A CONSTRUCTIVE LEARNING ALGORITHM BASED ON BACK-PROPAGATION

ANDREW DAVID LOWTON

Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

June 1995

The University of Aston in Birmingham

A Constructive Learning Algorithm Based on Back-Propagation

Andrew David Lowton

Doctor of Philosophy

1995

Summary

There has been a resurgence of interest in the neural networks field in recent years, provoked in part by the discovery of the properties of multi-layer networks. This interest has in turn raised questions about the possibility of making neural network behaviour more adaptive by automating some of the processes involved. Prior to these particular questions, the process of determining the parameters and network architecture required to solve a given problem had been a time consuming activity. A number of researchers have attempted to address these issues by automating these processes, concentrating in particular on the dynamic selection of an appropriate network architecture.

The work presented here specifically explores the area of automatic architecture selection; it focuses upon the design and implementation of a dynamic algorithm based on the Back-Propagation learning algorithm. The algorithm constructs a single hidden layer as the learning process proceeds using individual pattern error as the basis of unit insertion. This algorithm is applied to several problems of differing type and complexity and is found to produce near minimal architectures that are shown to have a high level of generalisation ability.

Keywords : Neural Networks, Dynamic Learning Algorithm, Generalisation.

Acknowledgements

Many thanks go to the following people:

Dr. Alan Harget for his help and support throughout my research.

Tina for supporting me and keeping me going when I wanted to give up.

All my friends and colleagues in the Computer Science Department at Aston.

Dave the Doc for an alternative and insightful view.

Th_AnX to Dave Hartley for the use of the rocket.

Judy for saving me in a crisis.

The 'L' block coffee club crew, especially Rose for her eternal optimism.

Mr. Wise Owl for coming through unscathed.

And finally to SERC for providing the necessary funding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the early 1970's, research into artificial neural networks was severely curtailed due to Minsky and Papert's criticism of the single layer perceptron which formed the basis of all neural network studies at that time (Minsky & Papert, 1969). Minsky and Papert showed that such networks were incapable of solving anything other than the most basic of problems. In recent years however, there has been a resurgence of interest in neural networks which utilise one or more hidden layers of neurons, for which learning algorithms are now available. The additional power provided by these multi-layer perceptron networks has been demonstrated by their successful application to a wide range of real-world problems.

Whilst much attention has been given to the theoretical aspects of artificial neural networks, few guide-lines exist for determining the optimum hidden layer architecture for a given problem. The optimum architecture is normally determined by an empirical approach in which many architectures are tested and their performance determined; this is a time consuming and computationally expensive process. Attempts have been made to automatically determine optimum architectures using several different approaches, the most common approach being the use of dynamic algorithms which attempt to determine the optimum architecture during the learning process. Other approaches have been taken to determine optimum architectures, including genetic algorithms which are another recently resurrected research topic. All these methods share the same basic appeal in that they remove the need for guess-work in the selection of a hidden layer architecture and additionally conform to the inherent adaptive nature of neural networks.

Dynamic algorithms can broadly be categorised into constructive, destructive and hybrid methods of architecture selection. Constructive algorithms build a hidden layer structure from an initially sparse network as learning proceeds. Destructive algorithms (commonly referred to as pruning algorithms) remove components from an initially overlarge network. Hybrid algorithms initially follow a constructive algorithm, upon the completion of which a destructive algorithm prunes unnecessary elements to leave a minimal network structure.

The stated aim of all these general approaches is usually to generate a minimal or near minimal hidden layer structure, since it is widely accepted that such an architecture produces optimal generalisation properties. That is to say, the network not only correctly classifies patterns drawn from the training set, but also other patterns that have not previously been presented to the network during the learning stage. This is partly due to the fact that a minimal network does not possess the capacity to simply 'memorise' the training set, and hence must extract the underlying features that characterise the data.

The aim of the work presented in this thesis is to develop a dynamic method of architecture selection based on the back-propagation learning algorithm. The selection process should be computationally efficient and capable of finding (near) minimal architectures for the neural network so that a high level of generalisation is attained. In this study we have adopted a constructive algorithm approach based on the errors of the individual members of the training set. The basic premise of the method is that the interaction between the members of the training set may indicate whether the addition of further hidden units is necessary for performance enhancement.

The results of applying the algorithm to several problems of different types and sizes are presented. Initially the algorithm was applied to small binary problems for which extensive neural network studies had previously been conducted. Once this analysis was completed, the algorithm was successfully applied to 'real-world' problems.

The thesis is organised as follows:

Chapter 2 gives a general overview of artificial neural networks, including a brief history and a description of some of the commonly used learning algorithms. The back-propagation algorithm is described in detail, together with some of its successful applications.

In chapter 3 some of the current major issues are dealt with, including neural network capabilities, speed of convergence and generalisation. The chapter goes on to discuss dynamic configuration and describes the major constructive, destructive and hybrid methods that have been developed.

Chapter 4 describes the implementation and validation of a software simulation of the back-propagation algorithm, together with the results of its application to three binary test problems. Chapter 4 also assesses the merits and problems associated with each of the broad categories of dynamic algorithms, concluding that a constructive approach should be followed. There then follows a description of how the errors of individual patterns in the training set may be used as the basis of a constructive algorithm. Some preliminary results in support of this premise are presented.

Chapter 5 describes the modifications made to the back-propagation simulator necessary to implement a constructive algorithm. The constructive algorithm is developed and applied to the test problems, with modifications made to enhance performance. The chapter concludes with a description of the results of applying the algorithm to 'real-world' problems.

A summary of the work completed and a discussion of the major conclusions is given in chapter 6.

Chapter 2

Neural Networks

2.1 General Overview

Researchers in the area of neural networks are attempting to model the workings of the human brain, which is complicated, of considerable power and as yet little understood. The human brain is extremely good at processing large amounts of complex information involved in tasks such as vision, natural language processing and speech recognition. Even using the most sophisticated AI techniques such processing has proven incredibly difficult to emulate, and has, as Kohonen (1988a) states, 'existed so far only in science fiction'.

Feldman (1985) argues that modern AI is not even close to simulating the natural intelligence of a small child or animal, even though a modern computer has a computing speed that is of the order of a million times greater than that of the human brain. The brain can perform a pattern recognition task in a few hundred milliseconds, and with a basic computational speed of around one millisecond this implies that the task is performed in approximately one hundred steps, whereas a conventional computer program would require millions of steps to perform the same task (Feldman & Ballard, 1982). This difference is largely due to the fact that digital computers and the human brain function in fundamentally different ways. Most digital computers work in a sequential manner following a set of instructions, whereas the brain is inherently parallel and is generally not thought to follow fixed sets of instructions in this way. The human brain is made up of approximately 10^{10} computational elements called neurons which are each connected to about 10,000 other neurons (Gevins & Morgan, 1988). A neuron is made up of a cell body, together with dendrites that receive input and an axon that carries the neuron's output signal to the dendrites of other neurons (Reece & Treleavan, 1988); this is shown in figure 2.1.

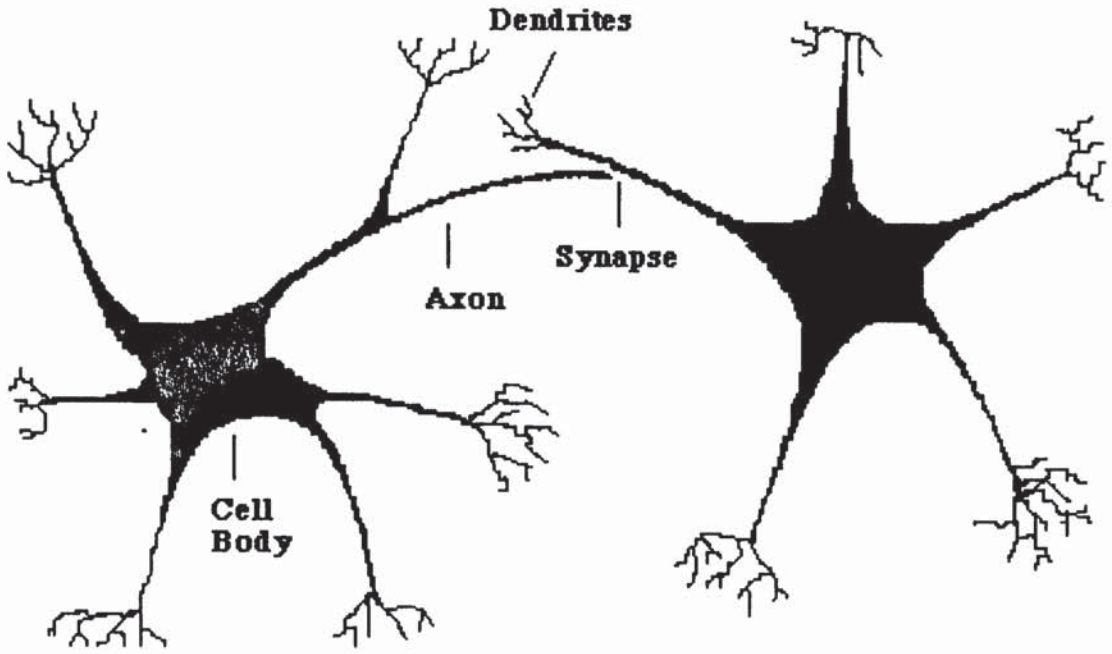


Figure 2.1 - A Biological Neuron

The signals are generated by a change in the potential of the membrane that covers the entire neuron, and are called action potentials or spikes (Braitenberg, 1973). Each spike is collected at the junction of the axon and dendrite, called the synapse, and adjusted in size by the synaptic strength; all the weighted signals for the particular neuron are added, and if the sum exceeds a certain threshold then the neuron fires, that is, it sends out a spike along its axon (Reece & Treleavan, 1988). Eliot (1988) notes that this view of a neuron as a simple processing element is a majority view, but it is argued by some that a single neuron could be similar to a VLSI chip in computational complexity. Most artificial neural networks simulate a very simple type of neuron in order to reduce this complexity.

2.1.1 The Neural Network Model

The basic artificial neuron, as shown in figure 2.2, is commonly referred to as a unit and consists of a set of inputs $y_1, y_2 \ldots y_n$, together with an associated set of weights

ω_1, ω_2 .... ω_n which modify the strengths of the input signals and act in an analogous way to the brain's synaptic strengths.

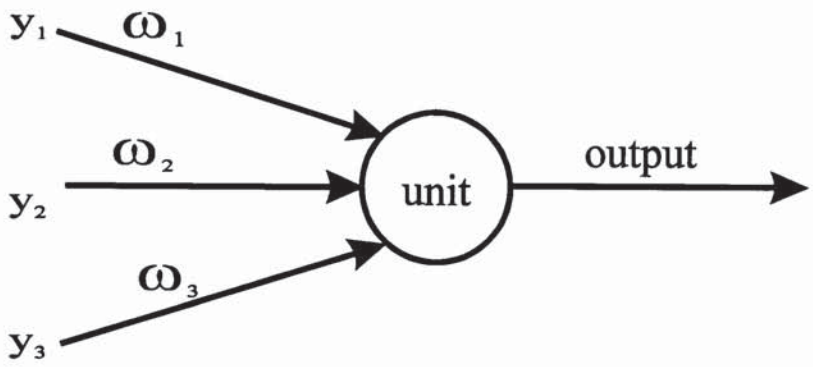


Figure 2.2 - An Artificial Neuron

The unit processes the weighted inputs (commonly a simple summation) to obtain the unit's activation value (or state), to which a function is then applied to obtain the unit's output value (Ellison, 1988). The units are connected together to form a network, and it is the manner in which they are connected that constitutes one of the major differences between neural network architectures. Most models divide the units into layers: figure 2.3 shows a simple network that consists of two layers.

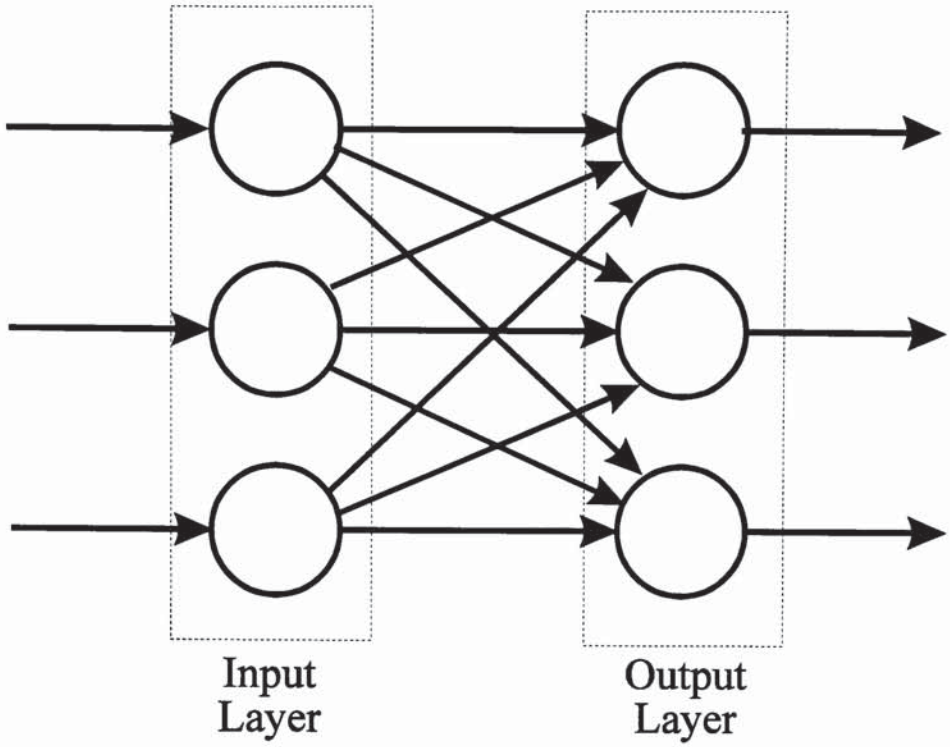


Figure 2.3 - A Simple Two layer Network

Each of the units in the input layer is connected to each of the units in the output layer, but the connections are unidirectional and neither layer is internally connected. It is possible to have networks that possess a fully connected architecture, that is to say, all units are connected to all other units, but this is not common and may only be used in conjunction with a small subset of learning algorithms (Peterson & Anderson, 1987).

Unlike a conventional computer, a neural network is not programmed; it learns by being presented with examples. This learning is achieved by adjusting the connection weights according to some mathematical rule, or possibly by changing the physical structure of the network by removing or creating new connections (Fahlman & Hinton, 1987). The brain seems to perform a similar operation by chemically altering its synaptic strengths (Braitenberg, 1973). As Wasserman & Schwartz (1988) state, most of these learning rules are variations of Hebb's law which was put forward by D.O Hebb in 1949. Hebb (1949) argued that if two adjacent neurons are active at the same time then the brain reinforces the excitatory link between them, causing an association to build up between the two neurons.

Mathematically the change in the connection weight between two neurons j and i, denoted by $\Delta\omega_{i,j}$, can be represented as follows;

$$\Delta\omega_{i,j} = k * (O_i * O_j) \qquad 2.1$$

where O_i and O_j are the outputs from unit i and unit j respectively, and k is a constant, usually referred to as the learning rate (Wasserman & Schwartz, 1988). If the adjacent neurons are of opposite sign, then the equation has the effect of making the connection between them more inhibitive. Zeidenberg (1987) maintains that this learning rule is a formalisation of associationist psychology which states that in the human brain, associations are built up between things that occur together.

Generally, the learning algorithms used to train neural networks are variations on a simple iterative process. A subset of the units in the network is clamped, that is to say their states are set to predetermined values. Subsequently, the remaining units and the weights in the network are allowed to change. This process is iterated over a set of examples until the network has learned to recognise the examples to a specified degree of accuracy. The subset of units that are clamped varies between different learning algorithms; it could comprise only the input units (collectively called an input vector) or an input vector together with an output vector.

There are many different network architectures and learning algorithms (some of which will be discussed later in this chapter), but all neural networks share interesting and useful properties. For example, in a conventional computer program the input generally must be of a precise type and must be accurate, whereas the input to a neural network may be corrupted by noise and yet the network still performs satisfactorily (Clark, 1987). Wasserman & Schwartz (1988) describe a system that was successfully trained to recognise printed characters, despite the fact that the input to the network was corrupted by 40% noise. Another important benefit is that neural networks have a high resilience to network

damage. Bedworth & Lowe (1988) reported that a neural network trained using the back-propagation learning algorithm showed a remarkable robustness to different forms of damage. This is due to the fact that neural networks use what has been called distributed representation (Rumelhart, Hinton & McClelland, 1986). In a conventional AI program, local representation is used, which means that a particular concept is represented by a particular node. If this node is damaged then the entire concept (and maybe adjacent concepts) is lost. In a neural network a concept is represented by many units and the particular pattern of connectivity between them. The analogy used by Fahlman & Hinton (1987) is that of a hologram where each part of the image is constructed from all parts of the film. Hence if part of the film is damaged, the hologram may be slightly degraded but no part of the image information is lost.

It is properties like these, together with new learning algorithms, analogue VLSI techniques and the greater availability of parallel processing systems that have generated considerable interest in neural networks (Lippman, 1987).

2.1.2 The History Of Neural Networks

The history of neural networks is a long and complex one. Although Greek philosophers put forward explanations of the workings of the brain and its thought processes around 400 BC (Kohonen, 1988), probably the earliest significant work in relation to current developments was due to Von Neumann, who showed that networks of this kind are equivalent to Turing machines in power and can be used to construct computer systems (Aleksander, 1988). Following this, McCulloch & Pitts (1943) hypothesised that a network, in which neurons are treated as Boolean devices, can compute logical functions. At this time however, it was not known how networks of neurons could learn, until Hebb, as previously described, put forward his theory of changing synaptic strengths. Using Hebb's ideas, Marvin Minsky and Dean Edmonds developed a machine in 1951 using a

randomly wired network of neuron like elements that exhibited learning (Rumelhart & Zipser, 1985). Other workers also developed models around this time, for example Farley & Clarke's 1954 model (Farley & Clarke, 1954), but the most famous and controversial model was the perceptron invented by Rosenblatt (1962).

The perceptron consisted of a set of binary input units which Rosenblatt referred to as *the retina* in relation to vision. These input units were connected in sets to another layer of units called the predicate or association cells, which performed a linear function on their input from the retinal units, and passed the result through a connection weight to the decision unit (which was a single unit output layer). All the inputs to the decision unit were summed and passed through a linear threshold function to give a binary result (Rosenblatt, 1962). A binary linear threshold function is a mathematical function that returns a 1 if the input to the function is greater than a given threshold value, and a value of 0 otherwise. Thus the perceptron was capable of classifying the input vectors into two distinct classes. Rosenblatt extended the original perceptron to have n decision units, implying that it had the capability to classify the input vectors into 2^n discrete sets (Rumelhart & Zipser, 1985). The perceptron was trained by applying an input vector to the retinal units, noting the output vector from the decision units and comparing this to the desired output vector. Depending on the nature of the error generated for each decision unit, the weights on the connections to that unit were incremented or decremented by a fixed amount, with no change if the output was correct (Rosenblatt, 1962).

Work continued on perceptron type networks throughout the sixties, a notable example being the Widrow Hoff least mean square learning algorithm which adjusts the weights in such a way as to minimise the mean square error between the actual and desired output (Widrow & Hoff, 1960). This was typical of the work in this period, which consisted of variations upon Rosenblatt's perceptron which, rather confusingly, is referred to as a two layer network. Figure 2.4 shows a two layer perceptron based network.

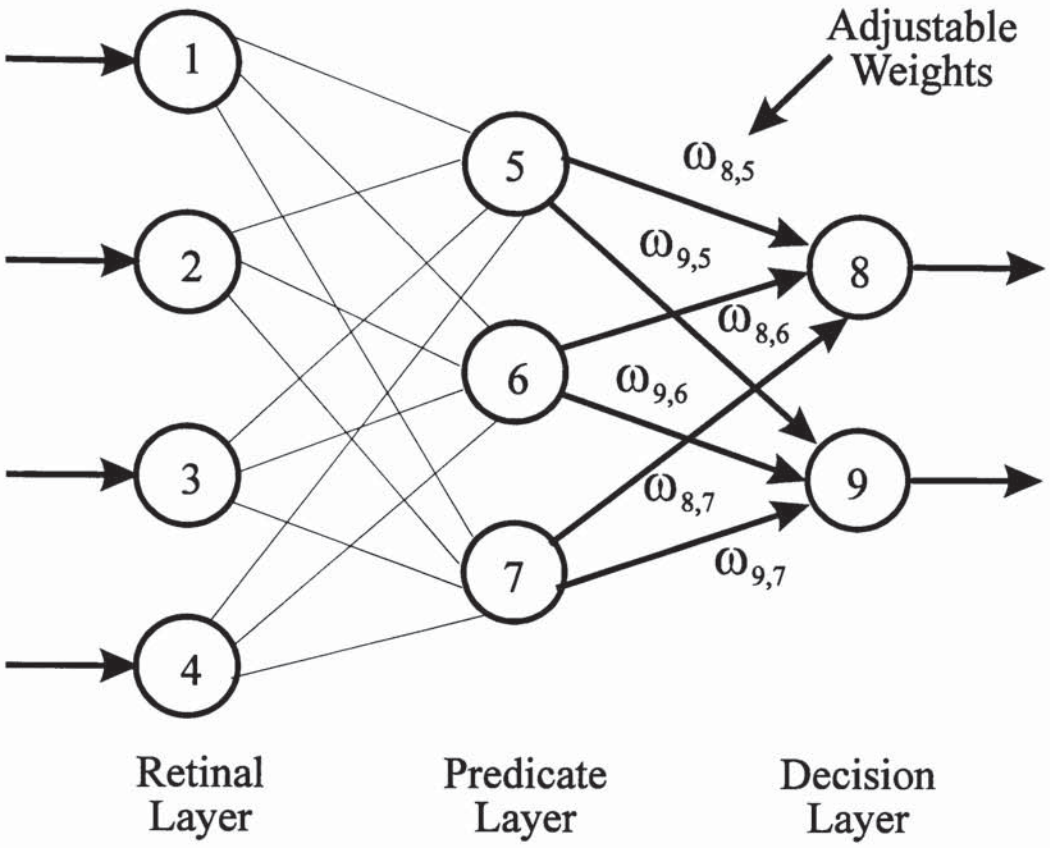


Figure 2.4 - The Original Perceptron

The network would appear to have three layers, but only two of these layers are connected by adjustable weights. This implies that nothing to the left of the predicate layer contributes to the learning process and the network could be drawn, as in figure 2.5, with only two layers, the retinal and predicate layers having been combined to form a more complicated input layer.

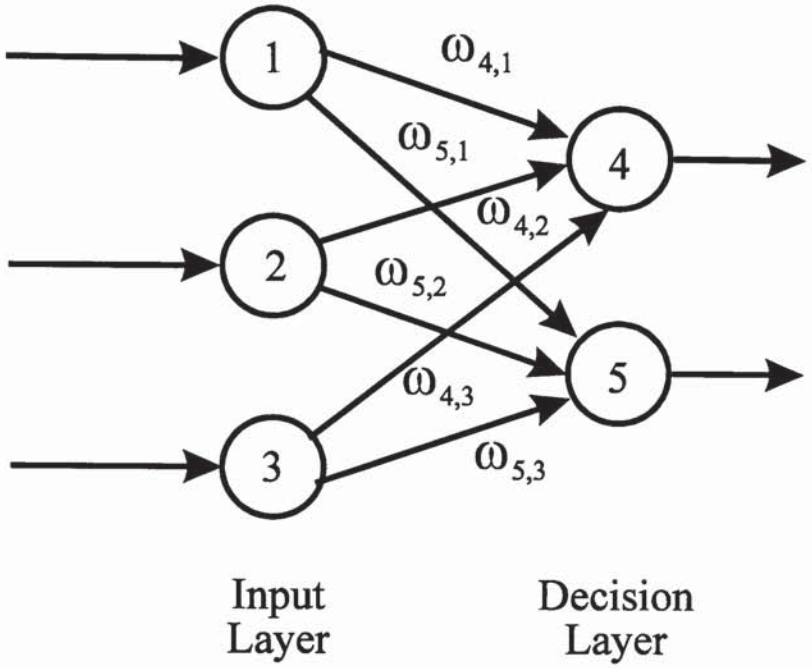


Figure 2.5 - A Condensed Perceptron

The perceptron based learning algorithm for two layer networks is guaranteed to find a set of weights that can map the input vectors to the correct output vectors, provided that such a set of weights exists (Hinton, 1985). Unfortunately, there exist quite simple problems, such as the exclusive OR (XOR) of two binary numbers, for which no such set of weights can be found. Minsky & Papert (1969), in their book 'Perceptrons', mathematically proved that the XOR problem is impossible to solve using a two layer network. This is the case because the relationship between the input and output is non linear and cannot be captured using a two layer network (Hinton, 1985). Hinton & Sejnowski (1986) illustrate this point in the following way. If the input and output units are considered together as a vector, the following set is used to train the network: (0,0,0), (0,1,1), (1,0,1) and (1,1,0). For this set of vectors, the first order probabilities are all 0.5 and the second order correlations are 0. That is to say that the probability of any one of the vector's elements being equal to one is 0.5 and the correlations between any two elements are zero. These are the same results that are obtained for the vector set consisting of all eight possible combinations of elements, and hence the two vector sets are indistinguishable using second order statistics. The

learning algorithms designed for two layer networks only capture second order statistics, and hence fail on the XOR problem which has a third order structure.

On the basis of this, Minsky & Papert (1969) condemned neural networks, reasoning that if they cannot solve a simple problem like the XOR problem, then how can they be used to model the complex workings of the human brain? They also stated that as there was no proof that networks with more layers would have greater power, and no learning algorithms for such networks existed, it would not be a profitable area of research. The result of these findings from such an eminent authority on the subject was that funding for neural network research in both the USA and USSR was severely curtailed (Ellison, 1988). However, some work in the area continued, most notably by Kohonen and Grossberg; consequently learning algorithms with proven convergence were produced for networks with more than two layers.

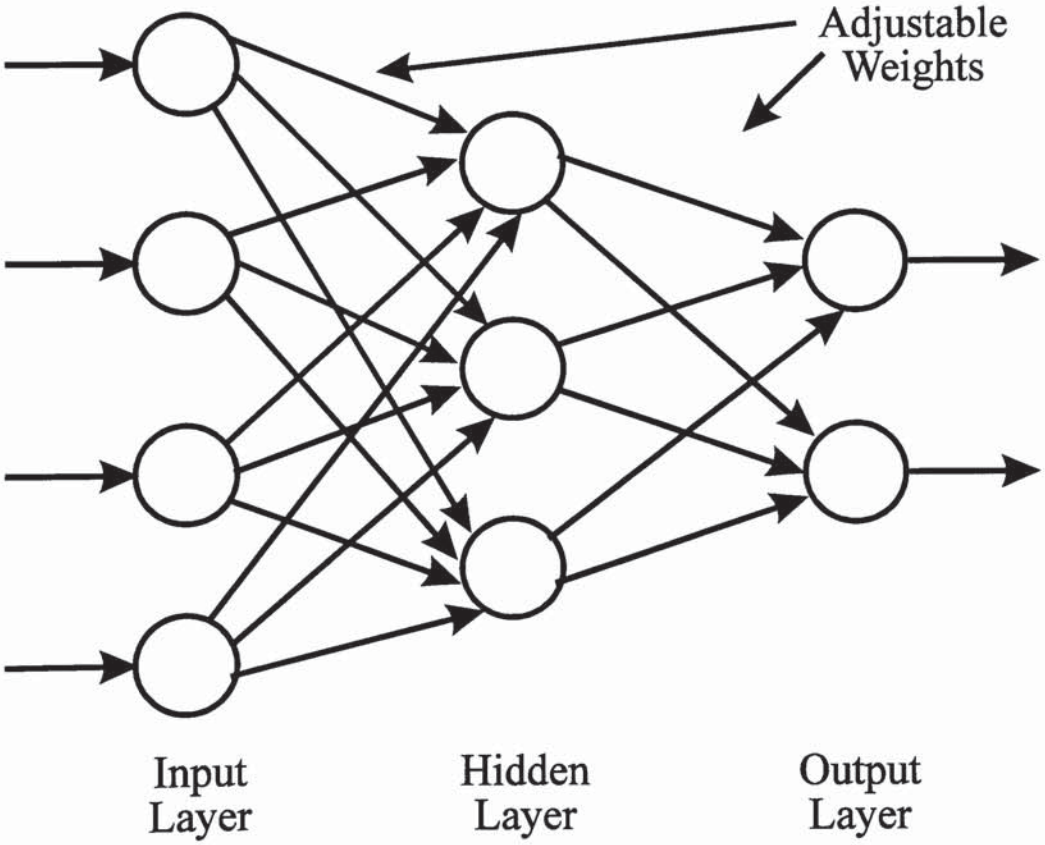


Figure 2.6 - A Three Layer Network

Figure 2.6 shows the most common of these multi-layer network architectures. The network has an extra, third layer of units, known as the hidden layer, and has two sets of adjustable weights to contribute to the learning process. These hidden units act as 'feature detectors', picking up higher order features with which to model the underlying structure of the input/output vectors (together called environmental vectors). Such networks exceed the performance of two layer networks and are capable of solving the XOR problem (Rumelhart, Hinton & McClelland, 1986). As more layers are added to the network, its power increases as it is able to model higher order functions, but the behaviour of the network becomes more difficult to predict and observe, and there is a speed penalty due to the increased number of weights that must be updated (Zeidenberg, 1987). The learning algorithms applied to these networks vary enormously in many different ways. Before describing some of the more popular ones and the network architectures they are carried out on, we need to differentiate between supervised and unsupervised learning.

2.1.3 Learning Paradigms - Supervised and Unsupervised Learning

During the learning phase of a supervised learning algorithm, both the input layer and output layer have their states clamped, which implies that the network is instructed which output vector to associate with each input vector (Rumelhart, Hinton & McClelland, 1986). This form of learning, also called associative learning, can be separated into further categories as follows. In auto-associative learning, each input vector is associated with itself, therefore if the learning process is completed, a particular input vector clamped to the input units will produce the same pattern on the output units even when the input pattern is corrupted by noise. The more general form, of which auto-association is a special case, is pattern association in which each input vector is associated with an arbitrary output vector (Rumelhart & Zipser, 1985).

When an unsupervised learning algorithm is used, only the input units are clamped and the network is allowed the freedom to 'decide' on the value of the states of the output units. In this case the network acts as a regularity detector, picking up the underlying structure of the input vector set (Rumelhart, Hinton & McClelland, 1986). Not all algorithms can be easily classified in this way; for example, both the Boltzmann machine learning algorithm (Ackley, Hinton & Sejnowski, 1985) and an algorithm developed by Moody & Darken (1989) use a combination of self-organisation and supervised learning. It is maintained by some of the proponents of unsupervised learning that it is the more biologically plausible form (Carpenter & Grossberg, 1988), although the majority of neural network models to date have been of the supervised learning type.

2.2 Neural Network Learning Algorithms

2.2.1 Competitive Learning

One unsupervised learning algorithm is competitive learning which forces the units in the network to compete against each other to respond to the input vectors presented. The architecture used has an arbitrary number of layers connected by unidirectional excitatory links. Within each hidden layer, the units are separated into non-overlapping clusters. The units in each cluster inhibit each other in such a way that only one of them may be active at once (an active unit having a state of value 1, and an inactive unit having a state value equal to 0) (Rumelhart & Zipser, 1985). The units within each cluster compete to respond to the output from the preceding layer, with the winning unit becoming active and inhibiting the other units in the cluster (Zeidenberg, 1987). A binary input vector is presented to the input units which are then clamped, and the units in each of the clusters in the next layer compete to respond. Only the weights on the connections to the 'winning' unit in each cluster are altered, which is achieved by transferring a fixed proportion of the

inactive unit's connection weights to active unit connections (Rumelhart & Zipser, 1985). This process is then repeated for each layer, using the output from the previous layer as input, until the states of the output units are determined. After iterating over the entire input vector set, the network will have partitioned this set into classes using the underlying structure (Rumelhart, Hinton & McClelland, 1986). Rumelhart & Zipser (1985) show that after presenting binary representations of the pairs of letters: AA, AB, BA, and BB, the set was split into two subsets (AA, AB) and (BA, BB), which indicated that the feature used to partition the set was the first letter of the pair. They achieved this result using clusters of size two; when clusters of size four were used, the subsets were simply the four members of the set. Rumelhart & Zipser (1985) state that in more complicated examples, it may be more difficult to determine the basis of partitioning.

2.2.2 Kohonen Networks

Kohonen(1982) developed a self-organising network capable of finding regularities and structure in input data and representing them as arrangements of feature detectors. The self organising feature map is a competitive network which is unsupervised, with the input units connected to an array of laterally connected output units.

Kohonen (1988) bases his model on the supposition that neurons in the brain which are close to each other respond to physically similar external features. The Kohonen net consists of two-dimensional arrays of output neurons, clusters of which respond to different inputs such that the greatest response is from the neuron in the centre of the cluster; the response of the neighbouring neurons decreases with distance. Kohonen(1988) suggests that lateral connections are a function of the distance between neurons, so that a given neuron will only influence a limited number of neurons within its vicinity. This is implemented in the model as the 'Mexican hat' function which is the derivative of the Gaussian function. As learning proceeds different areas of the network begin to respond to

different groups of input vectors, as features are extracted from the given environmental vector set.

Kohonen (1988a) applied his method to the problem of speaker independent detection of phonemes, a notoriously difficult problem. Each test phoneme was pre-processed into several different spectral channels, with 50 samples of each presented to the network as input. Kohonen (1988a) reports that after extensive post-processing to take into account the context in which each phoneme is spoken, the system could recognise speech from arbitrary speakers with an accuracy of between 92% and 97% depending on the experimental conditions.

2.2.3 The Hopfield Network

A neural network problem can be thought of as an optimisation problem in N dimensions that seeks to minimise a cost function subject to several constraints (Hinton & Sejnowski, 1986). A commonly used analogy is that the various possible configurations of the elements in the network form an N-dimensional surface with the different states of the network behaving like a ball bearing moving across the surface. The ball bearing is guaranteed to locate itself in a minimum, if there exists a cost function (corresponding to the 'height' of the surface) that is reduced every time the weights in the network are altered. Hopfield (1982) proved that such a cost function exists, which he called energy, for networks with symmetrical weights; that is to say the weight for the connection from unit i to unit j is the same as that from unit j to unit i. The connections in the network act as the constraints of an optimisation problem, in that two units joined by a positively weighted connection are reinforced in such a way that if one unit is active then so is the other; a similar case also exists for negatively weighted connections (Fahlman & Hinton, 1987). Unlike most optimisation problems, in which the constraints must be satisfied, the solutions in the Hopfield network may violate some of the constraints, incurring the penalty

of a gain in energy (Hinton & Sejnowski, 1986 and Ackley, Hinton & Sejnowski, 1985). Hinton & Sejnowski (1986) have noted that these so called 'weak constraints' are utilised in image processing applications where surface depth constraints must be relaxed at the surface's edges in order for the surface to be correctly interpreted. When a constraint in a Hopfield network is broken, the energy of the network is increased by the weight on the offending connection. This is due to the form of Hopfield's energy function, which is given by:

$$E = - \sum_{i<j} S_i S_j w_{i,j} + \sum_i S_i \phi_i \qquad 2.2$$

where i and j operate over the number of units in the network, S_i is the state of unit i (equal to either 1 or 0), $\omega_{i,j}$ is the weight on the connection between unit j and unit i and ϕ_i is a threshold for unit i (Fahlman & Hinton, 1987).

Using this equation, it can be seen that the difference in the energy of the network when a unit, i, is inactive and when it is active, is given by:

$$E_{ioff} - E_{ion} = \Delta E_i = \sum_i S_j \omega_{i,j} - \phi_i \qquad 2.3$$

which is used as a binary threshold device to determine the new state of a unit i. During the learning, the energy gap is used to update the states of the units that are not clamped, by setting the new state to 1 (regardless of the previous state) if the energy gap is positive, and setting it to zero otherwise (Ackley, Hinton & Sejnowski, 1986); Hebbian type learning is then used to alter the weights (Tank & Hopfield, 1987). Due to the fact that the energy is always decreased after every weight update, the solution obtained for each input vector corresponds to the nearest local minimum to the point on the energy landscape caused by the initial condition of the network. Hopfield has made use of this to design a network as a

content addressable memory, with each local minimum representing an item in that memory (Hinton, 1989). When presented with an input vector, the trained network finds the nearest local minimum which corresponds to the desired output vector (Hopfield, 1982). When presented with a noisy input vector, the network responds with the nearest local minimum, which is likely to be the one associated with the perfect input vector. This is demonstrated by Lippman (1987) who showed that a Hopfield network trained to recognise eight digits, could recognise the digit '3' when the input was corrupted by 25% noise. Further development of the Hopfield network led to a learning algorithm capable of finding the global minimum on an energy landscape using thermal noise. This algorithm is known as the Boltzmann machine learning algorithm.

2.2.4 The Boltzmann Machine

The Boltzmann machine is a development of the Hopfield network, which uses thermal noise to escape from local minima. A common analogy is that the various possible configurations of the network form an N-dimensional surface with the changing state of the network acting like a ball-bearing moving around the surface. Adding thermal noise is equivalent to physically shaking this surface, which enables the ball to 'jump' out of any local minima into the global minimum.

Kirkpatrick, Gelatt & Vechi (1983) developed an algorithm, called simulated annealing which uses ideas based on the thermodynamic properties of metal annealing. In order to find a very low energy state of a metal, its temperature is raised to a point at which its atoms have sufficient energy to move around easily, and in doing so align themselves in a minimum energy configuration. As the metal cools, the movement of the atoms becomes more restricted and their energy further decreases until the minimum energy configuration is reached. This process of successively heating and cooling the metal is known as annealing and results in a global minimum energy state (Cornana et al, 1987). Simulating

thermal noise in a Hopfield network is achieved by calculating the energy gap (as in section 2.2.3), and then setting the state of unit i to a value 1 with a probability P_i, given by:

$$P_i = (1+ e^{-\Delta E_i / T})^{-1} \qquad 2.4$$

where T corresponds to the temperature in the physical system. As the value of T approaches zero, the unit progressively approximates to, and eventually becomes, a linear threshold device as used in a pure Hopfield network. As the temperature is reduced, the probability of making a transition reduces and the network settles into a stable state known as thermal equilibrium (Aarts & Korst, 1988). When the network is at thermal equilibrium it is still changing states but is following a fixed probability distribution rather than changing states at random. Fahlman & Hinton (1987) liken this to shuffling a pack of cards; after shuffling the cards for a length of time, the probability of finding any particular card will remain constant at 1/52 even though the cards are continuously being re-ordered. Once the network has reached thermal equilibrium, the relative probability of the network being in one of two states, α or β, is given by:

$$\frac{P_\alpha}{P_\beta} = e^{-(E_\alpha - E_\beta) / T} \qquad 2.5$$

where P_α is the probability of being in the αth global state and E_α is the energy of that state (Sejnowski & Hinton, 1987); this is a Boltzmann distribution. For sufficiently long annealing schedules approaching a value of T=0, the global minimum can be achieved (Aarts & Korst, 1988). In practice the global minimum is not always reached, but the local minimum attained is usually sufficient to solve the problem.

An implementation of the algorithm is given in Lowton & Harget (1989), where the algorithm was found to be very robust despite the introduction of noise and damage into the network, but was found to require considerable computational processing for even relatively small problems. Peterson & Anderson (1987) achieved an increase in speed of

30 times by replacing the co-occurrence statistics used in the algorithm by solutions obtained from deterministic mean field theory equations; although again this has only been applied to small problems. Due to the high computational demands of simulated annealing, Boltzmann machine learning is not as widely used as the back-propagation algorithm. Although the latter does not possess a mechanism for avoiding local minima, it has been found to be capable of solving many classes of problems to a satisfactory level. Vogl *et al* (1988) argue that the search for the 'Holy Grail' of a global minimum is an unnecessary and largely fruitless activity given that other feasible points are capable of giving satisfactory results.

2.3 Back-Propagation Learning Algorithm

2.3.1 General Issues

Of the supervised learning algorithms, back-propagation is probably the most widely used due to its relative simplicity and success over a wide range of problem domains. Although there has been some dispute as to the origins of the algorithm, it is now widely accepted that the algorithm was developed by Werbos (1974) and independently derived by Parker in 1982 (Parker, 1987). Rumelhart, Hinton and Williams (1986a) were the first to prove its convergence subject to certain restrictions and were responsible for widely publicising the algorithm. The network has an input layer, an output layer and any number of hidden layers between, with full connection between layers but no lateral connections within each layer. This restriction has been removed by some workers (Scalettar & Zee, 1988), in the same way as many other modifications have been made to the algorithm to improve some particular aspect of performance. The basic algorithm however has remained relatively unmodified, its aim being to minimise the mean-squared error between the output

generated by the network and the desired output, over all the input/output pairs. This is given by:

$$E = \frac{1}{2} \sum_{c} \sum_{j} (y_{j,c} - d_{j,c})^2 \qquad 2.6$$

where c is an index over all input/output pairs, j is an index over the complete set of output units, $d_{j,c}$ is the desired (or target) value for unit j and case c and $y_{j,c}$ is the corresponding output given by the network. Convention dictates that the units are numbered sequentially from the input layer through each hidden layer to the output layer, as shown in figure 2.4.

2.3.2 Theory

In order to minimise the error function given by equation 2.6, the algorithm iteratively performs a two stage process, adjusting the weights until such time as the output of the network is correct to a pre-determined degree of accuracy.

The first stage begins with the states of the input units being clamped to the desired input vector, then layer by layer the states of the remaining layers are calculated in a feed-forward manner. For each unit , j, a value x_j is calculated using the equation:

$$x_j = \left(\sum_{i} y_i \omega_{j,i} \right) - \theta_j \qquad 2.7$$

where i is the index of the sum over the units connected into unit j, y_i is the output from unit i in the previous layer and $\omega_{j,i}$ is the strength of the connection between unit i and unit j. Each unit, j , has a threshold, θ_j, (used in equation 2.7) which can be treated in the same way as a 'standard' weight (Ackley, Hinton & Sejnowski, 1985). This is due to the fact that each threshold has exactly the same effect as a weight of size $-\theta_j$ connected to a unit that

always has a fixed output value of 1. Thus the threshold undergoes the same learning procedure as the weights, which helps to simplify the implementation of the algorithm. This value x_j is passed through a squashing (or activation) function which gives the output of the unit. Rumelhart, Hinton and Williams (1986b) state that the activation function can be any function that has a bounded derivative, but it is usual to use the equation:

$$y_j = (1 + e^{-x_j})^{-1} \qquad 2.8$$

The second stage then proceeds to alter the weights in the network in order to perform gradient descent in weight space (where 'height' in weight space corresponds to the error measure). The error can be calculated simply from equation 2.6, and is defined as the summed difference between the calculated response of the output units and their desired values. To minimise E by gradient descent, it is necessary to compute the partial derivatives of E with respect to each weight in the network. Since this is the sum of the partial derivatives for each input/output case, the index c will be suppressed for simplicity.

It can be shown (see Appendix A) that the partial derivative of E with respect to a weight $\omega_{j,i}$, where unit j is an output unit, is given by :

$$\frac{\delta E}{\delta \omega_{j,i}} = \delta_j y_i \qquad 2.9$$

where δ_j is given by :

$$\delta_j = (1 - y_j)\, y_j (y_j - d_j) \qquad 2.10$$

Hence, to perform gradient descent, the weight should be altered by an amount:

$$\Delta \omega_{j,i} = -\eta \delta_j y_i \qquad 2.11$$

where η is a constant of proportionality known as the learning rate.

Once the weight changes for the output layer are calculated, the error is propagated backwards to alter the remaining weights in the network. The partial derivative for a weight $\omega_{j,i}$, where unit j is not an output unit, is again given by equation 2.9 but in this case δ_j is determined by :

$$\delta_j = -y_j(1 - y_j)\sum_k \delta_k \omega_{k,j} \qquad 2.12$$

where k is an index running over the units in the next highest layer. The weights can then be altered using equation 2.11.

