NEURAL NET ALGORITHMS For DYNAMICAL SYSTEMS

Y.M. CHEUNG B.Eng, M.Sc.

This thesis is submitted to the Council for National Academic Awards in partial fulfilment of the requirements for degree of Doctor of Philosophy.

Nottingham Polytechnic,

Department Of Computing.

July 1992.

NEURAL NET ALGORITHMS FOR DYNAMICAL SYSTEMS

Y.M. Cheung

ABSTRACT

A neural net based algorithm is devised as an alternative to traditional analogue/numerical integration. The new algorithm consists of a multilayered neural net integrator model inspired by the neuron organisation of the vertebrate retina. A mixture of implicit weight setting, supervised and unsupervised learning is employed. The convergence of this approach proves to be fast when compared to existing models producing comparable results.

When the model is operating in a closed loop system it yields a consistent estimate of the derivatives of pictorial input profiles.

The mapping Of the resulting neural net models onto single and multiprocessor systems is examined. A general framework is formulated to permit arbitrary network definition and easy alterations of network parameters.

A parallel processing technique for distributed memory multiprocessor systems is devised. The parallel algorithm yields a large reduction in processing time.

ACKNOWLEDGEMENTS

The author would like to thank Dr. D Al-Dabass for his guidance and valuable advice throughout the duration of the research work.

CONTENTS Page

APPENDIX

Figures Page

CHAPTER 1 INTRODUCTION

CHAPTER 1 INTRODUCTION

The accuracy limitations of analogue integration led to the development of numerical software packages for simulating dynamical systems during the early 1950s. The speed limitations of numerical integration packages motivated the development of hybrid simulators in the late 1950s and early 1960s. In these simulators the dynamical equations were set up on analogue integrators while parameter changes during repeated simulation runs was supervised by digital machines. The advent of low cost microprocessors in the early 1970s motivated research into all digital multiprocessor systems for simulation [Aldabass,1976]. Specialized array machines such as the AP120B were also introduced with microprogramming facilities to speedup the execution of given arithmetic strings. The introduction of transputers and computing surfaces in the early 1980 provided yet another development of digital processors suitable for computing numerical integration algorithms.

However, despite the availability of such a seemingly wide range of computing tools no fundamentally different approach to the traditional twin techniques of analogue and numerical integration seems to have emerged. The relatively recent revival of interest in neural nets provided an opportunity to examine their possible utility for computing dynamical systems problems. In particular to investigate the capability of neural nets to perform integration, and consequently through a feedback arrangement to extract the derivatives of a given input trajectory.

1.0 Neural Nets

Recent progress in computer technology has resulted in high-performance computer systems which can process millions of instructions per second, but these 'symbolic' computer systems are still facing extreme difficulty in attempting to solve problems that human beings do well. Digital computers can be programmed for intelligent tasks. The problem is that the algorithmic solution to many information processing tasks is generally far too complex to be programmed. No computer can be programmed to match human capabilities in applications involving intelligent information processing such as to recognize, evaluate, adapt, learn and generalize. Even if they come close to performing any of these tasks many algorithms are still too computationally intensive to allow high-performance computers to find a solution in any reasonable period of time.

On the other hand, computers operating in a symbolic logic environment are much faster and more reliable at symbolic processing than the human brain. No human being can multiply two large numbers, calculate matrices or solve systems of differential equations at speeds performed by mini-computers. This indicates that better machines might be built by incorporating some of the properties of the biological nervous system/brain into the conventional von Neumann architecture.

Artificial systems that mimic biological nervous systems are commonly called Artificial Neural Networks or Neural Nets. By their very definition neural nets are information processing systems that have physical structures that closely parallel those of the biological nervous systems and are capable of solving problems that humans do well. Unlike traditional expert systems, where knowledge is made explicit in the form of rules, neural nets generate their own rules by learning encounters. [Abu-mos,1986] and [Arsenau,1989] have shown that neural nets are capable of solving any boolean computational problem. On the other hand, it is likely that this may come at high cost. Neural nets are not always the best solution for a given problem. They can handle processes difficult for conventional computers but performance is poor at precise computation.

Problems solved more effectively by the brain typically have two characteristics [Widrow,1990]: they are generally ill defined, and usually require an enormous amount of processing, typical examples are image and speech processing.

1.1 Basic Elements Of The Biological Brain

The brain is made up of a vast network of nerve cells called neurons. There are about 10^{11} neurons, probably more, in the human brain [Arbib,1989]. The structure of the brain is highly varied from one individual to another, as well as from one neuron to another. In fact, there is an enormous variety of neurons in the brain, with fundamental differences in structure, patterns of connections and the way that neurons send and receive signals.

A neuron receives inputs from many other nerve cells, sums the inputs and generates an output, which it then sends to another neurons. Figure 1.1 shows the synaptic connection between one neuron's axon and another neuron's dendrite. A neuron consists of a cell body with a number of input fibres called dendrites and a single long output cable-like extension called an axon [Strange,1989] [Callatay,1989].

