

$$y_j(n) = \varphi(v_j(n)) \tag{7.16}$$

Note that if the weight vector $\mathbf{w}_{ji}$ and the state vector $\mathbf{x}_i(n)$ are replaced by the scalars w_{ji} and x_i, respectively, the dynamic model of neuron reduces to the static model described in section 1.

FIR network is trained in a supervised manner similarly as an “ordinary” back-propagation network. The desired output is a time series value at each instant time. We may define an *instantaneous value* for the sum of squared errors produced by the network as follows:

$$I(n) = \frac{1}{2} \sum_{j=1}^{q} e_j^2(n) \tag{7.17}$$

where *q* is the number of outputs and *e*$_j$(*n*) is the error signal defined by $e_j(n) = y_j(n) - d_j(n)$. The goal is to minimize a *cost function* computed over all time:

$$J = \sum_n I(n) \tag{7.18}$$

Weight vectors will be again updated by using the idea of gradient descent:

$$\mathbf{w}_{ji}(n+1) = \mathbf{w}_{ji}(n) - \eta \frac{\partial J}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial \mathbf{w}_{ji}(n)} \tag{7.19}$$

where η is the learning coefficient and $\mathbf{w}_{ji}$ is the vector of weights in a synaptic FIR filter. It can be found from (7.15) that

$$\frac{\partial v_j(n)}{\partial \mathbf{w}_{ji}(n)} = \mathbf{x}_i(n) \tag{7.20}$$

where $\mathbf{x}_i(n)$ is the input vector applied to neuron *j*. This vector consists of delayed signals as shown in formula (7.14). Let us define the *local gradient* for neuron *j* as

$$\delta_j(n) = \frac{\partial J}{\partial v_j(n)} \tag{7.21}$$

Then equation (7.19) can be rewritten in a more familiar form

$$\mathbf{w}_{ji}(n+1) = \mathbf{w}_{ji}(n) - \eta \delta_j(n) \mathbf{x}_i(n) \tag{7.22}$$

Similarly as in the derivation of the standard back-propagation algorithm, the explicit form of the local gradient δ$_j$(*n*) depends on whether neuron *j* lies in the output layer or a hidden layer.

Case 1 – neuron *j* is an output unit

As shown in Figure 7-5, the output depends on the value of *v*$_j$(*n*) at time *n*. The cost function *J* can therefore be replaced by its instantaneous value *I*(*n*) (see equations (7.17) and (7.18)). The local gradient is expressed simply by

$$\delta_j^2(n) = \frac{\partial J}{\partial v_j(n)} = \frac{\partial I(n)}{\partial v_j(n)} = e_j(n)\varphi'(v_j(n)) \tag{7.23}$$

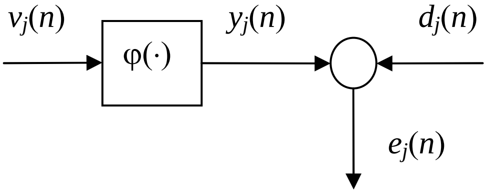


Figure 7-5: Output neuron *j*.

Case 2 – neuron *j* is a hidden unit

Neuron *j* located in a hidden layer feeds several output-layer neurons in a forward manner as shown in Figure 7-6. We use symbol $\mathcal{A}$ for the set of such neurons.

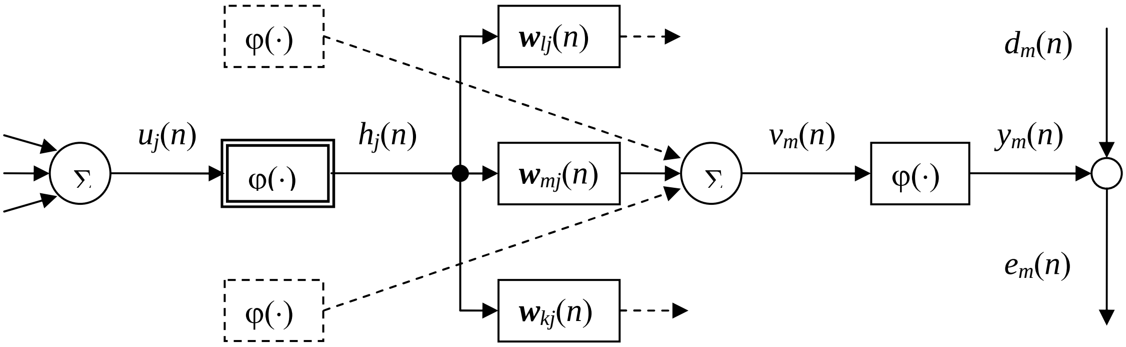


Figure 7-6: Hidden neuron *j*.

Let $v_m(n)$ denote the internal activation potential of the output neuron *m*. We may then write

$$\delta_j^1(n) = \frac{\partial J}{\partial u_j(n)} = \sum_{m\in\mathcal{A}} \sum_n \frac{\partial J}{\partial v_m(n)} \frac{\partial v_m(n)}{\partial u_j(n)} \tag{7.24}$$

Using the definition (7.21) (with index *m* used in place of *j*) we will rewrite (7.24) as

$$\begin{aligned} \delta_j^1(n) &= \sum_{m\in\mathcal{A}} \sum_n \delta_m^2(n) \frac{\partial v_m(n)}{\partial u_j(n)} \\ &= \sum_{m\in\mathcal{A}} \sum_n \delta_m^2(n) \frac{\partial v_m(n)}{\partial h_j(n)} \frac{\partial h_j(n)}{\partial u_j(n)} \\ &= \varphi'(u_j(n)) \sum_{m\in\mathcal{A}} \sum_n \delta_m^2(n) \mathbf{w}_{mj}(n) \end{aligned} \tag{7.25}$$

The sum $\sum_n \delta_m^2(n)\mathbf{w}_{mj}(n)$ represents an output from a synaptic FIR filter with $\delta_m^2(n)$ as input. The output from the filter is $(\boldsymbol{\delta}_m^2(n))^T \mathbf{w}_{mj}(n)$ (see Figure 7-4 and equation (7.13)), where

$$\boldsymbol{\delta}_m^2(n) = \begin{bmatrix} \delta_m(n) & \delta_m(n-1) & \cdots & \delta_m(n-M) \end{bmatrix}^T \tag{7.26}$$

The equation (7.25) can be rewritten as

$$\delta_j^1(n) = \varphi'(u_j(n)) \sum_{m\in\mathcal{A}} (\boldsymbol{\delta}_m^2(n))^T \mathbf{w}_{mj}(n) \tag{7.27}$$

We are now ready to summarize the weight update equations for *temporal back-propagation*. Comparing (7.28) with its counterparts (4.8) and (4.16) reveals a *vector generalization* of the standard back-propagation algorithm.

$$\begin{aligned}\mathbf{w}_{ji}^{2}(n+1) &= \mathbf{w}_{ji}^{2}(n) - \eta\delta_j^2(n)\mathbf{h}_i(n) \\ \mathbf{w}_{ji}^{1}(n+1) &= \mathbf{w}_{ji}^{1}(n) - \eta\delta_j^1(n)\mathbf{x}_i(n)\end{aligned} \tag{7.28}$$

