Review of Mapping Neural Networks in Classification Systems

R L Hing
Department of Computer Science, University of the Witwatersrand,Wits, 2050
SA : 122ron@witsvma, Int : 122ron.witsvma@f4.n494.z5.fidonet.org

Abstract

An old approach used in classification systems has been revived by the discovery of new algorithms which use this approach to approximate complex non-linear functions. This approach, known as neurocomputing, classifies input data in parallel. Neurocomputers should be exploited for their parallelism for use in problems that are currently being solved by serial classification systems such as knowledged-based and pattern recognition systems. This paper reviews single-layer and multi-layer feed-forward neural networks, termed mapping neural networks since they approximate a function or mapping, and describes their functionality in classification systems. They have been successfully used as knowledge-bases in expert systems, and have been shown to be functionally equivalent to several conventional classifiers used in pattern recognition.
Keywords : *neural networks, classification systems, non-linearly separable functions, Perceptron Convergence Algorithm, LMS Rule, Back-Propagation.*
Computing Review Categories : *I.2, I.5.*

Received November 1989, Accepted February 1990.

1. Introduction

Most of our current computers are serial Von Neumann computers that process information algorithmically. Neurocomputers on the other hand are highly parallel computers. They are neurologically inspired computer systems that process information non-algorithmically by learning from training examples. Although they are parallel computers, they can be simulated on serial computers quite effectively. They have also been termed *artificial neural network systems* [33] and *parallel distributed processing models* [42].

A neurocomputer consists of a highly interconnected network of simple processing elements. Each processing element is a simple multiply and accumulate *unit*, consisting of many *input signals* with associated *weights*, and a single *output signal* which can be connected to many other processing elements in the network. These computers learn from experience by adjusting the weights, which represent the knowledge stored by the network, of each processing element. The structure of a neurocomputer is similar to a simple model of the brain. The brain is thought of as consisting of an interconnected mass of neurons. Each neuron is made up of a cell body called the *soma* with many input connections called *dendrites*, and a single output connection called an *axon* which can be connected to many other neurons. The site at which the axon connects to one of the many dendrites is called a *synapse*. The brain stores and retrieves information by chemically adapting these synapses (a study and comparison of the brain and artificial neural networks can be found in [43,47]).

These and many other characteristics of neurocomputers make them powerful models for use in classification problems [9,33] as well as in highly non-linear control problems [50], and adaptive signal processing [49]. The inherent parallelism found in neural networks is believed to be needed for high performance speech and image recognition while its distributed representation has better generalization properties than local representations found in most symbolic processing systems, making neural networks more robust and damage resistant. Also, the capability of neural networks to learn from training examples removes part of the knowledge acquisition bottleneck found in knowledge-based system development.

This paper describes single-layer and multi-layer feed-forward neural networks, termed *mapping neural networks*, since they approximate a function or mapping $f: \mathbf{R}^n \supset A \rightarrow \mathbf{R}^m$ from a bounded subset A of n-dimensional Euclidean space to a bounded subset $f(A)$ of m-dimensional Euclidean space, where n is the number of units in the input layer, and m is the number of units in the output layer of the neural network. The network is trained on the examples (x_1,y_1), (x_2,y_2), ..., (x_n,y_n) where $y_k = f(x_k)$ [16]. This paper also discusses the functionality of mapping neural networks in classification systems. The main reasons for the renewed stimulation in neurocomputers, which were first discovered in the 1950's, is briefly outlined. New learning algorithms have now made it possible to train multi-layer feed-forward networks. These networks can be used to solve complex non-linearly separable problems, previously not solvable by single-layer networks. Finally, the paper discusses how existing knowledge-based systems and pattern recognition systems are classification systems [8] which can exploit the inherent parallelism found in neural networks. Feed-forward networks have been successfully used as a formalism for the knowledge-base in expert systems [3, 14, 32]. As a parallel classification method, neural networks can be used as a replacement for some existing serial

classification and clustering procedures used in pattern recognition [33].

2. Single-Layer Neural Networks

A neural network is defined as a computing system made up of a number of highly interconnected simple processing elements [4,5]. Each *processing element* or *neurode* performs simple multiply and accumulate operations therefore contributing very little to the overall system but collectively making up a powerful neurocomputer.

Before discussing the connectivity of many processing elements, the function of a single element is warranted. Each processing element, u_j, consists of a number of *input signals*, x_i, and a single *output signal*, o_j. Associated with each input signal is a *weight*, w_{ji}, corresponding to the ith input signal and the jth processing element. Figure 2.1 illustrates a typical processing element.

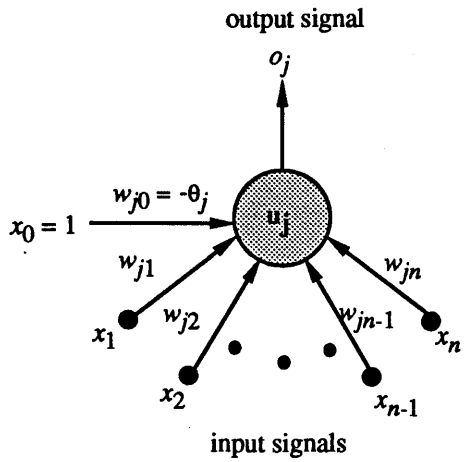

Figure 2.1. A typical processing element.

The *total* or *net input* of a processing element, designated by t_j, is calculated by summing the products of each input signal and its corresponding weight. The output signal or *activation* is then calculated by comparing the total input to a *threshold value* θ_j. Each processing element may be connected to a number of other processing elements in the neural network, where the output signal of the processing element contributes as an input signal to these other processing elements. The threshold term can be eliminated by introducing a specialized input signal x_0 fixed at the value 1. The corresponding weight of this input signal is the negative of the threshold value, and is known as the *bias*. The total input is now compared to zero instead of the threshold. It it written as follows :

$$t_j = \sum_{i=1}^{n} w_{ji} x_i - \theta_j$$

The input signals of a processing element, u_j, and their corresponding weights can be written as the components of an input and a weight vector. The input vector, $\mathbf{x}$, is $(x_1, x_2, ..., x_n)$ and the weight vector, $\mathbf{w}_j$, is $(w_{j1}, w_{j2}, ..., w_{jn})$. The weights of a processing element contain the processing element's knowledge, therefore the weight vector represents the knowledge of a processing element. These weights are adaptable since they can be trained to represent different types of knowledge. The product of an input signal and its corresponding weight can be positive therefore *exciting* the processing element, or negative causing *inhibition*, or zero which indicates there is *no connection* between the input signal and the processing element.