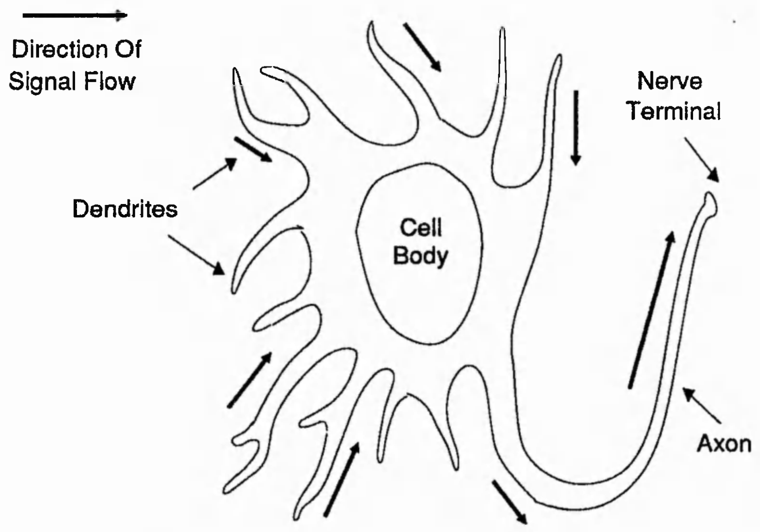


Figure 1.1 Schematic View Of A Neuron

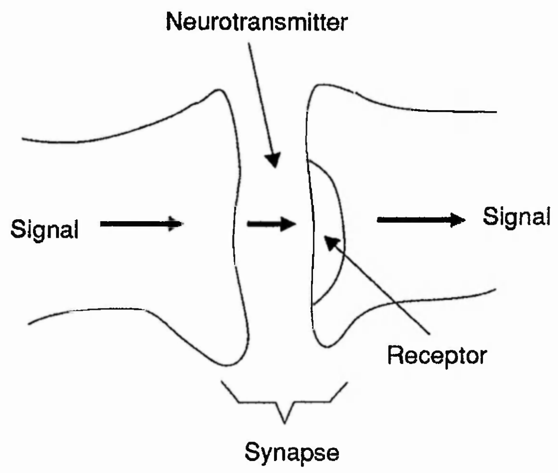


Figure 1.2 Connection Junction (Synapse)

Neurons are electrically excitable and capable of generating electrical signals called action potentials which are propagated down towards the end of the axon called the nerve terminal. Propagation of the signal occurs in one direction only, from the cell body to the nerve terminal. The nerve terminal of the axon is used for forming connections with the dendrites of other neurons and the connection junction between neurons is called a synapse.

The electrical signal generated by a cell body travels along the axon. When it reaches the nerve terminal a chemical, known as the neurotransmitter, is released. This chemical crosses the connection junction, the synapse, and interacts with specific sites called the receptors on the other side, as illustrated in Figure 1.2. The combination of neurotransmitter with receptor causes a change in electrical activity on the other side which may lead to continuation of electrical signalling in the next neuron.

There are many different neurotransmitters, but one neuron releases the same neurotransmitter from all of its nerve terminals. The amount of neurotransmitter released depends on the frequency of electrical signal in the axon, thus signalling at the synapse is in analogue form. This chemical transmission of information is also believed to be a means of storing information. Information can be stored between the synaptic junctions. The stronger the junction the more neurotransmitter is released for a given amplitude of triggering electrical signal.

The effect of different neurotransmitters on the post synaptic neuron can be either excitatory (positive) or inhibitory (negative) so that any particular neuron receives a mixture of positive and negative inputs. There are many inputs each with different 'strengths'. The neuron integrates the strengths and fires accordingly. The pattern of input strengths is not a fixed one, but is modified with use and this has relevance to 'learning' and 'storing' information.

1.2 Characteristics Of Artificial Neural Nets

Neural-style systems that mimic biological nervous systems are called Artificial Neural Nets. The inspiration for this approach came from the study of the structure of brain tissue rather than to emulate the workings of the brain. Consequently, a neural net is made of many simple processing elements, commonly refereed to as artificial neurons or simply neurons, which interact in parallel by means of the signals passing between them. Research in the field is of a very experimental nature, due to the mathematical complexity of these parallel non-linear systems. In general the approach taken is to obtain possible guidance from analytical methods, and then to conduct simulation experiments with software on conventional digital computers. Due to speed limitations, networks are usually kept relatively small. This vastly oversimplifies the structure of the real biological systems. However, even with small size networks some surprisingly difficult problems have been tackled. Computer based neural networks, for instance, have learned to speak [Sejnowk,1987], to detect speech [Newman,1990], to recognize handwriting [Fukushi,1988] [Yamada,1989] [lee,1988], to detect undersea objects [Gorman,1988] and many others listed in the references.

Neural network computer systems possess several useful features. First, the neural network is inherently parallel in nature. A parallel architecture provides a dramatic speed advantage over a conventional computer. Information is not stored in specific memory locations, but distributed over the interconnections of the network. Thus, the computation time for any particular problem, whether complex or small, would be the same. Second, massive parallelism means that the system is robust, fault-tolerant and functionally persistent, the loss of a few neurons and connections has negligible effect on the overall performance. Third, they are flexible, when confronted with a novel situation they will attempt to generalize, at worse returning a 'best fit' solution. Fourth, they have learning capabilities and can adapt to changes. Fifth, since conventional programming is not necessary, there is no requirement to completely understand the domain, thus, can be employed in poorly understood or experimental situations.

1.3 Artificial Neurons

Artificial neurons are the basic processing element of artificial networks, see Figure 1.3. It consists of three main components: a set of input paths, a transfer function block and a single output path. These three components are analogues to the dendrites (inputs), the cell body and the axon (output) of a biological neuron.

The set of input paths provide connection from other neurons. Each input is associated with a weight factor