7.3.1 Modelling time series

We will illustrate an application of the FIR multilayer network as a device for the nonlinear prediction of a time series. Consider a scalar time series $\{x(n)\}$, which is described by a *nonlinear regressive model of order p* as follows:

$$x(n) = f(x(n-1), x(n-2), \ldots, x(n-p)) + \varepsilon(n) \tag{7.29}$$

where f is a nonlinear *function* and $\varepsilon(n)$ is a residual. It is assumed that $\varepsilon(n)$ is drawn from a *white Gaussian noise process*. The nonlinear function f is unknown, and the only thing that is known is a set of *observables*: $x(1)$, $x(2)$, . . . , $x(N)$, where N is the total length of the time series. The requirement is to construct a physical model of the time series, given this data set. To do so, we may use an FIR multilayer perceptron as a one-step predictor of some order p. Specifically, the network is designed to make a prediction of the sample $x(n)$, given the past p samples $x(n\text{-}1)$, $x(n\text{-}2)$, . . . , $\mathrm{x}(n\text{-}p)$, as shown by

$$\hat{x}(n) = F(x(n-1), x(n-2), \ldots, x(n-p)) + e(n) \tag{7.30}$$

The nonlinear function F is the approximation of the unknown function f, which is computed by the FIR network. The actual sample value $x(n)$ acts as the *desired response*. Thus the FIR network is trained so as to minimize the squared value of the prediction error

$$e(n) = \hat{x}(n) - x(n), \quad p+1 \le n \le N \tag{7.31}$$

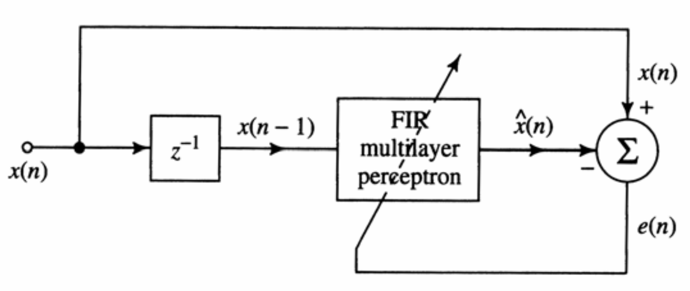


Figure 7-7: Open-loop adaptation scheme.

The open-loop adaptation method depicted in Figure 7-7 represents *feedforward prediction*. Once the training of the FIR network is completed, the generalization performance of the network is evaluated by performing *recursive prediction* in an autonomous fashion. Specifically, the predictive capability of the network is tested by using the closed-loop testing scheme, illustrated in Figure 7-8. Thus a “short-term” prediction of the time series is computed iteratively by feeding the sequence of one-step predictions computed by the network back into the input:

$$\hat{x}(n) = F(\hat{x}(n-1), \hat{x}(n-2), \ldots \hat{x}(n-p)) \tag{7.32}$$

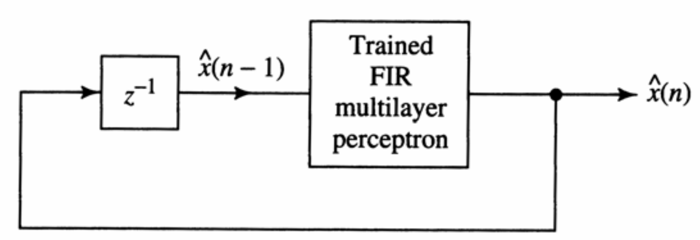


Figure 7-8: Closed-loop testing scheme.

Note that (after initialization) there is no external input applied to the network. We refer to the prediction so computed as short-term, because ordinarily it is only possible to maintain a reliable prediction of the time series through a limited number of time steps into the future. In the case of *chaotic phenomena*, the behaviour of which is *highly sensitive to initial conditions*, the quality of the prediction degrades rapidly once we go past a certain number of time steps into the future.

Wan (1994) has used an FIR network to perform nonlinear prediction on the time series plotted in Figure 7-9, which shows the chaotic intensity pulsations of an NH_3 laser. (This particular time series was distributed as part of the Santa Fe Institute Time-Series Competition, which was held in the United States in 1992.) For the laser data, only 1000 samples of the sequence (i.e. those to the left of the dashed line in Figure 7-9) were provided. The requirement was to predict the next 100 samples. During the course of the competition, the physical origin of the data set, as well as the 100 point continuations, were kept secret in order to ensure that there would be no bias built into the design of the predictor.

The FIR network was designed as a 1-12-12-1 fully connected feedforward network, configured as follows:

Input layer:
1 node
First hidden layer:
Number of neurons: 12
Number of taps per synaptic filter: 25
Second hidden layer:
Number of neurons: 12
Number of taps per synaptic filter: 5
Output layer:
Number of neurons: 1
Number of taps per synaptic filter: 5

For a model (prediction) order p = 25, the 100-step prediction obtained using this network is shown in Figure 7-10. This prediction was made on the basis of only the past 1000 samples, and no actual values were provided past 1000. The diagram shows that the prediction performed by the FIR network is indeed remarkable.

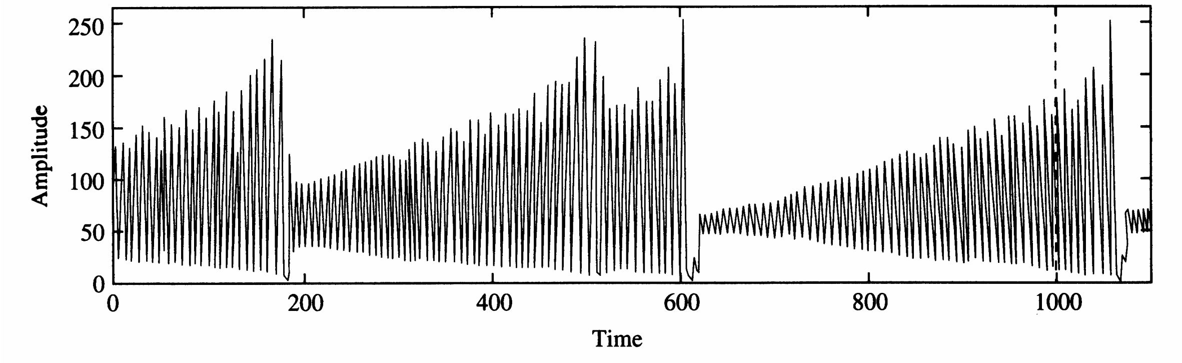


Figure 7-9: Chaotic time series.

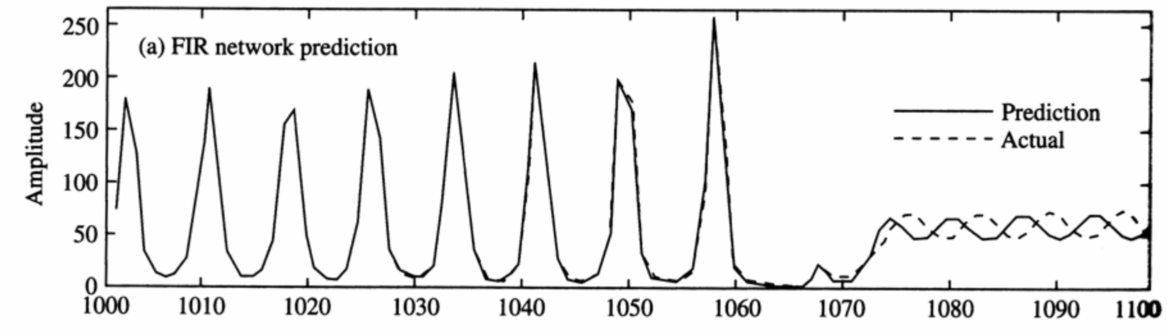


Figure 7-10: Recursive prediction using an FIR network trained on the first 1000 samples of the time series in Figure 7-9.

7.4 Real-time recurrent network

Consider a network consisting of a total of N neurons with M external input connections. Let $x(n)$ denote the M-by-1 external input vector applied to the network at discrete time n, and let $y(n+1)$ denote the corresponding N-by-1 vector of individual neuron outputs produced one step later at time $n+1$. The input vector $x(n)$ and one-step delayed output vector $y(n)$ are concatenated to form the $(M+N)$-by-1 vector $u(n)$, whose ith element is denoted by $u_i(n)$. Let $\mathcal{A}$ denote the set of indices i for which $x_i(n)$ is an external input, and let $\mathcal{B}$ denote the set of indices i for which $u_i(n)$ is the output of a neuron. We thus have

$$u_i(n) = \begin{Bmatrix} x_i(n), & i \in \mathcal{A} \\ y_i(n), & i \in \mathcal{B} \end{Bmatrix} \tag{7.33}$$

$$u(n) = \left[\begin{array}{cc|cccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \hline x_1(n) & x_2(n) & y_1(n) & y_2(n) & y_3(n) & y_4(n) \end{array}\right], \qquad \mathcal{A} = \{1, 2\}, \mathcal{B} = \{3, 4, 5, 6\}$$

One input will be equal to one (a threshold or bias). We may distinguish two distinct layers in the network, a *concatenated input-output layer* and a *processing layer* as shown in Figure 7-11 for $M = 2$ and $N = 4$. The network is fully connected; there are $M \times N$ forward connections and N^2 feedback connections. N of the feedback connections are in fact *self-feedback* connections.