Variations exist in the way that the weights are updated. One method known as the on-line method updates the weights after each input/output presentation, whereas the batch method sums the partial derivatives obtained from each presentation and updates the weights once after the entire training set has been presented; one presentation of the training set is known as a sweep or an epoch. The on-line method has the particular advantages that the partial derivatives do not need to be stored and the task can be learned in fewer epochs (Fahlman, 1988), but the batch version requires fewer weight updates, which can be important in large 'real-world' problems.

A deviation from 'true' gradient descent that has become a standard feature of the back-propagation algorithm and has been shown to decrease the number of epochs required (Rumelhart, Hinton & Williams, 1986b), , is the addition of a momentum term. Using momentum, equation 2.11 becomes :

$$\Delta\omega_{j,i} = -\eta\,\delta_j y_i + \alpha\,\Delta\omega_{j,i(n-1)} \qquad 2.13$$

where n is the number of the current epoch. The momentum term tends to reduce oscillation and 'push' the algorithm in the desired 'downhill' direction (Becker & Le Cun, 1988), but at the expense of introducing another parameter which must be defined.

The back-propagation algorithm has been applied successfully to a great variety of problems some of which will be discussed in section 2.3.3; but most work at the present time is concerned with improving the algorithm. Improvements are being sought in three major areas, namely speed of convergence, generalisation and network configuration. This work will be outlined in Chapter 3.

2.3.3 Applications

Back-Propagation has been successfully applied to a great many problems in a large number of problem domains ranging in diversity from feature location in facial images (Hutchinson & Welsh, 1989) to modelling natural gas markets (Werbos, 1988). Outlined below are three example applications of the algorithm.

Bounds, Lloyd & Mathew (1990) applied back-propagation to the problem of computer aided diagnosis; an area that had previously been tackled using statistical methods but had been found to require a very large patient sample set which would have taken a great deal of time to analyse. The particular area of study was the diagnosis of lower back pain, a common and yet difficult problem to diagnose correctly. Data was collected from 200 patients in the form of a large 'tick sheet' of symptoms which was coded into binary form to be utilised as the input data to the neural network. The data was analysed and each patient assigned one of four diagnostic categories which had been validated by follow up care.

Half of the patient data was chosen at random to form the training set, with the remaining data forming the test set used to evaluate the network's performance. Several different

experiments were performed varying the architecture of the network including a network of networks structure where four small networks, trained on each class individually, were combined to form a 'global' network. However, the best results were obtained from a network trained using all four categories from the inception of training. The results from this network, together with the results from a radial basis function based network (Broomhead & Lowe, 1988), provided better results than three groups of clinicians asked to make a diagnosis based on the patient's data. Perhaps the most interesting aspect of the work was the fact that in most cases of mis-diagnosis, the network was seen to make the same mistakes as the clinicians, implying that the network had modelled their mistakes as well as their correct diagnosis.

Another application of back-propagation that produced results that out-performed trained human subjects was reported by Gorman & Sejnowski (1988), who trained a network to classify sonar returns from the sea bed. The problem was to discriminate between cylinders and cylindrically shaped rocks detected on the ocean floor. The continuously valued sonar returns were pre-processed and normalised to be presented as input to the network, with the output being represented as mutually exclusive binary vectors to indicate the nature of each object. Experiments were performed on both aspect-angle independent and aspect-angle dependent training sets. The aspect-angle dependent training and testing sets were chosen to contain the correct ratio of returns from objects at each angle, whereas the aspect-angle independent training and testing sets were chosen at random from the total set of returns.

The results for the aspect-angle dependent data sets were slightly better than for the independent data sets, which indicates that selection of a data set that is representative of the data domain can be an important consideration in the design of neural networks. In both cases, the networks achieved 10% greater accuracy than human listeners trained on the same data which had been shifted in frequency to the auditory range. Gorman &

Sejnowski (1988a) state that this is an important result, as sonar target recognition has been an area were human subjects have consistently out-performed machines in the past.

Another area, and one important to AI research in general, is that of game playing. This is due to the fact that game playing, e.g. chess, requires not only a knowledge of the rules but also high levels of skill for good performance. Tesauro & Sejnowski (1989) used the back-propagation algorithm to teach a neural network to play backgammon. Backgammon was chosen because unlike some games (for example chess), the value of the current state of the board (and hence the best next move) can be judged without having to analyse several different moves ahead, and hence is more of a test of pattern matching skill.

The network was trained with several different moves for a given board position, each of which was given a rating. Bad moves were included in the possibilities in order for the network to respond more positively to good moves. The network was trained on around 3000 board positions culled from a variety of sources, but it was found that the network chose bad moves in certain situations. This was due to the special nature of the end-game in backgammon where a slightly different game playing strategy is used. As the network had been trained on very few end-game situations, their significance was severely diminished by the more numerous examples of general play. In order to overcome this problem, additions to the training set were made to force the network to learn the end-game strategy. Tesauro & Sejnowski (1989) argue that this is a necessary addition because the situations occur so infrequently that they would not appear in a randomly derived training set. In its raw form, this is an extremely difficult problem, and hence other features were added to the input data, such as the short and long term risk of each move. With these added features, the network was able to beat an intermediate level rule-based computer system in 60% of games and it was estimated that it could beat an expert human player in 35% to 40% of games. This application of back-propagation illustrates how well neural networks are able to generalise; there are approximately 10^{20} possible board positions in

the game of backgammon, but despite being trained on only a small fraction of these, the network performs well.

Chapter 3

Performance Issues and Dynamic Architectures

3.1 Performance Issues

3.1.1 What are Neural Networks Capable of?

Neural networks have been shown to be capable of solving real-world problems, although it has not been adequately demonstrated that they have universal application. Despite extensive studies, architectural features such as the number of layers or units within a layer required for a given task are generally unknown. Suitable architectures are found by trial and error, an approach which has been criticised for lacking a theoretical base. Cybenko (1990a) illustrates this, saying of artificial intelligence techniques *'...instead of being a unified theory and a body of techniques, it is just one darned thing after another!'*. Efforts have been made in recent years to address this issue which was first rigorously tackled in the late sixties by Minsky & Papert in their book '*Perceptrons*' (Minsky & Papert, 1969).

Minsky & Papert showed that a perceptron could only solve linearly separable problems and was therefore severely limited in its application, since such problems form only a very small fraction of real world problems (Hanson, 1990). It was suggested at the time that multi-layer networks might have wider application, but learning algorithms for such networks had not yet been defined. Research has subsequently shown that multi-layer perceptrons are capable of learning a considerable number of difficult and interesting non-linear functions. Work is currently being undertaken to determine the scope of neural network capabilities and hence lead to an understanding of what architecture is required for a given task.

It has been shown that a multi-layered perceptron in which the units can have an infinite number of states, as is the case of real valued output, is equivalent to a Turing machine (Hartley & Szu, 1987). Hartley & Szu (1987) state that this is theoretically important since there is no known stronger model of computation than the Turing machine, but that its practical value is less certain because of the limitations on the accuracy available. Wieland & Leighton (1987) show that any binary mapping (i.e. binary inputs mapped onto binary outputs) can be represented by a network with at least two layers of hidden units. This has been made more specific by Denker & Schwartz *et al* (1987) who show that any binary mapping can be represented by a network with just one layer of hidden units, but for some problems a large number of units within that layer may be necessary. Whilst showing that one hidden layer of units is sufficient, this result does not address the issue of the number of units to use within that layer, and also what architecture to adopt for non-binary functions.

The proof that any continuous function can be modelled by a neural network with two layers of hidden units comes from the unusual source of a Russian mathematician's solution of Hilbert's long standing thirteenth problem (Girosi & Poggio, 1989). Using Kolmogorov's theorem, it can be shown that any continuous function of many variables can be represented by a neural network having two hidden layers. Opinions on the abilities of networks with one hidden layer have changed in recent years. As recently as 1987 it was felt that there existed real valued functions that could not be represented by a network with only one hidden layer (Lippman, 1987). However, it has since been shown that neural networks with only one layer of hidden units can model any function to an arbitrary degree of accuracy (Cybenko, 1990). Hornik, Stinchcombe & White (1989) also state this proposition via a proof which shows that any Borel measurable function, that is to say any function with a countable number of discontinuities or as they state 'any function of interest', can be represented using a neural network with one hidden layer to any desired degree of accuracy.

This still leaves the unsolved question of how many hidden units within a layer are needed by a network to solve a given problem. Previous attempts to answer this question rely on limits to the number of units required in a single layer. Hecht-Nielsen (1987), again using an adaptation of Kolmogorov's theorem, shows that any continuous n-dimensional function can be represented by a network having a single hidden layer with 2n+1 hidden units. Also Sartori & Antsaklis (1991) prove that any training set of p patterns can be exactly represented using a single hidden layer of p-1 hidden units. Both of these figures have only limited use in the sense that they give only a theoretical upper limit to the number of hidden units required. For example, a continuous linearly separable function in 32 variables requires no hidden units at all, whereas the two schemes outlined above state that it is possible to solve this problem with 65 and 2^{32} - 1 hidden units respectively. Obviously this is an extreme example, and for many functions one or both of these schemes will give a reasonable upper limit on the number of hidden units, but this is likely to be higher than the minimum required. However, whilst the results quoted in this section give a theoretical basis for stating a general architecture for representing a given function, they do not address the issue of whether or not the representation can be learned in a reasonable amount of time.

The theories mentioned can only prove that in principle, functions of certain kinds can be represented using networks with certain architectures; this does not take into account factors such as initial conditions, the learning algorithm used, the problem of local minima, the resultant generalisation properties and whether a solution can be found in a reasonable time. An illustrative example of the problem of local minima and initial conditions is given by Brady, Raghavan & Slawney (1989), who show that there exist linearly separable problems that the back-propagation algorithm, given certain initial conditions, will be unable to solve due to local minima. The question of the length of time to learn a given function from a training set is addressed by Judd (1987) who shows that this task is NP-complete and hence has no efficient general solution. This is not a surprising result since it

would be a considerable mathematical breakthrough to prove otherwise, since every existing NP-complete problem could then be solved in polynomial time. This is due to the fact that any NP-complete problem can be converted into any other in polynomial time and hence be solved in polynomial time itself (Bruck & Goodman, 1990 & Penrose, 1989). It does however, present a difficult problem of scaling in neural networks if the aim of implementing neural networks is to solve real-world problems, which are invariably large and therefore computationally demanding. Several studies have attempted to address this problem by investigating methods to speed up the learning process, some of which are described in the following section.

3.1.2 Speed Of Convergence

One of the major problems with the back-propagation algorithm and of neural networks in general, is the speed at which the network converges to a solution, both in terms of the number of iterations required and the CPU time taken. One approach to improve this situation has been to design hardware specifically for neural network simulations; using VLSI (Graf, Jackel & Hubbard, 1988), electronic circuits (Tank & Hopfield, 1986, 1987), parallel architectures (Forrest *et al*, 1987) and a proposed optical implementation (Ticknor & Barrett, 1987). While these techniques have been largely successful, most work in the area has concentrated on improving existing algorithms and developing new ones.

As the amount of work in this area has increased, it has become apparent that there are many problems in conducting a realistic comparison of different algorithms in terms of performance. For example, in reporting speed improvements it is important to give both the number of iterations required and the CPU time taken since both are metrics of performance. Lang & Witbrock (1988) reported that back-propagation required 20,000 iterations to distinguish between two intertwined spirals, whereas the backgammon

network described above required only 50 iterations but took between 100 and 200 hours of CPU time (Tesauro & Sejnowski, 1989).

The back-propagation algorithm is sensitive to the initial values taken for the weights in the network and also the parameters used. For example Kolen & Pollack (1990) showed that the non-convergence rate on the XOR problem rose from less than 5% for weights in the range $|W| < 0.5$ to 50% for $2 < |W| < 4$. This shows that in order to compare results, researchers must use the same initial parameters and weight ranges and repeat each experiment several times with different starting weights in order to obtain statistically significant results.

When the algorithm fails to converge, alternative methods have to be employed to obtain a value for the average performance. Several different methods have been employed to circumvent this problem; Tesauro & Janssens (1988) use the inverse of the average learning rate which avoids the problem of infinite learning times but tends to give a result that is biased towards the faster trials. Jacobs (1988) merely reports the number of failed trials and gives an average value for the convergent trials, whilst Fahlman (1988) reports the number of failures but performs them again to obtain an average over the full number of trials. The lack of standardisation in this area makes comparison difficult, as the methods mentioned above represent a small proportion of the possibilities. Comparison is also made more difficult by the fact that the criteria used for successful convergence is not universal. The most commonly used criteria of success is when the mean squared error over the entire training set falls below a certain pre-determined value, but this is by no means the only method used. Yu & Simmons (1990) use what they call a correctness ratio to determine success; the real value of the output unit is thresholded (usually against 0.5) to obtain a binary value which is then compared to the binary target value. Once all the outputs for all the members of the training set are correct using this measure then learning is deemed to be complete. Yu & Simmons (1990) state that it is quite possible for learning

to terminate whilst still having a large value for the mean squared error, and hence comparison with other algorithms using that measure is difficult to make.

The choice of test problem may also uniquely affect the algorithm's performance in a way that would not be matched by other problems. For instance, Smieja & Richards (1988) developed a variation of back-propagation which they state is analogous to the way in which human beings gradually refine their skills from a basic set of initial skills. As the learning proceeds, the problem is gradually made more and more difficult until the desired problem is solved. This process can be seen as a deformation of a problem, which can be represented as a simple shape in some space, into a more difficult problem that is represented by the same surface forming a more complex shape in the same space. This algorithm managed to solve a complex problem that standard back-propagation was unable to solve, however, it is not applicable to a wide range of problems since it is not clear how most problems could gradually be made more difficult over the course of learning.

Some attempts have been made at standardisation so that a realistic comparison can be made between various algorithms. For example, Scott Fahlman published a paper on the subject (Fahlman, 1988), and started an electronic mail bulletin board designed specifically to deal with the problem of bench-marks. However, it has largely remained the case that results are stated without sufficient information to replicate them; while this situation persists, it will prove difficult to make comparisons between algorithms on performance issues such as speed.

One of the difficulties with using the back-propagation algorithm lies in the selection of parameters, since performance can be markedly altered by the parameter values chosen. Ideally, the learning rate should be large enough to avoid taking small steps along narrow ravines in weight space, whilst being small enough to avoid 'jumping' over narrow minima in areas of high curvature of the error space. The traditional method of determining the

'ideal' value has been described as a 'black art' (Fahlman, 1988), combining intuition and experience with trial and error.

Jacobs (1988) argues that such a global 'ideal' value of the learning rate may not exist in the vast majority of cases, since the complex shape of the error surface will mitigate against a constant learning rate, due to variations in curvature in different portions and directions of the surface. He proposed a variation of the back-propagation algorithm that has a separate learning rate for each modifiable parameter in the network. Thus if the learning rate for a particular parameter proves to be inappropriate at a later stage of learning, its value may be altered. If the current derivative of the error with respect to a parameter is in the same direction as the exponential average of the previous derivatives then the learning rate is increased; alternatively, if they have opposite signs then the learning rate is decreased. Applying this to several problems yielded faster learning times but a higher incidence of non-convergence.

A similar method is employed by Vogl *et al* (1988) using one learning rate for all the parameters in the network. Using the 'batch' method of weight updates, the learning rate is increased if the error is reduced between consecutive iterations. Whereas the learning rate is decreased if there is a rise in error; in this case the momentum is set to zero so that previous adverse weight changes are nullified. Applying this method to problems of moderate complexity gave better results but the algorithm seems to be very sensitive to the initial value of the learning rate, giving an optimum convergence rate for a value of 0.261 and no convergence at all for values greater than 0.263.

Cater (1987) also modified the back-propagation algorithm to dynamically alter the learning factors in the network, but in this case a separate learning rate is used for each training pattern to be learned. The patterns are presented to the network, after which the learning rate for the pattern resulting in the greatest error value is doubled with the remaining learning rates remaining constant. Unfortunately this technique results in

oscillation and so a second alteration was made to reduce the learning rates for all those patterns where the total error increases by more than 1% between successive iterations. Cater (1987) states that increases in speed of between 5 and 20 times are observed over the standard algorithm, with convergence in this case being defined as the point at which the error curve begins to monotonically decrease. The standard error curve tends to contain small fluctuations as learning proceeds which result in a higher number of iterations using this measure of convergence.

Another approach, which gives only limited success, removes the need to define the learning factor by linking it to the architecture of the network (Plaut *et al*, 1986). One area of research that has been applied to the problem of parameter selection and which is generating a large amount of interest is genetic algorithms. Marshall & Harrison (1991) have applied genetic algorithm techniques to the problem of determining optimum parameter values in back-propagation networks. The various different aspects of a back-propagation network (for example the initial weight range, learning rate, momentum factor etc.) are coded as a binary string, with each parameter taking a number of bits in the string and hence taking one of a finite number of values. Initially a large population of such strings are evaluated and each given a fitness value; those strings with the largest fitness values have the greater probability of being combined to form the next generation of the population. This process is repeated with small amounts of random mutation to avoid stagnation at a non-optimal point. In effect the genetic algorithm is simultaneously searching many points of the state space in a directed probabilistic manner to find the optimal set of parameters. This method was applied to a chest pain diagnosis problem and took 12 hours to find a set of parameters that had previously taken 12 weeks to determine using trial and error. In addition, a genetic algorithm to optimise network weights was also implemented, however, this can only find an approximate solution because of the binary representation used, and so the final refinement of the weights uses the back-propagation algorithm.

Gradient descent is a first-order search method, that is to say, it only uses information gained from first derivatives. It has been suggested that second-order methods that take into account the curvature of weight space, such as Newton's method, would converge much faster (Parker, 1987). In order to implement this however, the Hessian matrix of second derivatives must be calculated and inverted, which requires $O(n^3)$ operations where n is the number of modifiable parameters in the network. Back-Propagation requires O(n) operations and is consequently much less computationally demanding.

Watrous (1987) proposed a quasi-Newton method based on back-propagation and found that it converged more rapidly to a solution. Watrous' method used an approximation to the Hessian matrix and requires $O(n^2)$ operations. Despite requiring less iterations however, this method is considerably more computationally demanding especially for medium to large problems. Becker & Le Cun (1988) have implemented a second-order algorithm based on an approximation to Newton's method using a diagonal approximation to the Hessian matrix which requires O(n) operations, and found that it converges from 1.5 to 2.5 times faster than standard back-propagation. Despite this, they report that on the limited test problems studied, the algorithm seems to be more sensitive to the values of the learning rate and initial weights and that approximately 30% more CPU time is required for convergence. Perhaps the most promising of the second order methods is Quickprop developed by Fahlman (1988). This method is also based on Newton's method but combines this with the assumptions that the error against weight value curve can be approximated by a parabola and that the curves are not affected by simultaneous weight changes in the network. These assumptions (acknowledged by Fahlman (1988) to be 'risky') simplify the algorithm and ensure that it requires O(n) operations. Fahlman (1988) reports that the algorithm converges much faster over the range of problems tested than standard back-propagation and has the added advantage that it satisfactorily scales up to larger and more difficult problems.

Aside from the two major areas of learning factor modification and second order methods described above, there have been many other attempts to speed up the algorithm, using a wide variety of methods. When the back-propagation algorithm is used to learn binary training sets, the data is naturally enough presented as combinations of zeros and ones. However Stornetta & Huberman (1987) note that whenever an input unit is set to zero it is effectively excluded from the learning process and its weights remain unaltered. In order to remove this anomaly they altered the algorithm so as to represent the data as patterns of -1/2 and 1/2 and found that the algorithm performed 10% better than back-propagation and was less susceptible to fluctuations in performance. However, the description of how these results were achieved is vague and difficult to replicate.

Hoskins (1989) has developed a heuristic called 'focused attention back propagation' which alters the training by focusing attention on those members of the training set that are the most difficult to learn. The algorithm achieves an increase in speed of 50% on three real valued problems but has the disadvantage that a larger number of trials fail to converge.

The work described above is by no means an exhaustive survey (see for example Dahl (1987), Solla *et al* (1988), Silva & Almeida (1991) and Chan & Fallside (1987)) but covers the main approaches that have been taken to improve convergence rates for the back-propagation algorithm.

3.1.3 Generalisation

The performance of neural network algorithms is usually measured in terms of the number of iterations taken to learn the members of the training set to a certain degree of accuracy. Whilst correct classification of the training set (sometimes referred to as memorisation) is important, it is more important to capture the underlying structure of the environmental

vector set and hence respond correctly to members that were not present in the training set. This is stated by Wolpert (1990), who defines generalisation to be '*the ability of a system to take a learning set of input-output vector pairs from a particular pair of input-output spaces, and, based only on that learning set, make a "good" guess as to the output vector corresponding to an input vector not contained in the learning set*'. Denker *et al* (1987) define generalisation in terms of set theory, which is shown diagrammatically in figure 3.1.

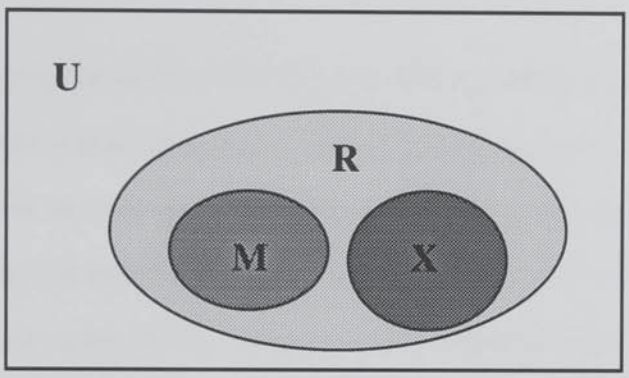


Figure 3.1 - **A** Diagrammatic Representation of Generalisation

In figure 3.1, U represents the universal set of all input-output pairs, of which R is a subset which represents the function (or rule) the network is to learn. M and X are disjoint subsets of R which represent the training (or memorisation) set and generalisation (or extraction) set respectively. Denker *et al* (1987) define generalisation to be the ability of the network to accurately represent set X, after being trained on set M.

Wolpert (1990a) argues that this ability to 'generalise' is the primary reason for the current interest in neural networks. In practical terms, generalisation is important because it relaxes the requirement that the training set should contain all possible members. This is not always a significant factor, for example Denker & Schwartz *et al* (1987) showed that a network which takes a binary input pattern and outputs a 1 or a 0 depending on whether the input pattern is odd or even achieved perfect generalisation because it learned that only the least significant bit was important and common to all possible examples. In general,

however, a large proportion of all possible patterns must be presented to the network to achieve reasonable generalisation performance. For example, the backgammon network described previously has 10^{20} possible board positions, which clearly constitutes an impossibly large training set to be adequately represented by a small subset of patterns using an arbitrary architecture (Tesauro & Sejnowski, 1989). Hence work has been done to determine those factors inherent in network architecture, the training set and the algorithms, that determine the network's ability to generalise.

Limiting themselves to the study of Boolean functions, Huyser & Horowitz (1988) set out to find a method that would guarantee correct generalisation. Firstly they divided the environmental vector set into two sets; the training set which incorporated those vectors that contain essential information, and the test set whose vectors included only redundant information. This was achieved by comparing the output of each vector against that of its nearest neighbours. If the output of any of the neighbours was different from that of the original vector, then the vector and its neighbours fell on a decision boundary and as such were classified as critical vectors which should be included in the training set. Huyser & Horowitz (1988) also state that to guarantee correct generalisation the minimum number of hidden units possible must be used, and the learning process must terminate at the global minimum of the error space. These conditions ensure that the network does not learn the training set by rote or learn a function that is adequate to describe the particular vectors in the training set but is unrepresentative of the data set as a whole. The method has been shown to provide accurate generalisation, but the authors concede that the minimum number of hidden units required to learn most Boolean functions is unknown. A limited empirical study carried out by Ahmad & Tesauro (1988) concluded that performance was enhanced when the network was trained using 'border patterns', which correspond to the critical vectors used by Huyser & Horowitz. They also conclude that further work must be done to investigate the many factors that influence the network's generalisation ability.

As noted by Huyser & Horowitz (1988), it is widely accepted that a minimal architecture improves the generalisation ability of neural networks. Denker *et al* (1987) show that the number of possible generalisations, given a particular training set, is very large. Given a Boolean problem with N inputs and P outputs, then the number of possible truth tables that exist is given by:

$$(2^{\mathrm{p}})^{2^{N}} \qquad 3.1$$

If the number of items in the data set is given by m, then the number of generalisations possible given this data is:

$$\mathrm{G} = \frac{(2^{\mathrm{p}})^{2^{N}}}{(2^{p})^{m}} \qquad 3.2$$

Hence, given a problem with 30 input bits, 1 output bit and a thousand training examples, there are $2^{10^{9}}$ possible generalisations that are perfectly valid given the training data (Denker *et al*, 1987). Baum & Haussler (1989) demonstrate that given m training examples, w weights and n nodes, then if a network can classify 1-e/2 of the training set (for suitably small e), then the network will correctly classify a fraction 1-e of unseen examples drawn from the same distribution as long as:

$$\mathrm{m} \geq O\left(\frac{w}{e}\log\left(\frac{n}{e}\right)\right) \qquad 3.3$$

It can be seen from the above equation that in order to force the network to adopt the required generalisation, then either the training set size must be increased which increases computational demands, or the number of hidden units must be reduced (Kruschke, 1989). This is summed up in a private communication from Rumelhart to Hanson & Pratt (1989), in which he argues that '*...the simplest most robust network which accounts for a data set*

will, on average, lead to the best generalisation to the population from which the training set has been drawn'.

Wieland & Leighton (1987) explain this by considering the learning process as one of curve-fitting where high levels of generalisation are analogous to good non-linear interpolation. If the number of hidden units is too large then it is possible that the curve will be over-fitted, resulting in poor generalisation. Networks containing a small number of hidden units, constituting an information bottle-neck, are likely to extract salient features from the training set. This is illustrated in figures 3.2 and 3.3 below:

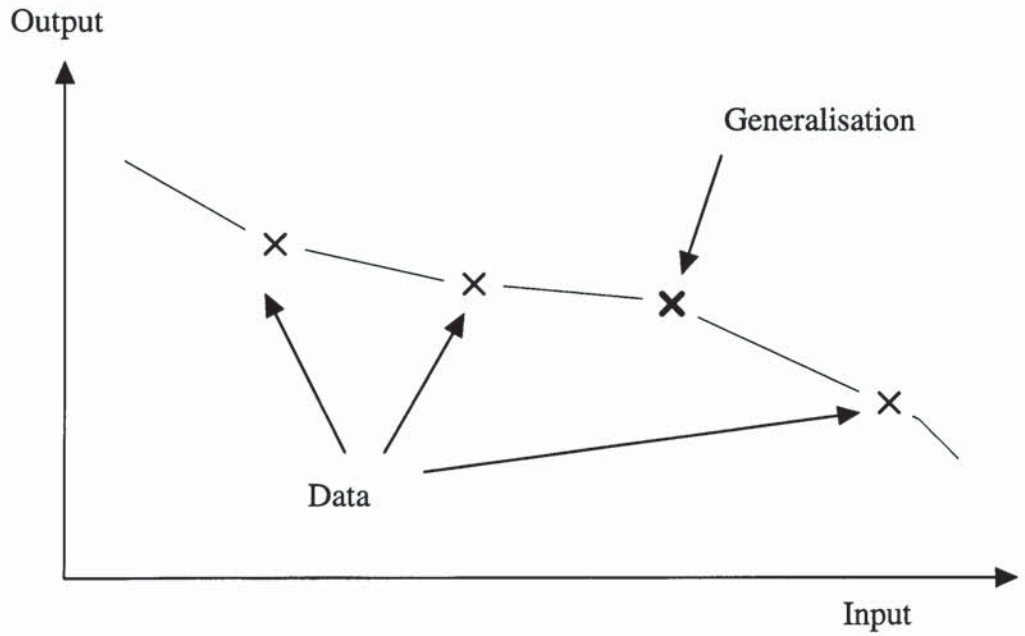


Figure 3.2 - A Curve-fitting Representation of Good Generalisation

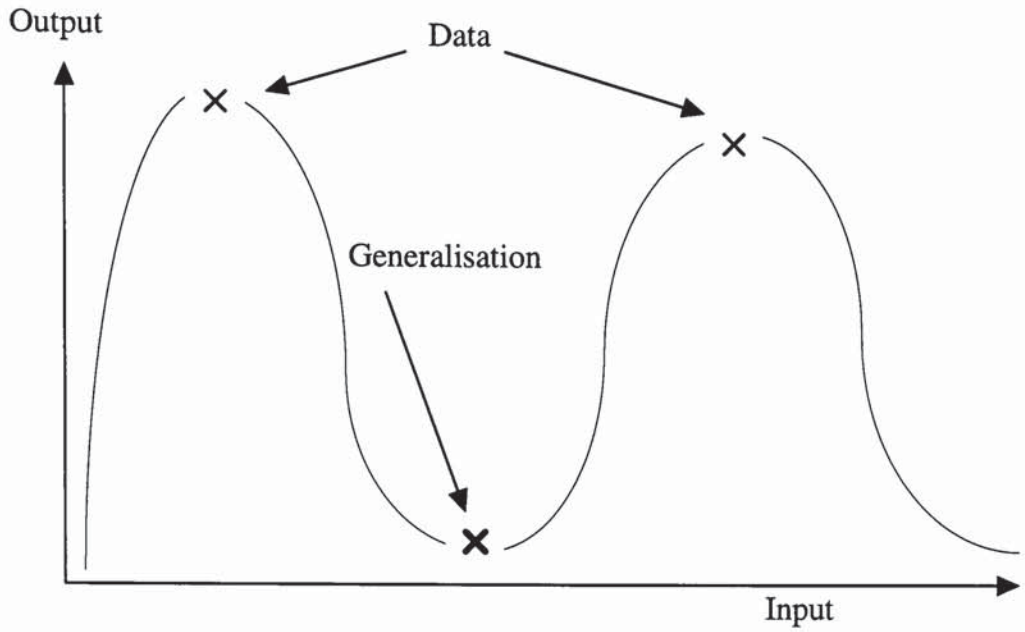


Figure 3.3 - A Curve-fitting Representation of Poor Generalisation

Figure 3.2 shows an example of good interpolation which results in improved generalisation, whereas the curve depicted in figure 3.3 shows a less smooth transformation representing an over-fitting of the data. Whilst accepting that constraining the number of parameters in the network results in improved generalisation, Le Cun (1989) argues that *a priori* knowledge of the problem must be incorporated into the design of the network architecture to achieve the best results. Le Cun (1989) improved the generalisation performance from 87% to 98.4% on an image recognition problem by introducing a hierarchy of shift invariant local feature detectors. This approach would not necessarily carry over directly to other problem domains but illustrates that incorporating problem specific information into the network design strategy can be helpful in improving generalisation. A similar approach was taken by Yu & Simmons (1990), who introduced extra output units into the network in order to increase the information content of the training set. The example considered is the parity problem which takes N inputs and has a single output unit to indicate the parity of the input pattern. Yu & Simmons (1990) point out that one way to decide on the value of the parity bit is to count the number of 1's in the pattern and decide whether it is odd or even. Hence extra output bits are included in the

network to encapsulate this information, resulting in an improved performance in terms of average training time and completed trials. In order to test the generalisation properties of the method, a similar approach was taken on a language understanding problem and resulted in a 5% improvement in generalisation ability (Yu & Simmons, 1990a).

Whilst methods such as the two outlined above, improve generalisation performance, they are problem specific and would not be simple to implement in more complex problems where relevant knowledge is not readily available. An alternative approach, and one we discuss in the next section, is to dynamically configure the network architecture as learning proceeds. This removes the necessity for performing a great many experiments to ascertain the optimum architecture, as the aim of these methods is for the optimum architecture to be generated automatically during the learning process.

3.2 Dynamic Architectures

A number of algorithms have been developed in recent years that dynamically configure the architecture of a neural network. The research has been instigated by a need to determine an architecture that offers good training and a high level of generalisation, without the need to perform vast numbers of trials. This ability also fits well into the adaptive nature of neural networks by introducing learning into the process of network architecture selection. In general the methods that have been developed can be separated into two distinct categories; destructive or pruning algorithms, and constructive or growth algorithms. Destructive algorithms begin learning with a large network and remove weights or units or indeed both as the learning proceeds, whereas constructive algorithms add units, or in some cases layers, to an initially small network. These two broad categories are discussed in sections 3.2.1 and 3.2.2 respectively, as well as hybrid systems in 3.2.3.

3.2.1 Pruning or Destructive Algorithms

Destructive algorithms dynamically configure the network architecture by removing elements of an initially large network as training progresses. One approach which is an extension of a common practice in the neural network community is to include a weight decay term in the learning procedure. This is usually done to deter the network from assuming very large weights which, because they result in a very low derivative value, can considerably impede learning. A logical extension of this method would be to allow weights to decay away completely, when they can be removed from the network, thus resulting in a reduction in network complexity.

Hanson & Pratt (1989) took this approach based on the unpublished work of Rumelhart. Rumelhart included a bias term in the cost function to be minimised, given by:

$$B = \sum_{i} \sum_{j} W_{i,j}^{2} \qquad 3.4$$

This implies that the back-propagation algorithm now minimises the expression:

$$O = E + B \qquad 3.5$$

where E is the standard mean squared error. This extra bias term forces the weights to remain small, and in effect completely removes some weights from the network. Hanson & Pratt (1989) note that this approach decays all weights equally and also causes the derivatives to decay resulting in smaller error changes. They introduced two different bias terms given by:

$$B = \frac{w_i}{(1 + \lambda w_i)} \qquad 3.6$$

referred to as a hyperbolic bias, and an exponential bias given by:

$$B = (1 - e^{-\lambda w_i}) \qquad 3.7$$

These bias terms allow very large and very small weights to decay at a high rate whilst leaving the more average weight values relatively unaltered. This is a desirable feature since large weights are undesirable for the reasons outlined above, and removing the small weights in the network offers a reduction in network size. Applying these methods to low order parity problems and removing any hidden units that were unnecessary resulted in smaller networks using these biases, but the authors concede that a high proportion (up to 75%) of the trials did not converge within a reasonable time (Hanson & Pratt, 1989).

A different type of bias term was introduced by Chauvin (1989), who includes a function of the activation values of the hidden units as a bias in an attempt to minimise the 'energy' of the hidden units. This has the effect that redundant hidden units which have an invariant activation value over the training set are effectively, though not 'physically', removed. By minimising the activations of units that contribute very little to the learning process, the remaining units are forced to act orthogonally, and hence more effectively. This method was applied to small binary problems and found to produce minimal architectures, though the network showed a tendency to increase weight values to compensate for the reduction in hidden unit activations (Chauvin, 1989). Hence, the weight decay term given by equation 3.7 was introduced, presumably along with the inherent problems mentioned above, although no indication of relative learning times is given.

In contrast to this gradual decay approach, several algorithms have been developed which completely remove a unit, together with its associated weights, from the network at some point during the learning process. Mozer & Smolensky (1989) developed a method called 'skeletonization' which calculates a relevance measure for each hidden and input unit at a

given point and removes the least relevant. A measure of the contribution of a unit, i, is given by:

$$P_i = E_{(without\ unit\ i)} - E_{(with\ unit\ i)} \qquad 3.8$$

but this has the problem that the entire training set must be presented twice to calculate the value of P_i. In order to avoid this, Mozer & Smolensky (1989) introduce a coefficient into each input and hidden unit which represents the 'attentional strength' of the unit, such that the output of a unit j is given by:

$$O_j = F\left(\sum_i \omega_{j,i}\alpha_i O_i\right) \qquad 3.9$$

where α_i is the attentional strength which effectively removes the unit if it has a value 0, and results in a normal unit if it has value 1. Thus the relevance measure of equation 3.8 can be re-written as:

$$P_i = E_{\alpha_i=0} - E_{\alpha_i=1} \qquad 3.10$$

which is approximated by a value $P`_i$ given by:

$$P`_i = \left.\frac{\partial E}{\partial \alpha_i}\right| (\alpha_1,.....\alpha_n) = (1,.....1) \qquad 3.11$$

and incorporated into the back-propagation learning procedure to avoid the necessity for extra passes through the network (Mozer & Smolensky, 1989). The algorithm proceeds as follows; the network is trained until the output unit activities are within a pre-determined range of their target values, at which point the unit with the lowest relevance value is removed and the learning process repeated. Applying this to several small problems resulted in a reduction in network size in a time comparable with ordinary back-propagation (Mozer & Smolensky, 1989). Whilst this paper demonstrates that destructive

algorithms can find less complex networks within a reasonable time frame, the concepts involved have not been developed into a generally applicable algorithm. In the case of the examples given in the paper, the number and type of units to remove were determined in advance in order to illustrate certain points, such as the ability of the network to recover from the 'damage' sustained by the loss of a hidden unit. Mozer & Smolensky (1989) state that a general algorithm could be realised from this work by using the relative sizes of the relevance measures to determine a terminating criteria for the trimming process, so allowing the algorithm to remove either type of unit, but this has yet to be implemented.

A similar method for the removal of hidden units based on a measure of their contribution to the performance of the network has been implemented by Sietsma & Dow (1991). This method has two stages of pruning; firstly non-contributing units are removed, where a unit is defined to be non-contributing if it satisfies one of three criteria. Firstly if it has an approximately constant output across the patterns in the training set, secondly if it gives the same outputs as another hidden unit, or lastly if it gives the opposite output to another hidden unit. The second stage consists of removing those units which are independent at one layer but contribute redundant information at the next layer (Sietsma & Dow, 1988). This second stage is necessary where several units are linearly independent but a subsection are excess to the minimum information required to separate the classes formed at that layer. This however can lead to a problem in that the classes formed at a hidden layer may not be linearly separable, and in this case a small network is trained to solve the problem and then inserted into the larger network (Sietsma & Dow, 1991). In some senses this becomes a hybrid method incorporating both constructive and destructive features. However this is not a hybrid method in the same sense as those to be discussed in section 3.2.3, since it is using a constructive feature as a convenient escape from an inherent problem of the method, rather than as a basis for the method.

Applying this approach to the task of learning the frequency of sine waves with different phase shifts, Sietsma & Dow (1991) report that the network was reduced from a size of 64-

20-8-3 to networks of various smaller sizes including 64-8-3 and 64-6-4-4-3-3 (where there are 64 input units and 3 output units). Comparing the generalisation performance of these pruned networks with that of trained networks of the original size, showed the smaller networks to be superior, but networks with several small hidden layers performed badly.

The above example illustrates some of the problems inherent in destructive algorithms, in that a reasonable guess must be made to the starting configuration of the network in the same way as the required number of hidden units must be estimated in standard back-propagation. If the estimate of the number of hidden units is considerably larger than the minimum number required (as with the 64-20-8-3 network above) then most of the learning is done with excessively large networks which is computationally demanding. Le Cun (1989) also points out that the pruning coefficients must be very accurately tuned to avoid catastrophic effects such as over-pruning, and that the process of determining which elements of the network to prune dramatically lengthens the learning process.

3.2.2 Constructive Algorithms

The basic principle behind these algorithms is to start from a very small network (possibly having no hidden units at all), and gradually add in hidden units or, in some cases, hidden layers until the network is capable of solving the given problem to a specified level of accuracy.

Ash (1989) developed a method called 'Dynamic Node Creation' which adds hidden units into a single hidden layer as learning proceeds until a solution is found. The method is based on the fact that if too few hidden units are present in the network, then it will produce an error curve similar to that shown in figure 3.4, in which the desired error level is never attained.

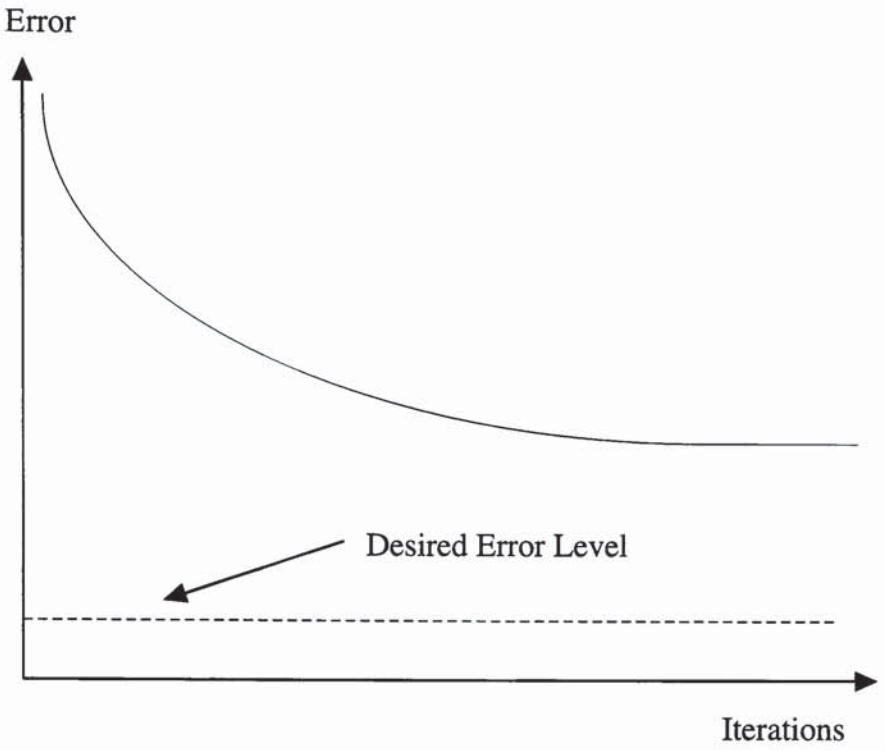


Figure 3.4 - An Error Plateau

The dynamic node creation method starts with a hidden layer of one unit and proceeds with back-propagation learning for a period of n iterations. The ratio of the reduction in error over the n iterations to the original error is calculated and a new hidden unit added if this slope is less than a pre-determined trigger value. Mathematically, a new hidden unit is added if:

$$\left|\frac{a_t - a_{t-n}}{a_{t0}}\right| < \Delta T \qquad 3.12$$

where a_t is the average squared error at time t, a_{t0} is the initial error and ΔT is the trigger slope (Ash, 1989). This process is repeated until the problem is solved to a satisfactory degree of accuracy. Applying this method to several small Boolean problems resulted in minimal architectures in the majority of trials, in a time comparable with standard back-propagation starting from the same architecture.

Meiosis networks, developed by Hanson (1990), also add hidden units into a single hidden layer, but use an algorithm based on the 'Stochastic Delta Rule'. This is based on the idea that neural signals in biological networks are best modelled by a distribution of weight values rather than the more usual single value. This is implemented by the weights in the network being based on a normal distribution with a finite mean and variance. As the learning proceeds, gradient descent is used to update the mean and variance of the weights in the network. An increase in variance is associated with a degradation in performance.

The constructive algorithm is implemented by considering the cumulative value of the weight distributions into each hidden unit. Hanson (1990) states that hidden units that have a high value of standard deviation indicate a high level of uncertainty and are hence performing badly. The meiosis method splits any hidden unit that has a ratio of standard deviation to mean of greater than 1, into two hidden units; half of the associated standard deviation is assigned to each new unit, with both new units retaining the same mean. Starting with an initial value of one hidden unit, the algorithm was applied to low order parity problems and a larger scale problem of blood disease analysis; small networks were produced in a time comparable with stochastic back-propagation. The algorithm was found to be sensitive to the values taken for the parameters and so fine tuning is required to obtain good performance. It is not clear how stochastic back-propagation compares to the standard algorithm and hence the convergence rates presented are difficult to assess.

An alternative approach to the single hidden layer constructive algorithm, is to build deep networks of many hidden layers. Several such algorithms have been developed, some of which are based on a modification of the perceptron learning algorithm called the pocket algorithm. It can be shown that the pocket algorithm is guaranteed to solve any binary problem with a probability approaching unity (Gallant, 1986). The algorithm simply stores 'in one's pocket' the weight vector that has produced the greatest number of correct classifications to patterns randomly chosen from the training set. The current best weight vector is slightly modified (according to the perceptron learning algorithm) and this new

weight vector tested against the old one; if it improves performance it replaces the current 'pocketed' vector, otherwise it is modified again and the process repeated (Gallant, 1990).