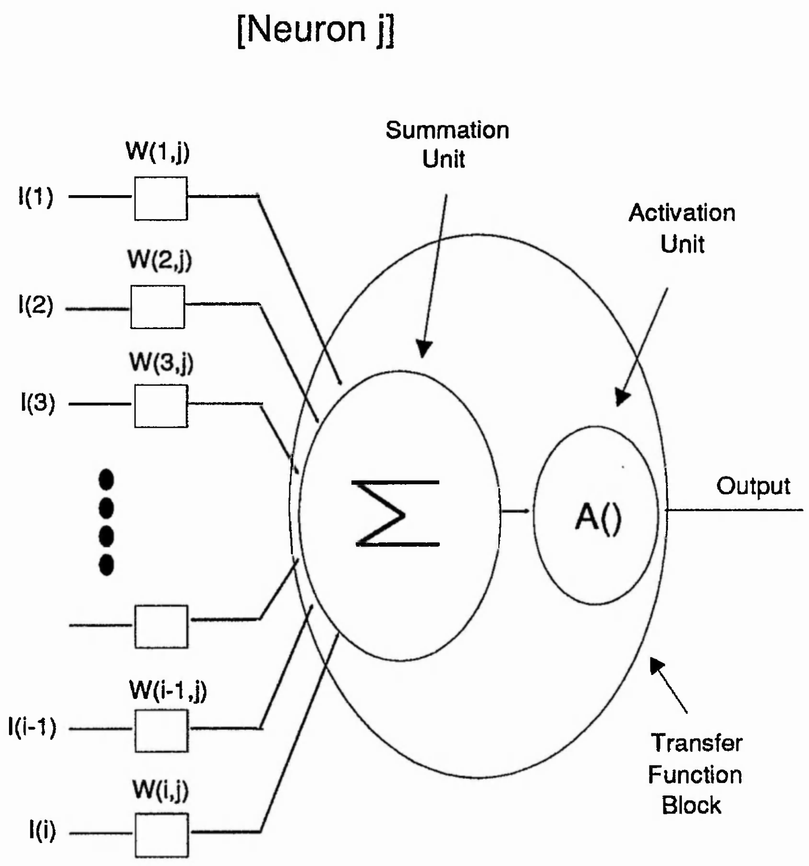


Figure 1.3 Model Of An Artifical Neuron

which determines the amount of connection strength one neuron has on the other. The signal received by each input is multiplied by the corresponding weight factor before propagating to the next stage.

The transfer function block consists of two units: the summation and the activation units. The summation unit performs arithmetic additions with the weighted inputs and produces an output signal. After summation, the net input of the neuron is fed to the activation unit to produce a new activation value. The transfer function of the activation unit defines how the activation value is output. In the simplest models, the activation function is simply a linear function, Figure 1.4(a), - the weighted sum of the neuron's external inputs. In more complicated models, non-linear transfer function are used. The binary threshold function, Figure 1.4(b), is the simplest; if the net inputs are greater than some fixed level (threshold) the neuron will output ONE, else it will output a ZERO. Sometimes, the transfer function is a saturation type function called Sigmoid function, Figure 1.4(c). It has high and low saturation limits and a proportionality range in between. This function is ZERO when the net input is a large negative number or is ONE when the net input is a large positive number and make a smooth transition in between.

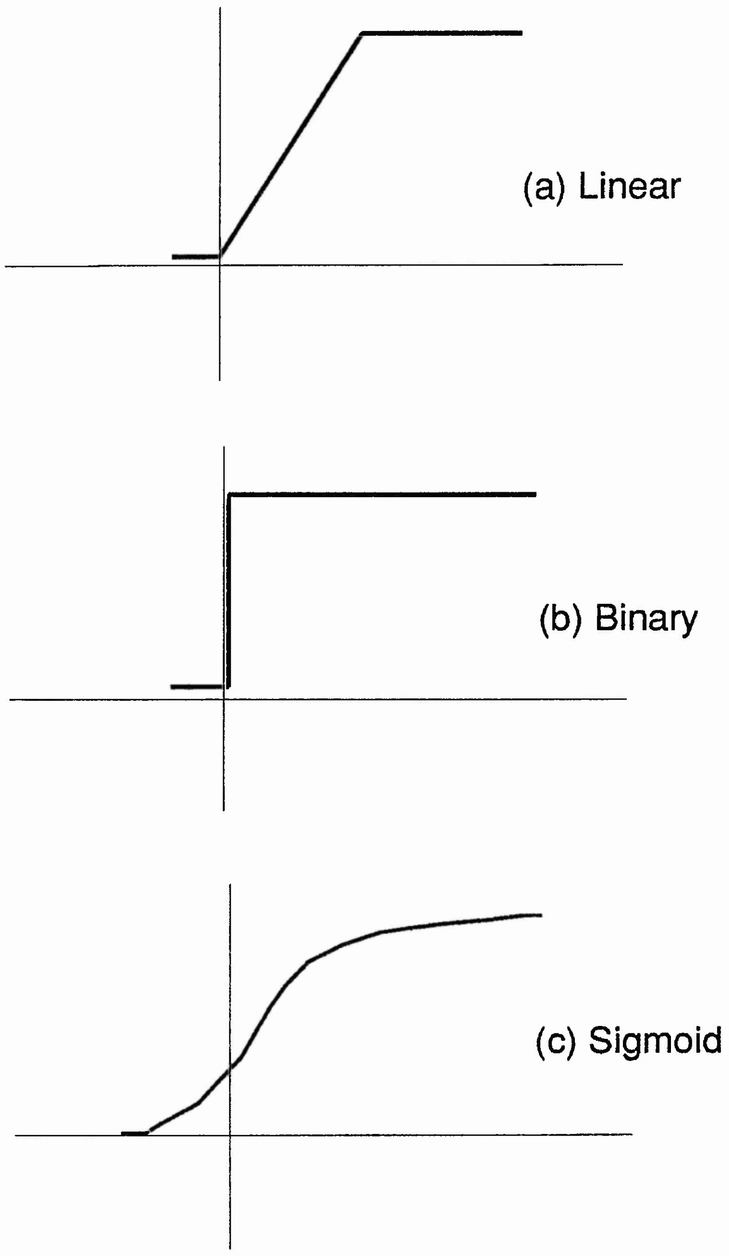


Figure 1.4 Common Neuron Output Functions

The characteristics of an artificial neuron can be best illustrated by the following general equation :-

$$Output_j = A\left(\sum_{i=1}^{n} Weight_{ij} Output_i \right) \qquad (1.1)$$

$Output_j$ represents the output of a neuron, A() represents the activation transfer function, - a function of the sum of product of inputs: $Output_j$, and the corresponding connection strength, W_{ij}. The subscripts i and j represent sending and receiving neurons respectively.

1.4 Network Structures

The behaviour of a neural network depends heavily on the way neurons are connected. Different neural network models have different network structures. In most models, the individual neurons are grouped into layers so the output from each neuron in one layer is fully interconnected with the input of all the neurons in the next layer. Similar to real biological systems, a neural net may include inhibitory connections from one neuron to another. Neurons can interact in many ways, by virtue of the manner in which they are interconnected. Common network configuration include: feed-forward only, feed-forward with feedback loop and neurons that are sparsely connected to a few other distant neurons.