Initially, the components of the weight vector are randomly initialized to values within a certain range or to zero. During the *training* or modification of these weights, many input vectors or *input patterns*, are presented to the processing element. Each input pattern is described as a binary vector of 1's and 0's, or 1's and -1's. Input patterns can also be analog, normally in the range [0,1] or [-1,1]. However, binary valued input patterns are the most often used. Associated with each input pattern is a *desired* or *teaching output signal*, d_j. Together, the input pattern and the desired output signal make up a *training example* or *training pattern*. The desired output signal can also be both binary or analog valued.

The presentation of a training pattern to the processing element where the weight vector is modified is known as a *training cycle*. During each training cycle the weight vector is modified by a *learning rule*. This method of training the processing element is termed *supervised learning* since the desired output signal is known. Supervised learning is concerned with placing or *categorizing* the input patterns into one of the desired output signals or *categories*. It can also be viewed as trying to find a suitable *mapping* or *association* from the input patterns to the desired output signals.

If the desired output signal is not known for each input pattern, *unsupervised learning* methods are needed. Unsupervised learning is then concerned with finding the categories to which each input pattern belongs. Unsupervised learning synonymous with *Kohonen's Self Organizing Feature Maps*, and *Carpenter/Grossberg Classifiers* are not covered in this paper but are described in [7, 28, 33].

A single processing element with several input signals and a single output signal has been examined, but generally problems solved using neural networks need many interconnected processing elements. There are many different ways of connecting processing elements; a particular configuration of processing elements and their interconnections is called the *architecture* of a neural network. The architecture of a neural network is important since each interconnection has a corresponding weight, and these weights are the components of each processing element's weight vector which represent the processing element's knowledge. Therefore, knowledge is a function of a neural network's architecture.

One type of architecture is the *single-layer feed-forward* neural network shown in figure 2.2. It is termed a feed-forward network since it contains no directed cycles, that is, the output signals cannot be directed back to become the new input signals, nor can adjacent processing elements be connected.

The input signals of a processing element can be viewed as the output signals of other processing elements. Therefore, a single-layer feed-forward neural network is sometimes called a two-layer neural network, consisting of two layers of processing elements, an input layer and an output layer. In this case, the input layer of a two-layer neural network can be thought of consisting of processing elements with a single input signal with a weight of 1. Input patterns are therefore passed unchanged through the input layer to the output layer. In this paper the term single-layer neural network will be used for a network with an input and an output layer.

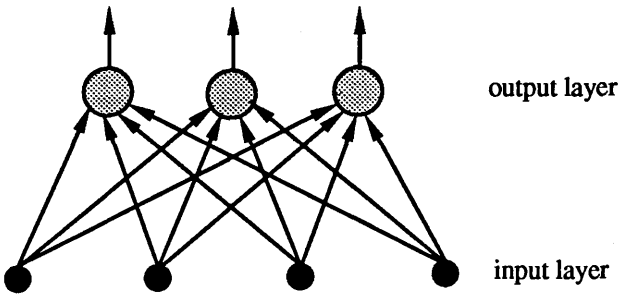

Figure 2.2 A single-layer feed-forward network.

Other network architectures that do contain directed arcs are called *feed-back* networks; networks where each processing element is connected to every other processing element including itself are termed *fully-interconnected* networks. Learning rules for these networks are not described in this paper; they can be found in [23, 33, 42]. The next section outlines supervised learning procedures for single-layer feed-forward networks.

Learning in Single-Layer Neural Networks

Neural networks were first modelled by Warren McCulloch and Walter Pitts [35], who showed how many simple interconnected neuron-like elements could be used as a computing device. Learning was later added to this model by Donald Hebb [15]. Hebb's basic definition states that if a processing element, u_j, receives an input from a processing element, u_i, and if both elements are highly active then the weight, w_{ji}, from u_i to u_j should be strengthened. Several learning rules based on Hebb's definition currently exist. These learning rules have been used in the area of pattern recognition [10] and adaptive signal processing [49] for over a decade.

Learning in a neural network is concerned with adapting the weight vectors of each processing element when the actual output signals are misclassified. Each component of the weight vector is modified by an amount Δw_{ji}. Shown below is the general form of a learning rule :

$$\mathbf{w}_{j_{new}} = \mathbf{w}_{j_{old}} + \Delta \mathbf{w}_j$$

A direct consequence of Hebb's definition is a simple Hebbian learning rule where each weight vector is modified by an amount which is equal to the product of the input signal and the actual output signal.

$$\Delta \mathbf{w}_j = o_j \mathbf{x}$$

This simple Hebbian learning rule is termed a *linear associator*, since the output signal of each processing element in the neural network is a function of the total input. This function, F, is termed an *activation function*, and is linear since it is the identity function (i.e. $o_j = I(t_j)$). Another simple Hebbian learning rule modifies each weight vector by an amount which is equal to the product of the input signal and the desired output signal.

$$\Delta \mathbf{w}_j = d_j \mathbf{x}$$

The problem with Hebbian learning is that if all the input vectors are not unit vectors, and are not orthogonal to one another the learning procedure cannot find an optimal set of weights. Input vectors can easily be normalized into unit vectors by dividing each component of the vector by the magnitude of the vector. However, input vectors not orthogonal to one another cause interference to the weight vector during learning. In this case a Hebbian rule will not find the correct association between the input and output vectors (see [86] for an analysis of Hebbian learning).

In Hebbian learning, each training pattern is presented to the neural network only once, since the correct association between the input patterns and the desired output signals will be found after one sweep through the training patterns. The limitations of these simple learning rules can be overcome by modifying these rules, and making them iterative learning procedures where the training patterns are presented to the neural network many times until some criteria is minimized. Perceptron learning [39,40] and the Pocket Algorithm [13,14] are iterative Hebbian learning algorithms that modify the weight vector of each processing element by the product of the input signal and the desired output signal.