Mezard & Nadal (1989) created the 'Tiling Algorithm' which utilises the pocket algorithm to dynamically construct a multi-layered network which is guaranteed to learn any binary mapping. At each stage of the algorithm a new layer of units, consisting of a master unit and several ancillary units, is appended to the network. The addition of such a layer at each stage, building outward from the input layer, is guaranteed to reduce the number of mis-classified patterns by at least one and hence guarantees convergence. The master unit is added first at each stage and is trained using the pocket algorithm to learn at least one pattern that was mis-classified by the previous master unit; the ancillary units are then added in order to ensure that previously learned patterns remain 'faithful' and are not lost. This process is iterated until the newly created master unit correctly classifies every pattern in the training set and hence becomes the output unit of the network. Mezard & Nadal (1989) report that applying this method to several small random mappings results in small efficient networks but that the number of layers (and the units within a layer) increase rapidly as the size of the problem increases. An average of seven layers is required to solve a random mapping of eight inputs with many of the ancillary units within each layer duplicating the functions performed by the ancillary units in the previous layer.

An improvement on the tiling algorithm in terms of the number of units within each layer was introduced by Nadal (1989), who removed the need for ancillary units by connecting each new master unit back into the input layer. Figure 3.5 shows the structure of such a network.

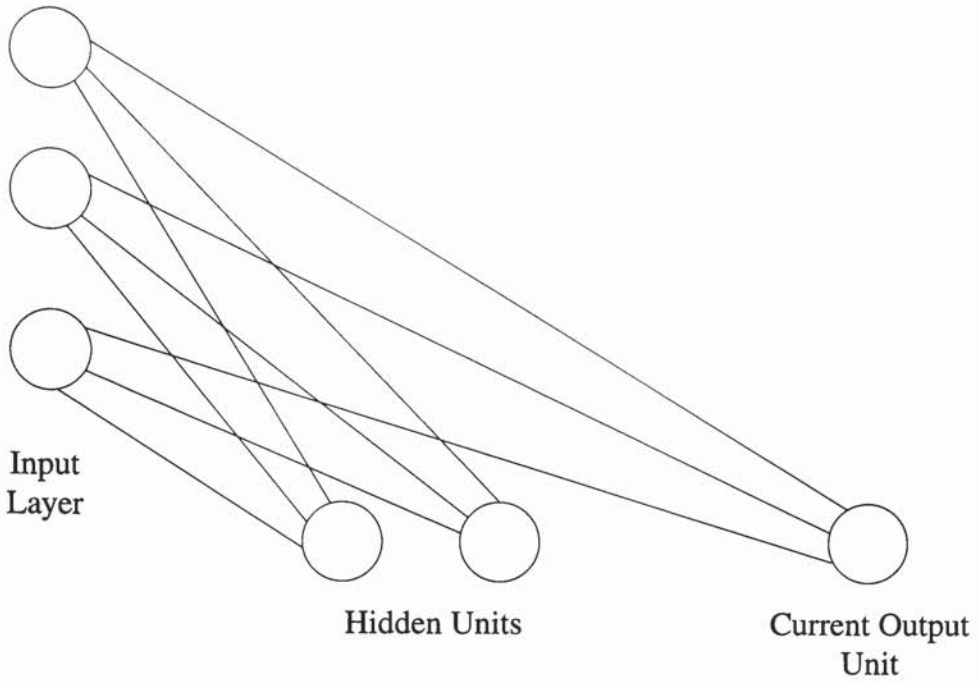


Figure 3.5 - A Network in which Each Hidden Unit Forms a Separate layer

Applying this algorithm to the same problems as the tiling algorithm gave similar results in terms of generalisation but generated very deep networks with a very much larger number of connections (Nadal, 1989)

Frean (1990) also made use of the pocket algorithm in the design of a constructive algorithm to solve binary valued problems. In contrast to the methods discussed above, which extend the network out from the input layer until the correct solution is reached, the 'Upstart Algorithm' inserts units between the input and output layers. If an output unit gives a mis-classification, for example a one where a zero was required, then this could be corrected using a new unit connected to the offending output unit with a large negative weight. This weight is only active for the input patterns mis-classified in this way, with a similar unit introduced to deal with the alternative mis-classification. This forms the basis of the upstart algorithm, which adds in extra units to correct the mistakes of the output unit; these additional units are themselves assigned targets values to achieve and the pocket algorithm is used to learn the new mappings. If mistakes are made by the new units, then further additional units are introduced between them and the input layer in order to correct

the mistakes. Frean (1990) proves that at each insertion into the network, fewer misclassifications are made and hence the algorithm is guaranteed to converge in a finite time.

Frean (1990) applied the upstart algorithm to N bit parity problems and found that the minimal solution of N hidden units was found in all cases. The algorithm also generated fewer hidden units than the tiling algorithm on a wide range of random mapping problems. No time comparisons are given but the algorithm may be altered in such a way as to construct a single hidden layer by connecting all the hidden units inserted into the network to a new output unit at each stage. This removes the large propagation time associated with networks with many layers.

An algorithm which uses a very similar error correction method to that of the upstart algorithm, and does construct a single hidden layer, is the 'Constructive Learning by Specialisation' (CLS) algorithm developed by Refenes & Vithlani (1991). The CLS algorithm builds a hidden layer by introducing a new hidden unit to correct each error as it occurs. A pattern is presented to the network, and a new hidden unit is inserted for each output unit that gives an incorrect response. The additional hidden unit is firstly connected to the offending output unit using a weight of sufficient magnitude to correct the output. Secondly, the new unit is connected to the input units and trained by perceptron learning to recognise this particular pattern but not to respond to any of the previously presented patterns. Once this perceptron based learning is complete for the new hidden unit, its weights are frozen at their current value, and the next pattern is presented to the network (Refenes & Vithlani, 1991). This process is repeated until all the members of the training set have been presented, at which point the network correctly recognises every pattern and hence convergence is guaranteed. Refenes & Vithlani (1991) state that the generalisation performance of the algorithm compares favourably with that of both the tiling and upstart algorithms, but no indication is given as to the relative speed of the algorithm.

The pocket algorithm, and its derivatives outlined above, are only capable of solving binary problems. Whilst it is possible to code real valued functions in a binary form, it is usually more desirable to utilise an algorithm that can learn the continuous mapping in its original form. One such constructive algorithm is the 'Cascade-correlation' algorithm developed by Fahlman & Lebiere (1990). Fahlman & Lebiere (1990) outline two of the major problems of back-propagation based algorithms; the 'step-size' problem which may be eliminated using the quickprop algorithm described in section 3.1.2, and the 'moving target problem'. The latter problem is caused by the fact that each hidden unit is only concerned with its inputs and the error propagated back from the output units, and hence attempts to solve the problem independently of the other hidden units in the network. This independence causes the evolution of each hidden unit in the network to be potentially detrimental to the performance of the other hidden units. The cascade-correlation algorithm is designed to eliminate this problem by only allowing the latest hidden unit to be added to the network to evolve, that is, to change its weights; the remaining hidden units in the network have their weights held constant.

The algorithm begins by training a network with no hidden units until a plateau state is reached, at which point the error level is compared to the desired error level and if it is satisfactory learning is terminated, otherwise a new hidden unit is inserted into the network. A new hidden unit operates on the outputs from all the input units and previously inserted hidden units, and hence constitutes a layer within the network in the same way as that shown in figure 3.5 above. The input weights are adjusted using the quickprop learning algorithm in order to maximise the correlation between the unit's output, V, and the error at each output unit, E_o, (the unit is not yet connected to the output units). This correlation measure is given by:

$$S = \sum_{o} \left| \sum_{p} (V_p - V_{av})(E_{p,o} - E_{av,o}) \right| \qquad 3.13$$

where o runs over each output unit, p runs over each pattern in the training set and V_{av} & $E_{av,o}$ are the values of V & E_o averaged over all patterns (Fahlman & Lebiere, 1990). Once a maximum value of S is attained, the unit is connected to the output units and has its input weights frozen at their current value. The cycle is then repeated with the output weights in the network trained until a new plateau state is reached.

Fahlman & Lebiere (1990) report that rather than use one new hidden unit at each stage, it is possible to use a 'pool' of hidden units from which the unit with the greatest correlation can be selected. On the applications reported, a typical pool of between 4 and 8 hidden units was used at each stage. The algorithm was applied to the interlocking spirals problem (described in Lang & Witbrock, 1988) and converged in an average of 1700 iterations creating an average of 15 hidden units. Fahlman (1988) reports that the quickprop algorithm requires at least 8000 iterations, with each individual iteration being more computationally demanding than those in the cascade-correlation algorithm which only require a forward pass through the network. The algorithm required 357 iterations on the 8-bit parity problem as opposed to 86 using quickprop (Fahlman, 1988), but the quickprop network used 16 hidden units whereas cascade-correlation resulted in networks of 4 and 5 hidden units on average (Fahlman & Lebiere, 1990). Wynne-Jones (1991) notes that whilst the multi-layered network has achieved impressive results due to enhanced feature detection, it has the disadvantage of variable delay between the application of an input pattern and the consequent output.

The constructive algorithms outlined above constitute the most widely known algorithms, but other algorithms have been developed in recent years. Smyth (1991) developed a constructive algorithm which at a point in training chosen by hand, adds a new layer into the network using the technique of 'singular value decomposition'. The algorithm was found to give slightly better results than standard back-propagation, but the author concedes that many layered networks are prone to local minima and have longer training times. There also still remains the problem of deciding on the number of hidden units

required in the initial hidden layer before the constructive process begins. Other constructive algorithms can be found in Gramss (1991), whose algorithm constructively adds feature detecting cells, Honovar & Uhr (1988) who recruit units from a pool as the learning proceeds, Wynne-Jones (1993) who uses principle component analysis to split nodes during the learning process, and Rujan & Marchand (1989) whose approach is derived from a geometric analysis of multi-layered networks.

3.2.3 Hybrid algorithms

Hybrid methods usually take the form of a constructive algorithm to build a network capable of learning the desired mapping, followed by a destructive algorithm to improve generalisation by minimising the architecture. One such algorithm based on standard back-propagation and decision trees has been proposed by Sankar & Mammone (1991). The 'Neural Tree Network' algorithm builds a feed-forward network of nodes that correspond to distinct classes within the training set. Initially the network consists of a root node which is a neural network that separates the training set into distinct regions. 'Child' nodes are then created for each region, and a network at these nodes further divides these regions, continuing this process until each region represents a single class (Sankar & Mammone, 1991a). Once the network has learned the training set to a pre-determined level of accuracy, a pruning algorithm is implemented to improve the generalisation performance. The pruning algorithm attempts to minimise a cost function given by:

$$D_T + \alpha \mid T_L \mid \qquad 3.14$$

where D_T is a measure of the classification error of the full tree, T, T_L is the set of leaf nodes of T and α is the penalty associated with each leaf (Sankar & Mammone, 1991). The algorithm finds the value of α for each node, t, such that retaining the node is preferable to retaining the branch of descendants of t. A set of pruned neural tree networks

is produced and the test set is applied to each in turn to determine the network that produced the best results (Sankar & Mammone, 1991a). The algorithm was applied to a speaker independent vowel recognition task and was found to give a better generalisation performance (54%) than back-propagation (51%) using fewer units (59 as opposed to 88).

Hirose *et al* (1991) proposed a hybrid method based on standard back-propagation, which also builds a network which is subsequently pruned. Starting from a network with one hidden unit, learning proceeds until a local minima is reached and a unit is added into a single hidden layer. A local minima is defined as the failure to achieve a 1% reduction in error over a period of 100 presentations of the training set. Units are added in this way until the convergence criteria is satisfied, at which point the pruning mechanism is implemented. The pruning algorithm records the current weight values in the network and then removes the most recently added hidden unit. If the pruned network achieves convergence then the new network weights are recorded and the process repeated. Once a pruned network fails to converge, the final topology is taken to be that consisting of the last recorded weight values (Hirose *et al*, 1991). The results obtained when applying the algorithm to the XOR problem illustrate one of the deficiencies of such hybrid systems. The number of hidden units in the network gave the pattern 1-2-3-4-3-2-1-2, which whilst achieving the desired minimal architecture, does so after a great many redundant iterations.

An alternative algorithm that does not separate the constructive and destructive aspects in the same way as those described above has been implemented by Chui & Hines (1991). The 'Dynamic Back-Propagation' algorithm implements the two dynamic aspects interchangeably as learning proceeds, starting from a large initial network. The methods are similar to those described above, in that units are added and deleted in a manner determined by the shape of the error curve. Applying the algorithm to the XOR problem resulted in an optimal network of 2-3-1-1 in 796 iterations which is overly complicated, but is probably due to the large initial network size.

3.3 Alternative Approaches

Dynamic approaches that insert or remove units as learning proceeds are the most prevalent methods of addressing this problem, although other methods exist. One such alternative strategy is to use genetic algorithms to determine the network architecture and parameters. Marshall & Harrison (1991) have adopted this approach and have achieved reasonable results. However, as Lynch *et al* (1991) observes, using genetic algorithms to evaluate each network is a computationally time consuming task, as every network in the population for each generation must be trained to completion to test its 'fitness'. Marshall & Harrison (1991) state that their directed search method is much quicker than trial and error by hand, but accept that the method is slow and requires faster processors.

An alternative analytical method that gives a 'ballpark' estimate of the minimum number of hidden units required for a given binary problem was proposed by Gutierrez *et al* (1989). The method estimates the number of 'conflicts' inherent in a given training set, where a conflict is a set of input/output relationships that require incompatible weight solutions when the responses of an output unit are learned on a single layer perceptron. Applied to a variation of NETtalk, a phonetic transcription of English text first demonstrated by Sejnowski & Rosenberg (1987), the algorithm yielded an architecture of 79 hidden units which was actually reduced to 71 since the algorithm gives a 10% over-estimation of results. This network achieved 96% correctness on the training set, but no results were given for generalisation. However it has been shown that the problem may be solved using an architecture containing as few as 45 hidden units. Gutierrez *et al* (1989) propose that their method should be used as a 'first guess' estimate of the number of hidden units and that its application may not extend across the range of possible binary problems.

Kruschke (1989) describes a difficulty arising from using the minimum number of hidden units for a given problem, since the network is more susceptible to noise and damage due

to the fact that each hidden unit is highly significant and must be more accurate. In order to retain the functional properties of the 'local bottle-neck' (i.e. high generalisation performance) whilst not physically reducing the number of hidden units in the network, Kruschke (1989) proposes a method which reduces the dimensionality of the weight vectors and clusters them in a lower dimensional space as learning proceeds. The resulting distributed bottle-neck would retain the properties essential to provide a high level of generalisation whilst utilising a network with sufficient number of hidden units to be relatively immune to damage. The proposed algorithm has yet to be applied and hence remains an untested theoretical proposition.

The alternative methods of architecture selection described above form only a very small percentage of the mainly dynamic approaches that have been developed, and it was felt that a dynamic configuration approach offered both a more profitable way forward, together with an appealing extension of the adaptive nature of neural networks.

Chapter 4

Preliminary Work

4.1 Introduction

The main aim of the proposed project is to develop an algorithm capable of dynamically configuring the network architecture as the learning process proceeds. As was noted in the previous chapter, dynamic methods of selecting the optimum architecture for a given problem have arisen from a need to remove the element of guess-work in selecting the number of hidden units. The process of finding the optimum neural network architecture can be a lengthy one, especially if the problem is large; not only does each architecture have to be tested several times using different starting weights, but the optimum set of learning parameters for each architecture must also be determined. This process is very time consuming since a number of different network architectures must be trained to completion and tested. The dynamic configuration approach also offers the intuitive appeal of complying with the inherent adaptive nature of neural networks.

The algorithm will be implemented as a software simulation for reasons of practicality and time considerations. In line with many of the dynamic configuration methods outlined in Chapter 3, it is felt that an adaptation of the back-propagation method would be the most profitable approach due to the large amount of literature available and the relative simplicity of the algorithm. In adapting the back-propagation algorithm it is desirable that the resulting method performs in a comparable way, both in terms of the number of iterations required to obtain a solution, and the complexity of each iteration performed. Also in line with standard back-propagation, the algorithm developed should be applicable to both binary and real valued training sets, and should as far as possible be problem

independent. Initially the method will be applied to small binary problems and then subsequently tested on 'real-world' problems.

4.2 Implementing Back-Propagation

4.2.1 System Requirements

During the initial stages of the project the back-propagation algorithm was implemented in Modula_2 on a VAX system running under the VMS operating system. Modula_2 was selected as the development language because of its compliance with the software engineering principles of a structured, strongly typed and modular language which makes software development more reliable and debugging simpler. Modula_2 is also relatively portable which proved to be invaluable when high performance SUN Sparcstations became available, utilising RISC technology for greater speed and efficiency. Since back-propagation is a computationally demanding iterative algorithm, the software simulation was subsequently transferred onto the SUN Sparcstations running under UNIX. The UNIX operating system provides the facility to run processes in the background which was of considerable advantage since the learning process can take several hours.

4.2.2 The Algorithm

The implementation of the back-propagation algorithm followed that described by Rumelhart *et al* (1986b), in which the mean squared error (given by equation 2.6) is minimised and a momentum factor is utilised to speed up the learning process. The networks generated by the simulation were fully connected, with thresholds implemented as weights connected to a unit producing a constant output of 1; the thresholds were

modified in the same way as the standard weights in the network which simplified the implementation of the algorithm. The training sets, weights and parameters used by the network were stored as independent files, with the output produced by the network also stored in a file. This allowed the algorithm to be applied to different problems and different parameters in a relatively straightforward manner.

The activation function employed in the network was the exponential function given in equation 4.1.

$$y_j = (1 + e^{-x_j})^{-1} \qquad 4.1$$

On applying the simulation to small binary problems, it was found that as the values of the weights in the network increased in absolute magnitude, equation 4.1 caused an overflow error to occur. In order to remove this problem a check was introduced on the value of x_j; if it is greater in absolute magnitude than 75 then y_j is assumed to be equal to 1 or -1 as appropriate and equation 4.1 is not evaluated.

4.2.3 Validation

Validation plays an important role in any computer simulation and it was crucial for the back-propagation simulator to be thoroughly tested before it was used to obtain results. The availability of standards or bench-marks can provide a reliable means of testing a system and comparing its performance. As noted in section 3.1.2, it can be difficult to compare results for neural network performance since the literature description frequently contains insufficient information to repeat the experiment and replicate the results. However, some suitable bench-marks do exist, notably the XOR problem.

Since Minsky & Papert (1969) demonstrated that the single-layer perceptron was incapable of solving the XOR problem, a great deal of effort has been directed towards this particular problem. Rumelhart *et al* (1986a) state the results of an experiment in which the XOR problem was solved using standard back-propagation to a global error of 0.01. This was achieved using 2 hidden units in a single hidden layer, a learning factor of 0.25, a momentum factor of 0.9 and small initial random weights. Rumelhart *et al* (1986a) report that the number of iterations required to reach an error of 0.01 was found to average about 245. Utilising the same initial conditions, this experiment was repeated and used to test the performance of the simulator. The average value over 12 trials was found to be 288 which is comparable to the above result.

The XOR problem is the two dimensional special case of the parity problem (described in 4.3.3) which is itself a popular problem to evaluate the performance of neural network algorithms. In the case of parity, no published results were found that were detailed enough to make a comparison possible. Instead, validation was achieved by comparing the simulator in this project with that developed independently by a colleague. Both simulators were applied to the six dimensional parity problem (parity6) using the same learning parameters and exactly the same initial weights. Comparing the outputs of both simulators showed that the error values at each iteration were exactly the same (correct to six decimal places). This level of agreement was achieved for several different sets of initial weights for both the parity6 problem and encoder8 problem described in section 4.3.2. These results were taken to indicate that both simulators were functioning correctly as they were independently derived and written in different programming languages.

Once the simulator was deemed to be functioning correctly, it was applied to several binary problems to form a basis of comparison with the proposed constructive algorithm.

4.3 Applying Back-Propagation

4.3.1 Introduction

The selection of binary problems was governed by two factors; firstly that they were of sufficient complexity to require the use of hidden units, and secondly that they did not require an inordinate amount of computer time to process.

4.3.2 The Encoder8 Problem

The encoder problem shown in table 4.1 was chosen as it satisfies both of the above criteria and is a commonly used bench-mark. This version of the problem is referred to as encoder8 as it has 8 input and output units.

Input Vectors	Output Vectors
0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0	0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0	0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0	0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0	0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0	0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0	0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0

Table 4.1 - The Encoder8 Function

In order to learn this function, the network must be presented with all eight possible combinations, and consequently it is not possible to assess the network's generalisation

abilities. However, it was decided that initially it would be beneficial to concentrate on the network's ability to learn the training set when considering small binary problems.

It is generally accepted that the minimum number of hidden units required for this problem is 3, although it has been reported by Fahlman (1988) that 2 hidden units are sometimes sufficient, but convergence is very difficult to achieve. The performance of the network was observed over a variety of points in parameter space, and it was found that a learning rate of 0.25, a momentum factor of 0.9 and an initial weight and threshold range of $0 < W_{i,j} < 0.5$ gave the best performance for several different hidden layer architectures.

Once the parameter values were decided upon, the back-propagation algorithm was applied to architectures containing different numbers of hidden units. Each architecture was trained 20 times to a global error value of 0.1 using different random initial weights; a statistical evaluation was then performed on the results. These results are shown in table 4.2.

Number of Hidden Units	Number of Convergent Trials	Mean Number of Iterations	Standard Deviation
2	0	-	-
3	19	359	54.0
4	20	200	12.1
6	20	157	16.3
8	20	115	14.5
10	20	140	26.5
12	20	190	6.8
14	20	356	129.6

Table 4.2 - Results for the Encoder8 Problem Trained to an Error of 0.1

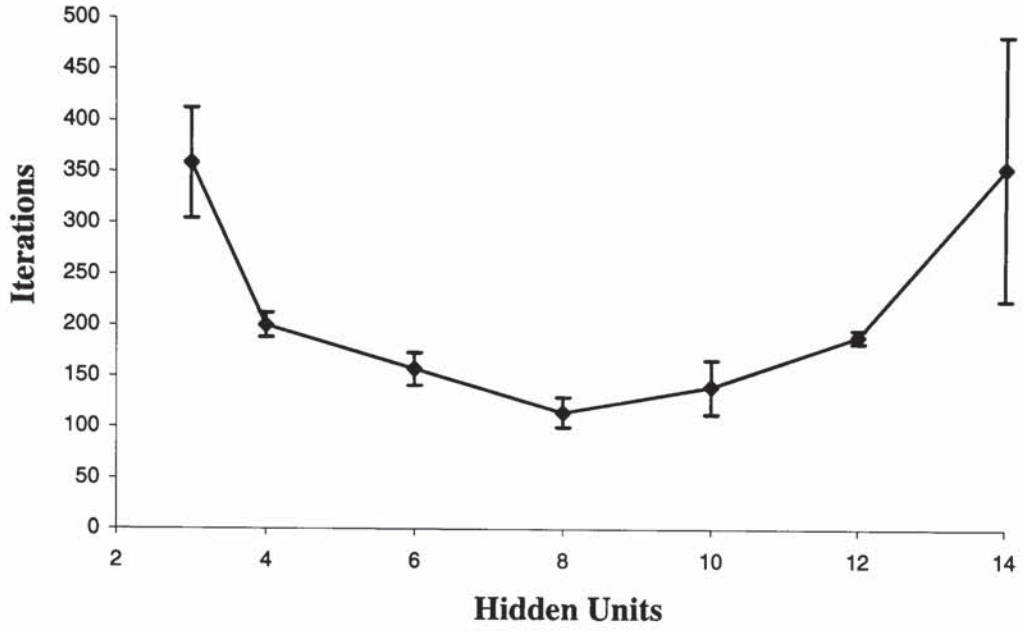


Figure 4.1 - Iterations Vs. Hidden Units for Encoder8 Trained to E = 0.1

Table 4.2 shows mean and standard deviation of the number of iterations required for convergence over the number of convergent trials for several different hidden layer architectures. Figure 4.1 shows a graph of the mean number of iterations against the number of hidden units, with the standard deviation represented as an error bar for each point.

The resulting curve is roughly parabolic in shape, having a minimum at a value of 8 hidden units. In order to determine the effect on the shape of the curve of making the error condition more stringent, the experiment was repeated for global errors of 0.01 and 0.001. As the error was reduced in value, it was found the general shape of the curve was retained but the minimum value was shifted to the right. The curves obtained for error values of 0.01 and 0.001 are shown in figures 4.2 and 4.3 below.

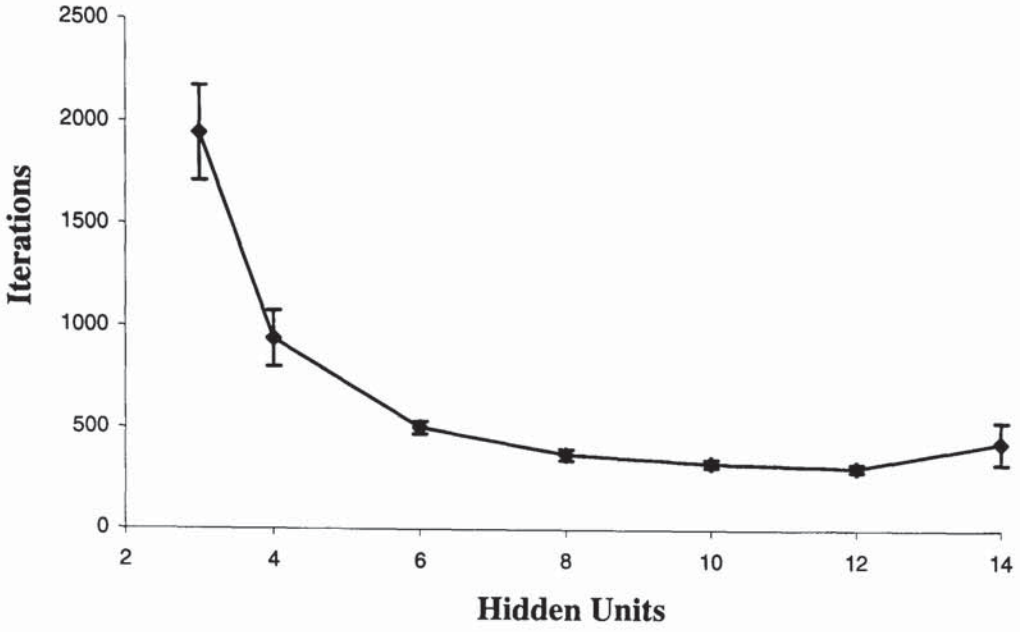


Figure 4.2 - Iterations Vs. Hidden Units for Encoder8 Trained to E = 0.01

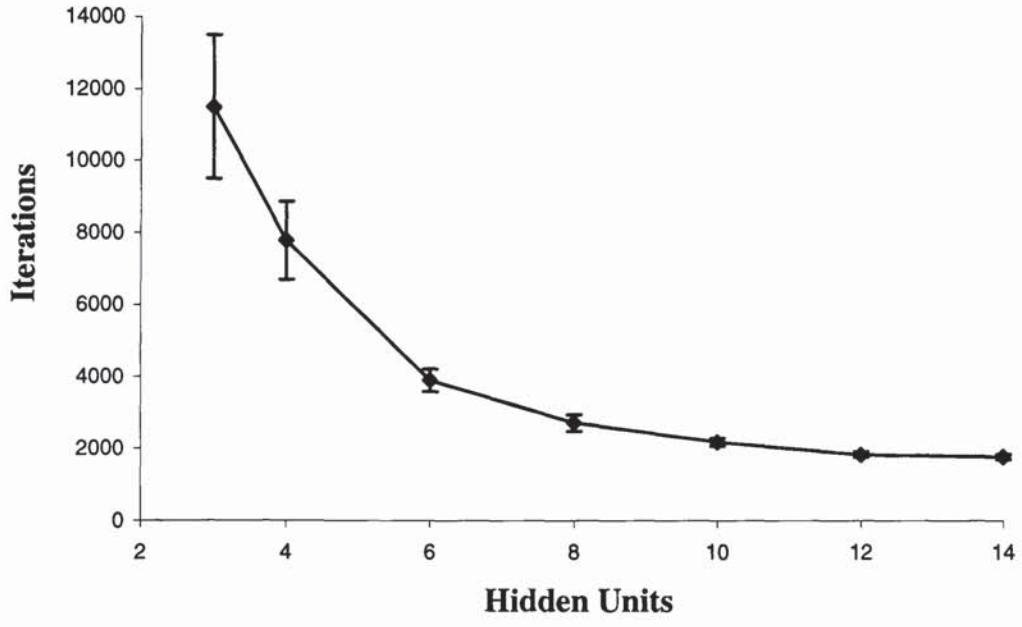


Figure 4.3 - Iterations Vs. Hidden Units for Encoder8 Trained to E = 0.001

Figure 4.2 has a similar shape to the curve obtained for E = 0.1 with a minimum number of iterations obtained for 12 hidden units. Training the network to E = 0.001 failed to give a definite minimum and so networks containing 16, 18, 20 and 24 hidden units were analysed; the results are shown in figure 4.4.

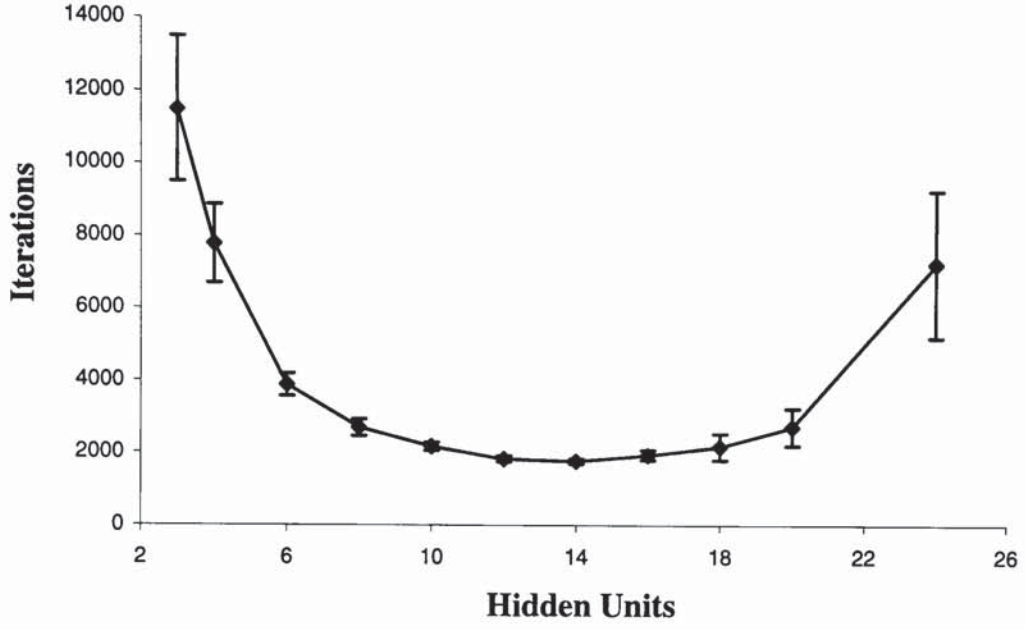


Figure 4.4 - Iterations Vs. Hidden Units for Encoder8 Trained to E = 0.001

This curve has the same parabolic shape as that obtained for an error of 0.1, but has a minimum number of iterations at 14 hidden units. This 'shift' to the right is to be expected because as the error condition is made more rigorous, it becomes more difficult for small networks to force the accuracy through the bottle-neck caused by the limited number of hidden units. Fahlman (1988) states that some researchers have found the same parabolic shape, but it is quite common to find that the average number of hidden units continues to decrease as the number of hidden units is increased. Further experiments were conducted on the encoder problem for different values of learning parameters. Of the further variation of parameters, only one resulted in a curve that continued to decrease as the number of hidden units was increased; this occurred for a small learning rate of 0.05, which resulted in very slow convergence.

4.3.3 The Parity6 Problem

The parity problem is a widely used bench-mark in neural network research, and as such it was felt that it would be appropriate to use in this study. In the terminology of Minsky and Papert (1988), a parity problem having N inputs is an order N problem, and thus a problem of considerable difficulty for relatively small values of N. The particular parity chosen for this study was even, that is the number of 1's contained in the input and output vectors combined must always be even. Table 4.3 shows sample data from the 64 input vectors for the 6-dimensional parity problem (parity6).

Input Vectors	Output Vectors
0 0 0 0 0 0	0
0 0 0 0 0 1	1
0 0 0 0 1 1	0
1 0 1 0 0 0	0
1 1 1 0 0 0	1
1 1 1 1 1 1	0

Table 4.3 - The Parity6 Problem

This problem is difficult to learn because if one bit of the input is altered then the opposite output is required, which nullifies the natural generalisation ability of neural networks. Fahlman (1988) argues that, for this reason, the parity problem is not a good bench-mark for neural networks, but despite this the problem has been extensively used. Given the historical importance and the complexity of the problem, it was decided to utilise parity6 as a bench-mark in this project. Using the entire training set of 64 input vectors, the minimum number of hidden units required to solve the problem is six.

Following the same parameter selection process as that used for the encoder8 problem resulted in a learning factor of 0.3, a momentum factor of 0.9 and initial weights and

thresholds randomly chosen in the range $-2 < W_{i,j} < 2$. Applying back-propagation to the complete parity6 training set over 20 trials to a global error of 0.1 gave the results shown in table 4.4.

Number of Hidden Units	Number of Convergent Trials	Mean Number of Iterations	Standard Deviation
6	0	-	-
7	12	1320	1232
8	13	1448	1318
10	18	777	649
12	19	309	304
14	19	236	222
16	19	180	73
18	19	277	184
20	20	327	322
22	19	245	242
24	19	143	100
30	18	170	163

Table 4.4 - Results for Parity6 Trained to an Error of 0.1

Table 4.4 shows that there were no convergent trials using the minimal architecture of six hidden units, and the learning time for networks of 7 and 8 hidden units far exceeded that obtained with larger networks. As can be seen from figure 4.5, these results are generally more erratic in nature than those obtained for the encoder8 problem.

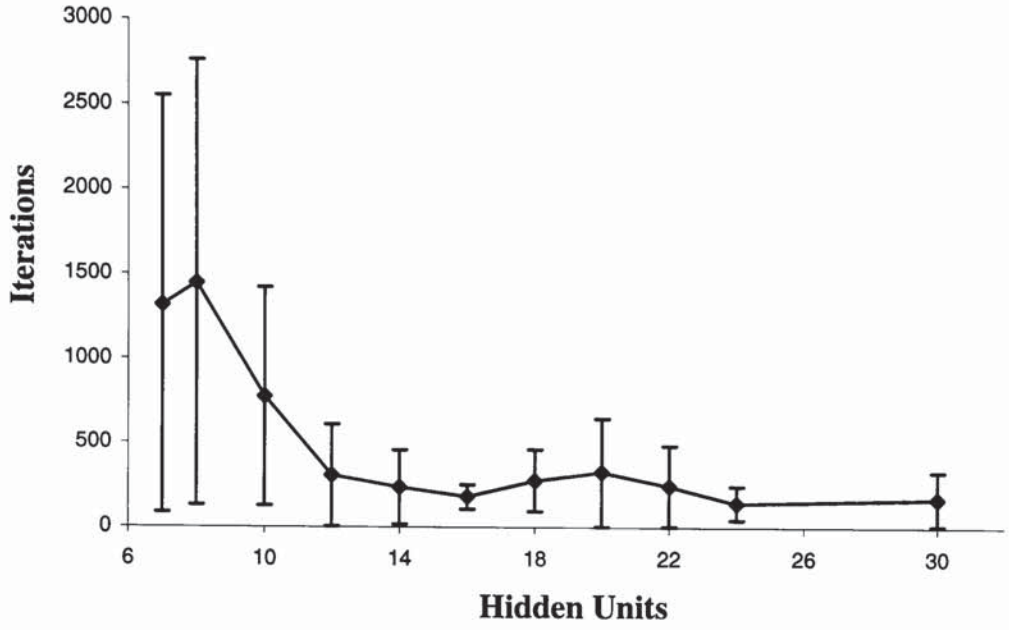


Figure 4.5 - Iterations Vs. Hidden Units for Parity6 Trained to E = 0.1

The erratic nature of these results is also reflected in the calculated standard deviations which are much greater than those obtained for the encoder8 problem. Table 4.5 shows the results obtained by using the more stringent error condition of E = 0.001.

Number of Hidden Units	Number of Convergent Trials	Mean Number of Iterations	Standard Deviation
6	0	-	-
7	9	3233	472
8	11	3544	681
10	18	3266	623
12	19	2674	274
14	18	2448	339
16	19	2447	259
18	19	2339	232
20	20	2422	604
22	19	2302	380
24	19	2212	264
30	18	2039	232

Table 4.5 - Results for Parity6 Trained to an Error of 0.001

The number of convergent trials tended to decrease as the error condition decreased from 0.1 to 0.001; this was because the learning process was terminated if convergence did not occur after 5000 iterations. In contrast to the encoder8 problem, the number of iterations required for convergence did not rise as the number of hidden units was increased, as can be seen from figure 4.6.

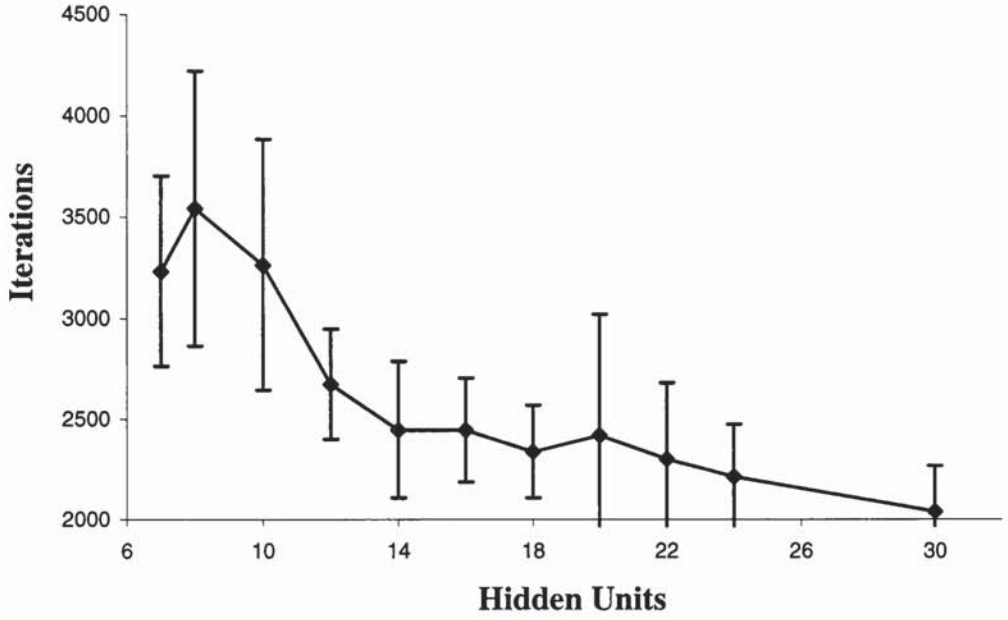


Figure 4.6 - Iterations Vs. Hidden Units for Parity6 Trained to E = 0.001

This type of curve was expected since it has been found in previous studies (Fahlman, 1988). An unusual feature of both the parity6 and encoder8 experiments is the decline in the relative value of the standard deviation as the error condition is made more stringent. As a percentage of the mean, the standard deviation (averaged over all the architectures tried) for the parity6 problem was reduced from 96.8% for E = 0.1 to 14.7% for E = 0.001, with a reduction from 18.8% to 8.8% for the encoder8 problem. This reduction was consistent for both problems, regardless of the architecture considered, and error values taken. This would seem to imply that the variation in the number of iterations required to reach the initial error is greater because of the random starting point in error space; once

this minima is attained the path followed to attain the next error measure within this region of error space is relatively constant.

4.3.4 The 4DC Problem

The third binary problem considered was the 2-or-more clumps problem, which is again widely used by researchers in the field (Denker *et al*, 1987). A 'clump' in this context is defined to be one or more '1's separated by zeros or the edge of the input. Table 4.6 shows some examples drawn from the four dimensional case.

Input Vector	N^{o} of Clumps	Output Vector
1 1 0 1	2	1
1 0 0 0	1	0
0 0 0 0	0	0
1 0 1 0	2	1

Table 4.6 - Examples from the Training Set of the 4DC Problem

Attempting to maximise performance over several different architectures for the 4-dimensional 2-or-more-clumps problem (4DC problem) resulted in a learning factor of 0.25, a momentum factor of 0.9 and weights and thresholds in the range $0 < W_{i,j} < 0.5$. Table 4.7 shows the results of applying back-propagation 20 times with different initial weights and the error minimised to a value of 0.1.

Number of Hidden Units	Number of Convergent Trials	Mean Number of Iterations	Standard Deviation
1	0	-	-
2	3	154	88.5
3	19	105	29.1
4	20	115	45.1
5	20	116	61.3
6	20	121	51.0
8	20	175	159.1
10	19	217	216.1
12	18	461	418.3

Table 4.7 - Results for 4DC Trained to an Error of 0.1

Table 4.7 shows that, in line with the parity6 problem, the convergence rate for the minimum number of hidden units, in this case 2, was very poor. As can be seen from figure 4.7, this problem shares the same parabolic shape as the encoder8 problem with a minimum value at 3 hidden units.

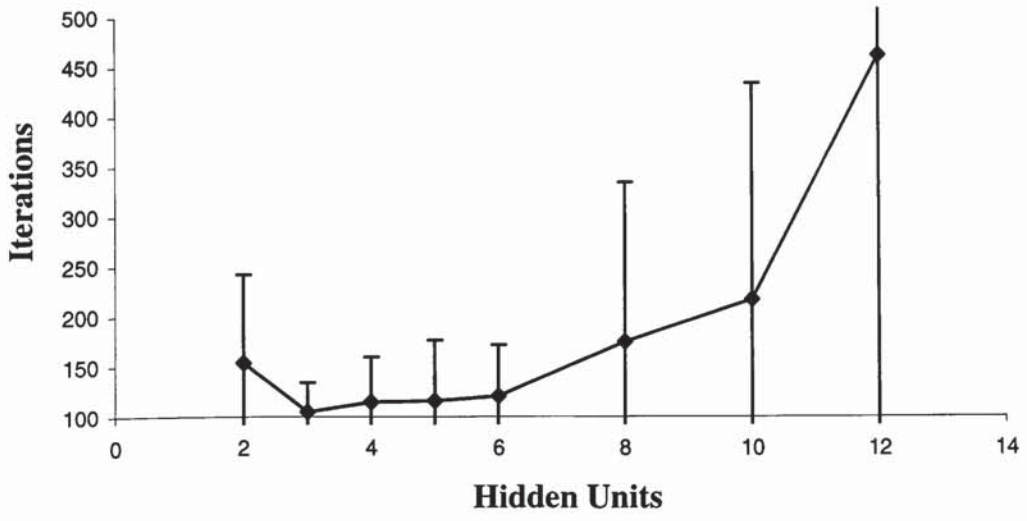


Figure 4.7 - Iterations Vs. Hidden Units for 4DC Trained to E = 0.1

Repeating this experiment for a global error of 0.001 gave rise to the results in figure 4.8.

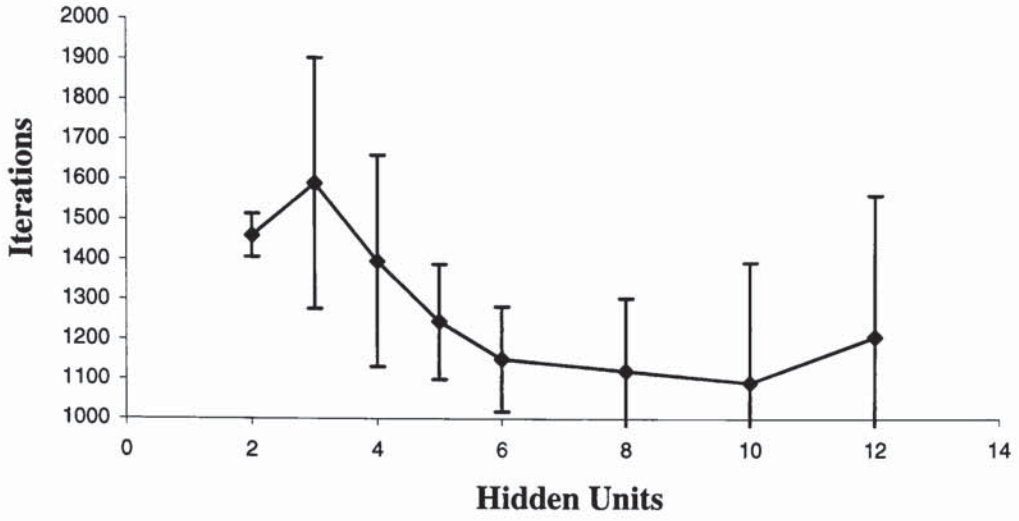


Figure 4.8 - Iterations Vs. Hidden Units for 4DC Trained to E = 0.001

These results are similar to those obtained for the earlier studies in that an increasing number of hidden units are required to cope with the increased accuracy demanded from the network. Furthermore, it was found that the standard deviation followed the same pattern as for both previous problems, reducing from 74.5% to 21.1% for a reduction in error from 0.1 to 0.001.