Many early models were single layer feed-forward networks, as shown in Figure 1.5 with the structure consisting of a single layer of neurons, in which each input is connected to all neurons but no output feedback. This structure has been extended in three different ways. Firstly, network connections can exit from neuron to neuron within a single layer. Secondly, a network can have multiple feed-forward layers in which neurons in a middle layer are hidden from the external inputs and outputs of the network, see Figure 1.6. Thirdly, networks can have feed-backward connections, Figure 1.7.

1.5 Learning

Neural nets store information by adjusting the connection strengths, weights, between neurons. Through appropriate adjustments, a network will be able to perform the input to output transformation desired. The adjustment process often refereed to as training or learning process and is governed by a set of learning rules. These rules can be classified into two main categories: Supervised and Unsupervised.

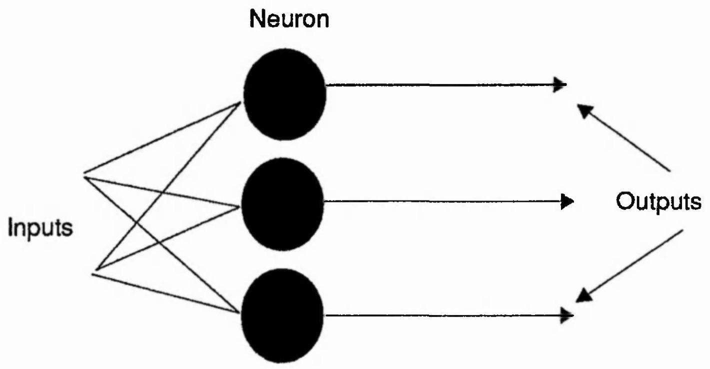


Figure 1.5 Single Layer Feed Forward Network

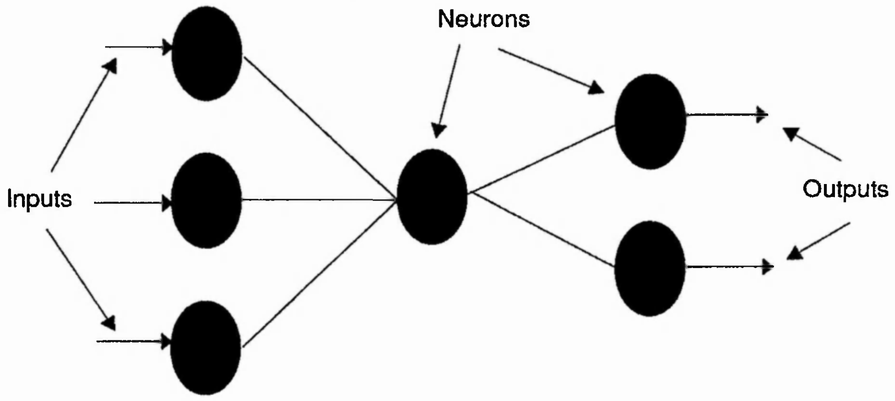


Figure 1.6 Multi-layer Feed Forward Network

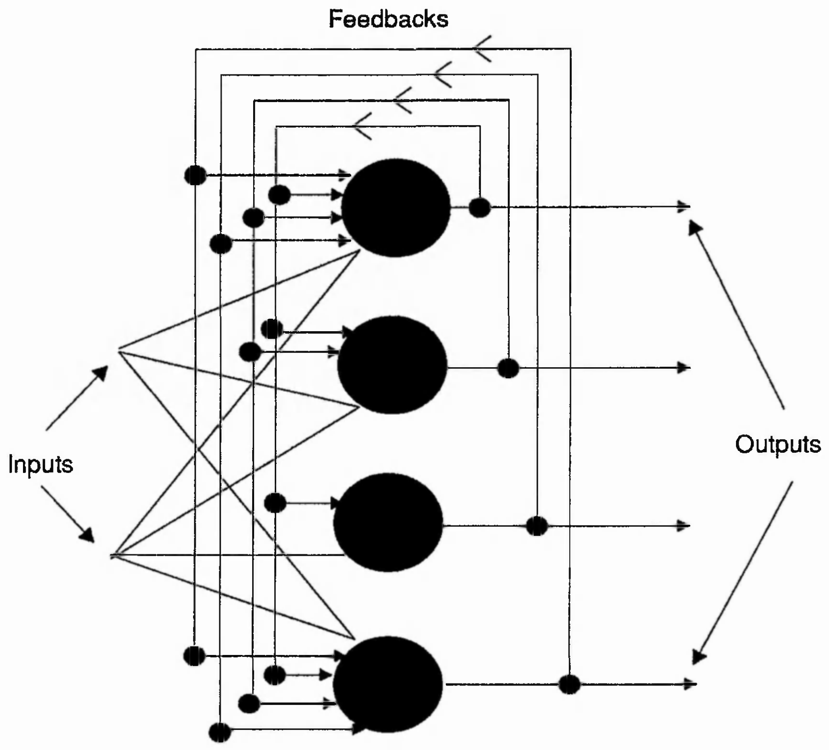


Figure 1.7 Single Layer Feed Backward Network

In supervised learning, the network output is compared to the ideal response, and any error made by the network is used to alter the connection strengths; learning is accomplished by changing the weights so as to reduce the errors. Rules in this category require a priori knowledge of what the result should be. Unsupervised learning differs in that the network must find the error and correct the network connections itself without making any comparison with ideal results.

Learning rules vary among different models. The most commonly known include: Hebbs Rule, Delta Rule, Backpropagation, Hopfield (Associative Memory), Boltzmann Machine, Competitive Learning.

1.6 Classification Of Models

Neural network models are distinguished by their learning strategies and the interconnection structure in which these strategies are embedded. The models described in this section can be grouped into four different classes [Yoon,1989]: correlational, competitive, error correction and stochastic.

In the 'correlational class', the interconnection strengths (weights) between neurons are adjusted according to the Hebbian Rule [Hebb,1949] in which the change in strength between two neurons is according to the output of the two neurons. This method is used in many models, both in supervised and unsupervised learning.