An improvement of Hebbian learning is a method whereby an *error-term* or *error signal*, e_j, is used to modify the weights. There are two learning procedures which use this rule to train a single-layer neural network. The first procedure, developed by Frank Rosenblatt, is called the *Perceptron Convergence Algorithm* [62]. It modifies the weights of a processing element called a *perceptron*. A perceptron is a threshold logic element, which is a processing element with a non-linear activation function. This non-linear function is a *hard-limiter* (figure 2.3). The weight vector is modified as follows (η is a learning constant that sets the step size during learning; it is normally in the range [0,1]) :

$$\Delta \mathbf{w}_j = \eta e_j \mathbf{x}$$

where

$$e_j = d_j - o_j$$

$$o_j = F(t_j) = \begin{cases} +1 \text{ if } t_j > 0 \\ -1 \text{ if } t_j \leq 0 \end{cases}$$

The second procedure, the *Delta Rule* or *LMS Rule* (Least Mean Square Rule), developed by Bernard Widrow and Marcian Hoff [48] and sometimes called the Widrow-Hoff Rule, modifies the weights of a processing element called an *adaline*. An adaline, for adaptive linear element, is a processing element that is also a threshold logic element except that the analog total input is used to calculate the error as opposed to the binary output signal from the hard-limiter. This has the effect of the LMS Rule updating the weights even when the input vector is correctly classified. The weight vector is modified as follows :

$$\Delta \mathbf{w}_j = \eta e_j \mathbf{x}$$

where

$$e_j = d_j - (\sum_{i=1}^{n} w_{ji} x_i - \theta_j)$$

A neural network made up of a single-layer of perceptrons or adalines is called a *non-linear associator* since the output signal is a function of the total input, where the activation function is non-linear. The hard-limiter can be represented graphically, and is shown in figure 2.3.

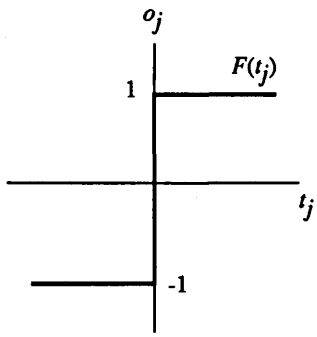

Figure 2.3 Hard Limiter.

Instead of using +1 and -1 as the limits of the hard-limiter, 0 and 1 can be used. If binary valued input vectors are used then they should be adjusted according to the limits of the hard-limiter used, that is use input vectors consisting of +1's and -1's, if using a hard-limiter with the limits +1 and -1. If the total input is zero then the activation function can output either +1 or -1, implementing two-valued logic. Alternatively, the activation function can output 0 if the total input is zero, implementing three-valued logic. This is shown below :

$$o_j = F(t_j) = \begin{cases} +1 \text{ if } t_j > 0 \\ 0 \text{ if } t_j = 0 \\ -1 \text{ if } t_j < 0 \end{cases}$$

Variants of the LMS Rule and the Perceptron Convergence Algorithm exist. Sometimes they are taken to be the same learning rule. The algorithm explained in this paper is the LMS Rule.

Widrow and Hoff showed that by using the Delta Rule the objective of training the neural network is to find a set of weights that minimizes the *total sum squared error* or *least mean squared error*, E, for each processing element, u_j, for all the input patterns, p. This total sum squared error is shown below (normally the total sum squared error is halved for mathematical convenience when formulating the Delta Rule as a gradient descent method, see [42]) :

$$E = \frac{1}{2} \sum_{p} \sum_{j=1}^{n} e_j^2$$

The Delta Rule is a *gradient descent* algorithm, since it minimizes the total sum squared error of the network using an ideal weight vector to achieve gradient descent. Mathematically, the total sum squared error is a function of the weight vector. This function is a quadratic, shown in figure 2.4 as a paraboloid for a weight vector with two components. It is a *hyperparaboloid* in n-space.

The paraboloid contains a bottom with a minimum value representing the least mean squared error which corresponds to the ideal weight vector. The function of the Delta Rule is to move the current weight vector to the point of the ideal weight vector. The Delta Rule achieves this in the most efficient way by moving the current weight vector along the negative gradient of the paraboloid which is the most direct path to the ideal weight vector. The negative gradient is found by differentiating the total sum squared error with respect to each the weight. This partial derivative is called a *delta* vector, and is equal to the $\Delta \mathbf{w}_j$ in the Delta Rule. It is computed for each processing element, u_j, and is parallel to the input vector.

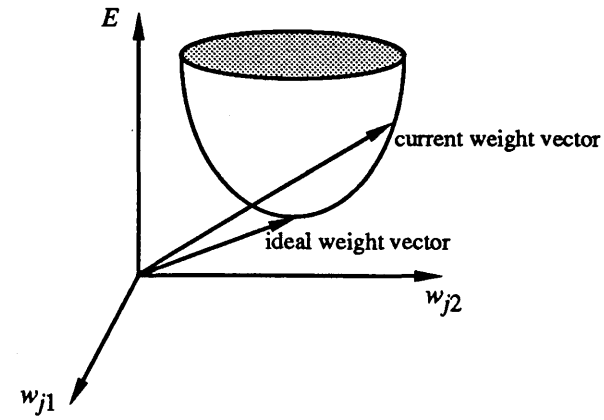

Figure 2.4 The minimum point of the paraboloid corresponds to the ideal weight vector.

Classification in Single-Layer Networks

The reason for having a quantized output signal, is that it makes a single processing element into a simple decision maker or *classifier*, which can be used to map certain functions. This dichotomised output signal then allows a single processing element to classify input patterns into one of two categories, +1 or -1. In the two dimensional case, that is a processing element, u_j, with two input signals, the input pattern space can be divided or separated into two classes by a single straight line, see figure 2.5.

This separating line forms two half-plane decision regions in the input pattern space. The equation for this straight line is :

$$g_j(\mathbf{x}) = w_{j0} + w_{j1}x_1 + w_{j2}x_2 = 0$$

where

$$\text{gradient} = -(w_{j1}/w_{j2})$$

$$\text{intercept} = -(w_{j0}/w_{j2})$$

In the n-dimensional case, that is a processing element with n input signals, the input pattern space is separated by a n-dimensional hyperplane.

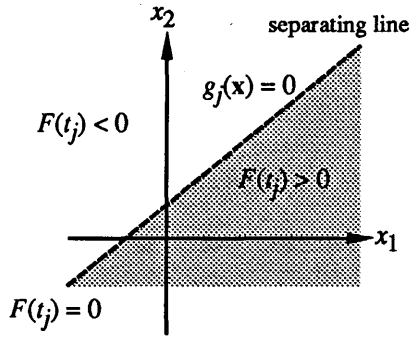

Figure 2.5 Input pattern space separated by a single line.