4.3.5 Summary

Section 4.3 has presented the results of applying the standard back-propagation algorithm to three small binary problems; these results will be used in Chapter 5 as a basis of comparison with the results of the developed constructive algorithm. These results show that applying standard back-propagation using a neural network of minimal architecture is a slow process, often resulting in non-convergence. This project will assess whether the process of using a constructive algorithm to gradually increase the size of the hidden layer has any beneficial effect on the training of networks.

4.4 Constructive or Destructive?

4.4.1 Introduction

Ideally the architecture built by the algorithm should be minimal for each given problem, and should result in the ability of the network to generalise. In practice a near minimal architecture may be satisfactory as long as generalisation performance is acceptable, and will be less computationally demanding than a minimal solution in terms of the number of iterations required for convergence.

The typical effect on the convergence rate as the number of hidden units increases is shown in figure 4.9 below, which shows a representative example of training the parity6 problem to a global error value of 0.001.

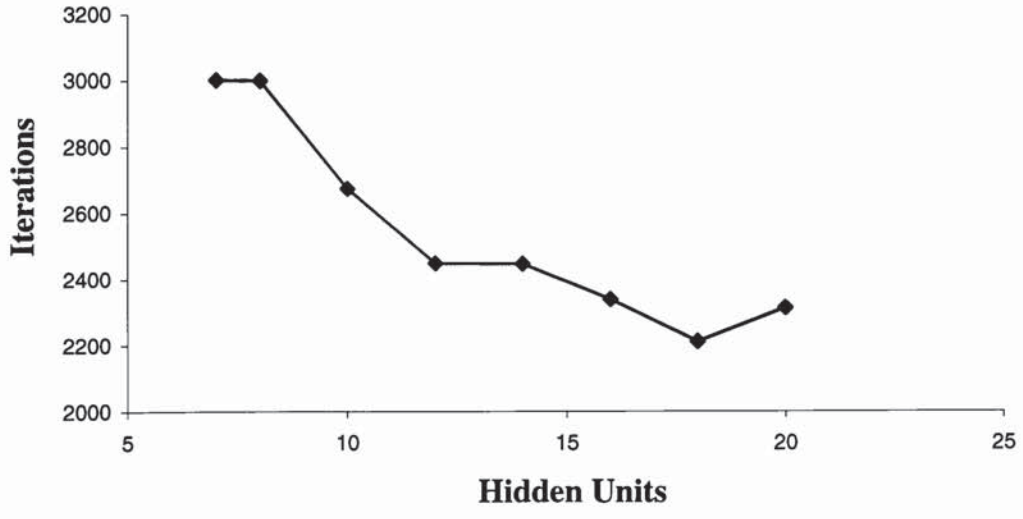


Figure 4.9 - Parity6 Trained to an Error of 0.001

As the number of hidden units is increased the number of iterations required for convergence falls steeply to a plateau, which can persist as the number of hidden units is increased (as shown above) or may rise steeply. As was noted in section 3.1.3, Huyser &

Horowitz (1988) state that in order to achieve perfect generalisation, a network containing the minimum number of hidden units is a necessary condition. However, figure 4.9 shows that the minimum number of hidden units (in this case six) does not correspond to the minimal number of iterations required for convergence and that an architecture consisting of 7 or 8 hidden units would be more pragmatic as it would demand less computational processing. Mozer & Smolensky (1989) also note the computational demand of a minimal solution, stating that a minimum number of hidden units leads to improved generalisation at the cost of a greater number of iterations. It is desirable therefore to strike a balance between the two performance criteria, whilst giving preference to improved generalisation since a network only requires a single appropriate training session.

4.4.2 Destructive Algorithms

The appeal of destructive algorithms lies in their relative simplicity, since it is an intuitively obvious process to remove those elements of the network that contribute least to the performance. The process of destruction relies heavily upon the neural network's ability to sustain large amounts of damage, whilst still attaining a high level of performance, perhaps after a further period of training. Bedworth & Lowe (1988) have shown this in their study of the fault tolerance of the multi layer perceptron, stating that performance can remain above the given error criteria even when hidden units are effectively removed from the network. Mozer & Smolensky (1989) argue that this is due to the network under utilising all the facilities available to it in achieving a solution, in other words a high level of redundancy is contained in the network. Figures 4.10 and 4.11 diagrammatically illustrate the redundancy and hence overcomplexity of a network with a greater number of hidden units than is actually required.

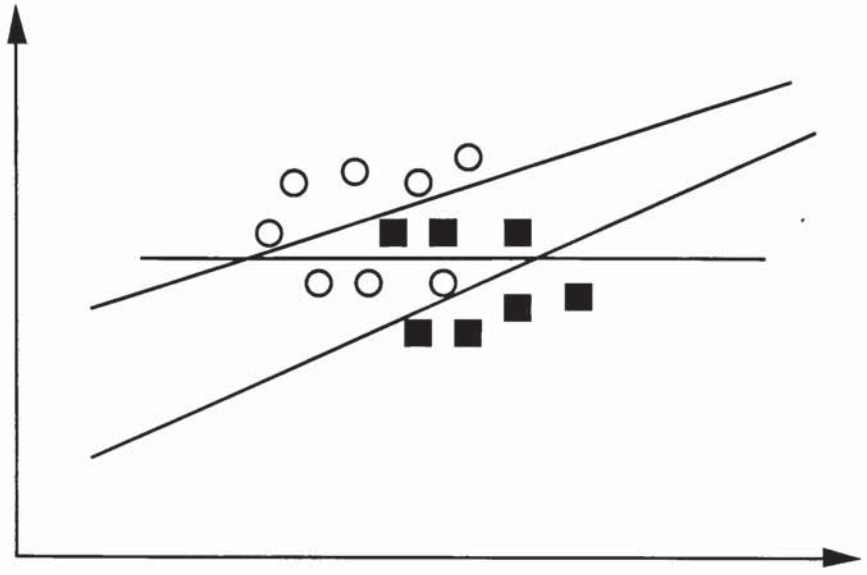


Figure 4.10 - A Feature Space Separated by 3 Hidden Units

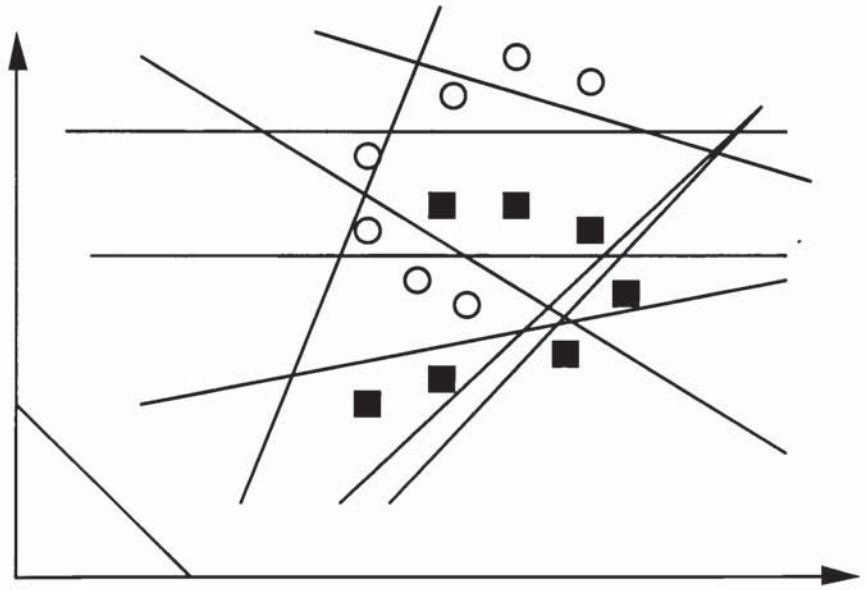


Figure 4.11 - A Feature Space Separated by Many Redundant Hidden Units

Figure 4.10 shows the feature space of a 'mesh' problem consisting of two categories, and the classification resulting from a network containing 3 hidden units. Figure 4.11 shows a back-propagation solution using 16 hidden units, of which 7 are outside the [0,1] unit plane. These 7 hidden units would be relatively simple to prune since they are not contributing to the process of classifying the two categories. However, pruning is only simple in the early stages of the process when there are relatively large numbers of hidden

units in the network. As the units are removed, it becomes increasingly difficult to decide which of the remaining units should be removed. This implies that there is no guarantee that the architecture may not become entrapped in an intermediate solution that satisfies the convergence criteria but is oversized for the given problem, resulting in arbitrary category separation and anomalous generalisation.

The problem of prematurely terminating the reduction process is a symptom of the fact that destructive methods tend to require fine tuning of the pruning parameters (Le Cun, 1989). If the pruning condition is too stringent then the network is reduced to an architecture that is no longer functionally capable of solving the given problem. Whereas a relaxed pruning condition may result in an overly large architecture, with the resultant loss of generalisation capability.

Computationally, destructive algorithms are inefficient in that they are dealing with architectures that are oversized for the major portion of the learning process (Gramss, 1991). An example noted in section 3.2.1 illustrates this; Sietsma & Dow (1991) pruned a network from a size of 64-20-8-3 down to 64-8-3 which implies that the vast majority of the original weights were unnecessary. The number of connections contained in a given architecture is a major factor in the computational demands of a given algorithm, since much of the training time is spent in updating the weight values. Le Cun (1989) also states that destructive algorithms require a great many iterations to achieve the convergence criteria, most of which are performed with networks that are too large. Dealing with larger networks has the added disadvantage that the error surface tends to be over-complicated, resulting in an increased likelihood of becoming entrapped in local minima (Smythe, 1991). Hanson & Pratt (1989), for example, report that introducing a weight decay method of pruning resulted in 75% less convergent runs, compared to standard back-propagation.

Honovar & Uhr (1991) states that dynamic configuration algorithms possess the highly desirable feature of removing the need for *a priori* knowledge of network architecture.

Destructive algorithms still require the user to over estimate the number of hidden units to solve a given problem. This is contrary to the aim of dynamic algorithms, which should remove the need for hidden unit estimation.

4.4.3 Constructive Algorithms

Despite the intuitive simplicity of a destructive approach, it was felt that constructive algorithms would be more appropriate given some of the limitations of the destructive methods outlined above. Constructive algorithms retain the advantage of requiring no *a priori* knowledge of hidden layer structure, since they may be initialised with a minimal structure regardless of the problem being considered. This level of problem independence provides an important benefit by removing one level of user intervention, and offers a potentially powerful approach in combining the re-weighting of links with dynamic network configuration.

In the same way that destructive algorithms mirror the biological capability of the brain to sustain large amounts of damage whilst retaining high performance levels, constructive algorithms mirror the brain's ability to create new structures. There is strong evidence to support the view that biological neural networks generate new structures throughout life in response to learning stimuli (Greenough & Bailey, 1988), and whilst it is not the aim of this project to produce a biologically plausible algorithm, it is one of the motivations behind constructive algorithms in general.

Constructive algorithms are theoretically computationally efficient due to the fact that a large proportion of the learning process is performed with networks that are smaller than the final (hopefully minimal) network architecture. Since most of the learning process is performed with small network configurations, the error space is relatively simple and hence less prone to the problem of local minima. If, however, during this process a local minima

is encountered, then the addition of a new unit should provide sufficient perturbation to the current network state to escape from the minimum. Wynne-Jones (1991) notes that in implementing constructive algorithms it is hoped that at each stage of the training process, the network approximates a solution which is incrementally improved with the addition of each new unit into the network.

As noted in section 3.2.2, this is true of those algorithms based on the pocket algorithm, which reduce the number of errors made at each stage by at least one with the addition of further units, therefore guaranteeing eventual convergence. Unfortunately these methods are applicable only to binary problems, which is a major limitation since they form only a small fraction of 'real-world' problems. These problems would require a conversion to a binary representation in order to apply the algorithms; a process which would inevitably increase the dimension of the input layer. Nadal (1989) reports that both the tiling algorithm and his subsequent improvement generate increasingly deep networks, the tiling algorithm rapidly increasing the number of units within each layer. This would make the application of these algorithms to large real valued problems prohibitive due to the computational demands involved. This project aims to produce an algorithm that is applicable to both binary and real valued problems and it is felt that this is best served by using analogue units .

A constructive algorithm that utilises analogue units and is hence directly applicable to real valued problems is cascade correlation (Fahlman & Lebiere, 1990). The method (described in section 3.2.2) introduces each new unit as a separate layer and hence results in a multi-layer structure. Whilst this results in a higher order of feature detection, the learning process in multi-layer networks can be slowed considerably with the insertion of a new layer (Smythe, 1991). This is due to the large number of connections that are introduced with the addition of each new layer, and which is further exacerbated by the fact that the degree to which a weight is altered during the backward pass of an algorithm is scaled by an amount which is less than unity at each layer. Thus, a network with a great many layers

will make increasingly smaller changes as the error is propagated backwards. Fahlman & Lebiere (1991) avoid the problem of diminishing weight updates by freezing the existing network structure on the introduction of each new unit. Whilst this seemingly restricts the area of the search space, the reported results seem to indicate that this is not problematic, although the algorithm does retain the problem of a variable delay between application of an input vector and production of a result. Since the proposed project is to be based on the back-propagation algorithm, which is computationally demanding in its own right, it was decided that the structure would be restricted to a single layer of hidden units. As has been noted earlier, such architectures are capable of learning any function of interest.

An algorithm that constructs a single hidden layer has been developed by Ash (1989). This method essentially adds hidden units when the learning curve attains a plateau state above the desired error level, the assumption being that there are insufficient hidden units present in the network and that convergence will never be attained. The drawback associated with this method of unit addition is that such plateau states may occur during the course of learning when sufficient numbers of hidden units are present in the network architecture. Figure 4.12 shows the learning curve obtained from the back-propagation algorithm applied to the parity6 problem with 8 hidden units (7 hidden units are sufficient to solve this problem, as shown in table 4.4).

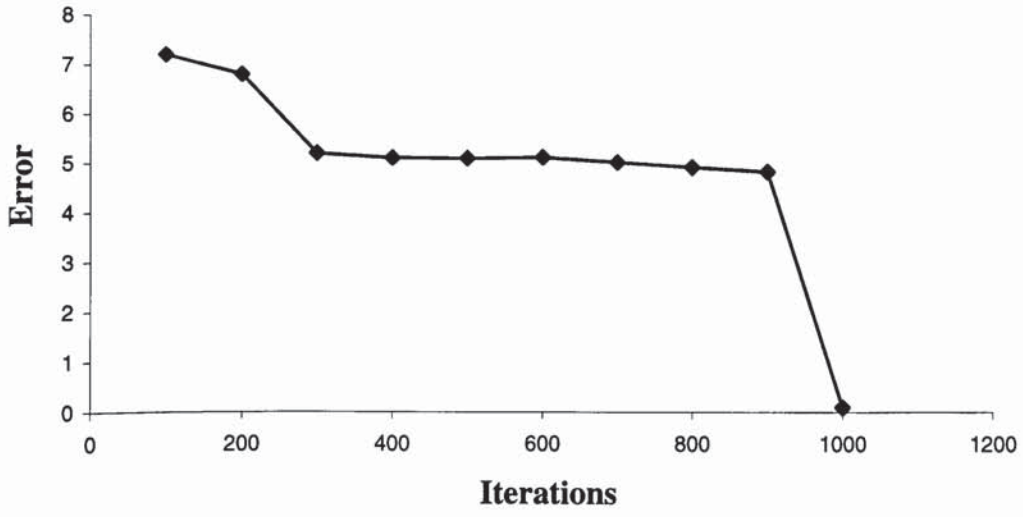


Figure 4.12 - Learning Curve for Parity6 with 8 Hidden Units

As can be seen from figure 4.12, the error curve shows a distinctive plateau suggesting that convergence will not occur, but this is followed by a rapid reduction in error sufficient to meet the desired error criteria. The addition of a further hidden unit during the plateau stage would therefore be unnecessary and possibly detrimental to the network's ability to generalise. One reason that these plateaus occur is that the network enters a long narrow ravine in error space, a typical example of which is shown in figure 4.13.

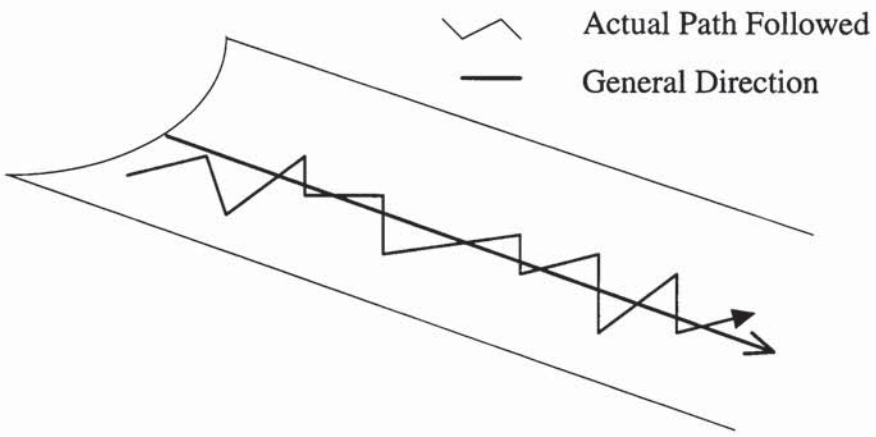


Figure 4.13 - A Narrow Ravine in Error Space

The network oscillates up and down the sides of the ravine because the gradient of the sides are much greater than those in the desired direction along the floor of the ravine. Ash (1989) overcomes this problem by using a large decision window of 1000 iterations, that is to say 1000 iterations elapse before a decision is made whether to add a further unit. For the small problems studied, this size of window seemed to be sufficiently large to capture delayed reductions in error. Unfortunately the approach is costly since it requires, by its very nature, a large number of iterations to achieve convergence.

Hirose *et al* (1991) follow a similar approach using a window of 100 iterations, although this resulted in a network with too many hidden units. They rectified this by applying a pruning algorithm once learning was complete. However, this leads to a prolonged training

phase since the architecture resulting from the constructive phase is then subjected to the pruning algorithm. In developing a constructive algorithm, we aim to avoid the necessity of a pruning stage by terminating the addition of hidden units at a near minimal architecture.

4.5 Method

4.5.1 Introduction

This section will describe the basis of a constructive algorithm, using the back-propagation method, which uses the errors of individual patterns in the training set to determine whether further hidden units are required in the network architecture. As noted above, the desired minimal architecture can be difficult to recognise and computationally expensive to attain. We aim to ascertain whether the smaller networks considered on the way to the final network make a positive contribution to the learning process. This may be done using a comparison of the number of iterations required with the final architecture, and the number required for standard back-propagation using this architecture. Ash (1989) has reported that the incremental development of the hidden layer is beneficial to the learning process for small binary problems, however no results were presented for large real valued problems which will be considered here.

The following sections describe the general network architecture, together with the training procedure and methods of parameter selection used for the problems in this study.

4.5.2 Individual Pattern Error

The normal procedure when applying the back-propagation algorithm to a given problem is to minimise some measure of the global error over all patterns present in the training set. It is not usually the case that the error generated for an individual pattern is analysed or utilised during the learning process. However, one such analysis given by Yu & Simmons (1990a) showed that it is possible to have a small global error whilst having high individual pattern errors. They introduced an extension of the back-propagation method called 'The Descending Epsilon Technique' which gradually makes the error condition for each pattern in the network more stringent as learning proceeds. This method, which is only applicable to binary problems, only propagates the error backwards for a given pattern if it is greater than epsilon. Errors less than epsilon are ignored. Once the error for every pattern is below epsilon, then the value of epsilon is reduced. Yu & Simmons (1990a) state that *'this gradual error reduction prevents the network from reducing some errors too rapidly at the expense of increasing other errors'.*

This 'see-saw' effect is also described by Fahlman & Lebiere (1990) who refer to it as the 'moving target problem'. They assert that reducing the error for one pattern may adversely affect the errors of some of the remaining patterns because the hidden units do not communicate with each other and only respond to local changes as the error is propagated backwards. Fahlman & Lebiere (1990) give the example of a training set that contains two computational sub-tasks A and B. If A generates a stronger error signal than B then the weights are likely to evolve in such a way as to tackle sub-task A at the expense of sub-task B. Once A is satisfactorily solved the network will move towards solving B which will be detrimental to sub-task A. This see-saw effect will repeat until both tasks are solved or the system settles into a local minima.

This effect is a global version of the interaction between the errors for individual patterns between consecutive iterations. In the above example the update of a pattern in sub-task B will degrade the performance of a pattern in sub-task A within one iteration, but the overall effect will be that sub-task A dominates due to its greater 'pull'. Whilst this effect occurs in the normal learning process with sufficient hidden units, it must occur persistently using an insufficient architecture because by definition it is incapable of classifying all the patterns correctly. A network with insufficient hidden layer facilities to solve a given problem will tend to see-saw between the different aspects of the training set, without being able to reconcile the differences between them. Figure 4.14 shows a simple example of this phenomenon.

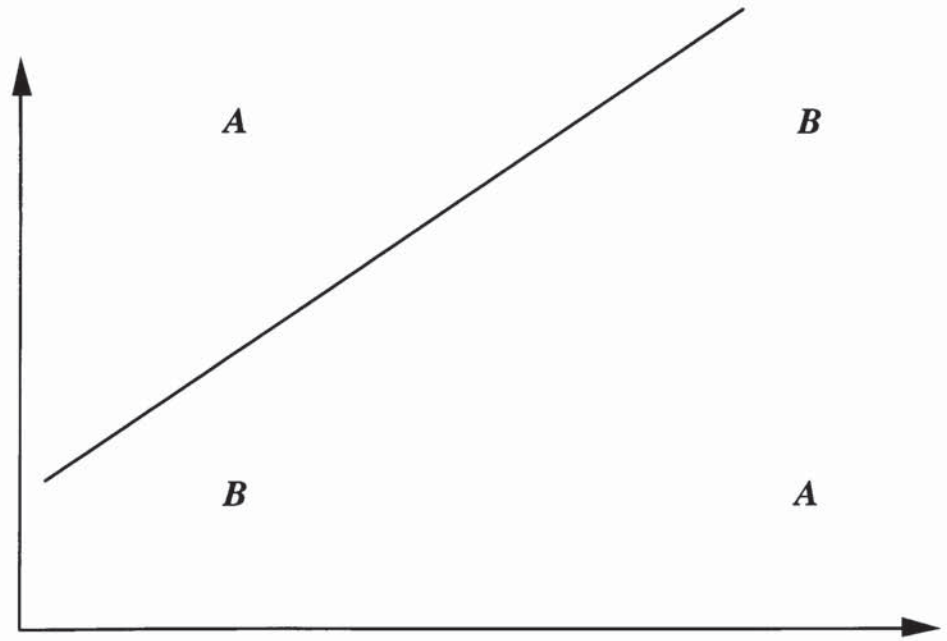


Figure 4.14 - A Two Category Feature Space

The diagram shows a feature space containing two categories A and B, and the decision surface provided by a single hidden unit. Clearly, the separation provided by the hidden unit is insufficient to solve the problem; a hidden layer consisting of two hidden units would be the minimum architecture for this particular problem. In this particular case during the learning process, adjusting the weights for the pattern belonging to category A

that is eventually mis-classified as belonging to category B will have a detrimental effect on the remaining patterns in the training set.

In terms of the back-propagation algorithm, a pattern is normally presented to the inputs and then all the weights updated before the next pattern is considered. This will normally decrease the cost function (equation 4.2 below) at each step allowing successive steps to adapt to the local gradient.

$$E = \frac{1}{2} \sum_{c} \sum_{j} (y_{j,c} - d_{j,c})^2 \qquad 4.2$$

However in certain circumstances, for example when the architecture of the network is insufficient to describe a particular set of functions to a given accuracy, then it might be expected that certain of the patterns in the training set might be in conflict. The simplest case of this is shown in figure 4.14 above. Given that the back-propagation update rule, shown in equation 4.3, is local and takes no account of global information, we might expect that the weights could be modified to accommodate one pattern at the expense of other patterns in the training set.

$$\Delta \omega_{j,i} = -\eta \, \delta_j \, y_i \qquad 4.3$$

This would be reflected in an increased error when the pattern was next presented to the network. If this phenomenon was observed to a significant extent over a period of time then it might suggest that the network architecture should be extended by the inclusion of a further hidden unit.

This project will determine whether or not it is possible to use this effect as the basis of a constructive algorithm. This will be possible if the see-saw effect is significantly more pronounced in networks with insufficient hidden units.

4.5.3 Using Individual Pattern Error as a Basis of Unit Insertion

As a preliminary test of the magnitude of this effect, the back-propagation algorithm was applied to the encoder8 problem (described in section 4.3.2) using various numbers of hidden units contained in a single hidden layer. Within each 100 iteration period, the number of occurrences of the error increasing between consecutive iterations for any of the 8 input patterns was recorded and totalled. An example of the results obtained are shown in table 4.8 below.

	Number of Error Increases Between Consecutive Iterations				
Number of Hidden Units	Iterations 0 - 100	Iterations 100 - 200	Iterations 200 - 300	Iterations 300 - 400	Iterations 400 - 500
1	400	192	315	507	543
2	239	85	334	274	123
3	244	48	116	205	19
4	220	0	0	0	0
5	240	0	0	0	0

Table 4.8 - Number of Increases in Each 100 Iteration Period for Encoder8

The minimum number of hidden units within a single hidden layer required to solve the encoder8 problem is 3; as can be seen from the above table the number of increases in error between consecutive iterations in each 100 iteration period is far greater for 1 and 2 hidden units than is the case for 3 or more. Using 4 and 5 hidden units, the errors for all the patterns never increased after the first 100 iterations. Repeating this process for different starting weights produced similar results; a typical example is shown in table 4.9.

	Number of Error Increases Between Consecutive Iterations				
Number of Hidden Units	Iterations 0 - 100	Iterations 100 - 200	Iterations 200 - 300	Iterations 300 - 400	Iterations 400 - 500
1	356	213	398	402	512
2	243	132	209	389	237
3	245	237	34	217	3
4	276	48	8	2	0
5	179	34	4	0	0

Table 4.9 - Number of Increases in Each 100 Iteration Period for Encoder8

The results shown in table 4.9 are similar to those shown in table 4.8 in that there is a marked and consistent reduction in the number of errors as the hidden layer architecture is built up, showing that the see-saw effect is much more pronounced for networks containing insufficient hidden units. Tables 4.10 & 4.11 show a similar set of results obtained for the parity6 problem.

	Number of Error Increases Between Consecutive Iterations				
Number of Hidden Units	Iterations 0 - 100	Iterations 100 - 200	Iterations 200 - 300	Iterations 300 - 400	Iterations 400 - 500
1	5012	5325	5678	5876	5587
3	4231	3258	3542	4124	4324
7	3756	2800	3246	3123	2780
10	3345	2012	1879	1755	1236
12	3012	2023	1786	1054	1482

Table 4.10 - Number of Increases in Each 100 Iteration Period for Parity6

Table 4.10 shows that the number of error increases was reduced as the number of units in the hidden layer was increased beyond 6, the theoretical minimum required.

	Number of Error Increases Between Consecutive Iterations				
Number of Hidden Units	Iterations 0 - 100	Iterations 100 - 200	Iterations 200 - 300	Iterations 300 - 400	Iterations 400 - 500
1	5234	4876	5898	5974	5924
3	4475	5468	5014	4654	4986
7	3127	2874	3125	3874	3046
10	2056	1802	1435	1563	1024
12	1828	1029	1246	1425	1158

Table 4.11 - Number of Increases in Each 100 Iteration Period for Parity6

The reduction in the number of error increases was not as great for the parity6 problem as for the encoder8 problem, but as noted in section 4.3.3, parity6 is a more difficult function for a neural network to model. However, the number of error increases between consecutive iterations was clearly reduced for networks with larger hidden layers, which suggested that this effect could be used as the basis of a constructive algorithm.

4.5.4 The Network

The initial network structure will be a fully connected architecture with one unit in the hidden layer as shown in figure 4.15 below.

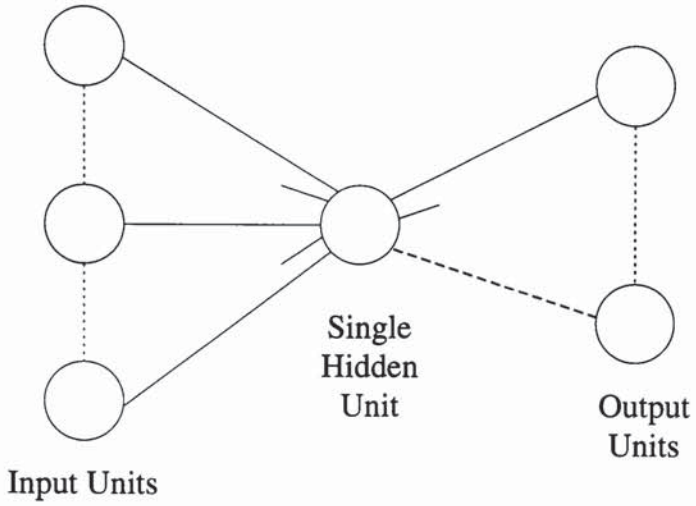


Figure 4.15 - The Initial Network Architecture

The dimensions of the input and output layers are dependent on the problem to be considered with the hidden layer dynamically configured as learning proceeds. It would be possible to initialise the network with a perceptron type architecture, having no hidden units, but the problems to be considered will not be linearly separable and hence will require at least one hidden unit. It may be beneficial, in the case of larger problems, to initialise the network with a larger number of hidden units, but initially a uniform starting architecture containing a single hidden unit will be utilised. Additional hidden units will be added when required into the single hidden layer and be fully connected to both the input and output layers. Only single hidden layer architectures will be considered since these avoid the large propagation delays and slower learning associated with multi-layered networks.

4.5.5 Parameter Selection

The standard back-propagation algorithm has been applied to a number of test problems in order that a comparison can be made with the results obtained by applying the dynamic configuration algorithm. The back-propagation algorithm was applied to architectures

containing a varying number of hidden units so that the optimum architecture was determined for each problem. Additionally, the optimum values for both the learning factor and momentum factor were determined for each problem, together with the optimum ranges for the initial random weights and thresholds.

Chapter 5

Developing and Applying a Constructive Algorithm

5.1 Developing a Constructive Algorithm

5.1.1 Introduction

Chapter 5 will describe the development of a constructive algorithm based on the back-propagation algorithm and its application to several problems of varying types and sizes. The necessary modifications to back-propagation will be described, together with the preliminary experiments performed to derive the detail of the constructive algorithm. The results of applying the algorithm to the binary problems outlined in Chapter 4 will be presented.

Several adaptations made to the basic algorithm will be given. These include adjusting the range of the added weights (including a study of the sensitivity of selecting a threshold value), freezing the existing weights in the network on the addition of a new unit and modifications designed to speed up the learning process.

5.1.2 Adapting Back-Propagation

Following the implementation of the standard back-propagation learning algorithm described in chapter 4, procedures were introduced to facilitate dynamic configuration. At this stage, a generic approach to network configuration was taken and hence procedures to add a hidden unit, delete a hidden unit and delete a weight were written. The procedure to

add hidden units was implemented in such a way as to add the new unit at the 'high' end of the layer in order to minimise the work done in inserting the unit (e.g. a new unit inserted into the network shown in figure 2.4 would be unit 8). The weights for the new unit were initialised in the same range as the initial weights in the network, and the previous weight changes used in the momentum term in equation 2.13 were set to zero.

All three additional procedures were thoroughly tested to ensure correct functionality. In this case, validation by comparison with existing bench-marks or other programs was not possible, and hence great care was taken to ensure the procedure's correct performance by printing the existing weights in the network, applying the appropriate procedure and then printing the relevant details of the new network.

5.1.3 Developing a Constructive Algorithm

Section 4.5.3 outlined the basis of a method for using the 'see-saw' effect between individual patterns in the training set to decide on the necessity for adding further units into a single hidden layer. The change in error between consecutive iterations was shown to predominantly increase for networks containing insufficient numbers of hidden units. As the number of hidden units increased beyond the minimum number, the error was shown to predominantly decrease for each pattern in the training set between consecutive iterations.

A modification to the above approach was made as follows: immediately a pattern is presented and the weights updated, the error for the pattern is calculated; this process is then repeated for the remaining patterns in the training set, the patterns being presented in a fixed cyclic manner. The error for each pattern is then calculated immediately before it is next presented to the network and hence before the weights are next altered in its favour. This value is then compared to the error previously determined and the magnitude of

change noted. The weights are then updated and the process repeated for the next pattern. In formal notation this is given by equation 5.1 below:

$$E_{p,n} - E'_{p,n+1} \qquad 5.1$$

where $E_{p,n}$ is the error for pattern p after weight update n, and $E'_{p,n+1}$ is the error for pattern p calculated immediately before weight update n+1.

This modification offers a more immediate indication of how each pattern is affected by the other patterns in the training set. This is because the comparison is carried out between an error measured immediately after a weight update, and the error recorded immediately before the next update for the same pattern. Using this method allows a more direct measurement of the effect that the patterns in the training set have on each other during the learning process. If the network has sufficient hidden units, then updating the weights after presenting a pattern should not significantly affect the remaining patterns in the training set.

In order to measure how patterns in a training set influence each other during the learning process, standard back-propagation was applied, using several different architectures and initial weights, to the three binary problems studied in chapter 4 namely encoder8, parity6 and 4DC. During each trial, the total number of times the error increased between measurements, the total size of the increases and the average increase over each 10 iteration period were recorded. The figure of 10 iterations was initially chosen as it provides a large number of data points within a relatively small total number of iterations, in this case 200. Figure 5.1 shows a graph of the total number of times in each 10 iteration period that the error increased when applying back-propagation to the encoder8 problem with a network architecture consisting of 8 input units, 1 hidden unit and 8 output units (i.e. an 8-1-8 architecture).

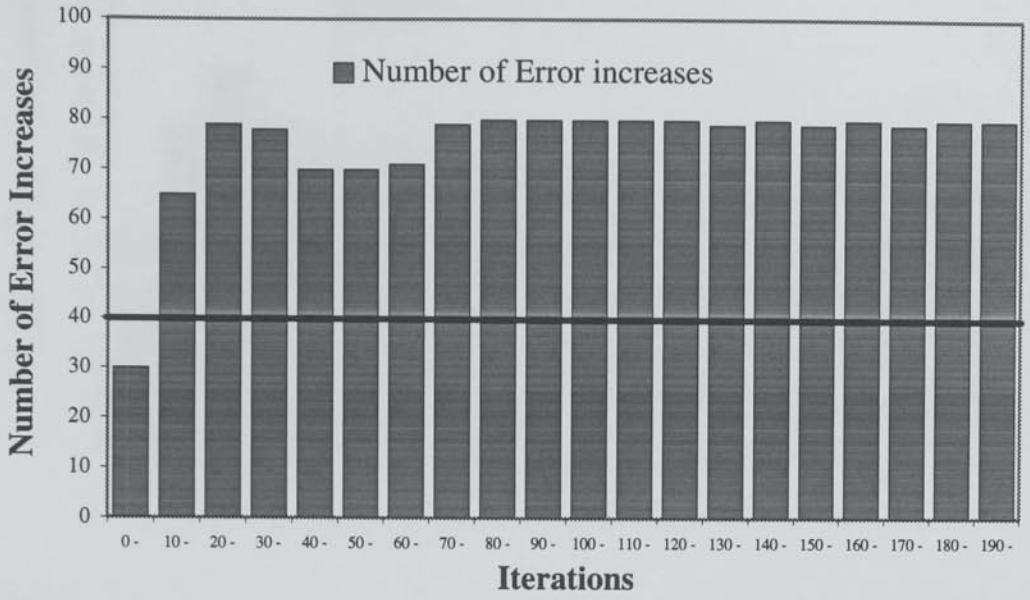


Figure 5.1 - Total Number of Error Increases for Encoder8 with 1 Hidden Unit

Figure 5.1 shows that the error increased between measurements in the majority of cases after the first 10 iterations. The result for the first 10 iterations was due to the network starting the learning process from a poor initial position in error space, from which improvement was inevitable. As the number of hidden units was increased towards the minimal architecture required to solve the problem, the total number of times that the error increased between consecutive measurements (hereafter referred to as 'error increases') was reduced. This is shown in figures 5.2 and 5.3 which show the number of error increases for the encoder8 problem using architectures of 2 and 3 hidden units respectively.

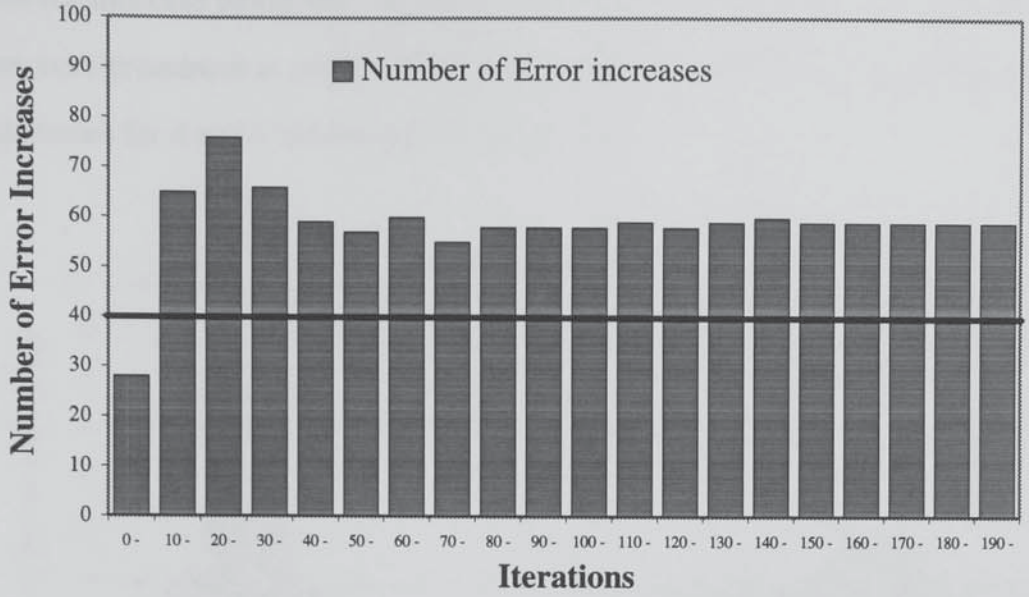


Figure 5.2 - Total Number of Error Increases for Encoder8 with 2 Hidden Units

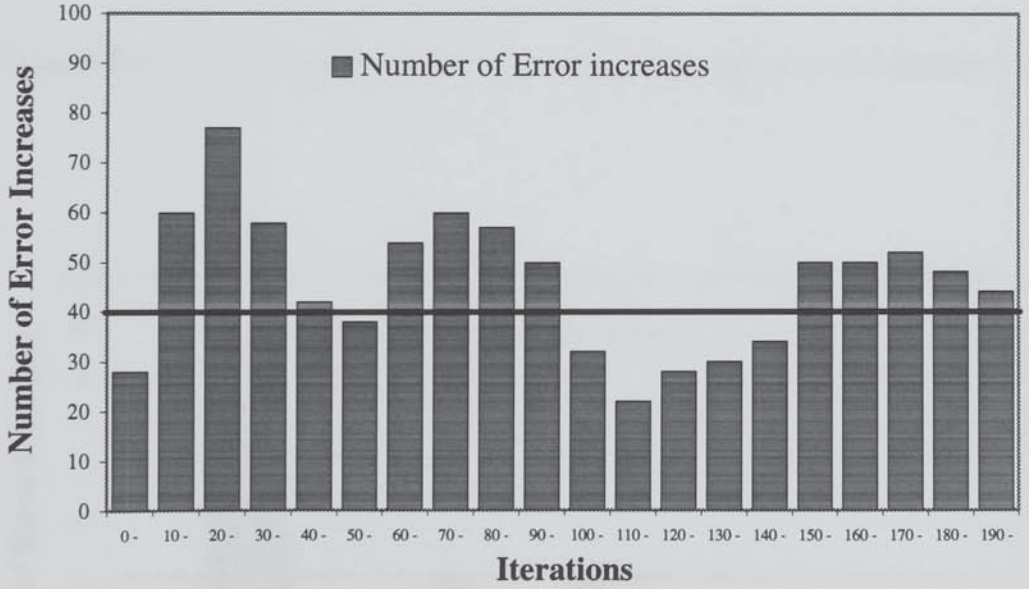


Figure 5.3 - Total Number of Error Increases for Encoder8 with 3 Hidden Units

Using 2 hidden units, the error predominantly increased, although less than for the network containing 1 hidden unit. Figure 5.3 shows that using a hidden layer architecture of 3 hidden units, which is a sufficient number to solve this problem, the number of error

increases fell below 50% for some 10 iteration periods (the total number of error increases possible for this case being 80). As the number of hidden units was increased, this effect became more prominent as can be seen from figures 5.4 and 5.5 which show the number of error increases for 4 and 5 hidden units respectively.

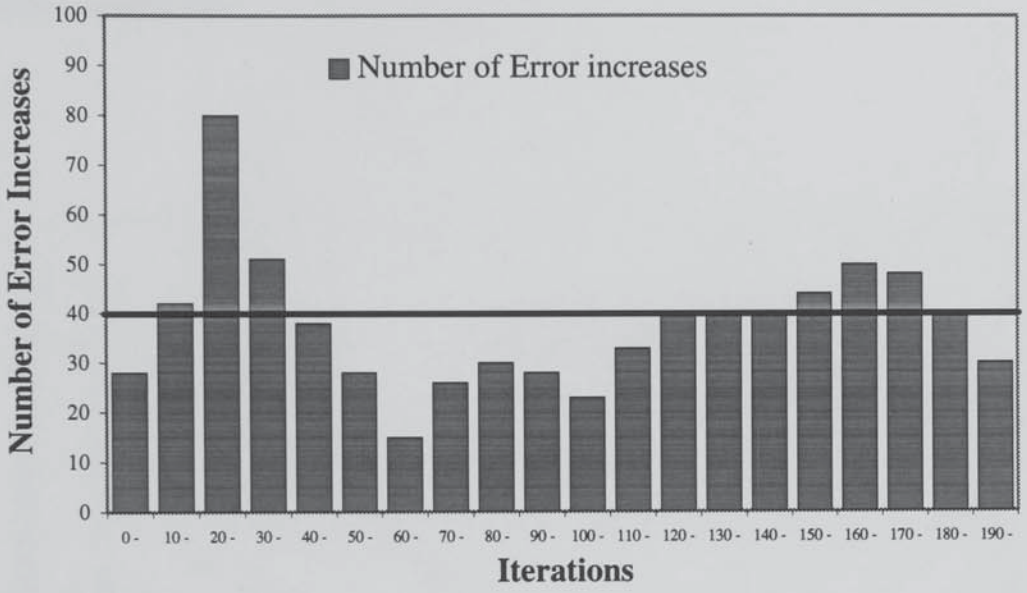


Figure 5.4 - Total Number of Error Increases for Encoder8 with 4 Hidden Units

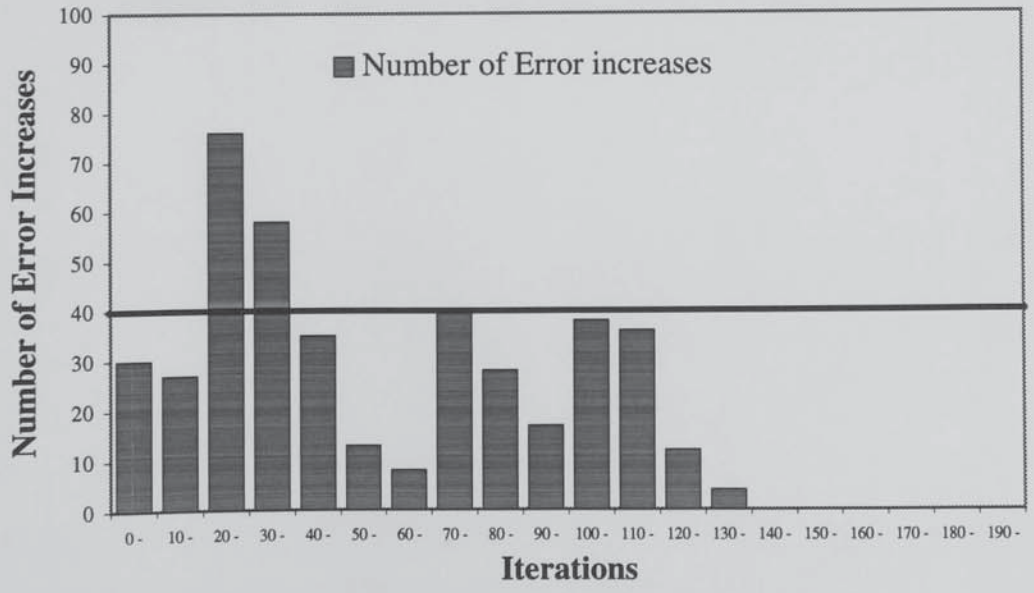


Figure 5.5 - Total Number of Error Increases for Encoder8 with 5 Hidden Units

As noted above, the total and average increase and decrease for each 10 iteration period were also recorded during this process. The results obtained using these measures were more variable than those obtained from the total number of error increases, as can be seen in figures 5.6 and 5.7 below, which were obtained using the encoder8 problem with a 4 hidden unit architecture.