Algorithms in the 'competitive class' are used in unsupervised learning. The output neurons compete until one dominates and the weights (connection strengths) that are connected to the dominant output are altered. The weights are changed according to the Hebbian Rule or a modification thereof.

Algorithms in the 'error correction' class are used in supervised learning. They are used in a variety of models, including the Perception [Rosenbl,1962] and Back-propagation [Rumelbh,1986(a)], [Rumbelbh,1986(b)] described later. Errors are computed from the difference between the network actual outputs and the desired outputs specified externally. The weights are then altered in an attempt to minimize the errors.

Algorithms in the 'stochastic class' use a statistical approach to train a network. Weights are adjusted in order to minimize a statistical quantity similar to the thermodynamic function. The Boltzman Machine [Hinton,1985] is one such model in this category.

1.7 The Hebbian Rule

The Hebbian Rule [Hebb,1949] is an unsupervised learning algorithm. Perhaps because it is a source of many later models, it is still one of the most commonly known learning algorithms. The original Hebbian Rule was not mathematically expressed, but was presented as a statement. The Hebbian Rule states that "When an axon of a nerve cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased". The following equation, proposed by [Sutton, 1981], is a widely accepted mathematical approximation of the Hebbian Rule :-

$$W_i(t+1) = W_i(t) + C\,X_i(t)\;Y(t) \qquad (1.2)$$

where $W_i(t+1)$ and $W_i(t)$ represent the next and present weights between a signalling neuron X_i and the receiving neuron Y, C is a positive constant determining the rate of learning.

1.8 Single-layer Perceptrons

The Perceptron [Minsky,1969] is one of the first simple networks with the ability to recognise simple patterns and was originally proposed by [Rosenbl,1962]. It is essentially a single-layer network with feed-forward interconnections, as illustrated in Figure 1.8.

In this simplified version, the single, neuron like processing element computes a weighted sum of the inputs and passes the result through a binary valued nonlinearity, thus providing two possible output values (generally +1 and -1). Each of the two possible outputs corresponds to a different classification response. The properties of this model can be expressed mathematically as :-

$$Y_j = \sum_{i=1}^{n} W_{ij} X_i \qquad (1.3)$$

where Y_j is the partial output of the processing element which formed a weighted sum of its inputs X_i , W_{ij} is the weight for the i-th inputs to the j-th processing element.

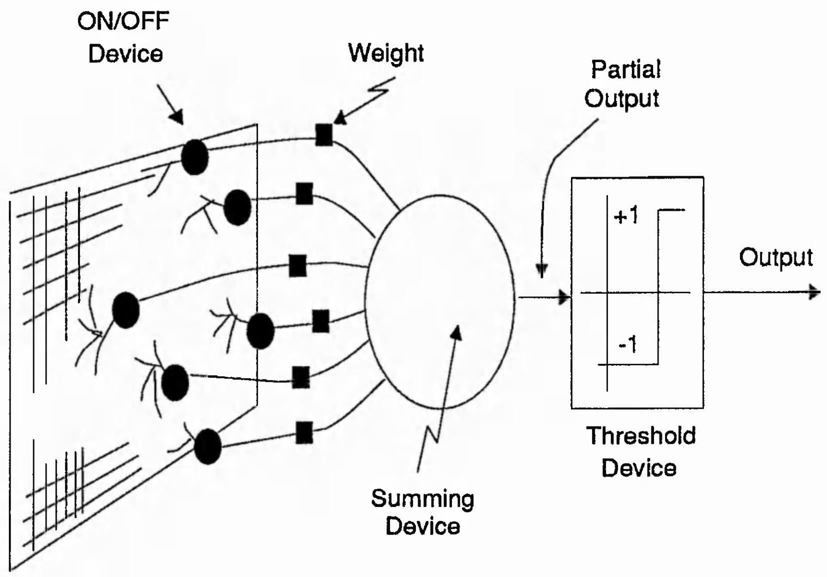


Figure 1.8 A Single Perceptron

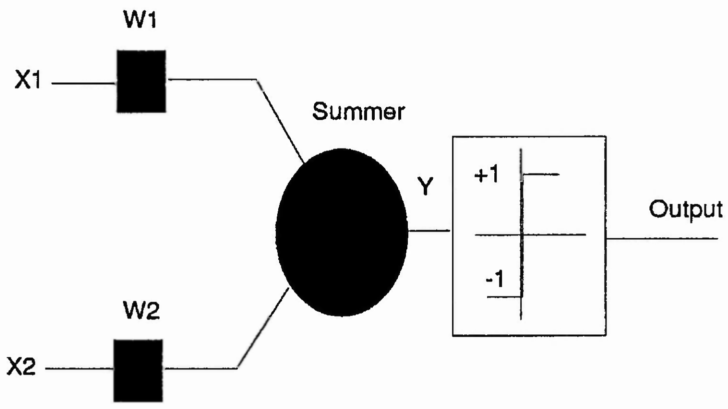


Figure 1.9 A Two Inputs Processing Unit

The partial result Y is then fed to a threshold device, T, so that:-

$$O_j = \begin{cases} +1 & \text{if } Y_j \geq T \\ -1 & \text{if } Y_j < T \end{cases} \tag{1.4}$$

Connection weights and the threshold in a Perceptron are typically adapted using the Perceptron Convergence Theorem [Rosenbl,1962].

The output of the processing element is computed by equations (1.4) and (1.5). The result is compared with the desired response (or target). If the output O_j is correct, then no change is made to the connection weights. If the output O_j is incorrect then the threshold and the weights are modified.

Mathematically, this amounts to the following: the change in the threshold, T, is given by :-

$$\Delta T = -(t_p - o_p) = -\delta_p \tag{1.5}$$

where p indexes the particular pattern being presented, t_p is the target value indicating the correct classification of that input pattern, and δ_p is the difference between the target and the actual output of the network. Finally the change in weights, ΔW_{ij} are given by:-

$$\Delta W_{ij} = \eta \ (t_p - o_p) \ X_i = \delta_p X_i \qquad (1.6)$$

where X_i is the input value propagating through the connections and η is the rate of weight change.