The logic OR-function is an example of a non-linear function which can be mapped by a single processing element with two input signals. This mapping consists of classifying input patterns into one of two classes, *true* or *false*, represented by +1 and -1. The four input patterns (1,1), (1,-1), (-1,-1), and (-1,1) can be depicted by co-ordinates in the input pattern space. Learning then corresponds to finding a set of weights, which represent a straight line in the input pattern space, that will give the correct separation of the input patterns. If an input pattern lies to the left of the straight line, then the output signal of the processing element will be negative, -1, and if the input pattern lies to the right of the line, the output signal will be positive, +1. The correct separation is achieved when the co-ordinates (1,-1), (1,1) and (-1,1) are to the right of straight line, and the co-ordinate (-1,-1) is to the left of the straight line. In this way we can see that a single processing element makes a decision or classification by forming a half-plane decision region in the input pattern space, this is shown in figure 2.6.

A single-layer feed-forward neural network can be built by connecting a number of processing elements adjacently in a single row. Training a single-layer of processing elements is fundamentally the same as training each processing element individually. Therefore, a single-layer neural network is analagous to many unconnected autonomous processing elements of which each can be used separately as a classifier. In this way, not only can simple classification problems be solved by a single-layer feed-forward network but it can be proved that the optimal set of weights can be found if the weights exist. This proof, called the *Perceptron Convergence Theorem*, is proved in [34]. Unfortunately, the single-layer neural network, no matter how it is interconnected, cannot always find a correct set of weights to separate the input patterns in the input pattern space.

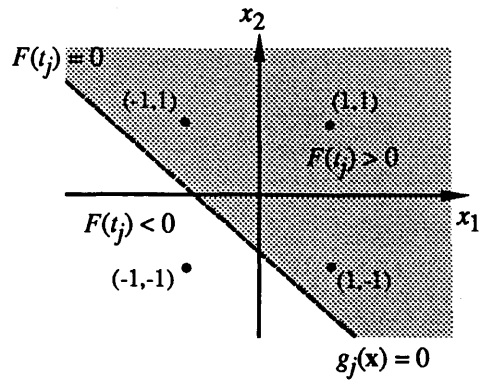

Figure 2.6 The required separation for the OR-function.

Non-linearly Separable Functions

A single processing element is shown to be capable of finding the correct mapping of a non-linear function. If the processing element has n input signals, and each input signal is binary valued, then there are $2^n = p$ input patterns. Now if the desired output signal is also binary valued then the processing element should be capable of learning 2^p different functions. Unfortunately, a single processing element can only learn a subset of these functions. The class of functions that cannot be solved by a single processing element, and in fact by a single-layer of processing elements since each processing element is autonomous, is the class of *non-linearly separable* functions.

One of the reasons for the extinction of neural network research in the 1960's was a book titled *Perceptrons* by Minsky and Papert [34], which exposed the flaws of the single-layer neural network using the example of the logic XOR (exclusive OR) function which is a non-linearly separable function. The XOR-function returns the same answer when totally opposite input signals are given to the function. When the two input signals are both true or both false, the XOR-function returns false. Using a processing element with two input signals, 0 or 1, to solve the XOR-function, the following inequalities are specified by the network :

$$\begin{aligned}
0w_{j1} + 0w_{j2} &< \theta_j \Rightarrow 0 < \theta_j \\
1w_{j1} + 0w_{j2} &> \theta_j \Rightarrow w_{j1} > \theta_j \\
0w_{j1} + 1w_{j2} &> \theta_j \Rightarrow w_{j1} > \theta_j \\
1w_{j1} + 1w_{j2} &< \theta_j \Rightarrow w_{j1} + w_{j2} < \theta_j
\end{aligned}$$

This set of inequalities cannot hold since it is impossible for w_{j1} and w_{j2} to be both greater than the threshold, θ_j, while the sum of w_{j1} and w_{j2} is less than θ_j. Graphically, a single line in the input pattern space, cannot make the required separation of the input patterns. It is impossible for a single separating line to form a decision region such that the co-ordinates (1,1) and (0,0) are grouped in one category, and the co-ordinates (0,1) and

(1,0) are grouped in the other category, since a single separating line can only form a half-plane decision region.

Non-linearly separable problems, however, can be solved by the combination of many separating hyperplanes which is achieved by combining the output signals of a single-layer neural network, to form the input signals of a second single-layer network, producing a two-layer neural network. If the input signals to a two-layer network is viewed as a layer of processing elements then the middle layer of elements is called an *intermediate* or *hidden* layer. There can be several hidden layers between the bottom or input layer, and the top or output layer, see figure 3.1. This paper will use the term two-layer neural network for a network with one hidden layer of processing elements.

The XOR-function can be mapped by a two-layer network with two intermediate units with separating hyperplanes $g_1(x)$ and $g_2(x)$ as in figure 2.7.

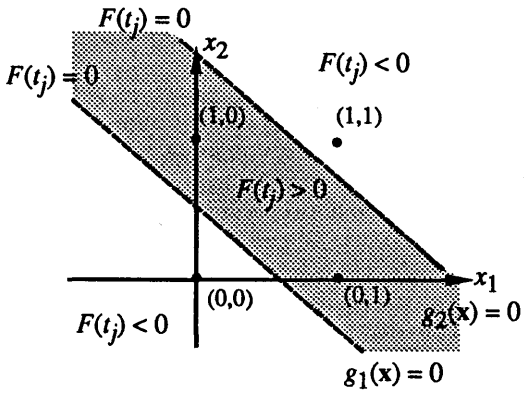

Figure 2.7 The required separation for the XOR-function.

In the 1960's it was clearly understood that many layers of processing elements could map non-linearly separable functions. However, at the time the learning algorithms, such as the Perceptron Convergence Algorithm and the Delta Rule, were limited to training single-layer neural networks (single-layer learning algorithms can be used to train multi-layer networks with intermediate units with fixed random weights, see [30,31]). This all changed twenty years later when a learning algorithm, the *Generalized Delta Rule* or *Back-Propagation*, was introduced by Rumelhart *et al.* [42, 44].

3. Multi-Layer Neural Networks

The fundamental issue concerning a multi-layer neural network is how to train the weights of the processing elements in the hidden layers. This section will outline learning in multi-layer systems using Back-Propagation [6,42,44] which evolved from the Delta Rule.