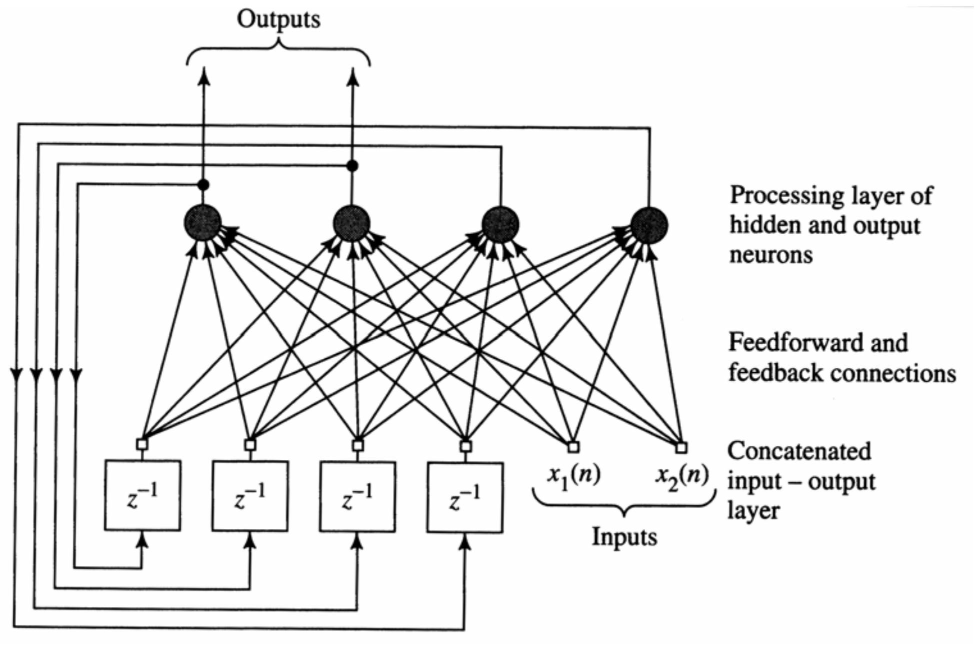

Outputs
Processing layer of hidden and output neurons
Feedforward and feedback connections
Concatenated input – output layer
z^{-1}
z^{-1}
z^{-1}
z^{-1}
$x_1(n)$
$x_2(n)$
Inputs

Figure 7-11: Diagram of a real-time recurrent network.

The internal activity of neuron *j* at time *n* for $j \in \mathcal{B}$ is

$$v_j(n) = \sum_{i \in \mathcal{A} \cup \mathcal{B}} w_{ji}(n) u_i(n) \tag{7.34}$$

At the next time step *n*+1, the output of neuron *j* is computed by passing $v_j(n)$ through the nonlinearity φ(·):

$$y_j(n+1) = \varphi(v_j(n)) \tag{7.35}$$

It is important to note that the external input vector *x*(*n*) at time *n* does not influence the output of any neuron until time *n*+1.

7.4.1 Real-time temporal supervised learning algorithm

Let $d_j(n)$ denote the desired response of neuron *j* at time *n*. Let $\mathcal{C}(n)$ denote the set of neurons that are chosen to act as *visible* units (providing measurable outputs) at time *n*; the remaining neurons of the processing layer are *hidden*. Then the error

$$e_j(n) = \begin{Bmatrix} y_j(n) - d_j(n), & j \in \mathcal{C}(n) \\ 0, & \text{otherwise} \end{Bmatrix} \tag{7.36}$$

An *instantaneous* sum of squared error at time *n* is

$$I(n) = \frac{1}{2} \sum_{j \in \mathcal{C}} e_j^2(n) \tag{7.37}$$

The objective is to minimize a cost function obtained by summing *I*(*n*) over all time:

$$J = \sum_{n} I(n) \tag{7.38}$$

To accomplish this objective we will again use the method of steepest descent, which requires knowledge of the *gradient matrix*

$$\frac{\partial J}{\partial w} = \sum_{n} \frac{\partial I(n)}{\partial w} \tag{7.39}$$

However, we want to develop a *real-time* learning algorithm so we will use an instantaneous estimate $\frac{\partial I(n)}{\partial w}$ of the gradient, which results in an approximation to the method of steepest descent.

The incremental change of $w_{kl}(n)$ will be

$$\Delta w_{kl}(n) = -\eta \frac{\partial I(n)}{\partial w_{kl}(n)} = -\eta \sum_{j \in \mathcal{C}} e_j(n) \frac{\partial e_j(n)}{\partial w_{kl}(n)} = -\eta \sum_{j \in \mathcal{C}} e_j(n) \frac{\partial y_j(n)}{\partial w_{kl}(n)} \tag{7.40}$$

To determine the partial derivative $\partial y_j(n) / \partial w_{kl}(n)$ we differentiate the network dynamics, i.e. equations (7.34) and (7.35) with respect to $w_{kl}(n)$:

$$\frac{\partial y_j(n+1)}{\partial w_{kl}(n)} = \frac{\partial y_j(n+1)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{kl}(n)} = \varphi'(v_j(n)) \frac{\partial v_j(n)}{\partial w_{kl}(n)} \tag{7.41}$$

$$\begin{aligned} \frac{\partial v_j(n)}{\partial w_{kl}(n)} &= \sum_{i \in \mathcal{A} \cup \mathcal{B}} \frac{\partial (w_{ji}(n) u_i(n))}{\partial w_{kl}(n)} \\ &= \sum_{i \in \mathcal{A} \cup \mathcal{B}} \left[w_{ji}(n) \frac{\partial u_i(n)}{\partial w_{kl}(n)} + \frac{\partial w_{ji}(n)}{\partial w_{kl}(n)} u_i(n) \right] \end{aligned} \tag{7.42}$$

The derivative $\partial w_{ji}(n) / \partial w_{kl}(n)$ equals 1 when $j = k$ and $i = l$; otherwise it is zero. Equation (7.42) can be, therefore, rewritten as

$$\frac{\partial v_j(n)}{\partial w_{kl}(n)} = \sum_{i \in \mathcal{A} \cup \mathcal{B}} w_{ji}(n) \frac{\partial u_i(n)}{\partial w_{kl}(n)} + \delta_{kj} u_l(n) \tag{7.43}$$

where δ_{kj} is a *Kronecker delta* equal to 1 when $j = k$ and zero otherwise. From the definition of $u_i(n)$ given in (7.33) we note that

$$\frac{\partial u_i(n)}{\partial w_{kl}(n)} = \left\{ \begin{matrix} 0, & i \in \mathcal{A} \\ \frac{\partial y_i(n)}{\partial w_{kl}(n)}, & i \in \mathcal{B} \end{matrix} \right\} \tag{7.44}$$

By combining equations (7.41), (7.43), and (7.44) we eventually get

$$\frac{\partial y_j(n+1)}{\partial w_{kl}(n)} = \varphi'(v_j(n))\left[\sum_{i \in \mathcal{B}} w_{ji}(n)\frac{\partial y_i(n)}{\partial w_{kl}(n)} + \delta_{kl}u_l(n)\right] \tag{7.45}$$

This is a difference equation for $\pi_{kl}^{j}(n) = \dfrac{\partial y_j(n)}{\partial w_{kl}(n)}$ for all $j \in \mathcal{B}$, $k \in \mathcal{B}$, and $l \in \mathcal{A} \cap \mathcal{B}$.