In spite of it's simplicity, the Perceptron Convergence Theorem guarantees that, if a Perceptron network could learn to correctly classify a set of patterns, it could do so within a finite number of iterations. In addition, if more than one element is used, the Perceptron Convergence Theorem can be applied independently to each of a set of processing element. However, this theorem will find the correct mapping from a set of inputs onto a set of outputs, only if the input classes are linearly separable. This is the major problem of the Perceptron Convergence Theorem. The classic simple example of a function that cannot be computed by the Single-layer Perceptron is the exclusive-OR logic function. Consider a two input processing element, as illustrated in Figure 1.9, the partial output, Y, can be expressed, in terms of weights and inputs as :-

$$Y = X_1 \ W_1 + X_2 \ W_2 \qquad (1.7)$$

By considering the relationship between X_1 and X_2 equation

(1.7) can be arranged so that X_2 is a function of X_1 :-

$$X_2 = \frac{Y}{W_2} - X_1 \left(\frac{W_1}{W_2} \right) \qquad (1.8)$$

equation (1.8) is that of a straight line. For the exclusive-OR problem, each input can only takes on two values (0 or 1) so there are only two corresponding points of interest on each axis, as illustrated in Figure 1.10. This pinpoints four points in the graph corresponding to the four possible input patterns: (0 0), (0 1), (1 0), (1 1). The straight line, defined by equation (1.8), divides the graph into two regions. Thus, different input patterns can be separated by moving the line to different locations. It is obvious that, no matter where the line is placed, it can never separate the points which define the exclusive-Or function.

1.9 Multi-layer Perceptrons

These are used to overcome the limitation of Single-layer Perceptrons. The additional layer or layers are isolated from the external world by the input and output layers. The properties, including differences and limitations, of the Single-layer and Multi-layer Perceptrons are detailed in

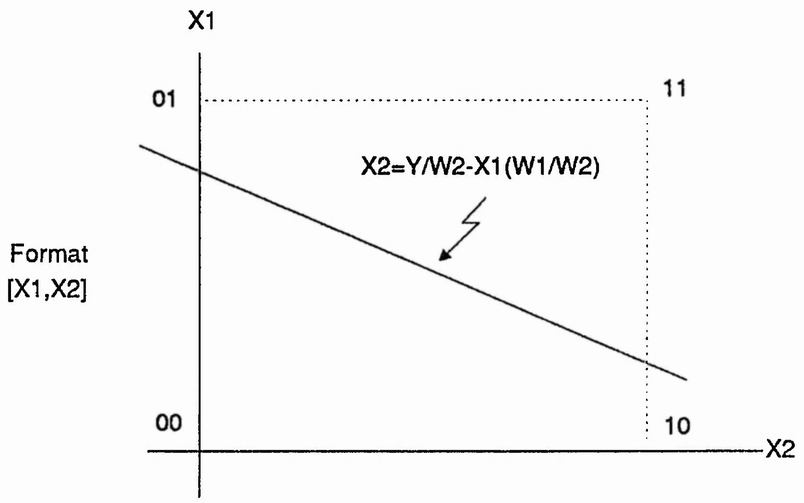


Figure 1.10 Geometrical Representation Of The XOR Problem

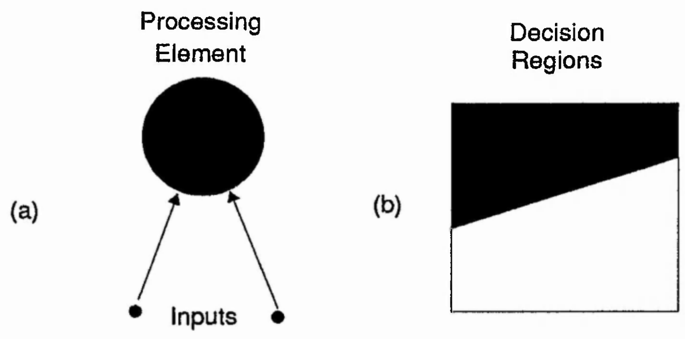


Figure 1.11 Decision Regions (b) Formed By Processing Element (a)

[Lippman,1987]. They distinguish Single- and Multi-layer Perceptron using the theory of linear discriminant functions and decision surfaces. Their discussions are summarized as :-

1) A Single-layer Perceptron will form two decision regions separated by a hyperplane. When a two input processing element is used the hyperplane is a straight line, see Figure 1.11 (a)(b).
2) In a Two-layer Perceptron, the separating regions are of the convex open type, as illustrated in Figure 1.12. It is obvious to see that the exclusive-OR function can be solved by a Two-lawyer Perceptron.
3) Arbitrary closed regions can be formed by using a Three-layer Perceptron. The complexity of the shape of the regions is limited by the number of processing units, see Figure 1.13.

It is clear that Multi-layer Perceptrons can overcome many limitations of the Single-layer Perceptron. It has not been used extensively in the past, mainly due to the lack of an effective weight adjustment algorithm. However, it has significantly affected the development of many later models, especially after the introduction of the Back-

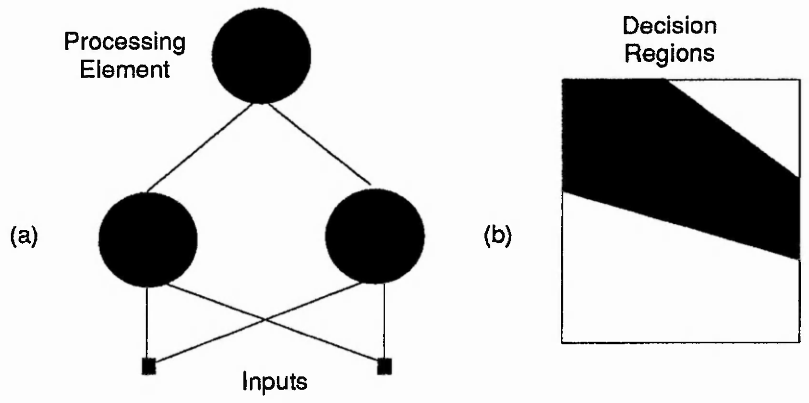


Figure 1.12 Decision Regions (b) Formed
By Topology (a)