Training with Back-Propagation takes place in two phases : the *forward activation flow* and the *backward error flow*, see figure 3.1. During the forward activation flow, an input pattern is presented to the processing elements of the input layer. The input layer in turn presents the input pattern unchanged to the first hidden layer. The output signals of each processing element in the first hidden layer are calculated using the input pattern and the current weights. These output signals then become the input signals to the next hidden layer, or the output layer if the network has only one hidden layer. Once the output signals of the output layer have been found, error-terms, the difference between the desired and actual output signals, can be computed. These error-terms are then used in the backward error flow phase to modify the weights of the network.

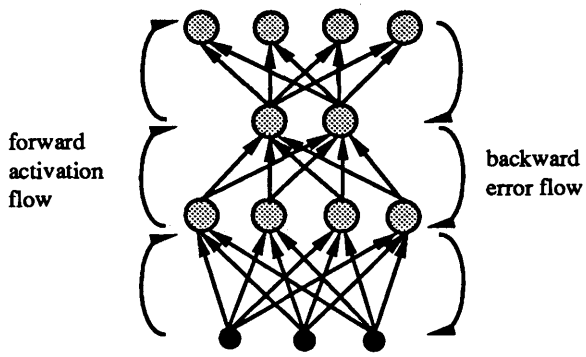

Figure 3.1 Back-Propagation is a two phase learning procedure.

The weights of the connections between the hidden layer and the output layer can be modified using the Delta Rule since the desired and actual output values of the output processing elements are known. However, the hidden units cannot use the Delta Rule since the desired output values for these processing elements are not known. This is where the Generalized Delta Rule is needed.

Back-Propagation is a generalization of the Delta Rule which is a gradient descent algorithm. To make use of gradient descent methods, Back-Propagation requires all the processing elements to use a *S-shaped, differentiable* activation function. The function most often used is the *sigmoid* activation function, shown in figure 3.2. To modify the weights of each layer in the neural network the same learning rule as in the Delta Rule is used, but the error-term, e_j, for each processing element is modified to account for some of the error or blame from the layer below. For the output layer, the error-term is the product of the difference between the desired and actual output signals, and the derivative of the sigmoid activation function, shown below :

$$e_j = (d_j - o_j)F'(t_j)$$

$$F(t_j) = \frac{1}{1 + \exp^{-t_j}}$$

$$F'(t_j) = o_j(1 - o_j)$$

The reason for propagating the error-terms back to the hidden layers is that the processing elements of the output layer can output the incorrect values either because the weights on their connections are incorrect, or because the processing elements of the hidden layers are providing them with incorrect input values. These errors from the

output layer must be propagated backward to the weights of the hidden layers. This is again achieved by using the same learning rule as in the Delta Rule except the error-term is the product of the derivative of the sigmoid activation function, and the total sum of the all the errors and the weights of the output layer before the weights have been updated; this is shown below :

$$e_j = F'(t_j)\sum_k e_k w_{kj}$$

If there is more than one hidden layer then the error-term for the mth hidden layer is the product of the derivative of the activation function, and the total sum of all the errors and the weights of the $(m+1)$th hidden layer, the layer above the mth hidden layer, before the weights have been updated. The purpose of the derivative in the error-term is to provide stability during learning, and to compensate for the blame from the layer below.

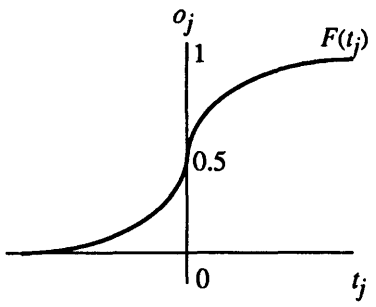

Figure 3.2 Sigmoid activation function.

Like the Delta Rule, Back-Propagation is a gradient descent algorithm that minimizes the total sum squared error, E, but the total sum squared error is not a quadratic function of the weights. The error surface is more complex containing not only one minimum value as in the paraboloid but several minima, see figure 3.3. Each minimum value can be either a *global* or a *local minimum*. Each global minimum corresponds to an ideal set of weight vectors and a minimized sum squared error for the network, where as each local minimum coincides with an unstable set of weight vectors. Back-Propagation can therefore find a solution which is one of the many local minima instead of a global minimum.

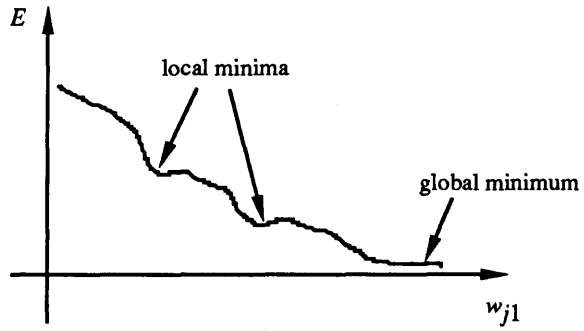

Figure 3.3 Error-surface of a multi-layer network in one-dimension.

A gradient descent algorithm tries to find the quickest and the most direct path to the solution. This path to the solution corresponds to the negative of the gradient of the sum squared error which is the most direct path. To dislodge a solution from a local minimum, a technique from *thermodynamics* called *annealing* can be applied. Annealing is the process of cooling a metal slowly to reach a stable state, instead of cooling the metal quickly reaching an unstable state or local minimum. *Simulated annealing* [18,19], an analagous process to annealing is used to move a solution out of a local minimum closer to the global minimum by adding momentum to the solution.

Momentum in the form of a momentum term is added to the Back-Propagation learning rule to allow the sum squared error to fall out of local minima, so that a global minimum can be reached. The learning rule is modified as follows (α is a momentum constant normally between [0,1]) :

$$\mathbf{w}_{j_{new}} = \mathbf{w}_{j_{old}} + \Delta\mathbf{w}_{j_{new}} + \alpha\Delta\mathbf{w}_{j_{old}}$$

This modification has the effect of moving the change of weights $\Delta w_{j_{new}}$ in the same direction as the previous change of weights $\Delta w_{j_{old}}$, therefore causing the network to find a global solution.

Back-Propagation is a complex learning rule with many design issues that need to be addressed before the network is trained. These include the number of hidden elements to be used, and the size of the learning constant, η. The use of too many intermediate elements will make the network lose its generalization capabilities; too few elements will increase the number of training cycles, and reduce the accuracy of the network. Training a multi-layer neural network using Back-Propagation can therefore become computationally expensive using a large number of training iterations; one of the reasons why parallel computers are needed. The learning constant can be used to speed up the training but too large a value will cause the network to oscillate, i.e. the network can be trapped in a local minimum; too small a value will slow the learning process therefore requiring a large number of iterations. These research issues are currently being investigated; they are discussed in [6,20,33,36].