For every time step *n* and all appropriate *j*, *k*, and *l* the dynamics of this system are governed by

$$\pi_{kl}^{j}(n+1) = \varphi'(v_j(n))\left[\sum_{i \in \mathcal{B}} w_{ji}(n)\pi_{kl}^{i}(n) + \delta_{kj}u_l(n)\right] \tag{7.46}$$

with initials conditions

$$\pi_{kl}^{j}(0) = 0 \tag{7.47}$$

In summary, the real-time recurrent learning (RTRL) algorithm proceeds as follows:

1. For every time step *n*, starting from *n* = 0, use the dynamic equations (7.34) and (7.35) to compute the output values of *N* neurons.
2. Use (7.46) to compute all $\pi_{kl}^{j}(n)$.
3. Calculate the weight update

$$\Delta w_{kl}(n) = -\eta \sum_{j \in \mathcal{C}(n)} e_j(n)\pi_{kl}^{j}(n) \tag{7.48}$$

4. Update the weight w_{kl} in accordance with

$$w_{kl}(n+1) = w_{kl}(n) + \Delta w_{kl}(n) \tag{7.49}$$

   and repeat the computation.

The learning coefficient η should be small enough so that the learning process is considerably slower than the network dynamics.

8 Radial-basis function networks

Function approximation seeks to describe the behaviour of very complicated functions by ensembles of simpler functions as shown in Figure 8-1.

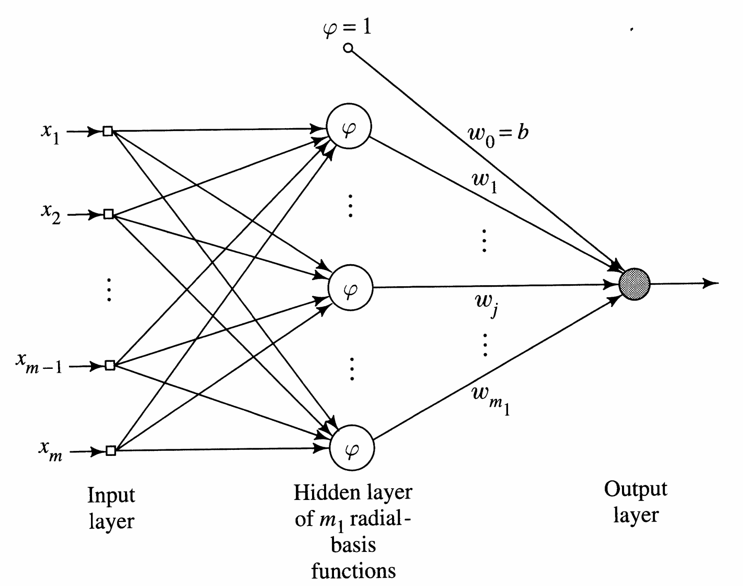


Figure 8-1: Radial-basis function network (RBFN).

Let $f(x)$ be a function of a vector $x = \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix}^T$. The goal of function approximation using radial-basis function network is to describe the behaviour of $f(x)$ by combination of simpler functions $\varphi_i(x)$:

$$\hat{f}(x, w) = \sum_{i=0}^{m_1} w_i \varphi_i(x) \tag{8.1}$$

where w_i are elements of the weight vector $w = \begin{bmatrix} w_0 & w_1 & \cdots & w_{m_1} \end{bmatrix}^T$ and $\varphi_0 = 1$. The difference between the function $f(x)$ and its approximant (8.1)

$$\left\| f(x) - \hat{f}(x, w) \right\| < \varepsilon \tag{8.2}$$

When one can find coefficients w_i that make ϵ arbitrarily small for any function $f(\cdot)$ over the domain of interest, we say that the elementary function set $\{\varphi_i(\cdot)\}$ has the property of *universal approximation*, or that the set of elementary functions $\varphi_i(\cdot)$ is *complete*.

Three basic decisions have to be made when designing a radial-basis function network:

1. The choice of elementary functions $\varphi_i(\cdot)$.
2. How to compute the weights w_i.
3. How to select the number of elementary functions m_1.

8.1 Basic RBFN

Radial-basis functions are typically centred at c_i:

$$\varphi_i(x) = G(\|x - c_i\|), \quad i = 1, 2, \dots, m_1 \tag{8.3}$$

The most popular choice of elementary function G is a Gaussian function

$$\begin{aligned} G(x, c_i) &= \exp(-\frac{(x-c_i)^2}{2\sigma^2}) && \text{one-dimensional} \\ G(x, c_i) &= \exp(-\frac{(x-c_i)^T \Sigma^{-1} (x-c_i)}{2}) && \text{multidimensional} \end{aligned} \tag{8.4}$$

with variance σ^2 or covariance matrix Σ.

Let training set T consists of N training samples:

$$T = \begin{bmatrix} \left[(x_1)^T \quad f(x_1)\right] \\ \left[(x_2)^T \quad f(x_2)\right] \\ \cdots \\ \left[(x_N)^T \quad f(x_N)\right] \end{bmatrix} \tag{8.5}$$

8.1.1 Fixed centre at each training sample

A basic RBFN contains $m_1 = N$ elementary functions with centres $c_i = x_i$, $i = 1, 2, \dots, N$. The hidden layer of the basic RBFN grows with the number of training data.

The width σ of Gaussian functions can be fixed according to the spread of the centres:

$$\sigma = \frac{d_{\max}}{\sqrt{2m_1}} \tag{8.6}$$

where m_1 is the number of centres and d_{max} is the maximum distance between the chosen centres. This formula ensures that the individual radial-basis functions are not too peaked or too flat; both of these extreme conditions should be avoided.

The width σ_k of a Gaussian function $G(x,c_k)$ can also be calculated by so called *P-nearest neighbour heuristic*. Consider a given centre c_k and assume that $c_1, c_2, \dots, c_P$ are P nearest centres. Then we set

$$\sigma_k = \sqrt{\frac{1}{P}\sum_{i=1}^{P}\|c_k - c_i\|^2} \tag{8.7}$$

The number of nearest neighbours is usually set to $P = 2^m$, where m is the number of inputs (the dimension of the input space).

The output of RBFN is a linear combination of elementary functions:

$$y = \hat{f}(x) = \begin{bmatrix} 1 & G(x, x_1) & \cdots & G(x, x_N) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_N \end{bmatrix} \tag{8.8}$$

where N is the number of training samples. Equation (8.8) must be satisfied for each training sample (measurement). The set of such equations is called the set of *normal equations* and the weight vector w is the solution of a set of linear algebraic equations.

$$\begin{bmatrix} 1 & G(x_1, x_1) & G(x_1, x_2) & \cdots & G(x_1, x_N) \\ 1 & G(x_2, x_1) & G(x_2, x_2) & \cdots & G(x_2, x_N) \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & G(x_N, x_1) & G(x_N, x_2) & \cdots & G(x_N, x_N) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_N) \end{bmatrix} \tag{8.9}$$

The above set of normal equations can be written in a more compact form

$$\mathbf{G} w = \mathbf{f} \tag{8.10}$$

and the weight vector calculated by using the *pseudoinverse* of matrix **G**:

$$w = \mathbf{G}^{+} \mathbf{f} \tag{8.11}$$

8.1.2 Example of function approximation – large RBFN

The same function as in section 4.14.4 is used. The training data consists of 100 points. RBFN has, therefore, $m_1 = 100$ hidden layer neurons with Gaussian activation functions (elementary functions) centred at training samples. All Gaussian functions have the same width calculated from (8.6).

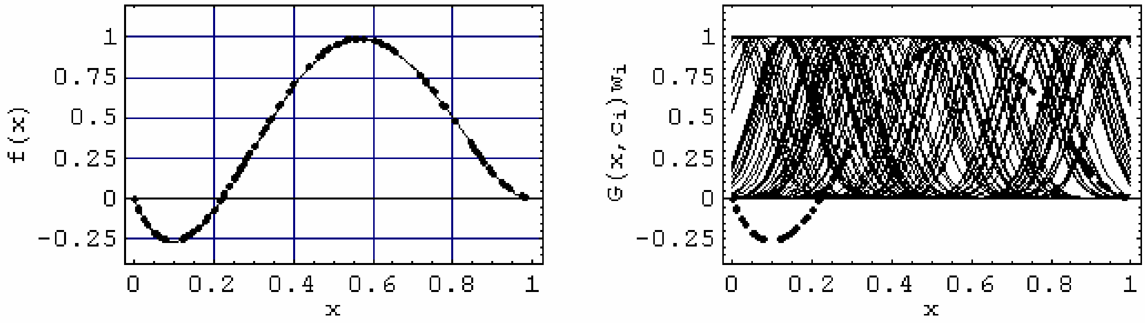


Figure 8-2: RBFN output (left, dashed line) and the ensemble of 100 elementary functions (right).