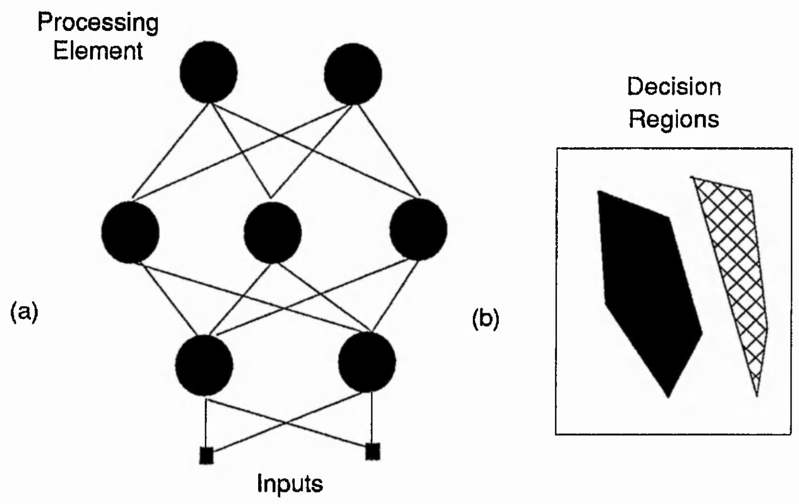


Figure 1.13 Decision Regions (b) Formed
By Topology (a)

propagation learning algorithm [Rumelbh,1986(a)], [Rumelbh,1986(b)].

1.10 The Delta Rule

Other efforts to overcome the limitations of the Single-layer Perceptron model include the learning algorithm by Widrow-Hoff or LMS (least mean square) or Delta rule [Lippman,1987]. The Delta Rule is an error correction rule which minimizes the mean square error between the desired and the actual outputs of Perceptron like nets, thus the amount of learning is proportional to the error computed.

The basic difference from the classical Perceptron model is the use of a linear threshold function instead of the hard-limiting output. Mathematically, it can be expressed by the following two equations:-

$$Y_j = \sum W_{ij} X_i \tag{1.9}$$

$$\Delta W_{ij} = (T - Y_j) X_i \eta \tag{1.10}$$

where Y_j is the output of the processing elements, W_{ij} is the connection weight between input X_i and the receiving unit Y_j, η is the constant of proportionality representing the learning rate, ΔW_{ij} is the weight change and T is the target value.

Although the Delta Rule was developed as an improvement of the classical Single-layer Perceptron, it is not able to solve a number of problems, such as the exclusive-OR.

1.11 The Back-propagation Algorithm

Functions which are linearly dependent cannot be solved by single-layer models, see section 1.8. To overcome this limitation it is necessary to use multi-layer architectures such as the Multi-layer Perceptron. Such models, however, were not generally used until the introduction of the Back-propagation learning algorithm.

The Back-propagation algorithm is currently one of the most popular algorithms. It employs a supervised error correction method similar to the Delta Rule. A detailed mathematical description can be found in [Rumelbh,1986(a)], [Rumelbh,1986(b)] and in appendix (A.2). The basic idea of the Back-propagation model is to propagate errors back from the output layer towards the input and use the computed errors to correct the connection weights. Output units calculate their errors from the differences between the target and actual outputs. These errors are then back propagated to the hidden layer and used to evaluate the corresponding errors of the units in the hidden layer.

In general, there are three sets of equations. One set for the normal feed-forward propagation of signals from input to output; one set for the backward propagation of error, from output to input; and one set for the adjustment of the connection weights.

The forward motion is governed by equations (1.11) and (1.12). They represent the characteristics of the processing units in each layer. The output of each unit is a Sigmoid function, f(), of the weighted sum, Y_j :-

$$O_j = \frac{1}{1 + \exp(-Y_j)} = f(Y_j) \tag{1.11}$$

$$Y_j = \sum_{i=1}^{n} O_i \, W_{ij} + \theta_j \tag{1.12}$$

There are two equations for the error calculations. One for the units in the output layer, and one for units in the hidden layer. In the output layer, the calculated output vector, o_k is compared with the target output vector, t_k. The difference between these vector components (i.e. $t_k - o_k$) defines the error. In other words, the input(s) of the output processing units should be corrected by δ_k to provide the target output vector t :-

$$\delta_k = (t_k - o_k) f'(Y_k) \qquad (1.13)$$

A detailed proof of the equations described in this section can be found in appendix (A.2). Introducing equation (1.11) for the output function, f(), yields the derivative hence the error :-

$$f'(Y_k) = \frac{df(Y_k)}{dY_k} = o_k (1 - o_k) \qquad (1.14)$$

(1.13) and (1.14) yield the error function for the units in the output layer.

$$\delta_k = (t_k - o_k) o_k (1 - o_k) \qquad (1.15)$$

The error equation for the units in the hidden layer is more complicated. The error from each output unit is back-propagated to the connected hidden units, see Figure 1.14,