Classification in Multi-Layer Networks

A single-layer of processing elements forms half-plane decision regions, which is not capable of solving the XOR-function since a single line cannot separate the input pattern space. However, using a two-layer network, convex bounded and unbounded decision regions can be formed in the input pattern space. A convex region is defined as a region such that if any two points lie on the boundary of the region then a straight line passing through these two points will always pass through points in the region. These convex regions are formed from the intersection of many half-plane regions formed from the first intermediate layer of processing elements. This is equivalent to combining all the processing elements in the first intermediate layer with the logical AND operation in the output layer. These convex regions have

at most the same number of sides as the number of processing elements in the first layer. Figure 3.4 shows bounded and unbounded convex decision regions for a two-layer network with five intermediate units.

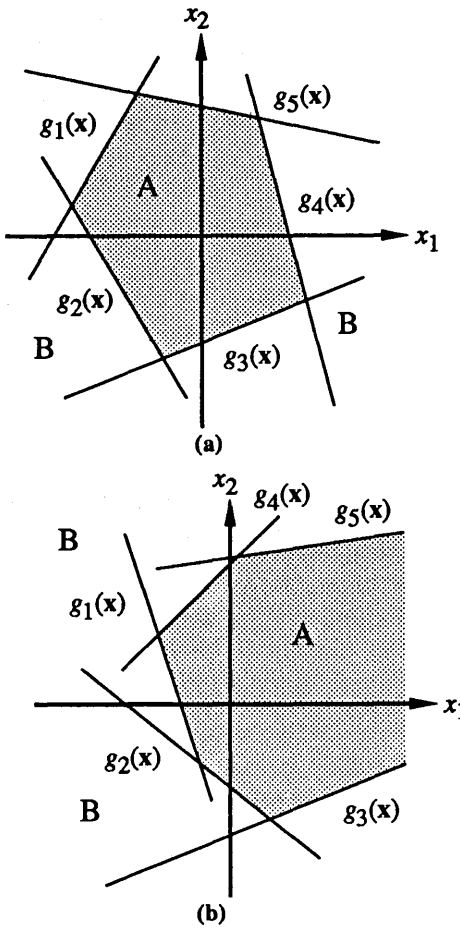

Figure 3.4 (a) Bounded convex decision region. (b) Unbounded convex decision region.

If the decision region to be formed in the input pattern space is non-convex or disconnected then an extra layer of processing elements is needed, one processing element for each disconnected region. Therefore, a three-layer network, i.e. a network with two hidden layers, can generate arbitrarily complex decision regions [33]. The second intermediate layer of a three-layer network is similar to the output layer of a two-layer network. Each processing element in the second intermediate layer corresponds to a disconnected region. This is equivalent to combining all the processing elements in the first intermediate layer with the logical AND operation in the second layer. The correct decision region is then formed from the union of the disconnected regions. This is equivalent to combining all the processing elements in the second intermediate layer with the logical OR operation in the output layer. Each output processing element normally corresponds to a category in the input pattern space, so if the input patterns must be classified into three categories, three output elements are needed. In the case of two categories, one or two processing elements in the output layer may be used. Figure 3.4 shows non-convex and a disconnected decision regions for a three-layer network with eight units in the first intermediate layer, and two units in the second intermediate layer.

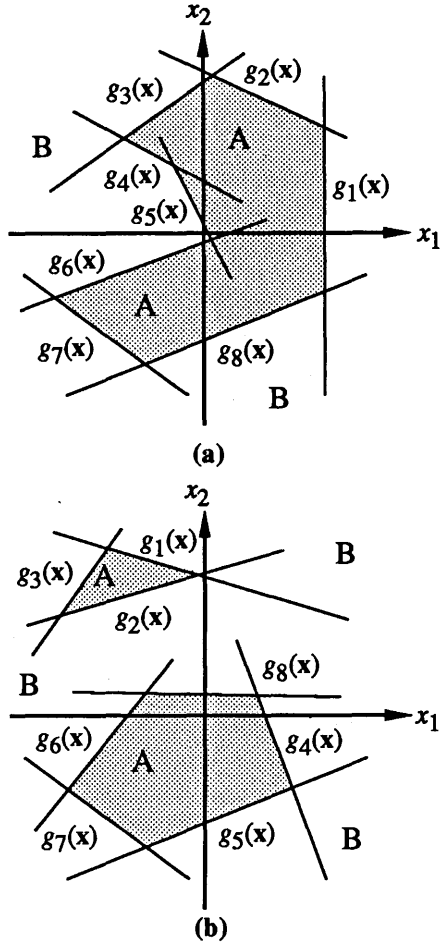

Figure 3.5 (a) Non-convex decision region. (b) Disconnected decision region.

4. Neural Networks and Classification

Classification is the task of placing specific input data into general categories. Knowledge-based systems, pattern recognition systems, and artificial neural network systems are classification systems [8]. It is the objective of this section to review how both mapping neural networks, and various other neural network architectures can be used in knowledge-based and pattern recognition systems.

It can be shown that several tasks solved by the knowledge-based system paradigm, such as diagnosis, and analysis, are classificatory in nature. These tasks can therefore be solved by neural network systems. Traditional knowledge-based or expert systems have two main components : a *knowledge base* and an *inference engine*. The knowledge base contains the knowledge of a human expert. This knowledge is normally represented in the form of *situation/action* statements or *if-then* rules. The inference engine uses the knowledge in the

knowledge base to conduct a dialogue with the user of the system. During this consultation the inference engine makes decisions and provides advice to the user based on the user's input and the information in the knowledge base. A key feature of expert systems is that the inference engine is capable of explicitly providing the path of reasoning it uses to make decisions.

Acquiring specialist knowledge from a human expert has been the primary bottleneck in expert system development [1,21,37]. This *knowledge acquisition* process is time consuming, requiring the expertise of a *knowledge engineer* or expert system builder. The knowledge engineer usually uses ad hoc manual knowledge acquisition methods to elicit the knowledge which then needs to be formulated into rules. This is not always possible since the expert makes use of heuristics or rules of thumb during his reasoning process which cannot be explicitly expressed as symbolic rules. Many techniques have been used to acquire implicit heuristics, such as techniques from cognitive science [38]. These techniques have been automated to form knowledge acquisition tools [2,11,27,26]. Such a tool typically interacts with the domain expert, organizes the knowledge it acquires, and generates an expert system. An alternative approach is the use of a mapping neural network to acquire knowledge.