The ensemble of 100 elementary functions shown in Figure 8-2 (right) centred at training samples approximate the original function very well (left).

The number of elementary functions $m_l = N$ grows with the number of training samples. The RBFN can, therefore, be prohibitively large.

8.1.3 Fixed centres selected at random

The simplest method how to reduce the size of the network is to select $m_l < N$ centres randomly from the training data set. This is considered to be a "sensible" approach provided that the training data are distributed in a representative manner for the problem at hand.

The set of normal equations (8.12) consists of N equations for m_l+1 weights, where $N > m_l$. The weights will be again calculated as shown in equation (8.11).

$$\begin{bmatrix} 1 & G(x_1,c_1) & G(x_1,c_2) & \cdots & G(x_1,c_{m_1}) \\ 1 & G(x_2,c_1) & G(x_2,c_2) & \cdots & G(x_2,c_{m_1}) \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & G(x_N,c_1) & G(x_N,c_2) & \cdots & G(x_N,c_{m_1}) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{m_1} \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_N) \end{bmatrix} \tag{8.12}$$

8.1.4 Example of function approximation – small RBFN

Training data selected randomly from the training set consist of 5 points. RBFN has, therefore, m_l = 5 hidden layer neurons with Gaussian activation functions (elementary functions) centred at selected training samples. All Gaussian functions have the same width calculated from (8.6).

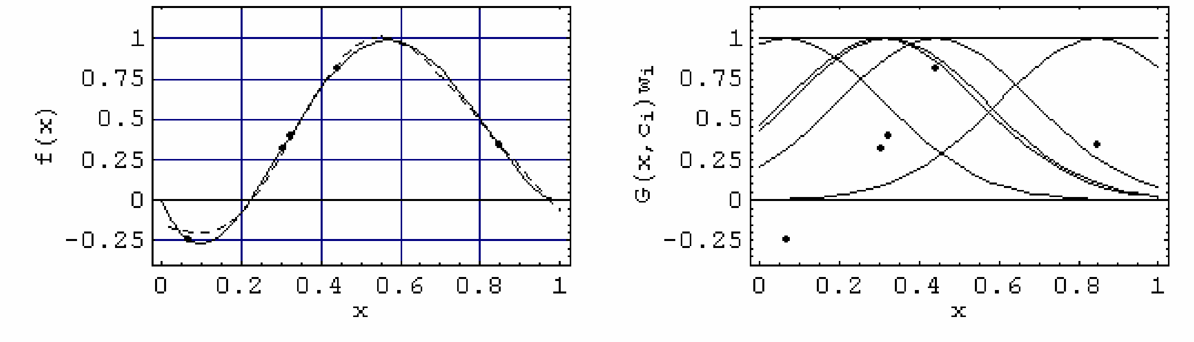


Figure 8-3: RBFN output (left, dashed line) and the ensemble of 5 elementary functions (right).

The ensemble of 5 elementary functions shown in Figure 8-3 (right) centred at training samples approximate the original function well (left).

8.1.5 Example of function approximation – noisy data

Uniformly distributed noise from the interval [-0.1, 0.1] was added to the training samples used in sections 8.1.2 and 8.1.4. The noisy data consists of 100 samples as shown in Figure 8-4.

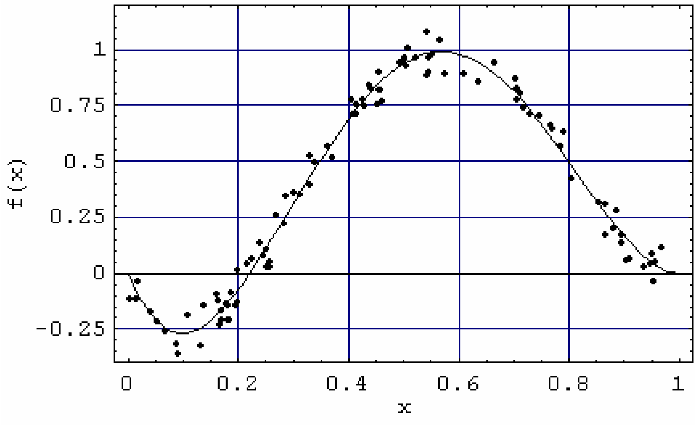


Figure 8-4: Noisy training samples.

The equation (8.11) gives the best approximate solution to the set of normal equations. In case that $m_1 < N$ (more equations than variables) the weights are calculated in such a way that the Euclidean norm $\|\mathbf{G}w - \mathbf{f}\|$ is minimized. We can, therefore, expect that the RBFN output will be less sensitive to the noise. This expectation is confirmed by computer experiments shown below.

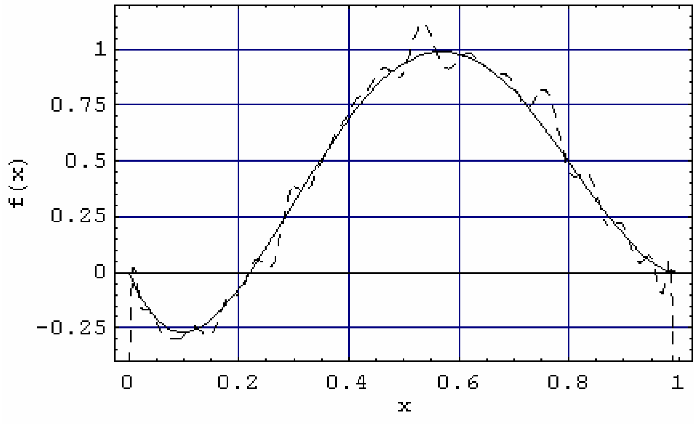


Figure 8-5: Gaussian functions are centred at each training sample; $m_1 = N = 100$.

Figure 8-5 shows that if $m_1 = N$ the impact of noise is not averaged and the RBFN output fluctuates around the desired values. The network learns noisy data and does not generalize well. If $m_1 < N$ the network contains smaller number of radial-basis functions (hidden neurons) and its generalization capability increases, as seen in Figure 8-6. All Gaussian functions have the same width calculated from (8.6).

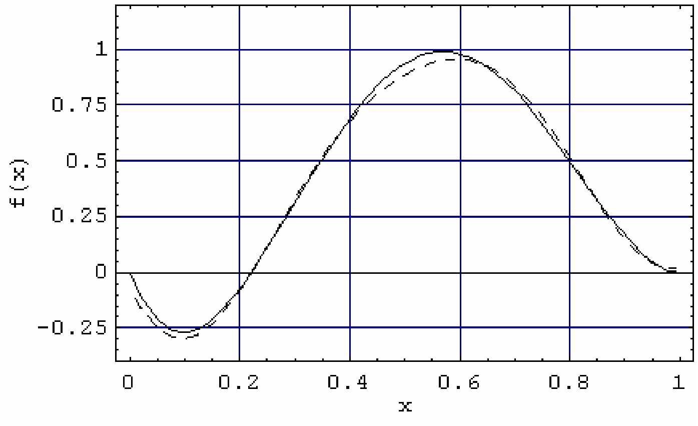


Figure 8-6: Gaussian functions are centred at 5 randomly selected training samples; m_1 = 5, N =100.

8.2 Generalized RBFN

The main problem with the method of fixed centres described in section 8.1 is the fact that it may require a large training set for a satisfactory level of performance. One way of overcoming this limitation is to use one of the following hybrid learning processes:

1. *Self-organized learning process*, the purpose of which is to estimate appropriate locations for the centres of the radial basis functions in the hidden layer.
2. *Supervised learning process*, the purpose of which is to estimate appropriate locations for the centres of the radial basis functions in the hidden layer, their widths, and weights.