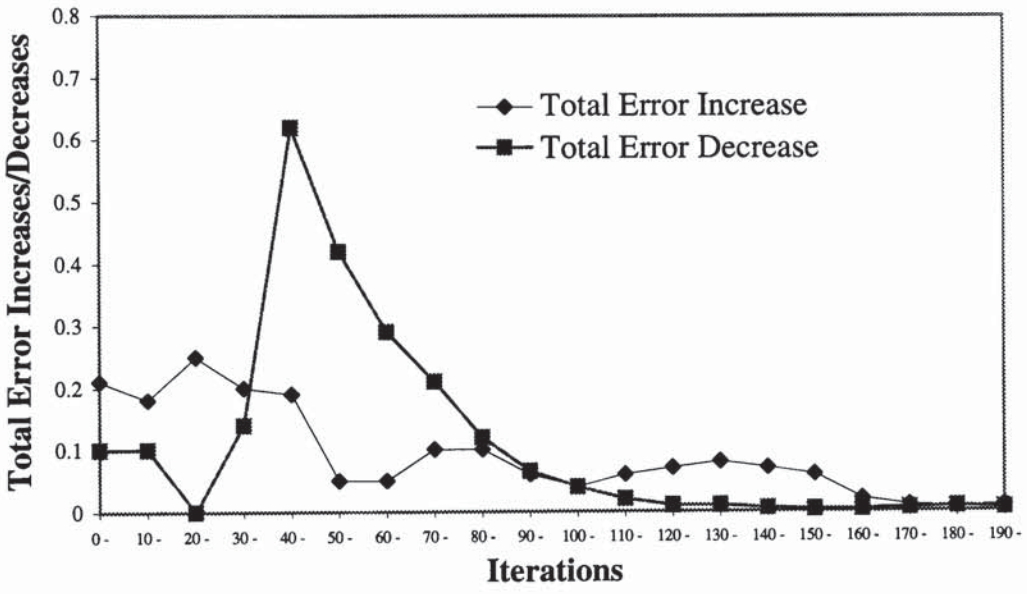


Figure 5.6 - Total Errors for Encoder8 with 4 Hidden Units

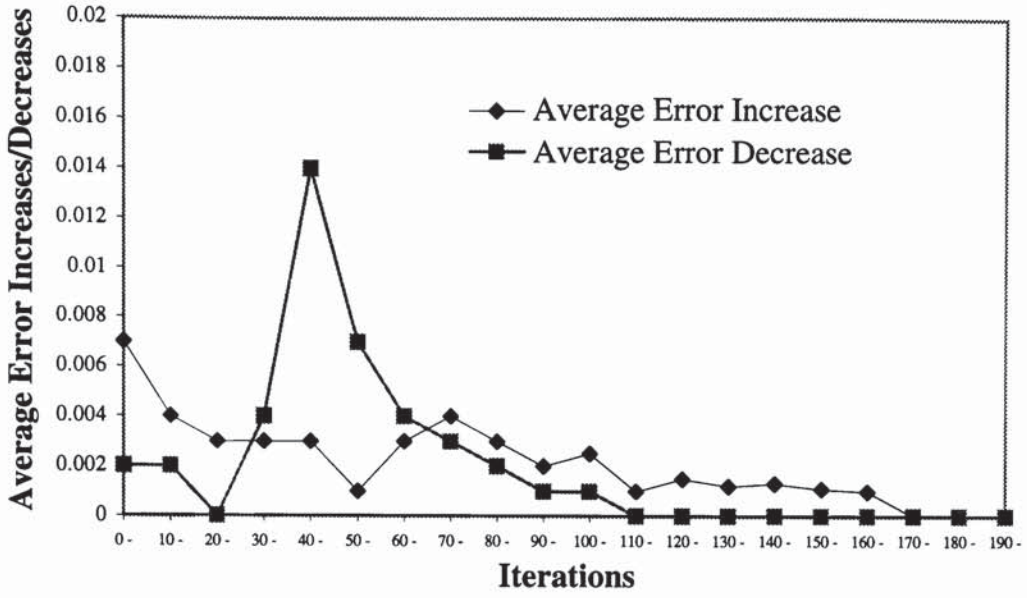


Figure 5.7 - Average Errors for Encoder8 with 4 Hidden Units

Figure 5.6 shows the total error increase and total error decrease over all patterns in each 10 iteration period, whilst figure 5.7 shows the average error increase and decrease in each period. Neither of these graphs support the fact that there are sufficient hidden units present in the network. There are large periods in which the error increase measures are greater than the decreases. This inconsistency was the case for both sufficient and insufficient hidden layer architectures on all three test problems using a variety of initial weights. For this reason, these measures were not utilised and hence no further examples are presented.

Examining the total number of error increases, the process was repeated for several initial weight ranges for the encoder8 problem and each was found to produce similar results to those presented in figures 5.1 to 5.5 above. The most interesting result was that the number of error increases never fell below 50% in any 10 iteration period for the insufficient networks of 1 or 2 hidden units. Once sufficient hidden units were present, the number of error increases fell farther below 50% with greater frequency as the number of hidden units was incremented.

This process was repeated on the parity6 problem; examples for 1, 6 and 8 hidden units are shown in figures 5.8, 5.9 and 5.10 below.

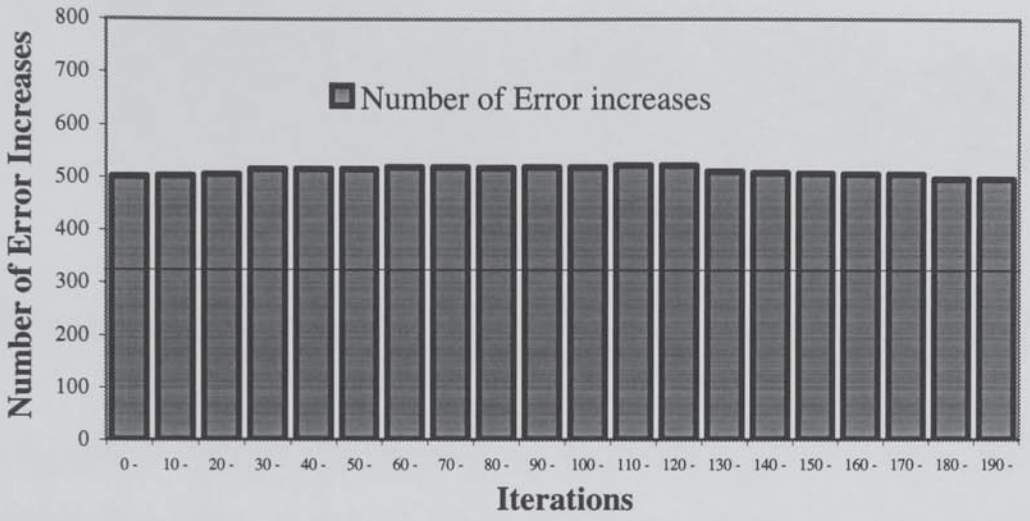


Figure 5.8 - Total Number of Error Increases for Parity6 with 1 Hidden Unit

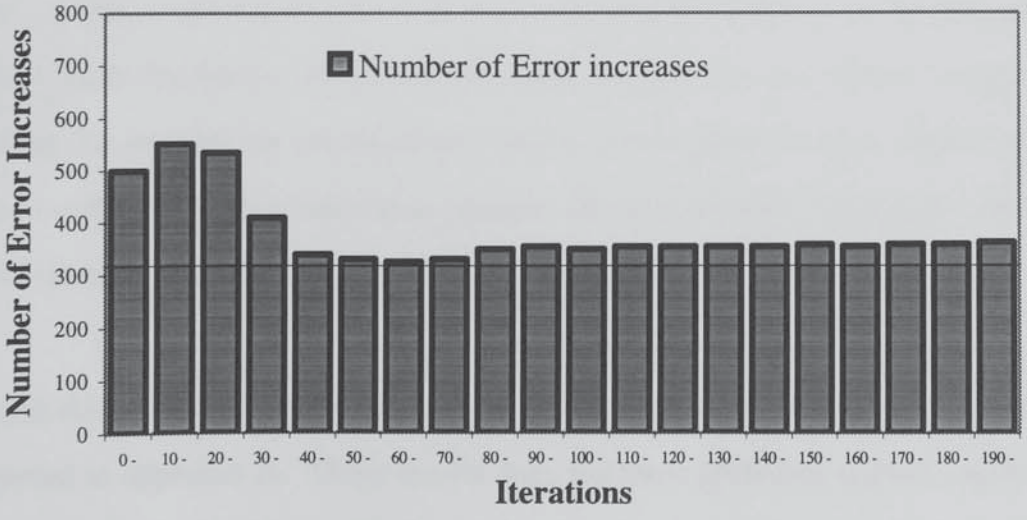


Figure 5.9 - Total Number of Error Increases for Parity6 with 6 Hidden Units

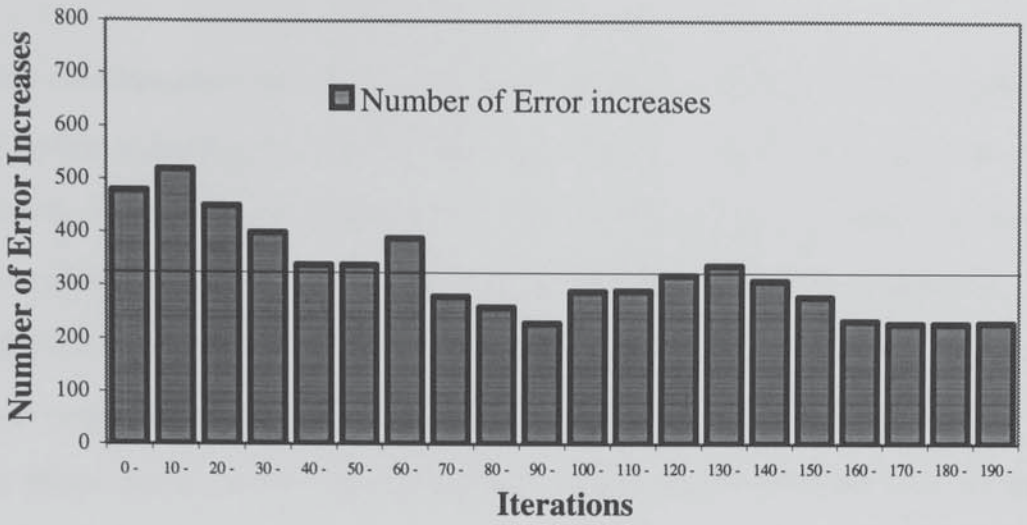


Figure 5.10 - Total Number of Error Increases for Parity6 with 8 Hidden units

As can be seen from figure 5.8, with a network of 1 hidden unit, the total number of error increases far outweighed error reductions, in line with what was found with the encoder8 problem. As the number of hidden units was increased to the minimum six hidden units required to solve the problem, shown in figure 5.9, the number of error increases was reduced but failed to fall below 50%. This is not surprising because unlike the encoder8 problem, convergence with the minimum number of hidden units is very difficult to achieve. As the number of hidden units was increased still further to 8, the total number of error increases fell below 50% of the possible number, as can be seen in figure 5.10. Repeating this process for several initial weight values produced very similar results, in that once sufficient hidden units were present, the error decreases began to outweigh the increases in each case.

Applying this process to the 4DC problem produced similar results, some of which are represented in appendix B. These results from the three problems studied implied that a viable constructive algorithm could be developed using the total number of error increases over all the patterns in the training set.

5.1.4 Criteria and Method of Unit Insertion

The first consideration was to decide how many iterations in the training phase should elapse before a decision could be made on whether a hidden unit should be inserted. Initially this was set at 100 iterations. Several different criteria for adding a hidden unit were considered on the basis of the results outlined above. It was decided that a unit would be added into the hidden layer if the total number of error increases did not fall below 50% in two consecutive 10 iteration periods within a 100 iteration decision window. On the binary values studied these values produced the best results, but in general the algorithm was shown to be relatively insensitive to these values, as shown in appendix F. This was coded into the back-propagation algorithm in the inverse sense, in that the algorithm added a unit into the hidden layer after every 100 iterations until the reverse of the above condition was found to be true. Therefore a decision was made that sufficient hidden units were present when the number of error *decreases* exceeded 50% in two consecutive 10 iteration periods. The addition criteria was coded in this way because the assumption was made that additional hidden units would be required initially, and therefore the aim was to terminate this process once sufficient units were present. The algorithm is therefore as follows.

Main Algorithm

```
Flag := FALSE;
REPEAT
Dynamic Learning;
    IF Decision to add unit is true THEN
        Add a new unit;
        ELSE
        Flag := TRUE;
    END
UNTIL Flag = True;
Continue with Standard Back-Propagation
```

As can be seen from the above outline, the algorithm continues to add hidden units until a decision is made that sufficient units are present to solve the given problem, at which point the standard back-propagation algorithm is applied. The dynamic learning algorithm is outlined below.

Dynamic Learning Algorithm

```
FOR Count := 1 TO 10
        FOR Count2 := 1 TO 10
                FOR Vector := 1 TO NumberOfVectors DO
                        Present Input Vector;
                        Calculate Output;
                        Calculate Error for this Input;
                        Compare Error with that Calculated
                        immediately after Weight Adjustment
                        during last Iteration;
                        Add to Count indicating Direction
                        of Error Change;
                        Update Weights;
                        Calculate Error for this Vector (for
                        use in next iteration's comparison)
                END
        END
END.
```

At the conclusion of this procedure, the count for each 10 iteration period within the 100 iteration decision window is passed to a procedure which determines whether a new unit should be inserted or not. If a new hidden unit is inserted then the new weights are

randomly generated in the same range as the weights were originally initialised in the network. The Modula_2 implementation of the algorithm is given in appendix C.

5.2 Selecting a Weight Range

5.2.1 Introduction

In general, reports of constructive algorithms in the literature do not contain information on the nature of the weights and thresholds of units inserted into the network. Hirose *et al* (1991), however, do state that their constructive algorithm added units with small random weights or zero weights; they found that zero weights caused the addition of a great many hidden units. One approach would be to re-initialise the network on the addition of each unit, but in doing this we would be forfeiting the weights already determined in the network which are likely to be representative of the final weights. This strategy will be investigated along with various alternative strategies of assigning weights to the inserted units, whilst retaining existing network weights and thresholds.

5.2.2 Varying the Weight Range of Inserted Weights

The algorithm was applied to all three binary problems a total of 20 times using different random starting weights in order to obtain average performance values. In order to minimise the time taken to obtain results, the networks were trained to a global error of 0.1. Table 5.1 shows the results of adding the new weights and thresholds in the same range as the initial weights in the network for the encoder8 and 4DC problems; the results for parity6 are summarised in section 5.2.3.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.3	3	472	64	14%
4DC	20	3.6	2	331	162	49%

Table 5.1 - Basic Method Of Addition

In table 5.1 and subsequent tables, the results presented can be divided into two parts. In terms of the hidden units in the network, the average number of hidden units in the final architecture, measured over the convergent trials, is shown, together with the mode which is the most common final hidden layer size. The speed of convergence is represented by the average number of iterations required to reach an error of 0.1. The standard deviation associated with this average is given, together with its size as a percentage of the average. The standard deviation values give a measure of the consistency of the results over the convergent trials. A trial was deemed not to have converged if the algorithm did not achieve an error value of 0.1 within a pre-set number of iterations. In the case of the binary problems studied this was 5000 iterations, since standard back-propagation is quite capable of solving these problems within this limit. A trial was also judged to be non-convergent if the algorithm continued to add hidden units despite the fact that the specified accuracy had been achieved, and also if the algorithm did not build a sufficient hidden layer architecture to solve the problem.

The results given in table 5.1 are good in the sense that every trial converged and the final number of hidden units was low (the minimum number of hidden units required to solve the encoder8 and 4DC problems are 3 and 2 respectively). One interesting feature of the results attained for the 4DC problem is the fact that the algorithm built networks with a hidden layer of 2 units a total of 9 times, converging in each case. As can be seen from table 4.7, standard back-propagation only converged 3 times in 20 trials with a hidden layer

of 2 hidden units. This would seem to imply that in this case, building the network gradually, aided the learning process for the minimal networks.

When hidden units are added into the network, the values of the existing weights in the network will have diverged from their initial random ranges. In the light of this it was felt that it may be profitable to add in units that have larger weights and thresholds, which would have a more immediate impact. The algorithm was modified such that once the decision to add a new unit was made, the algorithm searched through the existing weights and calculated both the maximum and minimum weights. The new weights and thresholds were then randomly assigned in the range min (weight) $< W_{i,j} <$ max (weight). The results of applying this to the binary problems studied are shown in table 5.2.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	14	3.7	3	1267	652	51%
4DC	9	4.2	3	446	228	64%

Table 5.2 - Weight Addition in Min/Max Range

These results are inferior to those in table 5.1 both in terms of the final number of hidden units and the number of iterations required to reach a solution. The fact that the final number of hidden units was in general greater, implies that the added weights and thresholds were not as appropriate. This result was due to the relatively large values assigned to the new weights and thresholds causing a major perturbation, and hence degrading the performance of the network.

An interesting aspect of the encoder8 results is that the network settled on 2 hidden units a total of 5 times, each of which converged. Standard back-propagation was not capable of achieving this result, never converging in 20 trials using a 2 unit hidden layer architecture.

This result would again seem to suggest that building a hidden layer is beneficial to the derivation and learning process of minimal networks.

In a variation of the above experiment, the thresholds of new units were selected in the range between the maximum and minimum existing thresholds; this was independent of the range of the weights. The results were similar but slightly inferior to those given in table 5.2. This is possibly due to the new threshold not being of the same order as the weights into it; this has the effect of forcing the activation function to its extreme values, which could require large modifications of the threshold value to correct. A brief discussion of the importance of threshold values is given in section 5.2.4.

In an attempt to add units with relevant weights without the adverse effects of using the minimum and maximum, the weights and thresholds were added in the range average (-ve weights) < $W_{i,j}$< average (+ve weights). The existing positive and negative weights were averaged separately in order to avoid them cancelling each other out in a global calculation. Table 5.3 shows the results of this amendment.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.2	3	453	34	8%
4DC	11	8.1	4	543	365	67%

Table 5.3 - Weight Addition in Average Range

These results are interesting because in the case of encoder8 the results are marginally the best outlined so far, both in terms of the final number of hidden units and the number of iterations required. However, the results for the 4DC problem show a significant degradation from previous results. Assigning the values of the thresholds in separate ranges as previously, resulted in similar but slightly inferior results for the encoder8

problem, whereas the results for the 4DC problem were greatly improved, achieving an average number of hidden units of 4.1 and an average number of iterations of 330.

Since the weights in the two layers may be of different magnitudes, the algorithm was altered to separately calculate the average of the weights in each layer. The thresholds were set in the same range as the weights in the first layer, because it was felt that they should be of a similar order of magnitude to the sum that is also passed to the activation function. If the threshold is of a similar magnitude, then the value passed to the non-linear function will be capable of becoming positive or negative relatively quickly. Alternatively if the threshold is set such that this value is a large positive or negative number but is of the opposite sign to that required by the network, then the algorithm may take a large number of iterations to reverse it. The results of this modification are shown in table 5.4.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.25	3	453	51	20%
4DC	8	4.9	2&3	572	591	103%

Table 5.4 - Weight Addition in Average Range with Separate Layers

The results for the encoder8 problem are again similar to the previous results, whereas the 4DC problem failed to reach the level of performance given by table 5.1 using the same initial weight range. This is due to the high percentage of cases in which hidden layers of 2 hidden units were built, but convergence was not achieved. For example, the results shown in table 5.4 include 13 cases of final architectures with 2 hidden units, only two of which converged.

In an attempt to improve the performance on the 4DC problem, the method used to create new weights was altered in the following way. As a new hidden unit was created, the new

weights were set with equal probability to either the average positive or average negative existing weight value. In this way the weights were larger than those that gave rise to table 5.4 and hence should have more effect. The results obtained are shown in table 5.5 below.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	18	3.4	3	493	126	25%
4DC	11	2.6	3	261	96	37%

Table 5.5 - Weight Addition as Equal to Average Positive or Negative Existing Weight

As can be seen from the above results, the performance on the 4DC problem was improved in comparison to the results in table 5.4, whilst giving comparable results for the encoder8 problem.

The encoder8 problem proved to be very stable in the sense that the results obtained show very little variation. Excluding the two worst cases (obtained by adding large weights to the network), the average value of H ranged from 3.2 to 3.5, the average number of iterations required rising from 449 to 493. Convergence was also excellent, never falling below 18 convergent cases from 20 trials. The results for the encoder8 problem improved when the added weights were larger than the initial range, but deteriorated if they were too large, as shown in table 5.2; this can also be seen in tables 5.4 and 5.5, where the larger weights used in table 5.5 result in a degradation of performance. An interesting aspect of these results is the ability of the algorithm to solve the encoder8 using only 2 hidden units. This occurred when the weights added to the network were large, which implies that these weights were large enough to 'push' the error in the right direction temporarily, but for sufficiently long to terminate the addition of further hidden units. The results obtained for the 4DC problem were less stable than those obtained for the encoder8. If, in a similar manner to above, we exclude the two worst cases then the average value ranged from 2.6 to

5, with the average number of iterations required for convergence varying from 361 to 572. The convergence rate was very poor, achieving 100% in only one experiment, and typically converging in around 50% of trials. This is due to the large number of trials in which the network settled on 2 hidden units, but failed to converge. This is not unexpected as standard back-propagation only converged in 15% of trials using networks of 2 hidden units. Bearing this in mind, the results shown in table 5.1 are very promising in that a value of 2 hidden units was attained 9 times, and convergence was achieved in all cases.

5.2.3 Parity6

The results obtained for the parity6 problem are not summarised in section 5.2.2 because it is not possible to represent them in the same form as those reported for the encoder8 and 4DC problems. This is due to the fact that most of the trials did not converge, and the disparate nature of the results in terms of the final number of hidden units made the generation of meaningful statistics difficult. Adding the new weights and thresholds in the same range as the initial network weights resulted in table 5.6.

Hidden Units	2	3	4	5	7	8	9	10	14	15	F
Occurrences	1	2	2	1	2	1	2	2	2	1	4
Convergence	0	0	0	0	0	1	1	2	2	1	-

Table 5.6 - Basic Method of Weight Addition for Parity6

Table 5.6 shows the number of occurrences of the final number of hidden units obtained by the algorithm, regardless of whether or not convergence was achieved, together with the number of these trials that converged. In order to terminate trials that continuously added hidden units without convergence, an upper bound of 20 hidden units was set. This value

was chosen because we found that if the algorithm built a hidden layer of this size, it would continue to add units indefinitely. These trials are denoted by 'F' in the final column. The results vary widely in the sense that some trials continue to add units through the 20 hidden unit boundary, several trials stop adding units before the minimum theoretical number is attained with the remaining trials varying between 7 and 15 hidden units. These results are disappointing both in terms of the number of convergent trials and the inconsistency of the final topology.

Adjusting the range of the new weights and thresholds in the network in the same way as for the encoder8 and 4DC problems above, gave the results shown in the tables that follow. Table 5.7 shows the effect of setting the weights of inserted units in the range: min (weight) < $W_{i,j}$ < max (weight).

Hidden Units	4	5	6	9	11	14	15	18	19	F
Occurrences	1	1	2	1	2	2	1	1	1	8
Convergence	0	0	0	0	1	1	1	0	1	-

Table 5.7 - Weight Addition in Min/Max Range for Parity6

This modification had the effect of increasing the number of hidden units present in the final hidden layer architecture. However, the size of the final architecture remained spread over a wide range of values, with some trials culminating in insufficient hidden units. Setting the inserted weights in the range: average (-ve weights) < $W_{i,j}$< average (+ve weights), gave the results shown in table 5.8 below.

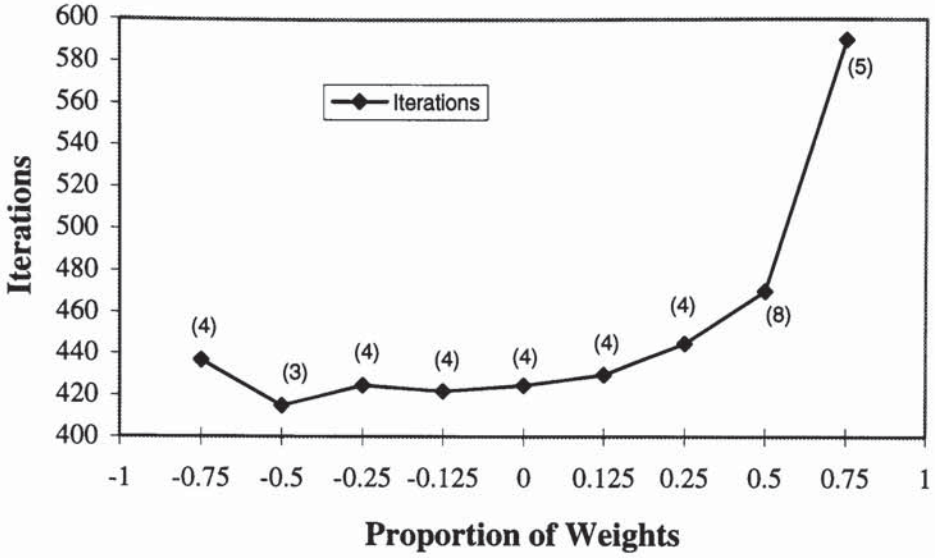
Hidden Units	2	3	5	7	8	11	14	15	16	17	18	19	F
Occurrences	1	1	1	1	1	1	2	1	1	2	1	1	6
Convergence	0	0	0	0	0	1	2	1	1	2	0	1	-

Table 5.8 - Weight Addition in Average Range for Parity6

The results shown in table 5.8 were marginally the best obtained for the parity6 problem, sharing the same negative features as those shown in table 5.6 & 5.7, together with the positive feature that final architectures of over 7 hidden units tended to result in convergent trials. The erratic nature of these results made a systematic and logical exploration difficult, however these experiments did show that the parity6 problem is relatively insensitive to the size of weights assigned to the new units, in contrast to the other binary problems studied,.

5.2.4 The Importance of Thresholds

The experiment that yielded the results shown in table 5.4 was repeated on the encoder8 problem until the second hidden unit had been added, and the weights and thresholds saved to file. The algorithm was then applied to this network in exactly the same way until the third hidden unit was to be inserted into the network. At this point, the weights assigned to the network were identical for each trial, but the threshold value was varied in order to determine its importance. The results quoted are therefore derived from identical networks with the threshold of the third hidden unit set as a proportion of the total of the weights into the unit. Figure 5.11 shows the resulting graph of the value of the inserted threshold (as a proportion of the total weights into the unit) against the number of iterations required to converge to a global error of 0.1. The figures in parentheses indicate the final number of hidden units in each case; the figures below the graph show some of the associated unit statistics.

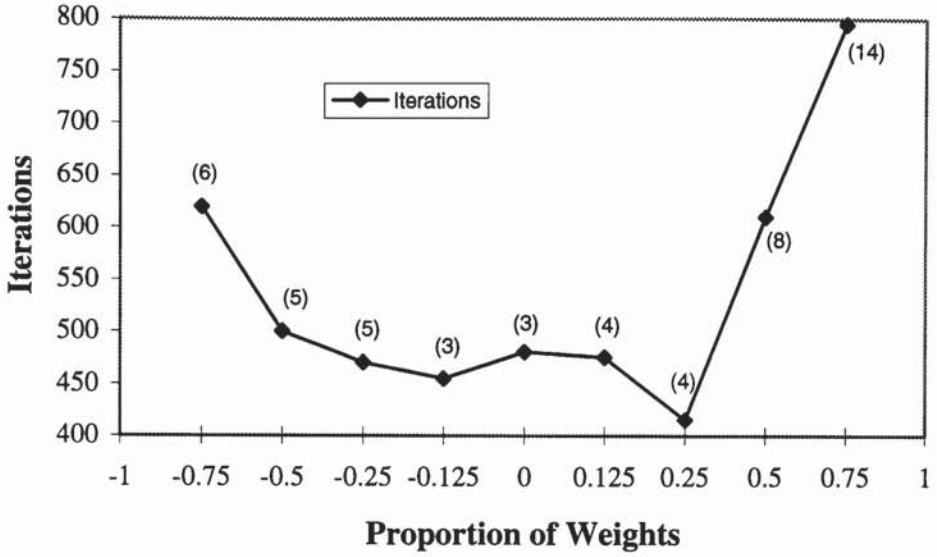


Total weights into hidden unit	=	-6.46
Total of positive weights into unit	=	0.698
Total of negative weights into unit	=	-7.15
Average positive weight into unit	=	0.35
Average negative weight into unit	=	-1.19

Figure 5.11 - An Example of Different Threshold Values for Encoder8

Figure 5.11 shows that the minimum value obtained, both in terms of iterations and final number of hidden units, occurs at a threshold value that is -0.5 of the total weights into the unit. The graph shows that the final number of hidden units is partially determined by the value of the threshold, since it varied from a minimum of 3 to a maximum of 5. The number of iterations required also shows a large variation rising from around 400 to nearly 600. Paradoxically in this particular example the only method of adding the weights and thresholds as outlined in section 5.2.2. that would add the threshold at or around its optimum value is that using the minimum and maximum existing weights. The results obtained using this method (shown in table 5.2) gave the worst results in terms of number of iterations and convergence, although despite this, a final value of 3 hidden units was the most commonly achieved result.

This process was repeated using several initial sets of weights, an example of which is shown in figure 5.12.



Total weights into hidden unit	=	7.95
Total of positive weights into unit	=	10.59
Total of negative weights into unit	=	-2.64
Average positive weight into unit	=	1.76
Average negative weight into unit	=	-1.32

Figure 5.12 - Alternative Example of Different Threshold Values for Encoder8

Figure 5.12 shows that, in this case, the process resulted in separate minima for the number of iterations and the final number of hidden units. The minimum number of hidden units occurred when the threshold was set at -0.125 of the total weights into the unit, with the minimum number of iterations occurring at +0.25. This result was found in other experiments in which the initial weights were varied, although a similar number of experiments gave results in which the minimum number of iterations and hidden units agreed. Over all the trials, these minima did not correspond with a particular proportion of the weights, varying between proportions of -0.75 to +0.5 without any discernible pattern.

In order to ascertain whether a pattern could be detected for other training sets, the process was applied to the parity6 problem. An example of this, in which the threshold of the 7th hidden unit was varied, is shown in figure 5.13.

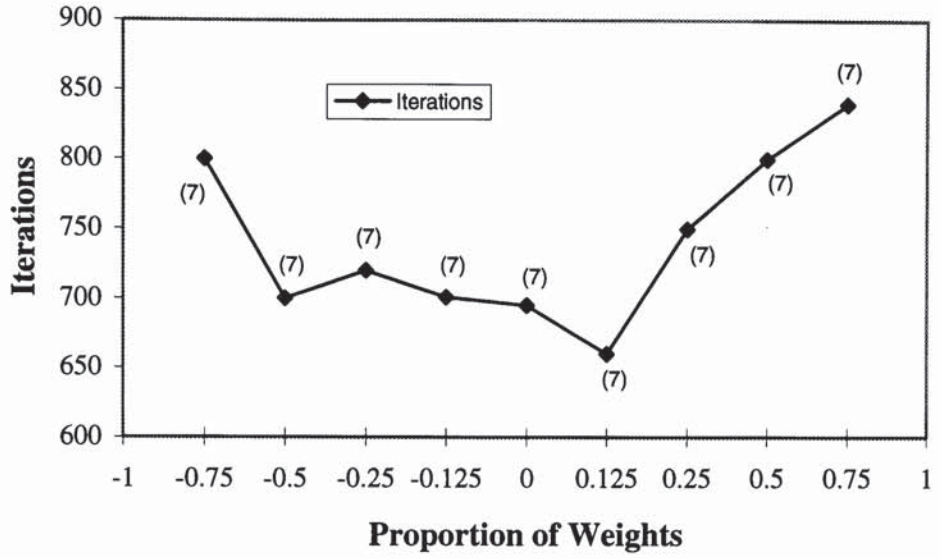


Figure 5.13 - An Example of Different Threshold Values for Parity6

In this case the value of the threshold had no influence on the final number of hidden units in the network, a result that was repeated in all the trials. However, the threshold did affect the number of iterations required to reach a solution, but did not have enough effect to determine whether or not a trial converged. That is to say, given an initial set of weights for which convergence was not achieved, varying the threshold of the seventh hidden unit had no effect on convergence. This again shows the relative insensitivity of the parity6 problem to parameter values. Whereas the performance of the algorithm on the encoder8 problem could be strongly affected by varying the threshold of a vital unit, only minor alterations in performance could be caused in the case of parity6.

In contrast, Smieja & Richards (1988) show that on the small rounding problem studied, the back-propagation algorithm is extremely sensitive to small perturbations in threshold values, illustrating that a small change in value can cause previously correct outputs to give an incorrect value. On the basis of this work, together with our results, we decided that the

development of a constructive algorithm based on varying the threshold values would not be a fruitful approach to take. One possible modification to determine an optimum threshold value, would have been to temporarily insert a pool of units into the network, each with a different threshold value. This method, which has been utilised by Fahlman & Lebiere (1990) to determine weight values, effectively tests several new units simultaneously, retaining the most effective after a number of iterations. This was not implemented as it is was felt that it would be computationally demanding to implement due to the number of alternatives that would have to be concurrently tested.

5.2.5 Re-Initialising the Network Weights and Thresholds

As stated in section 5.2.1, one possibility on the insertion of a hidden unit into the network, is to re-initialise all the network weights and thresholds, including those of units already present in the network. This method results in the learning that networks have achieved up to the addition of a new unit being lost. Table 5.9 shows the effect of this approach on the encoder8 and 4DC problems.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.4	3	543	90	17%
4DC	12	3.3	3	324	79	24%

Table 5.9 - Re-Initialising Network Weights for Encoder8 and 4DC

The results in table 5.9 show that re-initialising the network weights on the addition of a new unit did not have a significant effect on the results for the encoder8 problem, which showed a slightly increased average final architecture and slower convergence than the results in table 5.1. However, table 5.9 shows that this method had a significant

detrimental effect on the results obtained for the 4DC problem. Only 12 of the 20 trials converged, due to a large number of trials in which the algorithm settled on a 2 hidden unit architecture but did not subsequently achieve convergence. This level of convergence was in line with that achieved using the standard back-propagation algorithm, which was expected as all the information learned prior to the addition of a hidden unit is lost on re-initialisation of the network weights. Table 5.10 shows the results of applying this method to the parity6 problem.

Hidden Units	3	4	5	6	7	8	9	11	12	F
Occurrences	1	4	2	2	1	2	1	2	2	3
Convergence	0	0	0	0	1	2	1	2	2	-

Table 5.10 - Re-Initialising Network Weights for Parity6

In line with many of the results presented in this section, re-initialising the network weights had very little effect on the performance of the algorithm for the parity6 problem, with the final topology of the hidden layer varying from insufficient numbers of hidden units to too many.

Since retaining the existing network weights on the insertion of a new unit seemed to have a beneficial effect, it was decided that in subsequent experiments this method would be retained. The weights and thresholds of added units would be assigned randomly in the same range used for the initial network weights, since, whilst improvements in performance were possible for the encoder8 problem by varying the range of the weights and thresholds of additional units, no improvement was made for the 4DC problem and little for parity6.

5.3 Freezing the Existing Weights in the Network

5.3.1 Introduction

Ash (1989) briefly describes a method of freezing the existing weights and thresholds on the insertion of a new unit into the network, but does not implement the method. This was due to the fact that if part of the network is frozen then this effectively limits the area of search, since only part of the network is available for training. This is shown in figure 5.14, which depicts a two dimensional error surface frozen in one dimension.

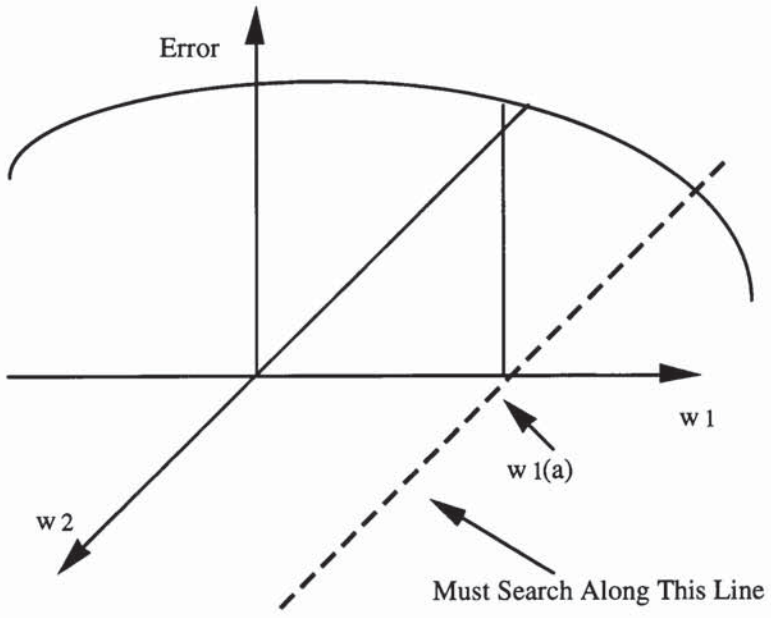


Figure 5.14 - A Three Dimensional Error Space

If weight W_1 is frozen at a value $W_1(a)$ then this limits the search to the W_2 dimension and the algorithm must search along the dotted line in search of a minimum. Obviously this is a simplification of what occurs in back-propagation applied to a multi-dimensional problem, but the general principle holds.

Whilst accepting that freezing limits the search space, it was felt that freezing the existing network temporarily could prove beneficial. The results outlined in section 5.2 indicated that performance was degraded when the weight values allocated to new units were much larger than the initial range of weights. It was felt therefore that adding new weights in this small range and freezing the existing network temporarily would allow the new unit to assume a significant role in the learning process. Fahlman & Lebiere (1990) have since utilised a permanent freezing approach as part of the cascade correlation algorithm, described in section 3.2.2, and obtained good results.

5.3.2 Freezing Algorithm

Freezing was introduced into the constructive algorithm in the following way.

Weight Freezing Algorithm

```
Run for 100 iterations to determine
whether a further hidden unit is to be
added.

WHILE Decision to add is TRUE THEN

        Add Unit;

        Multiply learning rate for new weights
        by a constant = Increase Factor;

        Multiply learning rate for old weights
        by a constant = Decrease factor;

        Run standard Back-Propagation for a
        period (initially 50 iterations);

        Return to normal learning rate;

        Run for 100 iterations to determine
        whether a further hidden unit is to be
        added;

END

Run standard Back-Propagation to convergence
```

The method of using increase and decrease factors was introduced in order to make the program more flexible and to facilitate the modification of the learning rates. Therefore it

is a relatively simple procedure to alter the learning rates for the different parts of the network for as many iterations as is required.

5.3.3 Applying Freezing

The basic freezing algorithm, that is to say an increase factor of 1 and a decrease factor of 0, was applied to the three test problems adding new weights and thresholds in the initial range. Table 5.11 shows the results for the encoder8 and 4DC problems.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.25	3	578	98	17%
4DC	20	3.5	2	470	400	85%

Table 5.11 - Increase Factor = 1, Decrease Factor = 0

These results were positive in that all trials converged for both problems resulting in a minimum architecture in the majority of cases. However, in comparison to the results shown in table 5.1, the number of iterations required for convergence is large. Therefore, the increase factor was raised to a value of 2 in order to force the new units to adapt quickly; these results are shown in table 5.12 below.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.2	3	561	49	9%
4DC	20	3.9	2	492	352	72%

Table 5.12 - Increase Factor = 2, Decrease Factor = 0

The results for the 4DC problem were very encouraging in the sense that all 20 trials converged, including 11 that resulted in hidden layers of 2 hidden units. Despite this, the average value of the final number of hidden units remains greater than that shown in table 5.1 because many of the trials resulted in 4 or 5 hidden units. Taking the results shown in table 5.1 as a base line, the results for encoder8 were also improved in terms of final architectures, but there was an increase in the average number of iterations due to the fact that the freezing algorithm effectively lengthens the decision window to 150 iterations.

In an effort to improve the results, the increase factor was held at a constant value of 2 and the decrease factor raised to a value of 0.5. In this way, the existing weights in the network were allowed to adapt at a low rate while the new weights were required to respond more rapidly during the learning process. Table 5.13 shows the results of this modification.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.05	3	552	53	10%
4DC	20	3.15	2	360	234	65%

Table 5.13 - Increase Factor = 2, Decrease factor = 0.5

Allowing the existing weights to adapt had the beneficial effect of decreasing the number of iterations required for convergence. In terms of hidden units, all but one case resulted in a minimal solution for the encoder8 problem, a minimal architecture was again achieved 11 times for the 4DC problem. The decrease factor was stepped through 1 and 1.5 and provided steady improvement to culminate in the results shown in table 5.14 below for a decrease factor of 2.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.05	3	490	53	11%
4DC	20	2.5	2	337	285	85%

Table 5.14 - Increase factor = 2, Decrease Factor = 2

These results were the best obtained from the work outlined thus far for both problems, in terms of both the architectures obtained and the fact that all trials converged. The number of minimal architectures obtained grew steadily as the learning rate for the existing part of the network was increased, culminating in table 5.14 which includes 19 and 13 minimal architectures for the encoder8 and 4DC problems respectively. In terms of the number of iterations, the results for both problems were only slightly higher than those obtained using the basic algorithm, despite the effective 150 decision window. Continuing this process by raising the value of the decrease factor to 3 gave the results shown in table 5.15.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.05	3	437	46	11%
4DC	8	3.1	4	979	807	82%

Table 5.15 - Increase Factor = 2, Decrease factor = 3

The results for the encoder8 improved in terms of the number of iterations required and remained constant in terms of the final architecture obtained. However, the results obtained for the 4DC problem suffered a sharp degradation, with only 8 of the 20 trials converging. This is to be expected as the algorithm was using a learning factor of 0.75 for a large portion of the learning process. This value is much larger than that traditionally

used in the back-propagation algorithm, and had the effect of making large changes to the weights which can cause the algorithm to overshoot the minima required to achieve the desired error levels. This same effect was observed for the encoder8 problem when a decrease factor of 4 was used since this caused an increase in both the number of iterations required and the dimension of the final architecture.