It is possible to replace the knowledge base of an expert system with a mapping neural network; these systems are called *connectionist expert systems* [3,14,32]. These networks can be trained by samples or examples from the expert's specialist area, using any of the learning algorithms described. The connectionist expert system exhibits favourable features from both the neural network and expert system paradigms. The knowledge base is capable of learning or adapting to new training information, and the inference and explanation capabilities of the expert system remain. Also, traditional expert systems infer knowledge serially from the knowledge base, although, attempts have been made to develop parallel inference engines for inferring knowledge from rules. Neural networks on the other hand are inherently parallel. The output signals of processing elements can be computed simultaneously.

Much research in the area of the neural network approach to knowledge-based system development is currently being done. Gallant [12,13,14] has shown how medical diagnosis problems can be solved using a connectionist expert system. He introduces MACIE (Matrix Controlled Inference Engine) which infers knowledge from a knowledge base consisting of a neural network representing medical domain knowledge. Lau Hing [29,32] has investigated using Back-Propagation trained multi-layer feed-forward networks with MACIE. Hinton and Touretsky [17] discuss how a neural network can be used to implement rules in a production system. Samad [45] describes how a connectionist architecture, dubbed RUBICON, can be used to implement rule-based systems. Using the connectionist approach in knowledge-based systems up to now has been faulted for their lack of an explicit explanation facility. However, models capable of generating explicit explanations have started to appear [14]. Also Lau Hing [29] has investigated explicit explanations in multi-layer systems.

The primary role of connectionist models up to now have been in areas that use traditional pattern and image recognition techniques where explanations are not essential. The perceptron model of Rosenblatt [39,40] is essentially a pattern recognition model. Several existing classification and clustering methods can be replaced by equivalent neural network models. In supervised learning the Hamming Net (a modified version of the Hopfield net, see [22]) can be used as an Optimum Classifier, perceptron learning satisfies the requirements of a Gaussian Classifier, the multi-layer perceptron defines complex decision regions similar to the k-Nearest Neighbour Mixture; in unsupervised learning the Carpenter/Grossberg Classifier forms clusters similar to the Leader Clustering Algorithm, and Kohonen's Self-Organizing Feature Maps also forms clusters, similar to the k-Means Clustering Algorithm [10,33]. These neural network models do not behave the same as the corresponding classifier but do perform the same functions. Any decision region required by any classification algorithm can be generated by a three layer feed-forward neural network [33].

Even though connectionist research started in the 1950's it is still currently in its infancy. Its future as a genuine tool for building intelligent classification systems is in the hands of current researchers and their applications. To date there have been many notable applications that have successfully used neural networks. NETtalk [46] converts strings of letters into strings of phonemes. This was accomplished with twelve hours of training, achieving a ninety two percent success rate. In constrast DECtalk, a rule-based system, marketed by Digital Equipment Corporation, achieved the same level of performance after several years of devising rules for every situation. To date NETtalk is one of the most successful implementations of a neural network but other well known problems have also been solved using this paradigm : the travelling salesman problem [23], the inverse kinematics problem from robotics [25], and word perception [41].

The problems solved alone do not make neurocomputers favourable classification and decision making tools, the greatest potential of these systems remain in high speed parallel processing. The architecture of neural networks are inherently parallel which make them parallel machines. Traditional serial classifiers can be replaced by an equivalent neural network, therefore speeding up the classification of data. Also, a neural network's distributed representation makes it more damage resistant, and its ability to learn to generalize a mapping from the input patterns to the desired output signals makes it capable of correctly classifying data that is not part of the training examples.

5. Summary and Conclusion

This paper highlights neural networks, specifically mapping neural networks, as a basis for classification systems. It is not emphasizing that every classificatory task must now be solved using the neural network paradigm. It is, however, providing an alternative approach to solving complex classification problems.

Classification is a natural method of information processing which the human cognitive system extensively uses; humans are natural pattern recognizers. The similarities in characteristics of humans and neurocomputers brings artificial intelligence a step closer to mimicking natural intelligence. Neurocomputers are parallel adaptable computers, the very ingredients necessary to model human intelligence.

References

[1] Berry D C, [1987], The problem of implicit knowledge. *Expert Systems - An International Journal of Knowledge Engineering*, August, **4**(3), 144-151.

[2] Boose J, [1984], Personal construct theory and the transfer of human expertise. *Proceedings of 4th AAAI Conference*, 27-33.

[3] Bounds D G, Lloyd P J, Mathew B and Waddell G, [1988], A Multi Layer Perceptron Network for the Diagnosis of Low Back Pain. *Proceedings of the 2nd ICNN*, 481-489.

[4] Caudill M, [1987], Neural Networks Primer Part I, *AI Expert*, December, 46-51.

[5] Caudill M, [1988], Neural Networks Primer Part II, *AI Expert*, February, 55-61.

[6] Caudill M, [1988], Neural Networks Primer Part III, *AI Expert*, June, 53-59.

[7] Caudill M, [1988d], Neural Networks Primer Part IV, *AI Expert*, August, 61-67.

[8] Chandrasekaran B and Goel A, [1988], From Numbers to Symbols to Knowledge Structures : Artificial Intelligence Perspectives on the Classification Task. *IEEE Transactions on Systems, Man, and Cybernetics*, May/June, **18**(3).

[9] DARPA, [1988], Classification and Clustering Models, In *DARPA Neural Network Study*, AFCEA. 87-95.

[10] Duda R O and Hart P E, [1973], *Pattern Classification and Scene Analysis*, Wiley, New York.

[11] Eshelman L and McDermott J, [1986], MOLE: a knowledge acquisition tool that uses its head. *Proceedings of 5th AAAI Conference*, 950-955.

[12] Gallant S I, [1985], Automatic generation of expert systems from examples, *Proceedings of the 2nd International Conference on AI Applications*. IEEE Press, New York. 313-319.

[13] Gallant S I, [1986], Optimal Linear Discriminants, *Proceedings of the 8th International Conference on Pattern Recognition*. IEEE Press, New York. 849-852.

[14] Gallant S I, [1988], Connectionist Expert Systems. *Communications of the ACM*, February, **31**(2), 152-169.

[15] Hebb D O, [1949], *The Organization of Behaviour*. Wiley, New York.

[16] Hecht-Nielsen R, [1989], Theory of the Back-Propagation Neural Network, *Proceeding of the IJCNN*, IEEE Press, 593-605.