8.2.1 Self-organized selection of centres

For the self-organized learning process we need a *clustering algorithm* that partitions the given set of data points into subgroups, each of which should be as homogeneous as possible. One such algorithm is the *k-means clustering algorithm*, which places the centres of the radial-basis functions in only those regions of the input space where significant data are present. Let m_1 denote the number of radial-basis functions; the determination of a suitable value for m_1 may require experimentation. Let $\{c_k(n)\}_{k=1}^{m_1}$ denote the centres of the radial-basis functions at iteration n of the algorithm. Then, the k-means clustering algorithm proceeds as follows:

1. *Initialization*. Choose random values for the initial centres $c_k(0)$; the only restriction is that these initial values be different.
2. *Sampling*. Draw a sample vector x from the input space with a certain probability. The vector x is input into the algorithm at iteration n.
3. *Similarity matching*. Let $k(x)$ denote the index of the best-matching (winning) centre for input vector x. Find $k(x)$ at iteration n by using the minimum-distance Euclidean criterion:

$$k(x) = \arg\min_k \|x(n) - c_k(n)\|, \quad k = 1, 2, \ldots, m_1 \tag{8.13}$$

where $c_k(n)$ is the centre of the kth radial-basis function at iteration n.

4. *Updating.* Adjust the centres of the radial-basis functions using the update rule:

$$c_k(n+1) = \begin{Bmatrix} c_k(n) + \eta[x(n) - c_k(n)], & k = k(x) \\ c_k(n), & \text{otherwise} \end{Bmatrix} \tag{8.14}$$

where η is a *learning-rate parameter* that lies in the range $0 < \eta < 1$.

5. *Continuation.* Increment n by 1, go back to step 2, and continue the procedure until no noticeable changes are observed in the centres c_k.

The k-means clustering algorithm just described is, in fact, a special case of a competitive (winner-takes-all) learning process known as the *self-organizing map*, which is discussed in section 6.

A limitation of the k-means clustering algorithm is that it can only achieve a local optimum solution that depends on the initial choice of cluster centres. Consequently, an unnecessarily large network may be created.

Having identified the individual centres of the Gaussian radial-basis functions using the k-means clustering algorithm, and their widths using the P-nearest neighbour heuristic or expression (8.6), the next and final stage of the hybrid learning process is to estimate the weights of the output layer. It can be done similarly as in section 8.1 using pseudoinverse as shown in equation (8.11).

8.2.2 Example – noisy data, self-organized centres

The same data as shown in Figure 8-4 is used. At first, $m_1 = 5$ centres were created randomly. The corresponding RBFN output does not approximate the underlying function well (see Figure 8-7).

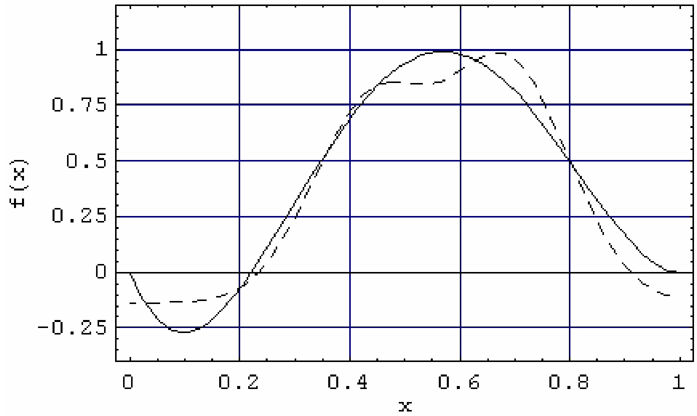


Figure 8-7: Function approximation with 5 initial centres [0.433, 0.373, 0.668, 0.343, 0.690].

Then the initial centres were self-organized by k-means clustering algorithm described in section 8.2.1. The corresponding RBFN produced much better approximation as shown in Figure 8-8.

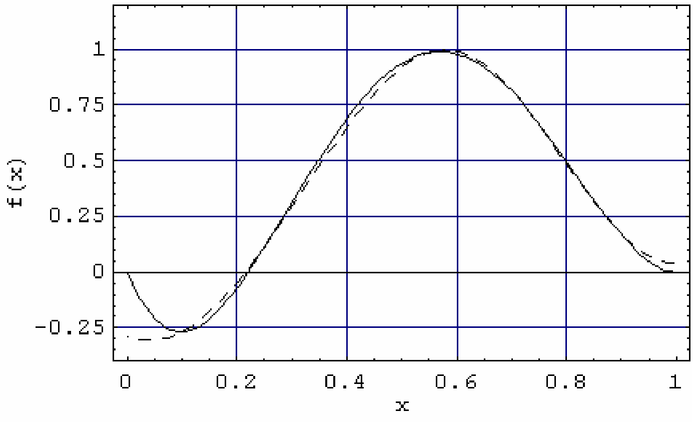


Figure 8-8: Function approximation with 5 self-organized centres [0.445, 0.262, 0.640, 0.105, 0.875].

8.2.3 Supervised selection of centres

In this approach, the centres, widths, and weights undergo a supervised learning process. A natural candidate for such a process is error-correction learning, which is most conveniently implemented using a gradient-descent procedure.

The first step in the development of such a learning procedure is to define the cost function

$$J = \frac{1}{2}\sum_{j=1}^{N} e_j^2 \tag{8.15}$$

where N is the size of the training set and e_j is the error signal defined by

$$e_j = y(x_j) - d_j = \sum_{i=0}^{m_1} G(x_j, c_i, \sigma_i) w_i - d_j \tag{8.16}$$

where $G(x_j, c_0, \sigma_0) = 1$ is the output from the bias neuron. The requirement is to find the free parameters w, c, and σ so as to minimize J.

Parameter update equations are

$$\begin{aligned} w_i(n+1) &= w_i(n) - \eta_w \frac{\partial J(n)}{\partial w_i(n)}, \quad i = 0,1,\ldots,m_1 \\ c_i(n+1) &= c_i(n) - \eta_c \frac{\partial J(n)}{\partial c_i(n)}, \quad i = 1,2,\ldots m_1 \\ \sigma_i(n+1) &= \sigma_i(n) - \eta_\sigma \frac{\partial J(n)}{\partial \sigma_i(n)}, \quad i = 1,2,\ldots m_1 \end{aligned} \tag{8.17}$$

where partial derivatives of J are calculated as follows:

$$
\begin{aligned}
\frac{\partial J(n)}{\partial w_i(n)} &= \sum_{j=1}^{N} e_j(n) G(x_j, c_i(n), \sigma_i(n)), & i &= 0,1,\ldots,m_1 \\
\frac{\partial J(n)}{\partial c_i(n)} &= \sum_{j=1}^{N} e_j(n) w_i(n) G(x_j, c_i(n), \sigma_i(n)) \frac{x_j - c_i(n)}{\sigma_i^2(n)}, & i &= 1,2,\ldots,m_1 \\
\frac{\partial J(n)}{\partial \sigma_i(n)} &= \sum_{j=1}^{N} e_j(n) w_i(n) G(x_j, c_i(n), \sigma_i(n)) \frac{(x_j - c_i(n))^2}{\sigma_i^3(n)}, & i &= 1,2,\ldots,m_1
\end{aligned}
\qquad (8.18)
$$

The above partial derivatives are calculated under an assumption that the RBFN has one input and Gaussian radial-basis functions.

The following points are noteworthy:

1. The cost function J is convex with respect to the linear parameters w_i, but non-convex with respect to the centres c_i and widths σ_i. The search for the optimum values of c_i and σ_i may, therefore, get stuck at a local minimum in parameter space.
2. The update equations for w_i, c_i, and σ_i are assigned different learning coefficients η_w, η_c, and η_σ, respectively.

For the initialization of the gradient-descent procedure, it is often desirable to begin the search in parameter space from a *structured* initial condition that limits the region of parameter space to an already known useful area. In so doing, the likelihood of converging to an undesirable local minimum is reduced.

9 Adaline (Adaptive Linear System)

The leading characteristic of neural and adaptive systems is their adaptivity (Figure 9-1).