Using the same initial twenty sets of starting weights, the results for the 4DC problem were analysed as the decrease factor was steadily raised from 0 to 3, with the increase factor fixed at a value of 2. Trials that settled on final architectures of 2 hidden units with a decrease factor of 0, continued to attain this architecture as the decrease factor was raised, with an associated reduction in the number of iterations. In the case of trials that initially resulted in networks of more than 2 hidden units, the number of hidden units was reduced. This reduction was gradual and constant across all the trials, and was coupled with an associated increase in the number of iterations required.

Retaining a decrease factor of 2, and altering the increase factor to various values did not improve results for either problem, as can be seen from tables 5.16 and 5.17 below, which show the results for increase factors of 1 and 3 respectively.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.25	3	480	25	5%
4DC	19	2.8	2	352	354	101%

Table 5.16 - Increase Factor = 1, Decrease factor = 2

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.35	3	429	46	10%
4DC	19	2.6	2	332	284	86%

Table 5.17 - Increase Factor = 3, Decrease factor = 2

The results shown in tables 5.16 & 5.17 did not improve on those of table 5.14, but remained reasonable, showing that the algorithm is relatively insensitive to these parameters. Temporarily tripling the learning rate for the weights of inserted units did not result in the sharp degradation in performance for the 4DC problem that was associated with tripling the rate for existing weights (shown in table 5.15). We conjecture that is due to new weights being relatively small and randomly valued, which dampens the detrimental effect of the high learning rate.

The results obtained from the basic approach described in this section are very promising. The encoder8 problem has a range of values of final architecture from 3.05 to 3.35 (roughly equivalent to obtaining a minimal architecture of three hidden units between 15 and 19 times from 20 trials), the number of iterations ranging from 429 to 578. For the 4DC problem, the ranges are between 2.5 and 3.9 (roughly equivalent to obtaining a minimal 2 hidden units between 11 and 13 times from 20 trials), the number of iterations required ranging from 337 to 492.

The results presented in this section are surprising in the sense that increasing the learning factor for the existing weights in the network improved performance. It was expected at the outset that increasing the learning factor for the new portion of the network whilst freezing the remainder, would enhance performance by allowing the new unit to quickly assume a significant role given the fixed state of the existing network. Increasing the learning rate for the entire network, however, improved performance beyond expectation,

possibly because the detrimental interactions between hidden units (referred to as the 'moving target problem' by Fahlman & Lebiere (1990)) were condensed into a short period of intense learning. Although acceptable results were obtained, it was felt that these could be improved, particularly in terms of the number of iterations required for convergence, which is discussed in section 5.4.

5.3.4 Parity6

The same approach applied to the parity6 problem produced little variation or improvement in the results. The results for an increase and decrease factor of 2 are shown in table 5.18.

Hidden Units	2	3	6	7	8	9	10	12	14	19	F
Occurrences	3	3	2	1	2	1	2	1	1	1	3
Convergence	0	0	0	0	0	1	0	1	0	1	-

Table 5.18 - Increase factor = 2, Decrease Factor = 2 for Parity6

The results obtained for the parity6 problem show similar agreement with those obtained earlier in that the results are inconsistent and variable. The number of hidden units varied between 2 and 20, and consequently a modification was sought that would make the results less variable. Applying standard back-propagation to the parity6 problem with weights in the range $-1 < W_{i,j} < 1$ or $-0.5 < W_{i,j} < 0.5$ gave inconsistent results in terms of the number of iterations and the number of trials that converged, with a weight range of $-2 < W_{i,j} < 2$ giving better performance. Kolen & Pollack (1990) show that using very small weights or very large weights for the XOR (2-dimensional parity) problem causes high levels of non-convergent and inconsistent behaviour. In view of the results obtained thus far for the parity6 problem, it was felt that assigning the weights in the range $-2 < W_{i,j} < -1$ and $1 < W_{i,j} < 2$, would be preferable since the area around the origin would

be avoided. Utilising this weight range, and applying the basic algorithm, i.e. setting both the increase and decrease factors to a value of 1, gave the results presented in table 5.19.

Hidden Units	3	4	5	6	8	15	F
Occurrences	7	5	3	2	1	1	1
Convergence	0	0	0	1	1	1	-

Table 5.19 - Increase factor = 1, Decrease factor = 1 for Alternative Parity6

Comparing these results with those shown in table 5.6 show that this weight range resulted in much more consistent results, since the number of hidden units in the final architecture was predominantly lower. Altering the increase and decrease factors using this initial weight range, culminated in the superior results shown in table 5.20 for increase and decrease factors of 2.

Hidden Units	3	4	5	6	8
Occurrences	2	6	8	2	2
Convergence	0	0	0	0	2

Table 5.20 - Increase factor = 2, Decrease factor = 2 for Alternative Parity6

Whilst the final architectures were still not optimal in that they were predominantly insufficient to model the function, the consistency in these results encouraged us to modify the algorithm so that minimal architectures could be found.

In an attempt to force the algorithm to add a larger number of hidden units and to build on the greater consistency shown in table 5.20, the condition that determines when to terminate the addition of hidden units was made more stringent. Rather than terminating the addition of further hidden units when the total number of error increases fell below 50% in two consecutive 10 iteration periods, this value was raised to 55%. This has the

effect of making the termination of the addition phase more difficult to achieve and hence potentially forces the addition of further hidden units. The results obtained by applying this modification to the parity6 problem using an increase and decrease factor of 2 is shown in table 5.21 below.

Hidden Units	5	6	8	F
Occurrences	10	4	4	2
Convergence	0	2	4	-

Table 5.21 - Termination Condition of 55%

As can be seen from table 5.21, the desired effect of increasing the size of the final architecture was achieved, though in the majority of cases, the final value remained below the minimum required. A further tightening of the termination condition from 55% to 60% gave the results shown in table 5.22.

Hidden Units	6	8	18	F
Occurrences	3	5	1	11
Convergence	2	4	1	-

Table 5.22 - Termination Condition of 60%

This modification had the effect of making the termination condition too stringent as can be seen above. In the majority of cases the algorithm continued to add hidden units until forced to stop by the algorithm once 20 hidden units were present. Relaxing the termination condition to a value of 57% gave the results outlined in table 5.23.

Hidden Units	6	7	8	18	F
Occurrences	4	3	8	1	4
Convergence	2	2	8	1	-

Table 5.23 - Termination Condition of 57%

This modification gave the best results obtained so far for the parity6 problem, with 75% of the trials resulting in minimal or near minimal architectures. However, it further illustrated the sensitivity of the parity6 problem to relatively minor changes in the algorithm. This is illustrated by the difference between the results shown in tables 5.23 and 5.22, where a relatively minor change to the termination condition resulted in a sharp degradation in performance. This is in contrast to the encoder8 and 4DC problems which required termination conditions of 75% to cause a similar degradation in performance. This demonstrates the inherent difficulty of the parity6 problem, in that it is not a typical 'neural network' type problem, despite being an important benchmark. However, these results again demonstrate that the 'freezing' process can make a positive contribution to the algorithm.

5.4 Speeding up Convergence

5.4.1 Introduction

In order to speed up the convergence of the back-propagation algorithm, Fahlman (1988) introduced two factors into the algorithm. These were an addition to the derivative of the sigmoid function, and secondly the use of a highly non-linear error function.

5.4.2 The Sigmoid Prime Modification

Fahlman (1988) states that the standard back-propagation algorithm is slowed due to the 'flat spots' of the traditionally used sigmoid activation function, given by equation 4.1. As the error is propagated backwards through the network, the error at each unit, j, is multiplied by the derivative of the sigmoid function evaluated at O_j, the current output of unit j. This derivative, referred to as sigmoid prime is given by equation 5.2 below.

$$O_j (1 - O_j) \qquad 5.2$$

Representing this function graphically in figure 5.15, it can be seen that its value approaches zero if the output of the unit is near to zero or one.

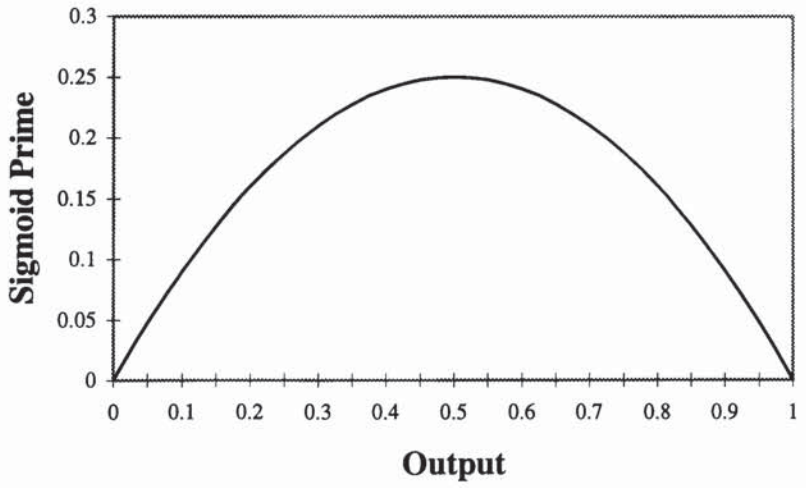


Figure 5.15 - The Sigmoid Prime Function

Consequently, a unit that has an output value that is incorrect but is close to zero or one, will propagate a small percentage of its error back through the network, and hence slows the learning process. In order to increase the error propagated through the network, Fahlman (1988) added a constant value of 0.1 to the sigmoid prime function, thus ensuring

that an error of at least 0.1 was propagated backwards through the network. Applying this modification, Fahlman (1988) reported a halving of the number of iterations required to solve the ten dimensional encoder problem.

It was felt that introducing this modification into the constructive algorithm would speed convergence, and also be beneficial since the results in section 5.3.2 showed improved performance as the learning rate increased. On the other hand, it was thought possible that adding new hidden units might destabilise the process.

5.4.3 The Atanh Modification

The hyperbolic arctangent of the difference between actual and desired values for a given unit, j, is given by equation 5.3 below.

$$\text{atanh}\ (t_j - O_j) \qquad 5.3$$

Using the standard error function, given by equation 2,6, in the back-propagation algorithm, the derivative of the total error for a particular pattern with respect to each output is the difference between the output value and the target value for that pattern. This difference is the value that is propagated backwards through the network. The effect of applying the atanh function to this value is to magnify the significance of large errors whilst leaving small errors relatively unchanged, hence speeding the learning process.

The atanh function is almost linear for small values of error but as the error increases, the function's value rises steeply, for example the atanh function, for an error of 0.5, gives a value of 0.55 and for an error of 0.9 a value of 1.47. Figure 5.16, shows the asymptotic nature of this function as the difference between actual and desired values approaches 1 and -1.

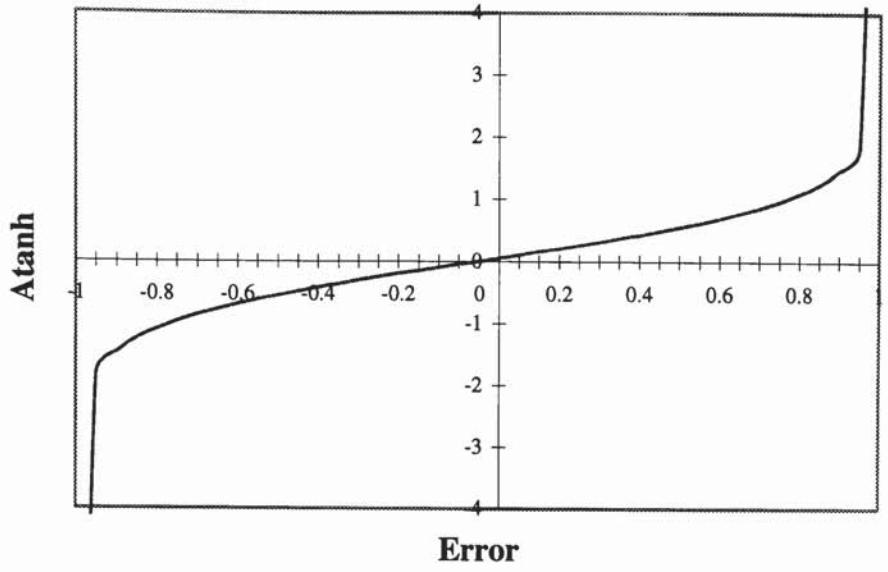


Figure 5.16 - The Atanh Function

Fahlman (1988) reports that applying this modification in conjunction with the sigmoid prime modification to the ten dimensional encoder problem reduced the number of iterations required for convergence by 25% compared to that achieved with the sigmoid prime modification alone.

By applying this modification in the context of our constructive algorithm, it was felt that the atanh function would have the benefit of making the error increases more apparent, hence aiding the process. This is because if there are insufficient hidden units present, some patterns will be mis-classified and hence have large errors. This error will be made larger by the atanh function and hence have a large negative effect on the remaining patterns. Once there are sufficient hidden units, this effect should be reduced and hence the discrepancy between sufficient and insufficient hidden units will be heightened. As the atanh function is unavailable in Modula_2, an exponential formulation was written together with a check that the output from the function did not become extremely large; any error greater than an absolute value of 0.99999 returned a value of plus or minus 17 as recommended by Fahlman (1988).

5.4.4 Results

The modifications outlined above were made to the basic version of the algorithm, rather than the 'freezing' version as any effects noted would be difficult to ascribe to any one modification. In the usual way the algorithm was applied 20 times to each problem in order to obtain average performance values. Applying the atanh modification together with a sigmoid prime constant of 0.1 gave the results summarised in table 5.24 below.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3	3	304	22	7%
4DC	17	3.88	4	364	50	14%

Table 5.24 - Atanh and Sigmoid Prime Constant = 0.1

The results for the encoder8 problem were the best results obtained using any method, converging to the minimal architecture in every trial, having the least number of iterations and the lowest standard deviation. However, the results for the 4DC problem were inferior to many of the results described previously, mainly because the algorithm never built a minimal hidden layer of 2 hidden units. It was felt that this may be due to the sigmoid prime constant affecting the process of deciding whether to add a further hidden unit or not, and hence it was reduced to a value of 0.05 which produced the results shown in table 5.25.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.05	3	337	25	7%
4DC	20	3.5	3	300	48	16%

Table 5.25 - Atanh and Sigmoid Prime Constant = 0.05

This modification now caused a slight degradation in performance for the encoder8 problem, but gave a slight improvement in the performance on the 4DC problem. In order to determine the effect of the sigmoid prime modification in isolation, a value of 0.1 was used with no atanh present.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.05	3	320	24	8%
4DC	16	4.6	5	420	49	12%

Table 5.26 - Sigmoid Prime Constant = 0.1

Comparing the results in table 5.26 with those obtained using the basic algorithm shown in table 5.1, shows that the sigmoid prime modification had a large detrimental effect on the performance of the algorithm on the 4DC problem, whilst serving to enhance the performance slightly on the encoder8 problem by reducing the number of iterations required for convergence. Isolating the atanh modification in the same way produced the results shown in 5.27 below.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3	3	421	28	7%
4DC	20	2	2	165	14	9%

Table 5.27 - Atanh Modification

In terms of the final architecture derived by the algorithm, these results could not be improved. A minimal architecture was produced in every case, all of which converged to a

solution. It is evident from these results that constructing a hidden layer is beneficial both in terms of convergence of minimal architectures and the number of iterations required at the minimal architecture. This is illustrated in the results shown above, in that once the minimal architecture was achieved, the algorithm only required 65 iterations for the 4DC problem as opposed to 154 for standard back-propagation, and 221 iterations for the encoder8 problem instead of 359. This is true not only for the results shown in table 5.27, but also for many of the other results presented in this chapter.

5.4.5 Parity6

Applying the same techniques to the parity6 problem with weights in the range $-2 < W_{i,j} < 2$ gave results very similar to those shown in table 5.18 above, the number of hidden units in the final architecture varying between 2 and 20. An example of this is shown in table 5.28 below, which shows the results obtained using the atanh modification alone.

Hidden Units	3	4	7	8	10	11	15	18	19	F
Occurrences	2	4	1	2	2	1	1	1	1	5
Convergence	0	0	0	1	2	0	1	0	1	-

Table 5.28 - Atanh Modification for Parity6 with Original Weight Range

The results in table 5.28 represent the best results using this weight range in conjunction with variations of sigmoid prime modification and atanh enhancement. Excluding the weight space around the origin from the initial weight range and that of added weights again improved the results. However, this was only the case for the atanh modification alone. The sigmoid prime modification resulted in poor performance, an example of which is shown in table 5.29.

Hidden Units	2	3	5	6	7	8	9	10	11	13	16	18	F
Occurrences	2	1	3	1	3	1	1	1	2	1	1	1	2
Convergence	0	0	0	0	1	0	1	1	2	1	1	0	-

Table 5.29 - Sigmoid Prime Modification of 0.1 for Alternative Parity6

In line with the other binary problems studied, table 5.29 shows that the sigmoid prime modification had a detrimental effect on the performance for parity6 using the revised weight range. The final architectures varied over a wide range of values, which was the case whenever the sigmoid prime modification was included, and whatever its value. The optimum results obtained utilising the alternative weight range occurred when this modification was removed but the algorithm included the atanh modification. The results obtained from applying this modification to the algorithm are shown in table 5.30 below.

Hidden Units	3	4	5	6	7	9
Occurrences	4	5	4	2	1	4
Convergence	0	0	0	0	1	4

Table 5.30 - Atanh Modification for Alternative Parity6

Again, this weight range had the benefit of making the results more consistent but generally resulted in a smaller than optimal hidden layer. Adopting the same strategy as with the freezing method in section 5.3, the termination criteria was made more stringent in an attempt to attain a greater number of hidden units in the final architecture. Increasing the termination condition from 50% to 55% generated the results shown in table 5.31 below.

Hidden Units	4	6	7	8	9	F
Occurrences	2	2	6	6	5	2
Convergence	0	1	5	5	3	-

Table 5.31 - Termination Condition of 55%

Table 5.31 shows that this modification had the desired effect of generally increasing the size of the final architecture, but some instances of insufficient hidden units remained. In an attempt to eliminate these remaining cases, the termination condition was further increased to 60%.

Hidden Units	6	7	8	9	15	F
Occurrences	1	5	6	3	2	2
Convergence	0	5	6	3	2	-

Table 5.32 - Termination Condition of 60%

Table 5.32 clearly shows the expected increase in dimension of the final architectures obtained, removing the cases of insufficient hidden units at the expense of an increase in the number of over-populated hidden layer architectures. As with the 4DC and encoder8 problems, the results for parity6 were improved by the use of the atanh function, though in this case the improvement was less dramatic. In all cases the modification seems to have the benefit of making the error increases more apparent which aids the constructive process.

In general the parity6 problem proved to be the most difficult of the artificial problems studied in that the algorithm did not attain the minimal architecture in 100% of cases as with the other problems. However, reasonable results were obtained given the inherent

difficulty of the problem, with the generation of near minimal architectures in 75% of cases. As stated in section 4.3.3, it has been argued that the problem is not a good benchmark for neural networks due to the lack of generalisation in the training set, but its historical importance makes it difficult to ignore.

5.5 Applying the Algorithm to Real-World Problems

Having attained a constructive algorithm that seemed to function reasonably well on artificial problems, it was decided that the algorithm should be applied to real-world problems, which are in general more difficult. Two problems from the medical domain were chosen, namely drug design and skin cancer diagnosis. A description of these problems, together with the results of applying our constructive algorithm to them are given in the sections below.

5.5.1 Diagnosis of Skin Melanoma

Skin melanoma is an increasingly common form of cancer which is predominantly caused by excessive exposure to the sun. Whilst most tumours are benign and easily removed, approximately 1 in 7 cases proves fatal (Concar, 1992). In the case of malignant tumours, early detection and removal is essential. It has been shown that classification into benign and malignant tumours may be performed by computer image analysis techniques, using features such as textural fractal dimension, structural fractal dimension and bulkiness of the lesion (Claridge *et al*, 1992). Using this data, Bostock *et al* (1993) have successfully applied neural network techniques to the classification of tumours.

Using statistical methods of classification on a previously unseen test set, Claridge *et al* (1992) achieved 91% correct classification of malignant tumours, with 69% of benign

tumours classified correctly. The percentage of correct classifications of malignant and benign tumours are referred to as sensitivity and specificity respectively. These values improved upon the work of clinicians who had achieved 47% sensitivity and 89% specificity. Bostock *et al* (1993) achieved 92% sensitivity and 68% specificity using a constructive algorithm based on the back-propagation algorithm and the work of Ash (1989). Using this method, units were added into the network every 300 iterations if the performance on the test set had not improved by at least 0.01%. This method is based on the assumption that generalisation performance deteriorates if too many hidden units are present.

It was decided to apply our constructive algorithm to this problem since it is a real world problem for which comparison with other methods is possible due to the availability of data and parameter values. The training and generalisation sets used were those utilised by Bostock *et al* (1993); both sets consist of 62 patterns, each of which has 3 real valued input values and 1 binary output. Examples taken from the training set are shown in table 5.33 below.

Input			Output
0.133059	0.1525392	0.454241	0
0.029291	0.4477691	0.274835	0
0.022411	0.8255010	0.445202	1
0.119636	0.4638780	0.223734	1

Table 5.33 - Skin Melanoma Training Set

An output value of 1 represents a malignant tumour, with a value of 0 indicating a benign tumour. Of the 62 patterns in the training set, 37 represented malignant tumours and 25 benign; the generalisation set was equally partitioned between the two categories. Bostock *et al* (1993) report that weights in the range $-2 < W_{i,j} < 2$ together with a learning rate of 0.25 and momentum equal to 0.9 produced the best results, and hence the results described

here were obtained using these values. Whilst utilising the standard least squares estimate of error during the learning process, Bostock *et al* (1993) state that a output of within 0.4 of the desired value was deemed to be correct once learning was complete . In order to facilitate direct comparison of results this method of reporting the results is used here.

Table 5.34 shows the results of applying the basic constructive algorithm to this problem, after which the responses to the patterns in both the training and generalisation sets were recorded.

	Sensitivity	**Specificity**	**Average Over Whole Set**
Training Set	95%	88%	92%
Generalisation Set	88%	62%	75%

Table 5.34 - Basic Method Applied to Skin Cancer Diagnosis Problem

These results compare favourably with those of Bostock *et al* (1993), who achieved an average of 93% on the whole training set together with 80% on the generalisation set made up of 92% sensitivity and 68% specificity. The lower percentage values obtained for specificity are most likely due to the imbalance of patterns in the training set, in which 60% of the patterns correspond to malignant tumours. Hence the learning process is biased towards these cases, which in medical terms is a pragmatic approach which would result in the unnecessary removal of a number of benign tumours.

In the same way as with the artificial problems studied earlier in the chapter, the freezing algorithm was applied to this problem with various values of increase and decrease factors. Of these results an increase factor of 2 together with a decrease factor of 1 gave marginally the best results as shown in table 5.35.

	Sensitivity	Specificity	Average Over Whole Set
Training Set	96%	87%	92%
Generalisation Set	90%	62%	76%

Table 5.35 - IF = 2, DF = 1 for Skin Cancer Diagnosis Problem

As can be seen from the above table, there was a slight improvement over the basic method but not the dramatic improvement that was reported for the 4DC problem. In the same way as the encoder8 problem, this problem seems relatively unaffected by changes in the algorithm. Also the generalisation ability of the network seemed to be independent of the number of hidden units present in the network once there were a sufficient number. Two of the individual results that contributed to table 5.35 illustrate this; one trial resulted in a network of 5 hidden units, and the second trial in 11 hidden units. On the generalisation set the 5 hidden unit architecture achieved a sensitivity value of 89% with a specificity value of 62%, whilst the 11 hidden unit architecture achieved a sensitivity value of 93% and specificity 60%. This result is contrary to what would normally be expected, since it is generally accepted that networks with smaller hidden layer structures give superior generalisation performance. Applying the standard back-propagation algorithm to this problem with networks consisting of various hidden layer architectures resulted in the graph shown in figure 5.17 below.

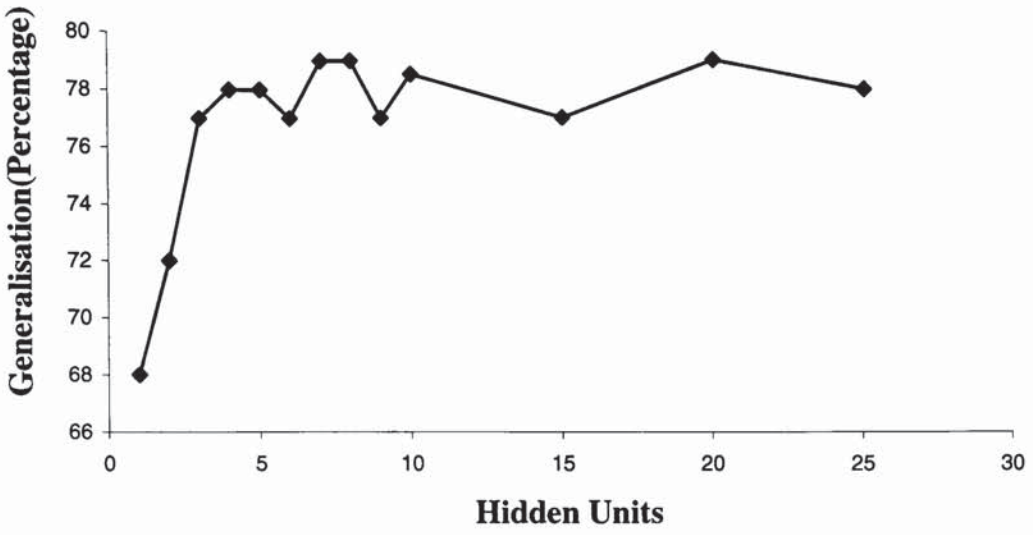


Figure 5.17 - Generalisation Performance of BP on Skin Cancer Diagnosis Problem

As can be seen from the above graph, the level of generalisation quickly rose as the number of hidden units was increased to a sufficient level, but then stabilised and maintained a high level of performance despite the fact that the number of hidden units was raised to a value of 25. This is contrary to what would normally be expected in that there was no drop-off in generalisation ability as is commonly shown in the literature when the number of hidden units is raised beyond the minimum required to any great degree (for example Denker *et al* (1987)). However there are some cases in the literature that have similar properties, Sietsma & Dow (1991) report that in training a network to detect sine wave frequency, *'...in some circumstances networks with many hidden units generalise better than networks with few hidden units'.*

In attempting to improve performance further, the techniques of sigmoid prime and atanh modification were employed in the same way as with the artificial problems. The sigmoid prime modification had a large detrimental effect on performance, causing the majority of trials to create very large hidden layer structures that produced final error values of large magnitude. However, the atanh modification again produced an improvement in performance, as shown in table 5.36.

	Sensitivity	Specificity	Average Over Whole Set
Training Set	95%	87%	91%
Generalisation Set	91%	65%	78%

Table 5.36 - Atanh Modification for Skin Cancer Diagnosis Problem

The final number of hidden units in the results in this table were reasonably large, in the majority of cases being around 12. It was felt that an improvement in generalisation, albeit small, could be gained by reducing this value. An attempt was made to make the termination condition more relaxed by reducing incrementally from 50% down to around 35%, but this only improved the generalisation performance by fractions of a percentage point. An alternative strategy was employed to attain the same ends by increasing the periods between unit additions from 100 to 200 iterations. This gives the network a greater chance to settle down and determine whether a further unit should be added to the network, thus potentially reducing the size of the final architecture. It was felt that this was an appropriate measure in this case as the network takes around 20-30,000 iterations to converge, and thus takes longer to respond to changes. The results of this modification are shown in table 5.37 below.

	Sensitivity	Specificity	Average Over Whole Set
Training Set	96%	85%	91%
Generalisation Set	92%	65%	79%

Table 5.37 - Increased Decision Window on Skin Cancer Diagnosis Problem

As can be seen from table 5.37, this modification improved the generalisation performance slightly by generating networks of around 5 hidden units. Further lengthening of the

decision window did not improve the results further, causing a slight degradation in specificity as the average size of final architecture was reduced.

In general the algorithm performed well on this problem, producing comparable results to that achieved by Bostock *et al* (1993), whose marginally better results were produced by a constructive algorithm that combines the method of Ash (1988) with a period of network freezing after the addition of each new unit.

5.5.2 Drug Design

When designing a drug, a medicinal chemist must determine its pharmacological activity, that is to say the fraction of the drug that will reach the required receptor in the body (Harget & Ellis, 1989). Two factors which are good predictors of drug activity are the partition coefficient and aqueous solubility of the drug, as they are indicators of the absorption, transport and distribution of the drug to the active site in the biological system (Harget & Bodor, 1992). Both of these factors may be determined experimentally, but this is not always appropriate and, in the case of yet unsynthesised drugs, is impossible.

Harget & Bodor (1992) state that a reliable theoretical approach to determining these factors would be beneficial to the medicinal chemist, in that unpromising candidate compounds could be ignored and effort concentrated on the synthesis of those with the greatest theoretical potential. Theoretical methods have been developed, but have been shown to be unreliable in certain cases. An example of this is the group contribution method proposed by Leo *et al* (1971), which has been shown to be unreliable in the design of steroids (Yalkowsky & Morozowich, 1980).

Harget & Bodor (1992) applied the back-propagation algorithm to the problem of determining these factors, concentrating on the partition coefficient. The training set

consisted of 302 compounds made up of a wide variety of organic molecules. Seventeen molecular properties were calculated for each compound, including charge, molecular surface area, volume and the number of carbon atoms. These factors were normalised to real values lying between 0 and 1, as were the experimentally derived partition coefficients. Harget & Bodor (1992) report that applying back-propagation with various single hidden layer architectures gave rise to an optimum value of 3 hidden units in terms of the performance on a previously unseen test set consisting of 21 compounds. The results were consistent with the experimental error and comparable with the results obtained using regression analysis, hence showing that neural networks could form the basis of a useful tool in this area of drug design.

Whilst these results were promising, a great deal of processing had to be done to determine the optimum architecture, as each run required approximately 30,000 presentations of the large training set and hence consumed a large amount of time. It was therefore felt that a dynamic approach could be beneficial as it removes the need for exhaustive testing of many different architectures. This problem has an increased level of difficulty over the previously discussed skin cancer diagnosis problem in that it is larger and has both real valued inputs and outputs. Also, rather than producing a category as an output, the value produced is an analogue value. As noted above the training set consists of 302 patterns, made up of 17 input values with one output representing the partition coefficient. The generalisation set is made up of 21 patterns of the same format. Harget & Bodor (1992) showed that the best performance on this problem using the standard back-propagation algorithm in terms of generalisation was achieved using a three hidden unit architecture, with improvement on the training set as architectures of increased size were employed. It was also reported that performance was relatively insensitive to the initial weight range used. The results presented here used initial weights in the range $0 < W_{i,j} < 1$, with a learning factor of 0.25 and momentum of 0.9.

Due to the length of time taken for convergence, only ten trials were performed for each variation in the algorithm. The network was trained to an error of 0.1 which typically takes around 30,000 iterations for this problem. Applying the basic algorithm to this problem produced the results shown in table 5.38 below.

Average Number of Hidden Units	Percentage of Convergent Trials	Percentage of Correct Classifications of Training Set	Percentage of Correct Classifications of Generalisation set
10	10%	85%	67%

Table 5.38 - Basic Method Applied to Drug Design Problem

The table shows the average final number of hidden units, averaged over the number of trials that were not forced to terminate the addition process; this number is shown as a percentage of the total number of trials. The main results of performance on the training and generalisation sets was calculated in the following way. If the output obtained was within 10% of the desired value, then the output was deemed to be correct; the figures in the table indicate the percentage of correct patterns. In this case only one of the ten trials terminated the addition of hidden units, resulting in a network of ten hidden units. Harget & Bodor (1992) report that the best generalisation results they obtained were gained using a network of 3 hidden units. The generalisation for the 10 hidden unit case shown in the above table is very poor, which was also shown by Harget & Bodor (1992), who showed that the generalisation performance of the networks studied steadily declined as the number of hidden units grew above a 3 unit structure.

In the same way as discussed earlier, the condition that determines whether to terminate the further addition of units was relaxed in order to reduce the size of the final architecture. Reducing from a value of 50% to 45% gave identical results to those shown in table 5.38, with a value of 40% giving the results shown in table 5.39 below.

Average Number of Hidden Units	Percentage of Convergent Trials	Percentage of Correct Classifications of Training Set	Percentage of Correct Classifications of Generalisation set
12	30%	90%	62%

Table 5.39 - Relaxed Termination Condition of 40% Applied to Drug Design Problem

Whilst an increased number of trials did not continuously add hidden units, the results remained poor. Relaxing the condition further had the tendency not to add any hidden units at all because a common factor in all the problems studied is an initial surge in improvement in the early stages of learning because of the randomness of the initial point in error space. If the termination condition is over relaxed then this initial activity can be sufficient to terminate addition immediately.

In order to improve these results, the alternative strategy of increasing the decision window was implemented, increasing initially to a value of 200 and then increasing further to 400, for which results are shown in table 5.40.

Average Number of Hidden Units	Percentage of Convergent Trials	Percentage of Correct Classifications of Training Set	Percentage of Correct Classifications of Generalisation set
7	50%	89%	69%

Table 5.40 - Decision Window of 400 Applied to Drug Design Problem

The average value of the final number of hidden units was decreased, resulting in a corresponding improvement in generalisation performance. Increasing the size of the

decision window further however did not improve the results significantly, and beyond a certain point, around 800, worsened performance.

Applying the freezing method followed a similar pattern to the basic method, the best results being obtained for an increase factor of 2 and a decrease factor of 2 together with a decision window of 400. Only 4 of the trials self-terminated with an average final architecture of 7 hidden units. In line with previous problems, the sigmoid prime modification had a detrimental effect on performance. Applying the atanh modification, however, improved the results, particularly in the case of a decision window of 400.

Average Number of Hidden Units	Percentage of Convergent Trials	Percentage of Correct Classifications of Training Set	Percentage of Correct Classifications of Generalisation set
5	60%	85%	78%

Table 5.41 - Atanh Modification Applied to Drug Design Problem

Whilst only 60% of the trials self terminated the addition process, the algorithm gave acceptable results, giving a high level of generalisation, and further showing that the algorithm is applicable to complex real-world problems.

Chapter 6

Conclusions

6.1 Introduction

The work presented here has shown that a viable constructive algorithm can be based on the errors associated with the individual patterns in the training set. Our algorithm is based on the back-propagation learning algorithm and constructs a single layer of hidden units to avoid the large propagation delays associated with multi-layer architectures. Initially utilising a hidden layer of one hidden unit, the algorithm constructs a single hidden layer resulting, in the majority of cases, in networks of minimal or near minimal hidden layer structure.

Initially, the back-propagation algorithm was applied to three binary test problems, as a basis of comparison with the subsequently developed constructive algorithm. The basic dynamic method was applied to these problems together with modifications which were applied to improve the performance of the algorithm. The algorithm was found to be successfully applicable to small or large, binary or real-valued problems.

6.2 Using Individual Pattern Error as a Basis for Unit Insertion

Chapter 4 outlined the basis of a method of using the 'see-saw' effect between individual patterns in the training set as the basis of a constructive algorithm. This work showed the direction of change in error between consecutive iterations to be predominantly an increase for networks with insufficient hidden units. It was shown that as the number of hidden

units is increased to an acceptable number to solve the given problem, the error tends to decrease between consecutive iterations for the majority of patterns in the training set.

In order to obtain a more immediate indication as to the effect on each pattern caused by the remaining patterns in the training set, the algorithm was modified. Rather than comparing the errors for each pattern between consecutive iterations, the error for each pattern is measured immediately after a weight update, and compared to the error recorded immediately before the next update. This directly measures the trade-off made to reduce the error on some patterns at the expense of others. It was found that if the network has sufficient hidden units, then updating the weights after presenting each pattern does not adversely affect the contribution of the remaining patterns in the training set. Whereas in networks with an inadequate hidden layer structure, the error tends to increase for the majority of patterns between consecutive iterations.

Incorporating this into the standard back-propagation algorithm was achieved by adding a further hidden unit into a single hidden layer if the total number of error increases did not fall below 50% in two consecutive 10 iteration periods within a 100 iteration decision window. The algorithm was shown to be relatively insensitive to the value of these parameters for the three binary problems, however, in the case of the real-world problems studied, the decision window was extended to enable the network to settle before a decision was made to increment the hidden layer.

6.3 Application of the Basic Algorithm

Applying the basic algorithm to the binary test problems produced promising results with all the trials converging and the majority producing minimal architectures for both the encoder8 and 4DC problems. One interesting finding was that the algorithm built minimal hidden layer architectures for the 4DC problem in almost 50% of the trials, all of which

converged. Convergence to a solution was only attained 3 times from 20 trials applying standard back-propagation to a network with the minimal hidden layer architecture. This implies that the process of building the hidden layer architecture in some way aided the convergence process. This was illustrated by performing trials in which the whole network structure was re-initialised on addition of a new hidden unit. None of the trials that resulted in a minimal architecture for the 4DC problem achieved convergence using this method. These results led us to conclude that despite an insufficient number of hidden units, the early phase of learning makes a positive contribution to the learning process as a whole. We believe that this is due to the network partially solving the problem, despite there being insufficient hidden units to model the function completely. Another example of this positive contribution were those cases in which the algorithm achieved convergence for the encoder8 problem with a hidden layer of two units, a result unattainable with standard back-propagation.

Varying the range of the random weights and thresholds of units inserted into the network did not result in a method that could be said to have significantly improved performance, but did raise some interesting issues. For example, the encoder8 problem was shown to be insensitive to variations in the range of added weights and thresholds, altering which produced little variation. However, it was shown that careful selection of threshold values could cause a significant improvement in performance. The encoder8 problem was solved with only 2 hidden units, which is a result unattainable with standard back-propagation. This only occurred when new units were added with relatively large weights and thresholds, which had the overall effect of causing the results to be erratic. These results varied from this minimal extreme of 2 hidden units to trials which resulted in architectures containing far too many hidden units. This would imply, that in the minimal case, the large weights pushed the network into a minimum deep enough to cause the cessation of unit addition and forcing convergence, whilst in other cases taking the network far from a solution which was only redressed on the addition of further hidden units. In contrast, adjusting the range of weights for additional units in the case of the 4DC problem resulted

in a wide variation in performance. None of these weight ranges improved on the initial experiment in which small random weights were inserted. The parity6 problem also remained unaffected by variations in the weight ranges, producing inconsistent results which proved difficult to remedy.

The basic form of the algorithm was shown not to perform as well on the complex real-world problems and parity6. However improvement was attained in subsequent experiments, through modification of the basic algorithm.

6.4 The Freezing Modification

Introducing the idea of freezing the existing network structure was implemented to enable a new unit to assume a useful role in the learning process given the constraints provided by the existing network structure. However, the most positive results were obtained by increasing the learning rate of both the existing network and the new unit for a short period. It is felt that this rather surprising result is due to condensing the detrimental interactions between hidden units into a very intense period of learning. This intense period is due to the temporarily large learning rate, which pushes the network towards a solution quickly. Therefore, during the period of the next decision window when the learning rates are returned to their normal level, these interactions have settled down for networks of sufficient units, and hence a decision to add further unnecessary units is less likely to occur.

Improvement was also achieved in the case of the parity6 problem using this technique, when combined with an alternative initial weight range. Removing the area around the origin in weight space had the effect of stabilising the results, which made the task of improvement more simple. This improvement also necessitated the alteration of the condition that determines the point at which no further hidden units are inserted into the

network. Whilst accepting that a single termination condition is unlikely to be applicable to all problem domains, the requirement to alter the condition is not desirable as it presents a further variable element that must be considered. One possibility that was considered was to gradually decrease the probability of a further unit being inserted. This would have the advantage of being dynamic and hence require no 'user intervention'. In practice it would be difficult to implement over a wide range of problems, as the requirement for hidden units varies so widely. For example a problem requiring 2 hidden units would require a small probability of adding the third hidden unit, whereas a problem requiring 10 hidden units would require that many more units were added beyond this point. The alternative employed for the real-valued problems, was to vary the size of the decision window. Whilst again being a value that must have a user defined size, it is a relatively simple value to roughly estimate given a problem's size and complexity.

6.5 Atanh and Sigmoid Prime Modification

In general, modification of the sigmoid prime function did not have a beneficial effect on performance. This was felt to be because it affects all the weights in the network throughout the learning process, causing changes of a greater magnitude than are usually required. This global effect, whilst having been shown to be beneficial in terms of speed for standard back-propagation, has a disruptive influence on the constructive process. This is because this method measures, and utilises, the relationship between individual pattern errors which are disturbed by this modification.

Taking the atanh of the discrepancy between the actual and desired values also distorts the errors caused by particular patterns in the training set, but it does so selectively. That is to say, only those patterns that produce large errors are affected. As networks with insufficient hidden layer structure are prone to producing large errors, the atanh modification has a large effect and emphasises the network's inability to correctly

categorise the whole of the training set. In the early stages of the learning process, this has the effect of causing further hidden units to be introduced, whereas once sufficient hidden units are present, its effect is reduced as the errors diminish. This focused error emphasis method produced the best results for all the problems studied in terms of the final number of hidden units produced and also in terms of consistency of results.

6.6 Future Work

In terms of this particular algorithm, the major area of further investigation could be carried out in introducing a further element of dynamic behaviour to cope with different problem domains. Despite the fact that the algorithm was shown to function on a variety of problems, this was only achieved with some manipulation of elements of the algorithm. One approach to reduce this user intervention would be for the algorithm to dynamically reduce its decision window as learning proceeds in order to gradually reduce the chance of further units being added. This would be especially useful for the larger problems, that have been shown to tend to add too many units if the decision window is of insufficient size.

Another area of investigation would be to add in varying numbers of hidden units depending on the degree to which the error increases were in the majority. This should in general increase the speed of attaining the desired architecture, but may in practice have to be coupled with a larger decision window after an addition of multiple units to allow the network to stabilise.

In general, it is felt that dynamic architectures conform to the overall ethos of neural networks and can be useful for reducing the workload, especially when a new function of unknown complexity must be modelled. However there remain many unknown variables in the network, upon which an approach such as genetic algorithms could be employed to

gain an approximate solution and hence narrow the parameter choice. A technique that provided an approximate solution would be a useful partner to the dynamic algorithm.

References

Aarts, E.H.L. & Korst, J.H.M. (1989), "Computations in Massively Parallel Networks based on the Boltzmann Machine: a Review", *Parallel Computing*, **9**, pp. 129-145.

Ackley, D.H., Hinton, G.E. & Sejnowski, T.J. (1985), "A Learning Algorithm for Boltzmann Machines", *Cognitive Science*, **9**(1), pp. 147-169.

Ahmad, S. & Tesauro, G. (1988), "Scaling and Generalisation in Neural Networks: a Case Study", in Hinton, G.E., Sejnowski, T.J. & Touretzky, D.S. (eds.), *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo, CA: Morgan Kaufmann, pp. 3-10.

Aleksander, I. (1988), "Neural Computing: Ideas from the brain", *Computer Bulletin*, **4**(1), pp. 14-15.

Ash, T. (1989), "Dynamic Node Creation in Back-Propagation Networks", ICS Report 8901, Institute for Cognitive Science, University of California, San Diego, La Jolla, California.

Baum, E.B. (1989), "A Proposal for a More Powerful Learning Algorithm", *Neural Computation*, **1**(2), pp. 201-207.

Baum, E.B. & Haussler, D. (1989), "What Size Net Gives Valid Generalisation", in Touretzky, D.S.(ed), *Advances in Neural Information Processing Systems*, **1**, San Mateo, California: Morgan Kaufmann, pp. 81-90.

Becker, S. & Le Cun, Y. (1988), "Improving the Convergence of Back-Propagation Learning with Second Order Methods", in Hinton, G.E., Sejnowski, T.J. & Touretzky, D.S. (eds.), *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo, CA: Morgan Kaufmann, pp. 29-37.

Bedworth, M.D. & Lowe, D. (1988), "Fault Tolerance in Multi-Layer Perceptrons: A Preliminary Study", SP4 Research Note 4, Pattern Processing and Machine Intelligence Division, RSRE, Malvern.