[17] Hinton G E and Touretsky D S, [1985], Symbols Among the Neurons : Details of a Connectionist Inference Architecture, *Proceedings of 9th IJCAI*, 238-243.

[18] Hinton G E, Ackley D H and Sejnowski T, [1985], A Learning Algorithm for Boltzman Machines, *Cognitive Science*, **9**, 147-169.

[19] Hinton G E and Sejnowski T, [1986], Learning and Relearning in Boltzman Machines. In *Parallel Distributed Processing : Explorations in the Microstructures of Cognition, Volume 1, Foundations*. Edited by Rumelhart, D E, McClelland, J L and the PDP Research Group, MIT Press, Cambridge, Massachusetts.

[20] Hinton G E, [1987], Connectionist Learning Procedures. *Carnegie-Mellon University, Department of Computer Science, Technical Report CMU-CS-87-115 version 2*.

[21] Hoffman R R, [1987], The Problem of Extracting the Knowledge of Experts from the Perspective of Experimental Psychology, *AI Magazine*, Summer, 53-67.

[22] Hopfield J J, [1982], Neural Networks and Physical Systems with Emergent Collective Computational Abilities, *Proceedings of the National Academy of Sciences*, **79**, 2554-2558.

[23] Hopfield J J and Tank D W, [1985], "Neural" Computation of Decisions in Optimization Problems, *Biological Cybernetics*, **52**, 141-152.

[24] Jordan M I, [1986], An Introduction to Linear Algebra in Parallel Distributed Processing, In *Parallel Distributed Processing : Explorations in the Microstructures of Cognition, Volume 1, Foundations*, Edited by Rumelhart, D E, McClelland, J L and the PDP Research Group. MIT Press. Cambridge, Massachusetts.

[25] Josin G, [1988], Integrating Neural Networks with Robots, *AI Expert*, August, **3**(8), 50-58.

[26] Kahn G, Nowlan S and McDermott J, [1985], MORE: an intelligent knowledge acquisition tool. *Proceedings of 9th IJCAI Conference*, 581-584.

[27] Klinker G, Bentolila J, Genetet S, Grimes M and McDermott J, [1987], KNACK - Report-Driven Knowledge Acquisition, *Proceedings of 6th AAAI Conference*.

[28] Kohonen T, [1982], Self-organized Formation of Topologically Correct Feature Maps, *Biological Cybernetics*, **43**, 59-69.

[29] Lau Hing R, [1990], *A Connectionist Expert System using Back-Propagation Learning*, Msc Research Report, Department of Computer Science, University of the Witwatersrand.

[30] Lau Hing R, [1990], Monte-Carlo Learning : the LMS Rule and Random Weights can Form Non-Convex Decision Surfaces and Generalize, *Department of Computer Science, University of the Witwatersrand, Technical Report 1990-02*, January.

[31] Lau Hing R, [1990], Monte-Carlo Learning : the LMS Rule and Random Weights can Form Non-Convex Decision Surfaces and Generalize, *Proceedings of the INNC*, Palais Des Congres, Paris, France, July.

[32] Lau Hing R, [1990], A Connectionist Expert System using Back-Propagation Learning, *Proceedings of the 2nd SA Conference on Expert Systems - Expert 90*, CSIR Pretoria, South Africa, May.

[33] Lippmann R P, [1987], An Introduction to Computing with Neural Nets. *IEEE ASSP Magazine*, April, **3**(4), 4-22.

[34] Minsky M L and Papert S, [1988], *Perceptrons : An Introduction to Computational Geometry, Expanded Edition*, MIT Press, Cambridge, Massachusetts.
[35] McCulloch W S and Pitts W, [1988], A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, **5**, 115-133.
[36] Plaut D C, Nowlan S J and Hinton G E, [1986], Experiment on Learning Back-Propagation. *Carnegie-Mellon University, Department of Computer Science, Technical Report CMU-CS-86-126.*
[37] Prerau D S, [1987], Knowledge Acquisition in the Development of a Large Expert System, *AI Magazine*, Summer,
[38] Olson J R and Reuter H H, [1987], Extracting expertise from experts : Methods for knowledge acquisition, *Expert Systems - An International Journal of Knowledge Engineering*, August , **4**(3), 152-168.
[39] Rosenblatt F, [1958], The Perceptron : A Probabilistic Model for Information Storage and Organization in the Brain, *Psychological Review*, **65**, 386-408.
[40] Rosenblatt F, [1962], *Principles of Neurodynamics*, New York, Spartan.
[41] Rumelhart D E and McClelland J L, [1981], An interactive activation model of context effects in letter perception : part 1, An account of the basic findings, *Psychological Review*, **88**,375-407.
[42] Rumelhart D E, McClelland J L and the PDP Research Group, [1986], *Parallel Distributed Processing : Explorations in the Microstructures of Cognition, Volume 1, Foundations.* MIT Press. Cambridge, Massachusetts.
[43] Rumelhart D E, McClelland J L and the PDP Research Group, [1986], *Parallel Distributed Processing : Explorations in the Microstructures of Cognition. Volume 2, Psychological & Biological Models*, MIT Press. Cambridge, Massachusetts.
[44] Rumelhart D E, Hinton G E and Williams R J, [1986], Learning representations by back-propagation errors, *Nature*, **323**, 533-536.
[45] Samad T, [1987], Towards Connectionist Rule-Based Systems, *Proceedings of the 1st ICNN*, 525-532.
[46] Sejnowski T J and Rosenberg C R, [1986], NETtalk : a parallel network that learns to read aloud, *The John Hopkins University Electrical Engineering and Computer Science Technical Report JHU/EECS-86/01.*
[47] Trelease R B [1988], Connectionism, Cybernetics, and the Cerebellum, *AI Expert*, August, 30-36.
[48] Widrow B and Hoff M E [1960], Adaptive Switching Circuits, *IRE WESCON Convention Record.* IRE, New York, 96-104.
[49] Widrow B and Winter R, [1988], Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition, *IEEE Computer*, March, **23**(3), 25-39.
[50] Widrow B and Nguyen D, [1989], The Truck Backer-Upper : An Example of Self-Learning in Neural Networks, *Proceedings of the IJCNN*, 357-363.

Acknowledgements : Many thanks to Anthony Low who in our many discussions crystalised my understanding of neural networks. Also thanks to Philip Machanick and Sheila Rock for helpful comments on an earlier draft.