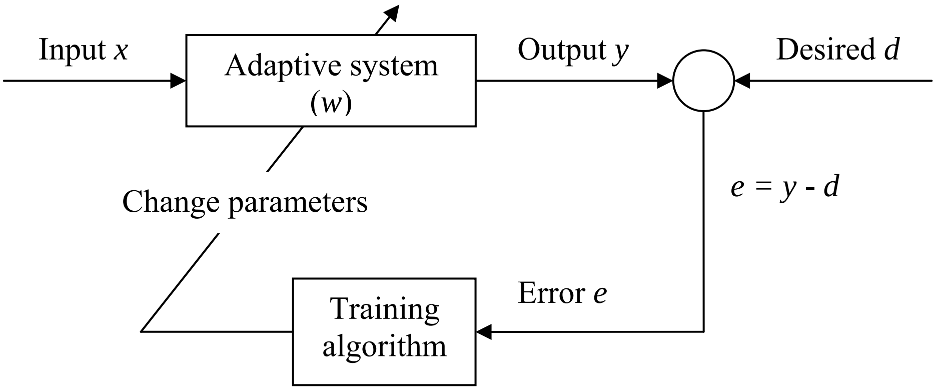


Figure 9-1: Adaptive system.

Instead of being built from specification, neural and adaptive systems use external data to automatically set their parameters. This means that neural systems are parametric. It also means that they are made “aware” of their output through a performance feedback loop that includes a cost function. The performance feedback is utilized directly to change the parameters through procedures called *learning* or *training rules*, so that the system output improves with respect to the desired goal (i.e. the error *e* decreases through training).

The data collection must be carefully planed to ensure that

- Data will be sufficient
- Data will capture the fundamental principles of the system
- Data is as free as possible form observation noise

Our brain is somehow able to extract much more information from pictures than from numbers, so data should be first plotted before performing data analysis. Plotting the data allows verification, assures the researcher that the data was collected correctly, and provides a “feel” for the relationships that exist in the data.

9.1 Linear regression

It will be instructive to start with a well known data modelling technique – *linear regression*. Mathematica notebook Linear model.nb illustrates how data can be fitted by various regression models, how those models can be statistically evaluated, and how the most appropriate model can be selected. Important measures used to compare various models are the *coefficient of determination* and the *adjusted coefficient of determination*. This Mathematica notebook also illustrates the *bias-variance dilemma*, which leads to the optimal size of a learning machine.

Mathematica notebook Noise.nb refreshes our knowledge of a normal distribution and shows that a too large data set may be expensive and only marginally beneficial for model building.

9.2 Linear processing element

An adaptive linear processing element (Adaline) is described by a linear equation

$$y = w_1 x_1 + w_2 x_2 + \cdots + w_p x_p + b = \mathbf{w}^T \mathbf{x} + b \qquad (8.19)$$

where w_i is the slope w.r.t. input x_i and b is the bias. The Adaline described by (8.19) has $p+1$ free parameters.

Let $\mathbf{T}$ be a training set consisting of N training samples:

$$\mathbf{T} = \begin{bmatrix} \mathbf{x}_1 & d_1 \\ \mathbf{x}_2 & d_2 \\ \vdots & \vdots \\ \mathbf{x}_N & d_N \end{bmatrix} \qquad (8.20)$$

Each row in the training set $\mathbf{T}$ represents one training sample consisting of input vector $\mathbf{x}_k = \begin{bmatrix} x_1 & x_2 & \cdots & x_p \end{bmatrix}^T$ and corresponding desired output d_k. The objective is to find the free parameters in such a way that model (8.19) fits training data (8.20) as well as possible.

It is unlikely that all the data will be approximated exactly. We can, therefore, write

$$d_i = y_i - \varepsilon_i = (\mathbf{w}^T \mathbf{x}_i + b) - \varepsilon_i \qquad (8.21)$$

where ε_i is the instantaneous error that is subtracted (or added) from the model output y_i. The best fit will be achieved if all instantaneous errors ε_i are as small as possible. The average sum of square errors (also called the *mean square error* – MSE) is a widely used criterion:

$$J = \frac{1}{2N} \sum_{i=1}^{N} \varepsilon_i^2 = \frac{1}{2N} \sum_{i=1}^{N} (y_i - d_i)^2 \qquad (8.22)$$

N is the number of observations (the size of the training set $\mathbf{T}$). Criterion (8.22) can be optimized analytically but we will focus on an adaptive solution using a gradient search method.

9.3 Gradient method

To reach the minimum of J, the search must be done in the direction opposite to the gradient. The update of free parameters is governed by this equation:

$$\bar{\mathbf{w}}(k+1) = \bar{\mathbf{w}}(k) - \eta \nabla J(k) \tag{8.23}$$

where η is a learning coefficient, $\nabla J(k)$ denotes the gradient of J at the kth iteration, and an augmented vector of free parameters

$$\bar{\mathbf{w}} = \begin{bmatrix} b & w_1 & w_2 & \cdots & w_p \end{bmatrix}^T \tag{8.24}$$

This search procedure is called the *steepest descent* method.

The gradient $\nabla J(k)$ is not known explicitly and must be estimated. In the late 1960s, Widrow proposed to use the instantaneous value I of the criterion instead of its true quantity J:

$$I = \frac{1}{2}\varepsilon^2 = \frac{1}{2}\left[(\mathbf{w}^T\mathbf{x} + b) - d\right]^2 \tag{8.25}$$

The weight update formula (8.23) changes to

$$\bar{\mathbf{w}}(k+1) = \bar{\mathbf{w}}(k) - \eta \nabla I(k) \tag{8.26}$$

where gradient

$$\nabla I(k) = \varepsilon(k) \begin{bmatrix} 1 \\ x_1(k) \\ x_2(k) \\ \vdots \\ x_p(k) \end{bmatrix} = \varepsilon(k)\bar{\mathbf{x}}(k) \tag{8.27}$$

The steepest descent equation (8.26) eventually becomes

$$\bar{\mathbf{w}}(k+1) = \bar{\mathbf{w}}(k) - \eta\, \varepsilon(k)\bar{\mathbf{x}}(k) \tag{8.28}$$

This equation is called the *least mean squares* (LMS) algorithm.

9.3.1 Batch and sample-by-sample learning

The LMS algorithm (8.28) is presented in a form in which the weight updates are computed for each input sample and the weights are modified after each sample. This procedure is called *sample-by-sample learning*, or *on-line training*. The estimate of the gradient is going to be noisy; that is, the direction toward the minimum is going to zigzag around the gradient direction.

An alternative solution is to compute the weight update for each input sample and store these values (without changing the weights) during one pass through the training set, which is called an *epoch*. At the end of the epoch, all the weight updates are added together, and only then will the weights be updated with the composite value. This method adapts the weights with a cumulative weight update, so it will follow the gradient more closely. This method is called the *batch training* mode, or *batch learning*. Batch learning is also an implementation of the steepest-descent procedure. In fact, it provides an estimator for the gradient that is smoother than the LMS. The agreement between the analytical quantities that describe adaptation and the ones obtained experimentally is excellent with the batch update.