Bostock, R.T.J., Claridge, E., Harget, A.J. & Hall, P.N. (1993), "Towards a Neural Network Based System for Skin Cancer Diagnosis", to appear in *Proceedings of the Third IEE International Conference on Artificial Neural Networks.*

Bounds, D.G., Lloyd, P.L. & Mathew, B.G. (1990), "A Comparison of Neural Network and Other Pattern Recognition Approaches to the Diagnosis of Low Back Disorders", *Neural Networks*, **3**(5), pp. 583-591.

Brady, M.L., Raghavan, R. & Slawney, J. (1989), "Back-Propagation Fails to Seperate Where Perceptrons Succeed", *IEEE Transactions on Circuits and Systems*, **36**(5), pp. 665-674.

Braitenberg, V. (1973), "*On the texture of brains*", Springer-Verlag, New York.

Broomhead, D. & Lowe, D. (1988), "Radial Basis Functions, Multi-variable Functional Interpolation and Adaptive Networks", **2**(3), pp. 269-303.

Bruck, J. & Goodman, J.W. (1990), "On the Power of Neural Networks for Solving Hard Problems", *Journal of Complexity*, **6**(2), pp. 129-135.

Carpenter, G.A. & Grossberg, S. (1988), "The ART of Adaptive Pattern Recognition by a Self Organising Neural Network", *IEEE Computer*, **21**(3), pp. 77-88.

Cater, J.P. (1987), "Successfully Using Peak Learning Rates of 10 (and greater) in Back-propagation Networks with the Heuristic Learning Algorithm", *Proceedings of the First IEEE International Conference on Neural Networks*, **1**, San Diego: IEEE, pp. 645-651.

Chan, L.W. & Fallside, F. (1987), "An Adaptive Training Algorithm for Back-Propagation Networks", *Computer Speech and Language*, **2** (3), pp. 202-218.

Chauvin, Y. (1989), "A Back-Propagation Algorithm with Optimal Use of Hidden Units", in Touretzky, D.S.(ed), *Advances in Neural Information Processing Systems*, **1**, San Mateo, California: Morgan Kaufmann, pp. 519-526.

Chiu, W.C. & Hines, E.L. (1991), "A Rule-based Dynamic Back-propagation (DBP) Network", *Proceedings of the Second IEE International Conference on Artificial Neural Networks*, London: IEE, pp. 170-174.

Claridge, E., Hall, P.N., Keefe, M. & Allen, J.P. (1992), "Shape Analysis for the Classification of Malignant Melanoma", *Journal of Biomedical Engineering*, **14**, pp. 229-234.

Clark, A. (1987), "Cognitive Science meets the Biological Mind", *New Scientist*, **116**(1581), pp. 36-38.

Concar, D. (1992), "The Resistible Rise of Skin Cancer", *New Scientist*, **134**(1821), pp. 23-28.

Cornana, A., Marches, I.M., Martini, C. & Ridella, S. (1987), "Minimising Multimodal Functions of Continuous Variables with the 'Simulated Annealing' Algorithm", *ACM Transactions on Mathematical Software*, **13**(3), pp. 262-280.

Cybenko, G. (1990), "Mathematical Problems in Neural Computing", CSRD Rpt. No. 905, Center for Supercomputing Research and Development, University of Illinois, Urbana, USA.

Cybenko, G. (1990a), "Complexity Theory of Neural Networks and Classification", Center for Supercomputing Research and Development, University of Illinois, Urbana, USA.

Dahl, E.D. (1987), "Accelerated Learning using the Generalised Delta Rule", *Proceedings of the First IEEE International Conference on Neural Networks*, **2**, San Diego: IEEE, pp. 523-530.

Denker, J., Schwartz, D., Wittner, B., Solla, S., Howard, R. & Jackel, L. (1987), "Large Automatic Learning, Rule Extraction and Generalisation", *Complex Systems*, **1**(6), pp. 877-922.

Eliot, L.B. (1988), "Predicting the Near-term future of Neural Networks", *IEEE Expert*, **3**(1), pp. 82-83.

Ellison, D. (1988), "The Perceptron and AI: A New Old way Forward", *Computer Bulletin*, **4**(1), pp. 27-30.

Fahlman, S.E. (1988), "Faster-Learning Variations on Back-Propagation: An Empirical Study", *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo, California: Morgan Kaufmann, pp. 38-51.

Fahlman, S.E. & Hinton, G.E. (1987), "Connectionist Architectures for Artificial Intelligence", *Computer*, **20**(1), pp. 100-109.

Fahlman, S.E. & Lebiere, C. (1990), "The Cascade-Correlation Learning Architecture", Technical Report TR-CMU-CS-90-100, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Farley, B.G. & Clark, W.A. (1954), "Simulation of Self-organising Systems by Digital Computer", *Institute of Radio Engineers - Transactions of Professional Group of Information theory*, **PGIT - 4**, pp. 76-54.

Feldman, J.A. (1985), "Connectionist Models and Parallelism in High Level Vision", *Computer Vision, Graphics and Image Processing*, **31**(2), pp. 178-200.

Feldman, J.A. & Ballard, D.H. (1982), "Connectionist Models and their Properties", *Cognitive Science*, **6**, pp. 205-254.

Forrest, B.M., Roweth, D., Stroud, N., Wallace, D.J. & Wilson, G.V. (1987), "Implementing Neural Network Models on Parallel Computers", *The Computer Journal*, **30**(5), pp. 413-419.

Frean, M. (1990), "The Upstart Algorithm: a Method for Constructing and Training Feed-Forward Neural Networks", Edinburgh Physics Department Pre-print No.89/469, Edingurgh University, Edinburgh, Scotland.

Gallant, S.I. (1986), "Optimal Linear Discriminants", *Proceedings of the 8th International Conference on Pattern Recognition, Paris*, San Diego: IEEE, pp. 849-851.

Gallant, S.I. (1990), "Perceptron-Based Learning Algorithms", *IEEE Transactions on Neural Networks*, **1**(2), pp. 179-191.

Gevins, A.S. & Morgan, N.H. (1988), "Applications of Neural-Network (NN) Signal Processing in Brain Research", *IEEE Transactions on Acoustics, Speech and Signal Processing*, **36**(7), pp. 1152-1161.

Girosi, F. & Poggio, T. (1989), "Representation Properties of Networks: Kolmogorov's Theorem Is Irrelevant", *Neural Computation*, **1**(4), pp. 465-469.

Gorman, R.P. & Sejnowski, T.J. (1988), "Learned Classification of Sonar Targets Using a Massively Parallel Network", *IEEE Transactions on Acoustics, Speech and Signal Processing*, **36**(7), pp. 1135-1140.

Gorman, R.P. & Sejnowski, T.J. (1988a), "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets", *Neural Networks*, **1**(1), pp. 75-89.

Graf, H.P., Jackel, L.D. & Hubbard, W.E. (1988), "VLSI Implementation of a Neural Network Model", *IEEE Computer*, **21**(3), pp. 41-49.

Gramss, T. (1991), "Fast Algorithms to find Invariant Features for a Word Recognising Neural Net", *Proceedings of the Second IEE International Conference on Artificial Neural Networks*, London: IEE, pp. 180-184.

Greenough, W.T. & Bailey, C.H. (1988), "The Anatomy of a Memory: Convergence of Results Across a Diversity of Tests", *Trends in Neuroscience*, **11**, pp. 142-147.

Gutierrez, M., Wang, J. & Grondin, R.O. (1989), "Estimating Learning Parameters for Two-Layer Perceptrons", *The International Journal of Neural Networks: Research and Applications*, **1**(3), pp. 160-164.

Hanson, S.J. (1990), "Meiosis Networks", in Touretzky, D.S.(ed), *Advances in Neural Information Processing Systems*, **2**, San Mateo, California: Morgan Kaufmann, pp. 533-541.

Hanson, S.J. & Pratt, L.Y. (1989), "Comparing Biases for Minimal Network Construction with Back-Propagation", in Touretzky, D.S.(ed), *Advances in Neural Information Processing Systems*, **1**, San Mateo, California: Morgan Kaufmann, pp. 177-185.

Harget, A.J. & Bodor, N. (1992), "Computer-Aided Drug Design: A Neural Network Approach", in Vogt, F.H.(ed), *Personal Computers and Intelligent Systems, Information Processing Volume III*, North-Holland: Elsevier Science Publishers, pp. 252-258.

Harget, A.J. & Ellis, D.S. (1989), "Classification of Steroid Partition Coefficients by a Pattern Recognition Technique", *Croatica Chemica Acta*, **62**(3), pp. 449-458.

Hartley, R. & Szu, H. (1987), "A Comparison of the Computational Power of Neural Network Models", *Proceedings of the First IEEE International Conference on Neural Networks*, **3**, San Diego: IEEE, pp. 15-22.

Hebb, D.O. (1949), *"The Organisation of Behaviour"*, New York: Wiley.

Hecht-Nielsen, R. (1987), "Kolmogorov's Mapping Neural Network Existence Theorem", *Proceedings of the First IEEE International Conference on Neural Networks*, **3**, San Diego: IEEE, pp. 11-13.

Hinton, G.E. (1985), "Learning in Parallel Networks", *Byte*, **10**(4), pp. 265-273.

Hinton, G.E. (1989), "Connectionist Learning Procedures", *Artificial Intelligence*, **40**(2), pp. 185-234.

Hinton, G.E. & Sejnowski, T.J. (1986), "Learning and Relearning in Boltzmann Machines", in Rumelhart, D.E. & McClelland, J.L. (eds), *Parallel Distributed Processing - Exploration in the Microstructure of Cognition*, **1**, Cambridge, Massachusetts: MIT Press, pp. 282-318.

Hirose, Y., Yamashita, K. & Hijiya, S. (1991), "Back-Propagation Algorithm which Varies the Number of Hidden Units", *Neural Networks*, **4**(1), pp. 61-66.

Honavar, V. & Uhr, L. (1988), "A Network of Neuron-Like Units that Learns to Perceive by Generation as well as Reweighting of it's Links", in Hinton, G.E., Sejnowski, T.J. & Touretzky, D.S. (eds.), *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo: Morgan Kaufmann, pp. 472-484.

Honavar, V. & Uhr, L. (1991), "Generative Learning Structures and Processes for Generalised Connectionist Networks", Technical Report #91-02, Department of Computer Science, Iowa State University, Ames.

Hopfield, J.J. (1982), "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Science (USA)*, **79**, pp. 2554-2558.

Hornik, K., Stinchcombe, M. & White, H. (1989), "Multilayer Feedforward Networks are Universal Approximators", *Neural Networks*, **2**(5), pp. 359-366.

Hoskins, J.C. (1989), "Speeding Up Artificial Neural Networks in the 'Real' World", M.C.C., Software Technology Program, 9390 Research Blvd., Kaleido II Bldg, Austin, Texas, 78758.

Hutchinson, R.A. & Welsh, W.J. (1989), "Comparison of Neural Networks and Conventional Techniques for Feature Location in Facial Images", *Proceedings of the First IEE International Conference on Artificial Neural Networks*, London: IEE, pp. 201-205.

Huyser, K.A. & Horowitz, M.A. (1988), "Generalisation in Connectionist Networks that Realise Boolean Functions", in Hinton, G.E., Sejnowski, T.J. & Touretzky, D.S. (eds.), *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo: Morgan Kaufmann, pp. 191-200.

Jacobs, R.A. (1988), "Increased Rates of Convergence Through Learning Rate Adaption", *Neural Networks*, **1**(4), pp. 295-307.

Judd, S. (1987), "Learning in Networks is Hard", *Proceedings of the First IEEE International Conference on Neural Networks*, **2**, San Diego: IEEE, pp. 685-692.

Kirkpatrick, S., Gelatt, C.D. & Vecchi, M.P. (1983), "Optimisation by Simulated Annealing", *Science*, **220**(4598), pp. 671-680.

Kohonen, T. (1982), "Self-Organised Formation of Topologically Correct Feature Maps", *Biological Cybernetics*, **43**, pp. 59-69.

Kohonen, T. (1988), "An Introduction to Neural Computing", *Neural Networks*, **1**, pp. 3-16.

Kohonen, T. (1988a), "The 'Neural' Phonetic Typewriter", *IEEE Computer*, **21**(3), pp. 11-22.

Kolen, J.F. & Pollack, J.B. (1990), "Backpropagation is Sensitive to Initial Conditions", *Complex Systems*, **4**(3), pp. 269-280.

Kruschke, J.K. (1989), "Distributed Bottlenecks for Improved Generalisation in Back-Propagation Networks", *The International Journal of Neural Networks: Research and Applications*, **1**(3), pp. 187-193.

Lang, K.J. & Whitbrock, M.J. (1988), "Learning to tell Two Spirals Apart", in Hinton, G.E., Sejnowski, T.J. & Touretzky, D.S. (eds.), *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo: Morgan Kaufmann, pp. 52-59.

Le Cun, Y. (1989), "Generalisation and Network Design Strategies", Technical Report CRG-TR-89-4, Department of Comp. Sci., University of Toronto, Ontario, Canada.

Leo, A., Hansch, C. & Elkins, D. (1971), *Chemical Review*, **71**, pp. 525.

Lippman, R.P. (1987), "An Introduction to Computing with Neural Nets", *IEEE Acoustics, Speech & Signal Processing Magazine*, **4**(2), pp. 4-22.

Lowton, A.D. & Harget, A.J. (1989), "On the Performance of a Neural Network Model - the Boltzmann Machine", *Proceedings of the IASTED International Symposium on Applied Informatics*, Anaheim: ACTA Press, pp. 63-66.

Lynch, M.R., Holden, S.B. & Rayner, P.J. (1991), "Complexity Reduction in Volterra Connectionist Networks Using a Self-structuring LMS Algorithm", *Proceedings of the Second IEE International Conference on Artificial Neural Networks*, London: IEE, pp. 44-48.

McCulloch, W.S. & Pitts, W.H. (1943), "A Logical Calculus of the Ideas imminent in Neurons Activity", *Bulletin of Mathematical Biophysics*, **5**, pp. 115-133.

Marshall, S.J. & Harrison, R.F. (1991), "Optimization and Training of Feedforward Neural Networks by Genetic Algorithms", *Proceedings of the Second IEE International Conference on Artificial Neural Networks*, London: IEE, pp. 39-43.

Mezard, M. & Nadal, J. (1989), "Learning in Feedforward Layered Networks: The Tiling Algorithm", *Journal of Physics A: Mathematical and General*, **22**(12), pp. 2191-2203.

Minsky, M.L. & Papert, S.A. (1969), *Perceptrons*, Massachusetts: MIT Press.

Minsky, M.L. & Papert, S.A. (1988), *Perceptrons: an Expanded Edition*, Massachusetts: MIT Press.

Moody, J. & Darken, C.J. (1989), "Fast Learning in Networks of Locally Tuned Processing Units", *Neural Computation*, **1**(2), pp. 281-294.

Mozer, M.C. & Smolensky, P. (1989), "Using Relevance to Reduce Network Size Automatically", *Connection Science*, **1**(1), pp. 3-16.

Nadal, J. (1989), "Study of a Growth Algorithm for a Feedforward Network", *International Journal of Neural Systems*, **1**(1), pp. 55-59.

Owl, W. (1986), *Adding Up and Taking Away*, London: World international Publishing Company.

Parker, D.B. (1987), "Optimal Algorithms for Adaptive Networks: Second Order Back-Propagation, Second Order Direct Propagation and Second Order Hebbian Learning", *Proceedings of the First IEEE International Conference on Neural Networks*, **2**, San Diego: IEEE, pp. 593-600.

Penrose, R. (1989), *The Emperor's New Mind: Concerning Computers, Minds, and the Laws of Physics*, London: Vintage, pp. 187.

Peterson, C. & Anderson, J.R. (1987), "A Mean Field Theory Learning Algorithm for Neural Networks", *Complex Systems*, **1**, pp. 995-1019.

Plaut, D.C., Nowlan, S.J. & Hinton, G.E. (1986), "Experiments on Learning by Back-propagation", Technical Report CMU-CS-86-126, Carnegie-Mellon University, Computer Science Dept., Pitsburgh, P.A.

Reece, M. & Treleavan, P. (1988), "Computing from the Brain", *New Scientist*, **118**(1614), pp. 61-64.

Refenes, A. N. & Vithlani, S. (1991), "Constructive Learning by Specialisation", in Kohonen, T., Makisara, K., Simula, O. & Kangas, J. (eds.), *Artificial Neural Networks*, Holland: Elsevier Science Publishers, pp. 923-929.

Rosenblatt, F. (1962), *Principles of Neurodynamics*, New York: Spartan.

Rujan, P. & Marchand, M. (1989), "A Geometric Approach to Learning in Neural Networks", *Proceedings of the IEEE INNS International Joint Conference on Neural Networks*, **2**, San Diego, California: IEEE, pp. 105-109.

Rumelhart, D.E., Hinton, G.E. & McClelland, J.L. (1986), "A General Framework for Parallel Distributed Processing", in Rumelhart, D.E. & McClelland, J.L. (eds), *Parallel Distributed Processing - Exploration in the Microstructure of Cognition*, **1**, Cambridge, Massachusetts: MIT Press, pp. 45-76.

Rumelhart, D.E, Hinton, G.E. & Williams, R.J. (1986a), "Learning Representations by Back-Propagating Errors", *Nature*, **322**(6008), pp. 533-536.

Rumelhart, D.E., Hinton, G.E. & Williams, R.J. (1986b), "Learning Internal Representations by Error Propagation", in Rumelhart, D.E. & McClelland, J.L. (eds), *Parallel Distributed Processing - Exploration in the Microstructure of Cognition*, **1**, Cambridge, Massachusetts: MIT Press, pp. 319-362.

Rumelhart, D.E. & Zipser, D. (1985), "Feature Discovery by Competitive Learning", *Cognitive Science*, **9**(1), pp. 75-112.

Sankar, A. & Mammone, R.J. (1991), "Optimal Pruning of Neural Tree Networks for Improved Generalisation", CAIP Center and Dept. of Electrical Engineering, Rutgers University, Piscataway, NJ.

Sankar, A. & Mammone, R.J. (1991a), "Speaker Independent Vowel Recognition using Neural Tree Networks", CAIP Center and Dept. of Electrical Engineering, Rutgers University, Piscataway, NJ.

Sartori, M.A. & Antsaklis, P.J. (1991), "A Simple Method to Derive Bounds on the Size and to Train Multilayer Neural Networks", *IEEE Transactions on Neural Networks*, **2**(4), pp. 467-471.

Scalettar, R. & Zee, A. (1988), "Perception of Left and Right by Feed Forward Net", *Biological Cybernetics*, **58**(3), pp. 193-201.

Sejnowski, T.J. & Hinton, G.E. (1987), "Separating Figure from ground with a Boltzmann Machine", in Arib, M.A. & Hanson, S.J. (eds.), *Vision, Brain and Cooperative Computation*, Cambridge, Massachusetts: MIT Press, pp. 703-724.

Sejnowski, T.J. & Rosenberg, C.R. (1987), "Parallel Networks that Learn to Pronounce English Text", *Complex Systems*, **1**, pp. 145-168.

Sietsma, J. & Dow, R.J.F. (1988), "Neural Net Pruning - Why and How", *Proceedings of the IEEE International Conference on Neural Networks*, **1**, pp. 325-333.

Sietsma, J. & Dow, R.J.F. (1991), "Creating Artificial Neural Networks that Generalize", *Neural Networks*, **4**(1), pp. 67-79.

Silva, F.M. & Almeida, L.B (1991), "Speeding-Up Backpropagation by Data Orthonormalization", in Kohonen, T., Makisara, K., Simula, O. & Kangas, J. (eds.), *Artificial Neural Networks*, Holland: Elsevier Science Publishers, pp. 1503-1506.

Smieja, F.J. & Richards, G.D. (1988), "Hard Learning the Easy Way: Backpropagation with Deformation", *Complex Systems*, **2**(6), pp. 671-704.

Smyth, S.G. (1991), "Building New Layers on Multi-layer Perceptrons", *Proceedings of the Second IEE International Conference on Artificial Neural Networks*, London: IEE, pp. 276-279.

Solla, S.A., Levin, E. & Fleisher, M. (1988), "Accelerated Learning in Layered Neural Networks", *Complex Systems*, **2**(6), pp. 625-640.

Stornetta, W.S. & Huberman, B.A. (1987), "An Improved Three-layer Back-propagation Algorithm", *Proceedings of the First IEEE International Conference on Neural Networks*, **2**, San Diego: IEEE, pp. 637-643.

Tank, D.W. & Hopfield, J.J. (1986), "Simple 'Neural' Optimisation Networks:An Analogue/Digital Converter, Signal Decision Circuit, and a Linear Programming Circuit", *IEEE Transactions on Circuits and Systems*, **33**(5), pp. 533-541.

Tank, D.W. & Hopfield, J.J. (1987), "Collective Computations in Neuronlike Circuits", *Scientific American*, **256**(4), pp. 62-71.

Tesauro, G. & Janssens, B. (1988), "Scaling Relationships in Back-propagation Learning", *Complex Systems*, **2**(2), 39-44.

Tesauro, G. & Sejnowski, T.J. (1989), "A Parallel Network that Learns to Play Backgammon", *Artificial Intelligence*, **39**(3), pp. 357-389.

Ticknor, A.J. & Barrett, H.H. (1987), "Optical Implementations in Boltzmann Machines", *Optical Engineering*, **26**, pp. 16-23.

Vogl, T.P., Mangis, J.K., Rigler, A.K., Zink, W.T. & Alkon, D.L. (1988), "Accelerating the Learning of the Back-Propagation Method", *Biological Cybernetics*, **59**(4), pp. 257-263.

Wasserman, P.D. & Schwartz, T. (1988), "Neural Networks Part 2", *IEEE Expert*, **3**(1), pp. 10-15.

Watrous, R.L. (1987), "Learning Algorithms for Connectionist Networks: Applied Gradient Methods of Nonlinear Optimization", *Proceedings of the First IEEE International Conference on Neural Networks*, **1**, San Diego: IEEE, pp. 619-627.

Werbos, P.J. (1974), "*Beyond Regression: New Tools for Production and Analysis in the Behavioral Sciences*", Ph.D. thesis, Harvard University."

Werbos, P.J. (1988), "Generalisation of Backpropagation with Application to a Recurrent Gas Market Model", *Neural Networks*, **1**(4), pp. 339-356.

Widrow, B. & Hoff, M.E. (1960), "Adaptive Switching Circuits", *1960 WESCON Convention Record Part IV*, pp. 96-104.

Wieland, A. & Leighton, R. (1987), "Geometric Analysis of Neural Network Capabilities", *Proceedings of the First IEEE International Conference on Neural Networks*, **3**, San Diego, California: IEEE, pp. 385-392.

Wolpert, D.H. (1990), "A Mathematical Theory of Generalisation: Part 1", *Complex Systems*, **4**, pp. 151-200.

Wolpert, D.H. (1990a), "A Mathematical Theory of Generalisation: Part 2", *Complex Systems*, **4**, pp. 201-249.

Wynne-Jones, M. (1991), "Constructive Algorithms and Pruning: Improving the Multi-Layer Perceptron", Research Initiative in Pattern Recognition, RSRE, St. Andrews Rd, Malvern.

Wynne-Jones, M. (1993), "Node Splitting: A Constructive Algorithm for Feed-Forward Neural Networks", *Neural Computing and Applications*, **1** (1), pp. 17-22.

Yalkowsky, S.H. & Morozowich, W. (1980), in Ariens, E.J.(ed), *Drug Design*, New York: Academic Press, **9**, pp. 121.

Yu, Y. & Simmons, R.F. (1990), "Extra Output Biased Learning", Technical Report AI90-128, Dept. of Computer Science & Artificial Intelligence Laboratory, Taylor Hall 2124, University of Texas at Austin, Austin, Texas.

Yu, Y. & Simmons, R.F. (1990a), "Descending Epsilon in Back-Propagation: A Technique for Better Generalisation", Technical Report AI90-130, Dept. of Computer Science & Artificial Intelligence Laboratory, Taylor Hall 2.124, University of Texas at Austin, Austin, Texas.

Zeidenberg, M. (1987), "Modeling the Brain", *Byte*, **12**(14), pp. 237-246.

Appendix A - Derivation of the BP Learning Algorithm

As noted in section 2.3.1, the error over the training set of input/output pairs is a mean-squared error given by :

$$E = \frac{1}{2}\sum_{c}\sum_{j}(y_{j,c} - d_{j,c})^2 \qquad \text{(A1.1)}$$

In most applications of the algorithm a value x_j is calculated for each unit j using the equation :

$$x_j = \left(\sum_{i} y_i * \omega_{j,i}\right) - \theta_j \qquad \text{(A1.2)}$$

which is then used in a squashing function given by :

$$y_j = (1 + e^{-x_j})^{-1} \qquad \text{(A1.3)}$$

to yield the output for each unit in the network. In order to perform gradient descent in weight space, it is necessary to calculate the partial derivative of E with respect each weight in the network. To derive these partial derivatives, two cases must be considered.

Case 1 - Unit j is an Output Unit

Using the chain rule :

$$\frac{\delta E}{\delta \omega_{j,i}} = \frac{\delta E}{\delta x_j}\frac{\delta x_j}{\delta \omega_{j,i}} \qquad \text{(A1.4)}$$

But :

$$\frac{\delta E}{\delta x_j} = \frac{\delta E}{\delta y_j} \frac{\delta y_j}{\delta x_j} \qquad \text{(A1.5)}$$

From (A1.3) :

$$\frac{\delta y_j}{\delta x_j} = y_j (1 - y_j) \qquad \text{(A1.6)}$$

From (A1.1) :

$$\frac{\delta E}{\delta y_j} = (y_j - d_j) \qquad \text{(A1.7)}$$

Substituting (A1.6) and (A1.7) into (A1.5) gives :

$$\frac{\delta E}{\delta x_j} = y_j (1 - y_j) (y_j - d_j) = \delta_j \qquad \text{(A1.8)}$$

From (A1.2) :

$$\frac{\delta x_j}{\delta \omega_{j,i}} = y_i \qquad \text{(A1.9)}$$

Substituting (A1.8) and (A1.9) into (A1.4) gives :

$$\frac{\delta E}{\delta \omega_{j,i}} = \delta_j y_i \qquad \text{(A1.10)}$$

Which implies :

$$\Delta \omega_{j,i} = -\eta \delta_j y_i \qquad \text{(A1.11)}$$

Case 2 - Unit j is Not an Output Unit

Re-writing equation (A1.4) and using equations (A1.6) and (A1.9) gives :

$$\frac{\delta E}{\delta \omega_{j,i}} = \frac{\delta E}{\delta y_j} y_j (1-y_j) y_i \qquad \text{(A1.12)}$$

In order to calculate $\delta E/\delta y_j$, consider the contribution (z say) of a unit k in the next highest layer and re-write :

$$z = \frac{\delta E}{\delta x_k} \frac{\delta x_k}{\delta y_j} \qquad \text{(A1.13)}$$

$$z = \delta_k \frac{\delta}{\delta y_j} \left(\sum_i y_i \omega_{k,i} \right) \qquad \text{(A1.14)}$$

$$z = \delta_k \omega_{k,j} \qquad \text{(A1.15)}$$

Then considering the contribution of all the units in the next layer gives :

$$\frac{\delta E}{\delta y_j} = \sum_k \delta_k \omega_{k,j} \qquad \text{(A1.16)}$$

Substituting (A1.16) into (A1.12) gives :

$$\frac{\delta E}{\delta \omega_{j,i}} = y_j (1-y_j) y_i \sum_k \delta_k \omega_{k,j} \qquad \text{(A1.17)}$$

But if (as in case 1) we make the substitution :

$$\frac{\delta E}{\delta x_j} = \delta_j \qquad \text{(A1.18)}$$

Then :

$$\frac{\delta E}{\delta \omega_{j,i}} = \delta_j y_i \qquad \text{(A1.19)}$$

And hence :

$$\Delta \omega_{j,i} = -\eta \delta_j y_i \qquad \text{(A1.20)}$$

QED.

Appendix B - Error Increases for 4DC Problem

The figures below show the total number of error increases in each 10 iteration period for the 4DC problem using various hidden layer architectures.

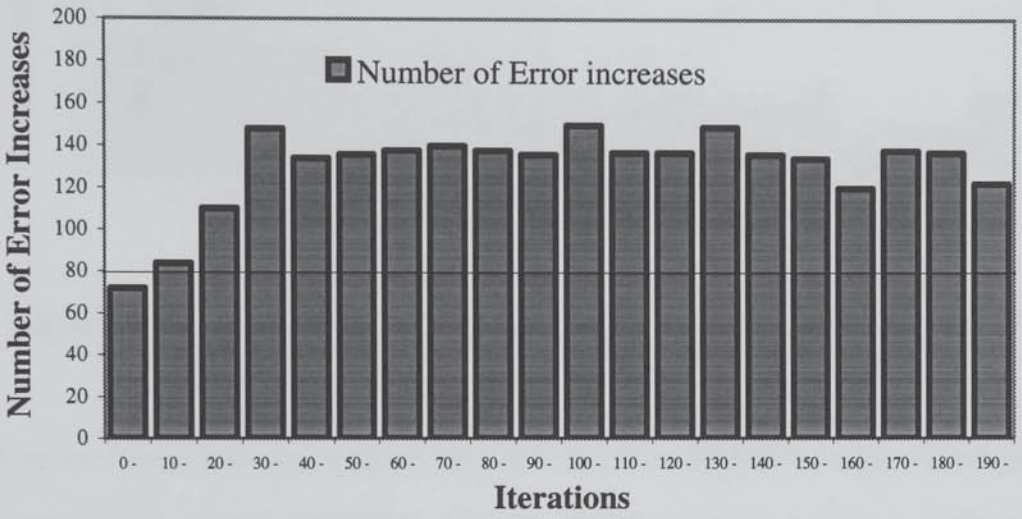


Figure B.1 - Total Number of Error Increases for 4DC with 1 Hidden Unit

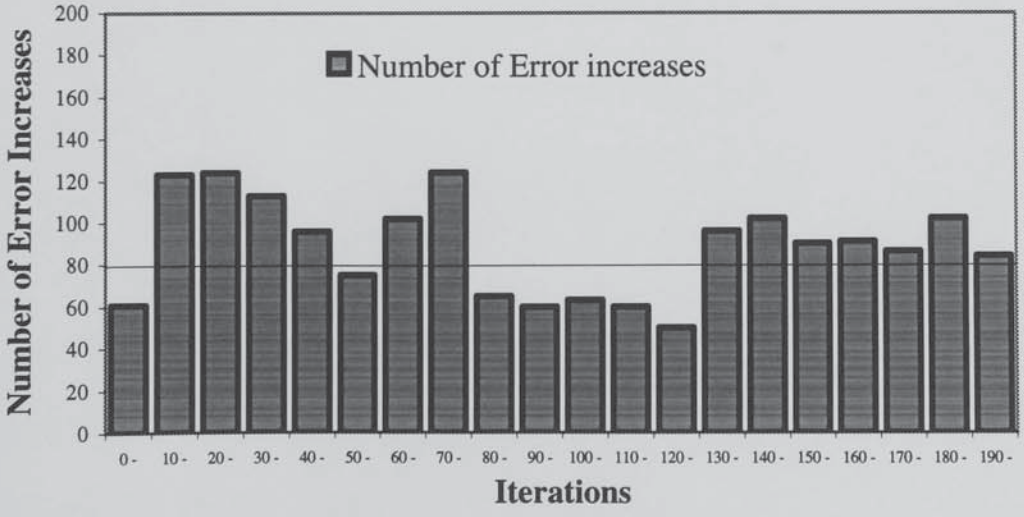


Figure B.2 - Total Number of Error Increases for 4DC with 2 Hidden Units

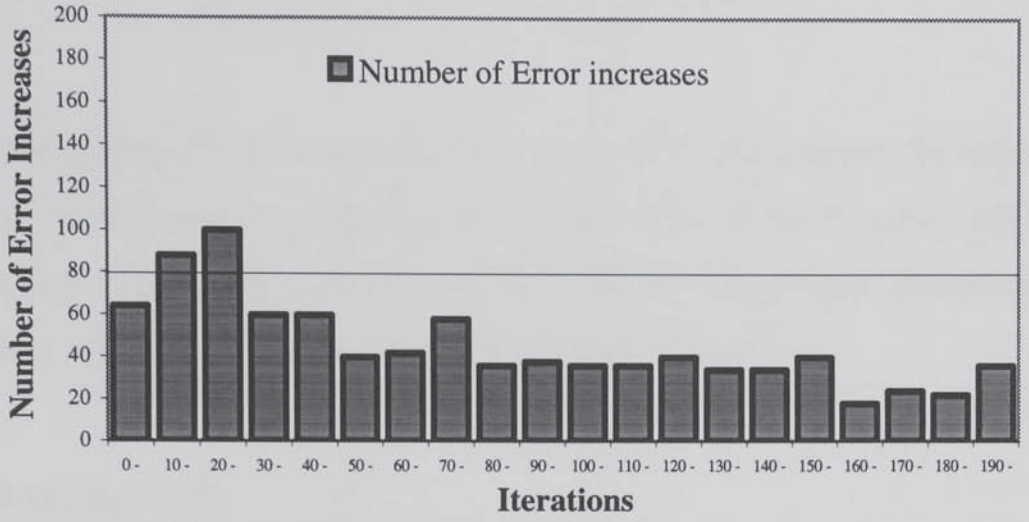


Figure B.3 - Total Number of Error Increases for 4DC with 5 Hidden Units

As can be seen from the figures above, the 4DC problem followed a very similar pattern to the other artificial problems studied, with the number of error increases reducing as the size of the hidden layer architecture was increased.

Appendix C - Program Listing

The program listed below implements the version of the constructive algorithm which includes the ability to freeze the existing weights in the network on the addition of a further unit. The program cycles round a loop reading the required data from external files and hence may perform several trials without user intervention.

```
(* Last update 1/10/90          *)
(* Has capability to add and delete units and to delete weights    *)
(* Writes weights as bytes        *)
(* Writes seed at end of weight file *)
(* Adds new weights and thresholds as -2<w<2  FOR BOTH SETS   *)
(* Has capacity for MULTI RUNS     *)
(* Implements Freeze              *)
(* IF =1 DF=0 Frozen iterations =50*)
(* Check range =400, Check frequency =10*)
(* Limit=10000*)
(* Stops adding if greater than 50% *)
(* Stops after 20 hidden units *)


MODULE BackPropogation;


FROM InOut IMPORT

   Read,WriteCard,ReadCard,WriteLn,WriteString,
   OpenInput,CloseInput,OpenOutput,CloseOutput,ReadInt,
   WriteInt,ReadString,Write;

FROM RealIO IMPORT ReadReal,WriteReal;

FROM SimpleIO IMPORT ReadLn;

FROM MathLib IMPORT

   Exp;

FROM SYSTEM IMPORT CCALL,ADR,TSIZE;

FROM Files IMPORT File,BinTextMode,ReadWriteMode,ReplaceMode,FileState,
          Open,Close,Create;
```

```
FROM Binary IMPORT WriteBytes,ReadBytes;


TYPE MaxRange = [1..256];
    NeuralArray = ARRAY MaxRange OF REAL;
    WeightArray = ARRAY MaxRange OF NeuralArray;
    LayerArray  = ARRAY [1..10] OF CARDINAL;
    DescisionArray = ARRAY MaxRange, MaxRange OF BOOLEAN;
    InputVectors = ARRAY MaxRange OF REAL;
    TargetVectors = ARRAY MaxRange OF REAL;

    EnvironmentalVector = RECORD

                    Input : InputVectors;
                    Target: TargetVectors

                    END;
   NameType = ARRAY MaxRange OF CHAR;

VAR Output,Delta,Activation: NeuralArray;
   Threshold,OldDeltaThreshold,DeltaThreshold: NeuralArray;
   Environment : ARRAY MaxRange OF EnvironmentalVector;
   Weight,OldDeltaWeight,DeltaWeight : WeightArray;
   WeightExists : DescisionArray;
   NumberOfLayers,NumberOfUnits,NumberInTrainingSet : CARDINAL;
   LearningFactor,MomentumFactor : REAL;
   NumberInLayer : LayerArray;
   NumberOfSweepsSoFar : CARDINAL;
   Choice : CHAR;
   Dummy  : CHAR;
   RealDummy : REAL;
   Error,OldError,ErrorSum : REAL;
   Response :CHAR;
   TrialVector : InputVectors;
   ErrorRep    : BOOLEAN;
   FileStream:File;
   Status:FileState;
   FileName:NameType;
   BytesReadSoFar:CARDINAL;
   Seed: CARDINAL;


PROCEDURE CreateWeightMatrix(VAR Seed : CARDINAL);

(*Creates the random initial weights for the network*)
```