9.4 Optimal hyperplane for linearly separable patterns (Adatron)

Consider again the training set $\mathbf{T}$ consisting of *N* training samples:

$$\mathbf{T} = \begin{bmatrix} \mathbf{x}_1 & d_1 \\ \mathbf{x}_2 & d_2 \\ \vdots & \vdots \\ \mathbf{x}_N & d_N \end{bmatrix} \tag{8.29}$$

Each row in the training set $\mathbf{T}$ represents one training sample consisting of input vector $\mathbf{x}_k = \begin{bmatrix} x_1 & x_2 & \cdots & x_p \end{bmatrix}^T$ and corresponding desired output d_k. We assume that the pattern (class) represented by the subset $d_i = +1$ and the pattern represented by the subset $d_i = -1$ are *linearly separable*. The equation of a decision surface in the form of a hyperplane that does the separation is

$$\mathbf{w}^T\mathbf{x} + b = \overline{\mathbf{w}}^T\overline{\mathbf{x}} = 0 \tag{8.30}$$

where $\overline{\mathbf{w}}$ and $\overline{\mathbf{x}}$ are augmented vectors

$$\begin{aligned} \overline{\mathbf{w}} &= \begin{bmatrix} b & w_1 & w_2 & \cdots & w_p \end{bmatrix}^T \\ \overline{\mathbf{x}} &= \begin{bmatrix} 1 & x_1 & x_2 & \cdots & x_p \end{bmatrix}^T \end{aligned} \tag{8.31}$$

To simplify the notation we will drop the over-bars and will assume that **w** and **x** are the augmented vectors. Thus the equation of the decision surface is $\mathbf{w}^T\mathbf{x} = 0$. We may also write

$$\begin{aligned} \mathbf{w}^T\mathbf{x}_i &\ge 0 \quad \text{for } d_i = +1 \\ \mathbf{w}^T\mathbf{x}_i &< 0 \quad \text{for } d_i = -1 \end{aligned} \tag{8.32}$$

or in a more compact form

$$d_i\,\mathbf{w}^T\mathbf{x}_i \ge 0 \tag{8.33}$$

For a given weight vector **w**, the separation between the hyperplane (8.30) and the closest data point is called the *margin of separation* denoted by ρ. The goal is to find the hyperplane for which the margin of separation ρ is maximized. Such a decision surface is called the *optimal hyperplane* as illustrated in Figure 9-2.

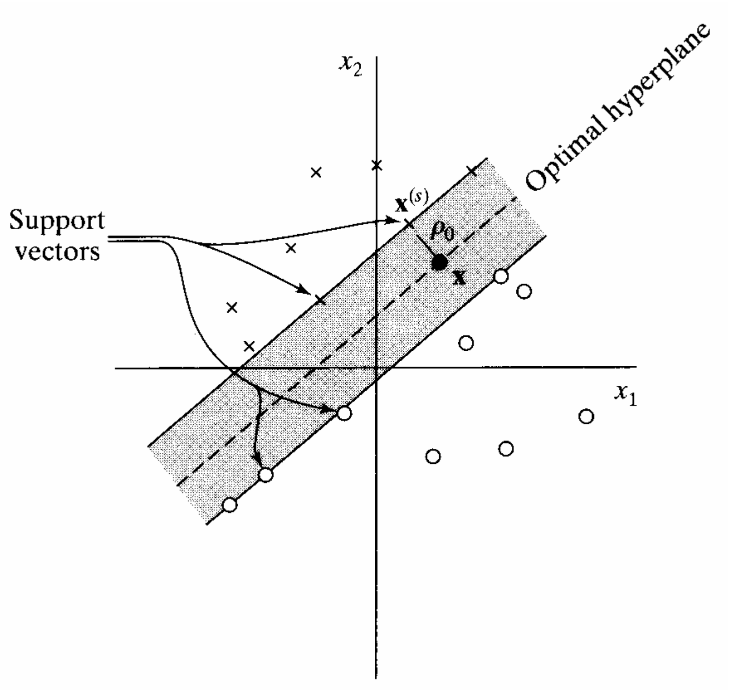


Figure 9-2: Optimal hyperplane for linearly separable patterns.

Let $\mathbf{w}_o$ denotes the optimum weight vector. The discriminant function

$$g(\mathbf{x}) = \mathbf{w}_o^T \mathbf{x} \tag{8.34}$$

gives an algebraic measure of the distance from **x** to the optimal hyperplane. As the decision surface is given by $g(\mathbf{x}) = \mathbf{w}_o^T \mathbf{x} = 0$ it follows that vectors $\mathbf{w}_o$ and **x** are perpendicular. We can, therefore, express any vector **x** as

$$\mathbf{x} = \mathbf{x}_p + r \frac{\mathbf{w}_o}{\|\mathbf{w}_o\|} \tag{8.35}$$

where $\mathbf{x}_p$ is the normal projection of **x** onto the optimal hyperplane, and *r* is an algebraic distance; *r* is positive if **x** is on the positive side of the optimal hyperplane and negative if **x** is on the negative side. Since $g(\mathbf{x}_p) = \mathbf{w}_o^T \mathbf{x}_p = 0$, it follows that

$$g(\mathbf{x}) = \mathbf{w}_o^T (\mathbf{x}_p + r \frac{\mathbf{w}_o}{\|\mathbf{w}_o\|}) = r \|\mathbf{w}_o\| \tag{8.36}$$

or

$$r = \frac{g(\mathbf{x})}{\|\mathbf{w}_o\|} \tag{8.37}$$

The particular data points $(\mathbf{x}_i, d_i)$ for which the first or second line of equation (8.32) is satisfied with the equality sign are called *support vectors*. The support vectors are those data points that lie closest to the decision surface and are therefore the most difficult to classify.

They have a direct bearing on the optimum location of the decision surface. If we knew the support vectors beforehand we would not need other training samples at all.

From equation (8.37) the *algebraic distance* from the support vector $\mathbf{x}^{(s)}$ to the optimal hyperplane is

$$r = \frac{g(\mathbf{x}^{(s)})}{\|\mathbf{w}_o\|} = \left\{ \begin{array}{ll} \frac{\rho_o}{\|\mathbf{w}_o\|} & \text{if } d^{(s)} = +1 \\ -\frac{\rho_o}{\|\mathbf{w}_o\|} & \text{if } d^{(s)} = -1 \end{array} \right\} \tag{8.38}$$

where the plus sign indicates that $\mathbf{x}^{(s)}$ lies on the positive side of the optimal hyperplane and the minus sign indicates that $\mathbf{x}^{(s)}$ lies on the negative side of the optimal hyperplane.

The optimal hyperplane defined by equation $g(\mathbf{x}) = \mathbf{w}_o^T\mathbf{x} = 0$ is unique in the sense that the optimum vector $\mathbf{w}_o$ provides the maximum possible separation between positive and negative samples. This optimum condition is attained by minimizing the Euclidean norm of the weight vector **w**.

A computationally efficient procedure for finding the optimal hyperplane is described e.g. in Haykin S: *Neural networks*, Prentice Hall 1999 (2nd ed.), p. 322-324. As the optimum hyperplane is defined by support vectors $\mathbf{x}^{(s)}$ the corresponding neural network is called a *support vector machine*.

9.4.1 Separation boundary generated by a NeuroSolutions SVM

Support vector machine implemented in NeuroSolutions uses Gaussian kernel functions and generates a nonlinear separation boundary. As can be seen in Figure 9-3, the boundary is positioned in the middle of data clusters. As the clusters are linearly separable in this case, the separation boundary could have been a straight line.

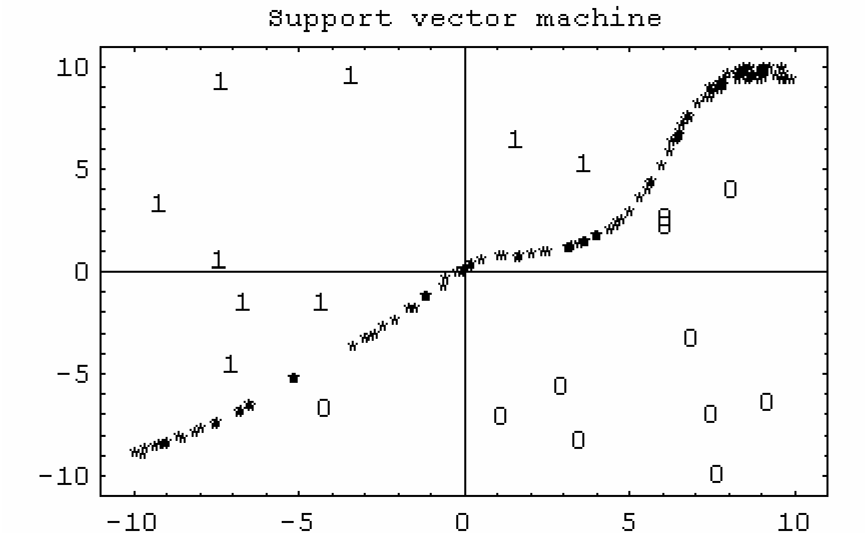


Figure 9-3: Optimum separation boundary.