Appendix C

```
VAR i,j,Layer,UpperLimit,Total : CARDINAL;
   NegativeFactor : CARDINAL;

BEGIN

  (*Set relevant weights to random values i.e.fully connected net*)

  Total:=0;

  (*Seed the C random number generator*)

  CCALL ( "srand", Seed );

  FOR Layer:=1 TO (NumberOfLayers-1) DO

    UpperLimit:=Total+NumberInLayer[Layer];

    FOR i:=(UpperLimit+1) TO (UpperLimit+NumberInLayer[Layer+1]) DO

      FOR j:=(Total+1) TO UpperLimit DO

        Weight[i,j]:=((FLOAT( CCALL ( "rand" ) )/4294967294.0)*8.0)-2.0;

        WeightExists[i,j] := TRUE;

        OldDeltaWeight[i,j]:=0.0

      END

    END;

    Total:=Total+NumberInLayer[Layer]

  END;

  (*Sets the threshold of each unit to a random value*)

  FOR j:=1 TO NumberOfUnits DO

    Threshold[j]:=((FLOAT( CCALL ( "rand" ) )/4294967294.0)*8.0)-2.0;

    OldDeltaThreshold[j]:=0.0

  END

END CreateWeightMatrix;
```

```
PROCEDURE PromptForFileName;
BEGIN
 WriteLn;
 WriteString('Please key in the name of the weight file');
 WriteLn;WriteLn;
 ReadString(FileName)
END PromptForFileName;

PROCEDURE WriteVectorFile;
(*Writes the environmental vector set to an external file*)
VAR i,j : CARDINAL;
BEGIN
 WriteString('Please key in name of vector file');
 WriteLn;WriteLn;
 OpenOutput('.dat');
 WriteCard(NumberOfLayers,5);
 WriteLn;
 FOR i:=1 TO NumberOfLayers DO
  WriteCard(NumberInLayer[i],5);
  WriteLn
 END;
 WriteCard(NumberInTrainingSet,5);
 WriteLn;
 WriteCard(NumberOfUnits,6);
 WriteLn;
```

```
 FOR j:=1 TO NumberInTrainingSet DO
  WITH Environment[j] DO
   FOR i:=1 TO NumberInLayer[1] DO
    WriteReal(Input[i],32,8);
    WriteLn
   END;
   FOR i:=1 TO NumberInLayer[NumberOfLayers] DO
    WriteReal(Target[i],32,8);
    WriteLn
   END
  END
 END;
 CloseOutput
END WriteVectorFile;

PROCEDURE WriteWeightFile;
(* Writes the current network weights to an external file *)
VAR i,j,Total,UpperLimit,Layer : CARDINAL;
   YesOrNo : CHAR;
BEGIN
 Create(FileStream,FileName,binMode,replace,Status);
 Open(FileStream,FileName,binMode,readWrite,Status);
 WriteBytes(FileStream,ADR(NumberOfLayers),TSIZE(CARDINAL),Status);
 FOR j:=1 TO NumberOfLayers DO
  WriteBytes(FileStream,ADR(NumberInLayer[j]),TSIZE(CARDINAL),Status);
 END;
```

```
WriteBytes(FileStream,ADR(NumberOfUnits),TSIZE(CARDINAL),Status);
Total:=0;
FOR Layer:=1 TO (NumberOfLayers-1) DO
 UpperLimit:=Total+NumberInLayer[Layer];
 FOR i:=(UpperLimit+1) TO (UpperLimit+NumberInLayer[Layer+1]) DO
  WriteBytes(FileStream,ADR(i),TSIZE(CARDINAL),Status);
  FOR j:=(Total+1) TO UpperLimit DO
   WriteBytes(FileStream,ADR(Weight[i,j]),TSIZE(REAL),Status);
   IF WeightExists[i,j] THEN
     YesOrNo := 'y';
     WriteBytes(FileStream,ADR(YesOrNo),TSIZE(CHAR),Status)
   ELSE
     YesOrNo := 'n';
     WriteBytes(FileStream,ADR(YesOrNo),TSIZE(CHAR),Status)
   END
  END
 END;
 Total:=Total+NumberInLayer[Layer];
END;
Total:=0;
FOR Layer:=1 TO (NumberOfLayers-1) DO
 UpperLimit:=Total+NumberInLayer[Layer];
 FOR i:=(UpperLimit+1) TO (UpperLimit+NumberInLayer[Layer+1]) DO
  WriteBytes(FileStream,ADR(i),TSIZE(CARDINAL),Status);
```

```
    FOR j:=(Total+1) TO UpperLimit DO
      WriteBytes(FileStream,ADR(OldDeltaWeight[i,j]),TSIZE(REAL),Status)
    END
   END;
   Total:=Total+NumberInLayer[Layer]
  END;
  FOR j:=1 TO NumberOfUnits DO
   WriteBytes(FileStream,ADR(Threshold[j]),TSIZE(REAL),Status)
  END;
  FOR j:=1 TO NumberOfUnits DO
   WriteBytes(FileStream,ADR(OldDeltaThreshold[j]),TSIZE(REAL),Status)
  END;
  WriteBytes(FileStream,ADR(LearningFactor),TSIZE(REAL),Status);
  WriteBytes(FileStream,ADR(MomentumFactor),TSIZE(REAL),Status);
  WriteBytes(FileStream,ADR(NumberOfSweepsSoFar),TSIZE(CARDINAL),Status);
  WriteBytes(FileStream,ADR(Error),TSIZE(REAL),Status);
  WriteBytes(FileStream,ADR(Seed),TSIZE(CARDINAL),Status);
  Close(FileStream,Status)
END WriteWeightFile;

PROCEDURE WriteWeightFileToScreen;
(* Writes the current network weights to an external file *)
VAR i,j,Total,UpperLimit,Layer : CARDINAL;
BEGIN
 (*OpenOutput('.dat');*)
```

```
WriteCard(NumberOfLayers,8);WriteLn;
FOR j:=1 TO NumberOfLayers DO
 WriteCard(NumberInLayer[j],8);WriteLn
END;
WriteCard(NumberOfUnits,8);WriteLn;
Total:=0;
FOR Layer:=1 TO (NumberOfLayers-1) DO
 UpperLimit:=Total+NumberInLayer[Layer];
 FOR i:=(UpperLimit+1) TO (UpperLimit+NumberInLayer[Layer+1]) DO
  WriteCard(i,8);WriteLn;
  FOR j:=(Total+1) TO UpperLimit DO
   WriteReal(Weight[i,j],32,8);
   WriteLn;
   IF WeightExists[i,j] THEN
    Write('y')
   ELSE Write('n')
   END;
   WriteLn
  END
 END;
 Total:=Total+NumberInLayer[Layer];
END;
Total:=0;
FOR Layer:=1 TO (NumberOfLayers-1) DO
 UpperLimit:=Total+NumberInLayer[Layer];
```

```
  FOR i:=(UpperLimit+1) TO (UpperLimit+NumberInLayer[Layer+1]) DO
   WriteCard(i,8);WriteLn;
   FOR j:=(Total+1) TO UpperLimit DO
    WriteReal(OldDeltaWeight[i,j],32,8);
    WriteLn;
   END
  END;
  Total:=Total+NumberInLayer[Layer]
 END;
 FOR j:=1 TO NumberOfUnits DO
  WriteReal(Threshold[j],32,8);
  WriteLn
 END;
 FOR j:=1 TO NumberOfUnits DO
  WriteReal(OldDeltaThreshold[j],32,8);
  WriteLn
 END;
 WriteReal(LearningFactor,32,8);
 WriteLn;
 WriteReal(MomentumFactor,32,8);
 WriteLn;
 WriteCard(NumberOfSweepsSoFar,8);
 WriteLn;
 WriteReal(Error,32,8);
```

```
 WriteLn;

 WriteCard(Seed,8);

 (*CloseOutput*)

END WriteWeightFileToScreen;


PROCEDURE DisplayWeightFile;

(* Reads in weight file in byte form and writes it out as floating point *)

BEGIN

  PromptForFileName;
  ReadWeightFile;
  WriteWeightFileToScreen

END DisplayWeightFile;


PROCEDURE ReadVectorFile;

(* Reads environmental vectors from external file *)

VAR i,j : CARDINAL;

BEGIN

  WriteString('Please key in the name of the vector file');

  WriteLn;WriteLn;

  OpenInput('.dat');

  ReadCard(NumberOfLayers);

  FOR i:=1 TO NumberOfLayers DO

    ReadCard(NumberInLayer[i])

  END;

  ReadCard(NumberInTrainingSet);

  ReadCard(NumberOfUnits);

  FOR i:=1 TO NumberInTrainingSet DO
```

```
   WITH Environment[i] DO
    FOR j:=1 TO NumberInLayer[1] DO
     ReadReal(Input[j],ErrorRep);
     ReadLn
    END;
    FOR j:=1 TO NumberInLayer[NumberOfLayers] DO
     ReadReal(Target[j],ErrorRep);
     ReadLn
    END
   END
  END;
  CloseInput
END ReadVectorFile;

PROCEDURE ReadWeightFile;
(* Reads network weights from external file *)
VAR i,j,Total,UpperLimit,Layer,Index : CARDINAL;
   YesOrNo : CHAR;
BEGIN
  Open(FileStream,FileName,binMode,readWrite,Status);
  ReadBytes(FileStream,ADR(NumberOfLayers),TSIZE(CARDINAL),
            BytesReadSoFar,Status);
  FOR j:=1 TO NumberOfLayers DO
     ReadBytes(FileStream,ADR(NumberInLayer[j]),TSIZE(CARDINAL),
            BytesReadSoFar,Status);
  END;
```

```
ReadBytes(FileStream,ADR(NumberOfUnits),TSIZE(CARDINAL),
            BytesReadSoFar,Status);

Total:=0;

FOR Layer:=1 TO (NumberOfLayers-1) DO

    UpperLimit:=Total+NumberInLayer[Layer];

    FOR i:=(UpperLimit+1) TO (UpperLimit+NumberInLayer[Layer+1]) DO

        ReadBytes(FileStream,ADR(Index),TSIZE(CARDINAL),
                BytesReadSoFar,Status);

        FOR j:=(Total+1) TO UpperLimit DO

            ReadBytes(FileStream,ADR(Weight[i,j]),TSIZE(REAL),
                BytesReadSoFar,Status);

            ReadBytes(FileStream,ADR(YesOrNo),TSIZE(CHAR),
                BytesReadSoFar,Status);

            IF YesOrNo = 'y' THEN

                WeightExists[i,j] := TRUE

            ELSE WeightExists[i,j] := FALSE

            END

        END

    END;

    Total:=Total+NumberInLayer[Layer]

END;

Total:=0;

FOR Layer:=1 TO (NumberOfLayers-1) DO

    UpperLimit:=Total+NumberInLayer[Layer];

    FOR i:=(UpperLimit+1) TO (UpperLimit+NumberInLayer[Layer+1]) DO

ReadBytes(FileStream,ADR(Index),TSIZE(CARDINAL),BytesReadSoFar,Status);

FOR j:=(Total+1) TO UpperLimit DO
```

```
        ReadBytes(FileStream,ADR(OldDeltaWeight[i,j]),TSIZE(REAL),
                        BytesReadSoFar,Status)

    END

  END;

  Total:=Total+NumberInLayer[Layer]

END;

FOR j:=1 TO NumberOfUnits DO

  ReadBytes(FileStream,ADR(Threshold[j]),TSIZE(REAL),BytesReadSoFar,Status)

END;

FOR j:=1 TO NumberOfUnits DO

  ReadBytes(FileStream,ADR(OldDeltaThreshold[j]),TSIZE(REAL),
                        BytesReadSoFar,Status);

END;

ReadBytes(FileStream,ADR(LearningFactor),TSIZE(REAL),BytesReadSoFar,Status);

 ReadBytes(FileStream,ADR(MomentumFactor),TSIZE(REAL),
            BytesReadSoFar,Status);

 ReadBytes(FileStream,ADR(NumberOfSweepsSoFar),TSIZE(CARDINAL),
        BytesReadSoFar,Status);

  ReadBytes(FileStream,ADR(Error),TSIZE(REAL),BytesReadSoFar,Status);

  ReadBytes(FileStream,ADR(Seed),TSIZE(CARDINAL),BytesReadSoFar,Status);

  Close(FileStream,Status)

END ReadWeightFile;


PROCEDURE CreateVectorFile;

(* Obtains environmental vectors from user *)

VAR i,j : CARDINAL;

BEGIN
```

```
WriteLn;WriteLn;WriteLn;
WriteString('Please key in the number of layers in the net');
WriteLn;
WriteLn;
ReadCard(NumberOfLayers);
WriteLn;

NumberOfUnits := 0;

   FOR i:=1 TO NumberOfLayers DO

     WriteLn;
     WriteString('Please key in the number of units in layer ');
     WriteCard(i,5);
     WriteLn;
     ReadCard(NumberInLayer[i]);
     WriteLn;
     NumberOfUnits:=NumberOfUnits+NumberInLayer[i]

    END;

    WriteLn;
    WriteString('Please key in the number of vectors in the training set');
    WriteLn;WriteLn;
    ReadCard(NumberInTrainingSet);


FOR i:=1 TO NumberInTrainingSet DO

  WITH Environment[i] DO

    WriteLn;WriteLn;
    WriteString('Please key in input vector number ');
    WriteCard(i,4);

    FOR j:=1 TO NumberInLayer[1] DO

      WriteLn;WriteLn;
      WriteString('Please enter element ');
      WriteCard(j,5);
      WriteString(' of input vector number ');
      WriteCard(i,5);
      WriteLn;WriteLn;
      ReadReal(Input[j],ErrorRep);
      ReadLn

    END;
```

```
    FOR j:=1 TO NumberInLayer[NumberOfLayers] DO

      WriteLn;WriteLn;
      WriteString('Please enter element ');
      WriteCard(j,3);
      WriteString(' of the corressponding target vector');
      WriteLn;WriteLn;
      ReadReal(Target[j],ErrorRep);
      ReadLn

    END

  END

 END;

 WriteVectorFile;

END CreateVectorFile;


PROCEDURE CreateWeightFile;

(* Generates a weight file using information supplied by the user *)

VAR i : CARDINAL;

BEGIN

 WriteLn;WriteLn;WriteLn;
 WriteString('Please key in the number of layers in the net');
 WriteLn;
 WriteLn;
 ReadCard(NumberOfLayers);
 WriteLn;

 NumberOfUnits := 0;

  FOR i:=1 TO NumberOfLayers DO

    WriteLn;
    WriteString('Please key in the number of units in layer ');
    WriteCard(i,5);
    WriteLn;
    ReadCard(NumberInLayer[i]);
    WriteLn;
    NumberOfUnits:=NumberOfUnits+NumberInLayer[i]

  END;
```

```
 WriteLn;WriteLn;
 WriteString('Please key in the number of units in the net');
 WriteLn;WriteLn;
 ReadCard(NumberOfUnits);WriteLn;WriteLn;
 WriteString('Please key in a positive integer');
 WriteLn;WriteLn;
 ReadCard(Seed);
 WriteLn;

 CreateWeightMatrix(Seed);

 WriteLn;
 WriteString('Please enter the required learning factor');
 WriteLn;
 ReadReal(LearningFactor,ErrorRep);
 WriteLn;WriteLn;ReadLn;
 WriteString('Please enter the required momentum factor');
 WriteLn;
 ReadReal(MomentumFactor,ErrorRep);
 WriteLn;WriteLn;ReadLn;

 NumberOfSweepsSoFar:=0;

 Error:=0.0;

 PromptForFileName;

 WriteWeightFile

END CreateWeightFile;


PROCEDURE ClampInput (InputVector : InputVectors);

(*Applies an input vector to the input units *)

VAR j : CARDINAL;

BEGIN

  FOR j:=1 TO NumberInLayer[1] DO

    Output[j]:=( InputVector[j] )

  END

END ClampInput;
```

```
PROCEDURE CalcOutput;

(* Given an input, this procedure generates an output vector *)

VAR Total,Count,i,j : CARDINAL;
   Temp : REAL;

BEGIN

  Total:=0;

  FOR Count:= 2 TO NumberOfLayers DO

    Total:= Total + NumberInLayer[Count-1];

    FOR j:= (1+Total) TO (Total+NumberInLayer[Count]) DO

      Temp:=0.0;

      FOR i:=(Total-NumberInLayer[Count-1]+1) TO Total DO

        (* Avoid adding to total if weight does not exist *)

        IF WeightExists[j,i] THEN

          Temp:= Temp+(Output[i]*Weight[j,i])

        END

      END;

      Activation[j]:=Temp + Threshold[j];
      (* Avoids overflow in sigmoid function *)

      IF ABS(Activation[j]) > 75.0 THEN

        IF Activation[j]>0.0 THEN Output[j]:=1.0;

        ELSE Output[j]:=0.0

        END

      ELSE

        Output[j]:=1.0/(1.0+Exp(-Activation[j]))

      END;
```

```
    END

  END

END CalcOutput;


PROCEDURE DisplayOutput;

(* Prints the output of the network to the screen *)

VAR j:CARDINAL;

BEGIN

 FOR j:=1 TO NumberInLayer[NumberOfLayers] DO

   WriteReal(Output[NumberOfUnits-NumberInLayer[NumberOfLayers]+j],32,8);

   WriteLn

 END

END DisplayOutput;


PROCEDURE  WeightAdjustment(CurrEnvNum:CARDINAL);

(* Calculates and makes the weight adjustments after presentation of a pattern *)

VAR Total,j,Layer,k: CARDINAL;
   Sum: REAL;


 PROCEDURE WeightChange(CurrTotal,CurrLayer:CARDINAL);

  (* Actually updates the weights *)

 VAR i,j : CARDINAL;

 BEGIN

   FOR j:= (CurrTotal-NumberInLayer[CurrLayer]+1) TO CurrTotal DO

     FOR i:= (CurrTotal-NumberInLayer[CurrLayer]) TO
            (CurrTotal-NumberInLayer[CurrLayer]-NumberInLayer[CurrLayer-1]
                  +1) BY -1 DO

      (* Avoid weight update if weight does not exist *)
```

```
      IF WeightExists[j,i] THEN

        DeltaWeight[j,i]:=( LearningFactor*Delta[j]*Output[i] )
                    +(MomentumFactor*OldDeltaWeight[j,i]);

        Weight[j,i]:=Weight[j,i]+DeltaWeight[j,i]

      END

     END

    END

  END WeightChange;

BEGIN

  Total:=NumberOfUnits;

  WITH Environment[CurrEnvNum] DO

  FOR j:=(Total-NumberInLayer[NumberOfLayers]+1) TO (NumberOfUnits) DO

    Delta[j] := (Target[j-(Total-NumberInLayer[NumberOfLayers])]-Output[j])*
                    Output[j]*(1.0-Output[j])

  END;

  WeightChange(Total,NumberOfLayers);

  FOR Layer:= (NumberOfLayers-1) TO 2 BY -1 DO

    Total:=Total-NumberInLayer[Layer+1];

    FOR j:=(Total) TO (Total-NumberInLayer[Layer]+1) BY -1 DO

      Sum:=0.0;

      FOR k:=(Total+1) TO (Total+NumberInLayer[Layer+1]) DO

        (* Avoid adding factor of weight that does not exist *)

        IF WeightExists[k,j] THEN

          Sum:= Sum+(Delta[k]*Weight[k,j])

        END
```

```
    END;

    Delta[j]:=Sum*Output[j]*(1.0-Output[j])

  END;

  WeightChange(Total,Layer)

   END

 END;

 FOR j:=1 TO NumberOfUnits DO

  DeltaThreshold[j]:=(LearningFactor*Delta[j])+(MomentumFactor
                *OldDeltaThreshold[j]);

  Threshold[j]:=Threshold[j]+DeltaThreshold[j]

 END

END WeightAdjustment;


PROCEDURE  FrozenWeightAdjustment(CurrEnvNum:CARDINAL);

(* Implements the frozen version of learning *)

CONST IncreaseFactor = 1.0;
    DecreaseFactor = 0.0;

VAR Total,j,Layer,k: CARDINAL;
  Sum: REAL;

 PROCEDURE FrozenWeightChange(CurrTotal,CurrLayer:CARDINAL);

 VAR i,j : CARDINAL;

 BEGIN

  FOR j:= (CurrTotal-NumberInLayer[CurrLayer]+1) TO CurrTotal DO

   FOR i:= (CurrTotal-NumberInLayer[CurrLayer]) TO
          (CurrTotal-NumberInLayer[CurrLayer]-NumberInLayer[CurrLayer-1]
                +1) BY -1 DO

    (* Avoid weight update if weight does not exist *)

    IF WeightExists[j,i] THEN
```

```
        IF (i=NumberOfUnits-NumberInLayer[3]) OR ( j=
                    NumberOfUnits-NumberInLayer[3]) THEN

        DeltaWeight[j,i]:=( IncreaseFactor*LearningFactor*Delta[j]*Output[i])
                  +(MomentumFactor*OldDeltaWeight[j,i]);

        ELSE

        DeltaWeight[j,i]:=( DecreaseFactor*LearningFactor*Delta[j]*Output[i])
                  +(MomentumFactor*OldDeltaWeight[j,i]);

        END;


        Weight[j,i]:=Weight[j,i]+DeltaWeight[j,i]

      END

     END

    END

  END FrozenWeightChange;

BEGIN

  Total:=NumberOfUnits;

  WITH Environment[CurrEnvNum] DO

  FOR j:=(Total-NumberInLayer[NumberOfLayers]+1) TO (NumberOfUnits) DO

    Delta[j] := (Target[j-(Total-NumberInLayer[NumberOfLayers])]-Output[j])*
                  Output[j]*(1.0-Output[j])

  END;

  FrozenWeightChange(Total,NumberOfLayers);

  FOR Layer:= (NumberOfLayers-1) TO 2 BY -1 DO

    Total:=Total-NumberInLayer[Layer+1];

    FOR j:=(Total) TO (Total-NumberInLayer[Layer]+1) BY -1 DO

      Sum:=0.0;

      FOR k:=(Total+1) TO (Total+NumberInLayer[Layer+1]) DO
```

```
      (* Avoid adding factor of weight that does not exist *)
      IF WeightExists[k,j] THEN
        Sum:= Sum+(Delta[k]*Weight[k,j])
      END
    END;
    Delta[j]:=Sum*Output[j]*(1.0-Output[j])
  END;
  FrozenWeightChange(Total,Layer)
 END
 END;
 FOR j:=1 TO NumberOfUnits DO
  IF j=NumberOfUnits-NumberInLayer[3] THEN
   DeltaThreshold[j]:=(IncreaseFactor*LearningFactor*Delta[j])+(MomentumFactor
                *OldDeltaThreshold[j]);
  ELSE
   DeltaThreshold[j]:=(DecreaseFactor*LearningFactor*Delta[j])+(MomentumFactor
                *OldDeltaThreshold[j]);
   END;
  Threshold[j]:=Threshold[j]+DeltaThreshold[j]
 END
END FrozenWeightAdjustment;

PROCEDURE PrintError;
BEGIN
 WriteString('Error = ');
 WriteReal(Error,32,8);
```

```
 WriteString(' after ');
 WriteCard(NumberOfSweepsSoFar,8);
 WriteString(' iterations.');
 WriteLn
END PrintError;
PROCEDURE RemoveWeight(IndexJ,IndexI : CARDINAL);
BEGIN
  WeightExists[IndexJ,IndexI] := FALSE;
  Weight[IndexJ,IndexI] := 0.0;
  OldDeltaWeight[IndexJ,IndexI] := 0.0
END RemoveWeight;

PROCEDURE AssignValuesForNewUnit ( UnitNumber : CARDINAL);
VAR Index : CARDINAL;
BEGIN

  CCALL ( "srand", Seed );
  FOR Index:= 1 TO NumberInLayer[1] DO
    Weight[UnitNumber,Index]:=((FLOAT( CCALL ( "rand" ) )/4294967294.0)
                                                          *8.0)-2.0;
    WeightExists[UnitNumber,Index]:=TRUE;
    OldDeltaWeight[UnitNumber,Index]:=0.0
  END;
  FOR Index := (NumberOfUnits-NumberInLayer[NumberOfLayers] +1) TO
               NumberOfUnits DO
    Weight[Index,UnitNumber]:=((FLOAT( CCALL ( "rand" ) )/4294967294.0)
                                                          *8.0)-2.0;
    WeightExists[Index,UnitNumber]:=TRUE;
```

```
    OldDeltaWeight[Index,UnitNumber]:=0.0

  END;

  Threshold[UnitNumber]:=((FLOAT( CCALL ( "rand" ) )/4294967294.0)*8.0)-2.0;

  OldDeltaThreshold[UnitNumber]:=0.0

END AssignValuesForNewUnit;


PROCEDURE AddUnit;

VAR Index, Index2 : CARDINAL;

BEGIN

  (* Move output units along one *)

  FOR Index := NumberOfUnits TO (NumberOfUnits -
        NumberInLayer[NumberOfLayers]
              +1) BY -1 DO

    Threshold[Index+1]:= Threshold[Index];

    OldDeltaThreshold[Index+1]:= OldDeltaThreshold[Index];

    FOR Index2 := (NumberInLayer[1] +1) TO (NumberOfUnits-NumberInLayer[
                        NumberOfLayers]) DO


      Weight[Index+1,Index2]:=Weight[Index,Index2];

      OldDeltaWeight[Index+1,Index2]:=OldDeltaWeight[Index,Index2];

      WeightExists[Index+1,Index2]:=WeightExists[Index,Index2]

    END

  END;

  NumberOfUnits:=NumberOfUnits+1;

  NumberInLayer[2]:=NumberInLayer[2]+1;

  AssignValuesForNewUnit(NumberOfUnits-NumberInLayer[NumberOfLayers]);

  WriteLn;
```

```
  WriteString('Unit added ');WriteLn

END AddUnit;

PROCEDURE RemoveUnit(UnitNumber : CARDINAL);

VAR Index, Index2 : CARDINAL;

BEGIN

  IF UnitNumber <> (NumberOfUnits-NumberInLayer[NumberOfLayers]) THEN

    FOR Index := (UnitNumber+1) TO (NumberOfUnits-
        NumberInLayer[NumberOfLayers])
                              DO

      Threshold[Index-1]:=Threshold[Index];

      OldDeltaThreshold[Index-1]:=OldDeltaThreshold[Index];

      FOR Index2 := 1 TO NumberInLayer[1] DO

        Weight[Index-1,Index2]:=Weight[Index,Index2];

        OldDeltaWeight[Index-1,Index2]:=OldDeltaWeight[Index,Index2];

        WeightExists[Index-1,Index2]:=WeightExists[Index,Index2]

      END

    END

  END;

  FOR Index:= (NumberOfUnits-NumberInLayer[NumberOfLayers]+1) TO
NumberOfUnits DO

    Threshold[Index-1]:=Threshold[Index];

    OldDeltaThreshold[Index-1]:=OldDeltaThreshold[Index];

    IF UnitNumber <> (NumberInLayer[1]+1) THEN

      FOR Index2 := (NumberInLayer[1] +1) TO (UnitNumber-1) DO

        Weight[Index-1,Index2]:=Weight[Index,Index2];
```

```
          OldDeltaWeight[Index-1,Index2]:=OldDeltaWeight[Index,Index2];
          WeightExists[Index-1,Index2]:=WeightExists[Index,Index2]
        END
      END;
      IF UnitNumber <> (NumberOfUnits-NumberInLayer[3]) THEN
        FOR Index2:= (UnitNumber+1) TO (NumberOfUnits-NumberInLayer[3]) DO
          Weight[Index-1,Index2-1]:=Weight[Index,Index2];
          OldDeltaWeight[Index-1,Index2-1]:=OldDeltaWeight[Index,Index2];
          WeightExists[Index-1,Index2-1]:=WeightExists[Index,Index2]
        END
      END
    END;
    NumberOfUnits:=NumberOfUnits-1;
    NumberInLayer[2]:=NumberInLayer[2]-1
END RemoveUnit;

PROCEDURE LearningAlgorithm;

CONST Limit = 10000;
    Divisor = 50;
    CheckRange = 400;
    CheckFrequency = 10;

TYPE DescisionArray = ARRAY [1..(CheckRange DIV CheckFrequency)] OF
CARDINAL;

VAR ErrorCount,BigLoop: CARDINAL;
  ErrorArray  : ARRAY MaxRange OF REAL;
  MainFlag,FreezeFlag : BOOLEAN;
  TotalRight:DescisionArray;

PROCEDURE NonDynamicLearning(FreezeFlag:BOOLEAN);

VAR  m,l,k: CARDINAL;
```

Appendix C

```
BEGIN
    ErrorSum:=0.0;
    FOR l:=1 TO NumberInTrainingSet DO
     ClampInput(Environment[l].Input);
     CalcOutput;
     IF FreezeFlag THEN
       FrozenWeightAdjustment(l)
     ELSE
       WeightAdjustment(l)
     END;
     FOR k:=1 TO NumberOfUnits DO
      FOR m:=1 TO NumberOfUnits DO
       OldDeltaWeight[k,m]:=DeltaWeight[k,m]
      END
     END;
     FOR k:=1 TO NumberOfUnits DO
      OldDeltaThreshold[k]:=DeltaThreshold[k]
     END
    END;
    FOR l:=1 TO NumberInTrainingSet DO
     ClampInput(Environment[l].Input);
     CalcOutput;
     FOR k:=1 TO NumberInLayer[NumberOfLayers] DO
       ErrorSum:=ErrorSum + ((Environment[l].Target[k]-Output
            [NumberOfUnits-NumberInLayer[NumberOfLayers]+k])*
```

```
                    (Environment[l].Target[k]-Output[
                NumberOfUnits-NumberInLayer[NumberOfLayers]+k]))
      END

    END;

    NumberOfSweepsSoFar:=NumberOfSweepsSoFar + 1;

    Error:=0.5*ErrorSum;

    IF Error<= ErrorArray[ErrorCount] THEN

      PrintError;

      ErrorCount:=ErrorCount+1

    ELSIF FLOAT(NumberOfSweepsSoFar)/FLOAT(Divisor) = FLOAT(
              NumberOfSweepsSoFar DIV Divisor) THEN

      PrintError

    END

END NonDynamicLearning;


PROCEDURE DescisionToAdd(Right:DescisionArray):BOOLEAN;

VAR i:CARDINAL;
   Percentage1,Percentage2:REAL;
   Descision:CARDINAL;

BEGIN

  Descision:=1;

  i:=2;

  REPEAT

    Percentage1:=100.0*FLOAT(Right[i])/(FLOAT(NumberInTrainingSet)*
                          FLOAT(CheckFrequency));

    Percentage2:=100.0*FLOAT(Right[i+1])/(FLOAT(NumberInTrainingSet)*
                          FLOAT(CheckFrequency));

    IF (Percentage1 >= 50.0) AND (Percentage2 >= 50.0) THEN

      Descision:=0
```

```
    END;
    i:=i+1;
  UNTIL (Descision=0) OR (i=(CheckRange DIV CheckFrequency));
  IF Descision = 1 THEN
    RETURN TRUE
  ELSE
    RETURN FALSE
  END
END DescisionToAdd;


PROCEDURE DynamicLearning;

VAR  m,l,k,Count,Count2: CARDINAL;
    TempSum: REAL;
    TempValue: ARRAY MaxRange OF REAL;


BEGIN
  FOR Count:=1 TO NumberInTrainingSet DO
    TempValue[Count]:=0.0
  END;
  FOR Count:=1 TO (CheckRange DIV CheckFrequency) DO
    TotalRight[Count]:=0;
    FOR Count2:=1 TO CheckFrequency DO
      ErrorSum:=0.0;
      FOR l:=1 TO NumberInTrainingSet DO
        ClampInput(Environment[l].Input);
        CalcOutput;
```

```
TempSum:=0.0;

FOR k:=1 TO NumberInLayer[NumberOfLayers] DO

  TempSum:=TempSum+ ((Environment[l].Target[k]-Output
       [NumberOfUnits-NumberInLayer[NumberOfLayers]+k])*
            (Environment[l].Target[k]-Output[
       NumberOfUnits-NumberInLayer[NumberOfLayers]+k]))

END;

IF TempValue[l]>TempSum THEN

 TotalRight[Count]:=TotalRight[Count]+1

END;

WeightAdjustment(l);

FOR k:=1 TO NumberOfUnits DO

 FOR m:=1 TO NumberOfUnits DO

   OldDeltaWeight[k,m]:=DeltaWeight[k,m]

 END

END;

FOR k:=1 TO NumberOfUnits DO

 OldDeltaThreshold[k]:=DeltaThreshold[k]

END;

TempValue[l]:=0.0;

CalcOutput;

FOR k:=1 TO NumberInLayer[NumberOfLayers] DO

  TempValue[l]:=TempValue[l]+ ((Environment[l].Target[k]-Output
       [NumberOfUnits-NumberInLayer[NumberOfLayers]+k])*
            (Environment[l].Target[k]-Output[
       NumberOfUnits-NumberInLayer[NumberOfLayers]+k]))

END;
```

```
      END;

      FOR l:=1 TO NumberInTrainingSet DO

       ClampInput(Environment[l].Input);

       CalcOutput;

       FOR k:=1 TO NumberInLayer[NumberOfLayers] DO

          ErrorSum:=ErrorSum + ((Environment[l].Target[k]-Output
               [NumberOfUnits-NumberInLayer[NumberOfLayers]+k])*
                    (Environment[l].Target[k]-Output[
               NumberOfUnits-NumberInLayer[NumberOfLayers]+k]))
       END

      END;

      NumberOfSweepsSoFar:=NumberOfSweepsSoFar + 1;

      Error:=0.5*ErrorSum;

      IF Error<= ErrorArray[ErrorCount] THEN

       PrintError;

       ErrorCount:=ErrorCount+1

      ELSIF FLOAT(NumberOfSweepsSoFar)/FLOAT(Divisor) = FLOAT(
               NumberOfSweepsSoFar DIV Divisor) THEN

       PrintError

      END

   END;
 WriteCard(TotalRight[Count],8)

  END

END DynamicLearning;


PROCEDURE FrozenLearning;

CONST Iterations = 50;

VAR Count : CARDINAL;
```

```
BEGIN
  FOR Count:=1 TO Iterations DO
    NonDynamicLearning(TRUE)
  END
END FrozenLearning;

BEGIN
  ReadVectorFile;
  Read(Dummy);
  WriteLn;
  WriteString('Please input the errors you wish to print');
  WriteLn;
  ErrorCount:=0;
  REPEAT
    ErrorCount:=ErrorCount+1;
    WriteString('Please input error number ');
    WriteCard(ErrorCount,4);
    WriteLn;
    ReadReal(ErrorArray[ErrorCount],ErrorRep);
    ReadLn;
    WriteLn;
  UNTIL ErrorArray[ErrorCount]<=0.0;
  FOR BigLoop:=1 TO 10 DO
    CreateWeightFile;
    ReadWeightFile;
    PromptForFileName;
    ErrorCount:=1;
    MainFlag := FALSE;
    REPEAT
```

```
      DynamicLearning;
      IF DescisionToAdd(TotalRight) THEN
        AddUnit;
        FrozenLearning
      ELSE
        MainFlag:= TRUE
      END;
    UNTIL (MainFlag=TRUE) OR (NumberInLayer[3]> 20);

    REPEAT
      NonDynamicLearning(FALSE);
    UNTIL (ErrorArray[ErrorCount]<=0.0) OR (NumberOfSweepsSoFar >= Limit);

    WriteWeightFile;
  END;
  Choice:='7'
END LearningAlgorithm;

PROCEDURE InputTrialVector;
VAR j : CARDINAL;
BEGIN
  FOR j:=1 TO NumberInLayer[1] DO
    ReadReal(TrialVector[j],ErrorRep);
    ReadLn
  END
END InputTrialVector;
```

```
PROCEDURE TestResponse;

BEGIN

 ReadVectorFile;

 PromptForFileName;

 ReadWeightFile;

 REPEAT

  InputTrialVector;

  ClampInput(TrialVector);

  CalcOutput;

  DisplayOutput;

  WriteString('Want another go ?');

  Read(Response);

  Read(Dummy);

 UNTIL  Response = 'n'

END TestResponse;


PROCEDURE RepeatedTest;

VAR division : REAL;
   RangeCount, Range, Range2 : CARDINAL;

BEGIN

  ReadVectorFile;
  ReadWeightFile;

  WriteString('Input divisions');
  WriteLn;WriteLn;
  ReadReal(division,ErrorRep);

  RangeCount:=TRUNC(1.0/division);
```

```
  FOR Range:=0 TO RangeCount DO
    TrialVector[1]:=FLOAT(Range)/FLOAT(RangeCount);
    FOR Range2:=0 TO RangeCount DO
      TrialVector[2]:=FLOAT(Range2)/FLOAT(RangeCount);
      ClampInput(TrialVector);
      CalcOutput;
      WriteReal(TrialVector[1],5,2);WriteString('  ');
      WriteReal(TrialVector[2],5,2);WriteString('  ');
      DisplayOutput;
      WriteLn;WriteLn
     END;
    END;
END RepeatedTest;

BEGIN (* Main Program *)
REPEAT
   (*ClearScreen;*)
   WriteString('MENU');
   WriteLn;
   WriteLn;
   WriteString(' 1) Create new environmental vector file');
   WriteLn;
   WriteLn;
   WriteString(' 2) Create new weight file');
   WriteLn;WriteLn;
   WriteString(' 3) Apply learning algorithm to existing weight file');
   WriteLn;WriteLn;
   WriteString(' 4) Obtain results');
   WriteLn;WriteLn;
   WriteString(' 5) Repeated Test');
   WriteLn;WriteLn;
   WriteString(' 6) Display a weight file');
   WriteLn;WriteLn;
```

```
WriteString(' 7) Exit from this superb program');
WriteLn;WriteLn;WriteLn;
WriteString(' Please enter your choice');
WriteLn;
Read(Choice);
WriteLn;

CASE Choice OF

  '1' : CreateVectorFile
| '2' : CreateWeightFile
| '3' : LearningAlgorithm
| '4' : TestResponse
| '5' : RepeatedTest
| '6' : DisplayWeightFile;
| '7' : WriteString('byeeeeeeeeeee')

ELSE

  WriteString('ERROR');

END


UNTIL Choice = '7'


END BackPropogation.
```

Appendix D - Program User Guide

Introduction

This appendix provides a user guide for the software implementation of the constructive algorithm described in this thesis. The program may be run interactively, with the user prompted for inputs, or in batch mode, where the information is stored in a file and no user intervention is required once the program is running. Both methods are described.

The program was written purely as a research tool and as a result is not 'user friendly'. In order to change certain rarely modified parameters, the code must be altered and compiled before execution. All common parameters, for example the learning factor, are entered by the user when the program is executed.

Compilation and Execution

Given a modula_2 source code file named *constructive.mod*, an executable is produced by invoking the compiler with the following commands:

m2c constructive.mod
m2c -e constructive

This produces a file called *a.out*, which may be executed by typing its name at the command line. If the user desires an alternative name for the executable, other than this UNIX default, for example *construct*, then the second compiler command above should be replaced by:

m2c -e constructive -o construct

This produces an executable called *construct* which again may be executed by typing its name at the command line.

The Main Menu

On executing the program, the user is presented with the following menu of options from which to select.

1) Create new environmental vector file
2) Create new weight file
3) Apply learning algorithm to existing weight file
4) Obtain results
5) Repeated test
6) Display a weight file
7) Exit
Please enter your choice

Option 1

The user is prompted for the parameters necessary to define a environmental vector set. The number of layers in the network, the number of units in each layer, the number of vectors in the training set and then each vector in turn are entered by the user. Once this information has been entered, it is stored in a file and the user is returned to the main menu.

Option 2

This option is not normally utilised as the creation of initial weights was subsequently incorporated into the learning process itself, since a new set of initial weights is normally required each time the learning algorithm is applied. In order to create an initial weight file, the user is prompted for the number of layers in the network, the number of units in each layer, the total number of units which is required as a check and a positive integer with which to seed the random number generator. The learning factor and momentum factor are also requested and stored with the initial weights in a file named by the user. Having entered this data the user is returned to the main menu.

Option 3

The user is prompted for the name of the file which contains the training set to be learned, followed by a list of error values. This list consists of specific error values with which the user wishes to associate a number of iterations, for example the user desires to record the number of iterations required to reach error values of 0.1, 0.01 & 0.001. This list must be terminated with a negative integer. The user is then requested to input the information required to create an initial set of weights as outlined in the description of option 2, together with the name of a file which is used to store the final values of the weights.

Once this information has been entered, the learning algorithm is applied to the specified weights and network structure. The program displays the error at each 100 iteration period, together with the number of iterations required to reach the specific error levels entered by the user. Each time a hidden unit is added to the network, the program displays a message, informing the user of the change in network structure. On completion of the algorithm, the user is returned to the main menu.

Options 4 & 5

These options are a legacy from the development phase of the program and have not been ported from the VAX VMS compiler.

Option 6

The weight files utilised by the program are stored in binary format, and hence option 6 is included to allow the user to view a named weight file.

Running in Batch Mode

Due to the necessity of performing large numbers of trials, a facility was added to the program to allow multiple applications of the learning algorithm. This simply repeats option 3 several times without returning to the main menu. Most benefit is gained from this by using the UNIX facility of input/output re-direction, which allows the input to a program to be read from a file. By placing the inputs the program requires in a file, the results for several trials can be obtained without any user intervention.

If the executable program is named *a.out* and the input required for ten trials is placed in a file called *batchfile*, then the user enters the following command:

a.out < batchfile > results_file

This runs the program, which takes its input from *batchfile* and writes the results to *results_file*. The contents of the batch file are simply those inputs required by the program. Part of such a file is shown below:

3	- menu option 3
0.1	- first error in error list
0.01	- second error in error list
-9	- terminator of list
3	- number of layers in network
6	- number of units in layer 1
1	- number of units in layer 2
1	- number of units in layer 3
8	- total number of units in the network
456345	- integer to seed random number generator
0.25	- learning factor
0.9	- momentum factor
w1	- name of file in which to store initial weights
w2	- name of file in which to store final weights

This shows the necessary information to perform one trial, but normally this list would continue with the values required for the subsequent trials. This approach is extremely flexible in allowing a large number of trials to be performed with variations in a large proportion of the parameters.

Modifications to the Basic Algorithm

The major modifications described in this thesis, for example freezing and the atanh modification do not affect the interface to the user as they are modifications of the learning algorithm and are stored as separate programs.

Appendix E - Description of Program

Introduction

This appendix provides a semi-formal English description of the program written to implement the constructive algorithm described in this thesis. This description focuses on the menu option of applying the learning algorithm as the remaining options are very straight forward and described in appendix D.

At each level of the description of the program, the portions displayed in **bold** are those sections that are expanded. Comments are enclosed between starred parenthesis, for example, (*This is a comment*).

The Main Menu

```
BEGIN (*Main program*)
REPEAT
        Display menu options to user
        CASE Choice OF
                1: Create a vector file
                2: Create a weight file
                3: Apply Learning Algorithm
                4: Test response to an input vector
                5: Display weight file in text format
                6: Repeated test
                7: Exit from the program
        ELSE
                Display error message
UNTIL User selects menu option 7
END
```

Apply Learning Algorithm

The learning algorithm procedure reads in the vector (or training) set, and applies the constructive learning algorithm to it a pre-determined number of times, generating a new initial set of weights each time. For each application of the algorithm, the number of hidden units is incremented as learning proceeds until a decision is made that further additions are unnecessary, at which point standard back-propagation is applied until convergence is achieved.

```
BEGIN (* Learning algorithm*)
        Read in vector file
FOR (The number of times user wishes to apply learning algorithm) DO
        Create a weight file
        MainFlag := False
        REPEAT
                Perform Dynamic Learning
                IF (Decision to Add a Further Hidden Unit) THEN
                        Add a Further Unit
                ELSE
                        MainFlag := TRUE
        UNTIL (MainFlag := TRUE) OR Number of hidden units has grown too large
        Apply standard back-propagation
        Write weight file to disk on conclusion of back-propagation

END (*End of FOR loop*)
Main menu choice = 7
END (*End of learning algorithm*)
```

Perform Dynamic Learning

The dynamic learning procedure performs standard back-propagation and counts the number of times the error goes 'the right way' for a pre-determined number of periods. Initially this was set at each ten iteration period within a hundred iteration check range. The error for each pattern is calculated immediately before a weight update and compared with a check error calculated immediately after the weight update in the previous iteration. The total number of times there is a decrease in error is passed back to the main learning algorithm procedure and used as the basis of the decision whether to add a further hidden unit.

```
BEGIN (*Dynamic Learning*)

Set the Check Error for each pattern to zero (* Must be set to zero as there is no previous
                                                iteration initially*)

FOR (Each period in which we are counting the number of times the error
        goes 'the right way') DO

    TotalRight(for this period) := 0

    FOR (Each iteration in this period) DO

        FOR (Each member of the training set) DO

            Present input vector

            Calculate the error for this vector

            IF (Check Error is greater than the current error) THEN

                Total number of times the error has gone the right way
                in this period is incremented by 1

            END (*End of IF*)

            Adjust weights using error generated by this pattern

            Calculate Check Error for this pattern

        END (* End of FOR*)
```

```
        END (*End of FOR*)
END (*End of FOR*)
END (*End of dynamic learning*)
```

Decision to Add a Further Hidden Unit

This procedure is passed the number of times the error goes 'the right way' for each period and utilises this information to return a Boolean value to the main learning algorithm procedure which indicates whether a further hidden unit should be added to the network.

```
BEGIN (*Decision to add a further unit*)
Decision to add a further unit is TRUE
FOR (Each pair of consecutive periods in the check range)
        IF (The error went the 'right way' more than 50% in both periods) THEN
                Decision to add a further unit is FALSE
        END (*End of IF*)
END (*End of FOR*)
RETURN (Decision to add a further unit)
```

Appendix F - Selection of Initial Parameter Values

Introduction

The decision to add a further hidden unit into the network if the total number of error increases did not fall below 50% in two consecutive 10 iteration periods within a 100 iteration decision window, was taken after varying these parameters to determine their effect on the performance of the algorithm. The results obtained using these parameters for the encoder 8 and 4DC problems are shown in table F.1.

Binary Problem	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
Encoder8	20	3.3	3	472	64	14%
4DC	20	3.6	2	331	162	49%

Table F.1 - Results for the Parameters Selected

Varying the 50 % Termination Condition

Keeping the check range fixed at a value of 10 and utilising a decision window of 100 iterations, table F.2 shows the results of varying the 50% termination condition for the encoder8 problem.

Termination Condition	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
30%	14	2.9	3	1463	1554	106%
40%	18	3.1	3	864	1124	130%
50%	20	3.3	3	472	64	14%
60%	20	3.7	4	427	38	8%
70%	20	4.1	4	433	48	11%

Table F.2 - Varying the 50% Termination Condition for the Encoder8 Problem

These trials gave the expected results in that they showed that as the termination condition was made more stringent, the average size of the final architecture was increased. When the termination condition was very low, at 30%, the algorithm settled on 2 hidden units in the majority of trials, which accounts for the low level of convergence. However, in 4 cases convergence was achieved with 2 hidden units, a result not achievable with standard back-propagation. Despite the low convergence rate in this case, the encoder8 problem proved to be very insensitive to the value of this parameter, with the final number of hidden units varying from 2.9 to 4.1 over a very wide range of termination conditions. Repeating this experiment for the 4DC problem gave the results shown in table F.3.

Termination Condition	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
30%	0	-	-	-	-	-
40%	8	2.3	2	224	70	30%
50%	20	3.6	2	331	162	49%
60%	20	3.7	2	309	148	48%
70%	20	4.3	2	344	154	45%

Table F.3 - Varying the 50% Termination Condition for the 4DC Problem

As with the encoder8 problem, the average number of hidden units increased as the termination condition was made more stringent. The 4DC problem was more sensitive to the value of this parameter than the encoder8 problem, in that convergence was never achieved for a termination condition of 30%. However, over the range 40% - 70% the average size of the final architecture only varied from 2.3 to 4.3.

On the basis of these results it was decided to use a value of 50%, since reasonable results were achieved for both the size of the final architecture and the number of iterations required for convergence.

Varying the Size of the Decision Window

Varying the size of the decision window for the encoder8 problem gave the results shown in table F.4 below.

Decision Window Size	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
50	20	4.5	5	279	37	13%
100	20	3.3	3	472	64	14%
200	20	3.2	3	668	68	10%

Table F.4 - Varying the Decision Window Size for the Encoder8 Problem

As the decision window size was increased, the size of the final architecture was reduced. This is due to the fact that a larger decision window offers more opportunity for the algorithm to meet the criteria for terminating the addition of further hidden units. The penalty associated with a large decision window size is that the number of iterations required for convergence is necessarily larger, as more iterations are performed before sufficient hidden units are present. Similar results were obtained for the 4DC problem as shown in table F.5 below.

Decision Window Size	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
50	20	5.2	3&4	264	133	50%
100	20	3.6	2	331	162	49%
200	20	2.8	2	503	190	38%

Table F.5 - Varying the Decision Window Size for the 4DC Problem

Similarly to the encoder8 problem, the average size of the final architecture was decreased as the decision window was made larger. Again the penalty associated with this reduction was an increase in the number of iterations required for convergence. We decided that a

100 iteration decision provided a good compromise between the requirements for a near minimal hidden layer structure and fast convergence to a solution.

Varying the Size of the Check Period

Retaining a decision window of 100 iterations and varying the check period size for the encoder8 problem gave the results shown in table F.6 below.

Check Period Size	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
5	20	3.2	3	489	56	11%
10	20	3.3	3	472	64	14%
20	20	3.5	3&4	462	52	11%

Table F.6 - Varying the Check Period for the Encoder8 Problem

The results shown in table F.6 show that reducing the length of the check period also reduced the size of the final architecture. This is because the algorithm terminates the addition of hidden units when the number of error decreases is greater than 50% in two consecutive check periods. Halving the size of the check period is approximately equivalent to the number of error decreases being greater than 50% in 1 check period of size 10. Repeating this experiment for the 4DC problem generated the results shown in table F.7.

Check Period Size	Number of Convergent Trials	Average Hidden Units	Mode for Hidden Units	Average Iterations	Standard Deviation	Standard Deviation as Percentage
5	10	4.2	4	451	133	29%
10	20	3.6	2	331	162	49%
20	20	3.8	2	315	155	49%

Table F.7 - Varying the Check Period for the 4DC Problem

In this case the average final hidden layer architecture appears to be larger when the check period is reduced. This is due to the 10 trials that did not converge and whose results are not included in the statistics. All ten of these trials produced networks with 1 hidden unit, which is an insufficient number to learn this function. The results for check periods of 10 and 20 are very similar, but the average final architecture is larger when the check period is doubled.

The results for the encoder8 problem again show that it is relatively insensitive to the parameters chosen, with the average final architecture ranging from 3.2 to 3.5 hidden units. The 4DC is more sensitive to the value of this parameter, but excluding the result for a check range of 5, the results are relatively uniform. In light of these results, it was decided that a check period of 10 would be utilised, since it gave stable results over both problems.

Parity6

The parity6 problem was shown to be insensitive to the values of all the parameters discussed here in that the results were uniformly poor. Similar trends to those observed for the encoder8 & 4DC problems were seen, for example the average number of hidden units

increased as the termination condition was increased, but convergence was very poor and the results were erratic in general.

Conclusion

In the light of the results presented here, we decided that a further hidden unit would be inserted into the network if the total number of error increases did not fall below 50% in two consecutive 10 iteration periods within a 100 iteration decision window. These parameters gave very good results for the 4DC and encoder8 problems, both in terms of the size of the final hidden layer architecture and the number of iterations required for convergence.