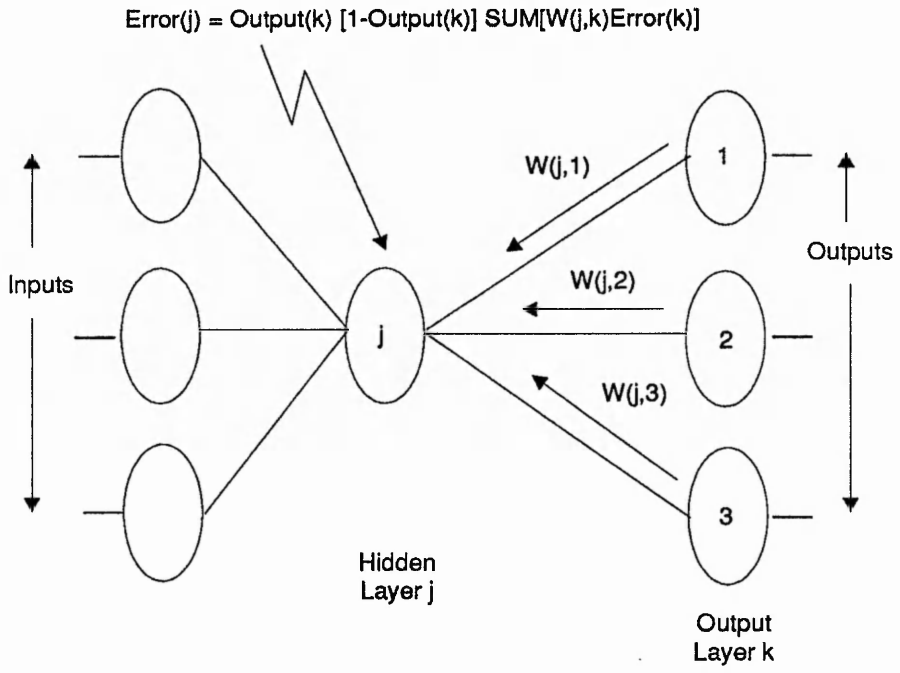


Figure 1.14 Backpropagation Architecture

and the error of each hidden unit is then calculated as :-

$$\delta_j = f'(Y_j) \sum \delta_k W_{kj} = o_j (1 - o_j) \sum \delta_k W_{kj} \qquad (1.16)$$

Equations (1.15) and (1.16) are used to adjust the connection weights between hidden-to-output and input-to-hidden layers. For hidden-to-output, weights are adjusted by :-

$$\Delta W_{kj} = \eta \, \delta_k \, o_j \qquad (1.17)$$

For input-to-hidden, weights are adjusted by:-

$$\Delta W_{ij} = \eta \, \delta_j \, o_i \qquad (1.18)$$

The Back-propagation learning algorithm can be summarized as :-

1) Initialize all connection weights and the thresholds to small random values.
2) Present an input from a class and specify the desired output.
3) Calculate the actual output of all the units using the present value of weights and inputs by equation (1.12).

4) Find the error terms δ_j and δ_k for all the output and hidden units by equations (1.15) and (1.16).
5) Adjust weights between hidden-to-output and input-to-hidden by equations (1.17) and (1.18).
6) Present another input and go back to step (2) until weights stabilize.

One difficulty with the Back-propagation algorithm is that many presentations of the training data are frequently required for weights to converge to an acceptable level especially when the decision regions, see Figure 1.13, or the desired mappings are complex. It is a time-consuming algorithm. Fortunately, this only occurs during learning and does not affect the response time during normal forward computation.

1.12 The Hopfield Model

The Hopfield model [Hopfield,1982] is a self-organizing, associative memory model, see appendix (A.1). A Hopfield network is composed of a single-layer of neurons that act as both input and output. The neurons are symmetrically connected; the connection weight W_{ij} from neuron j to neuron i is the same as the connection weight W_{ij} from neuron i to neuron j (i.e. $W_{ij} = W_{ji}$), see Figure 1.15. Hopfield networks are made of nonlinear binary threshold units which are capable of producing two output values: -1 (off) and +1 (on), the same type of unit as in the Perceptron.

In the Hopfield model, the connection weights are pre-calculated and set in advance so a Hopfield network does not perform any learning. The weight matrix is created by taking the outer product of each input pattern with itself and adding all the outer products.

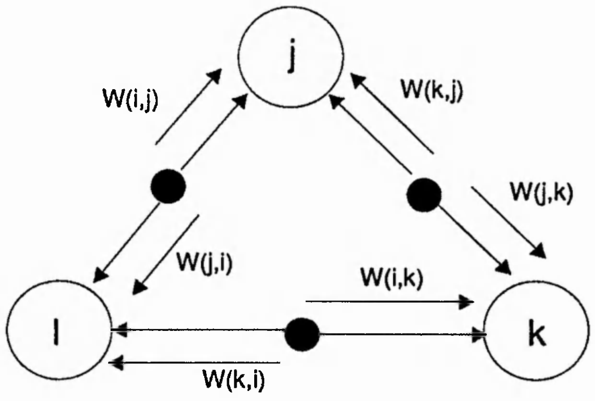
The Hopfield model associates an energy with the states of a network and once an input pattern is given to a network, the system seeks an energy minima. The global energy of the system is defined as:-

$$E = -\sum W_{ij} S_i S_j + \sum \theta_i S_i \tag{1.19}$$

$$\Delta E = \sum W_{ij} S_i - \theta_i \tag{1.20}$$

where S_i is the state of the ith neuron (-1 or +1), θ_i is the threshold, and ΔE is the change in energy due to the unit changing its state.

The Hopfield network is subject to an Updating Rule, as



Neurons Are Symmetrically Connected

Figure 1.15 A Simple Hopfield Net

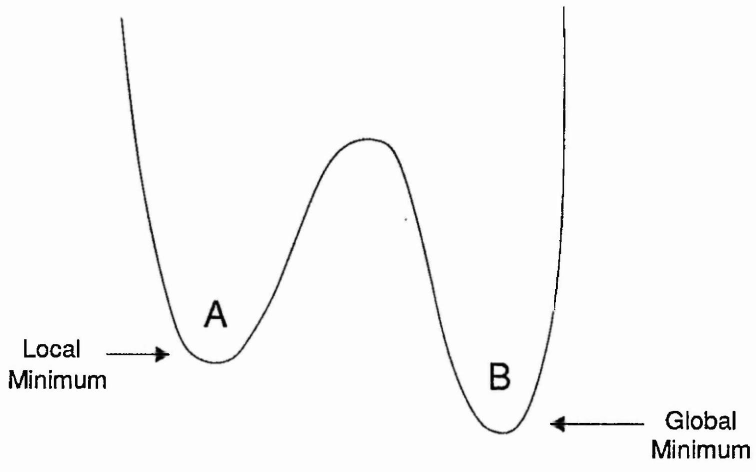


Figure 1.16 An Energy Function

follows: "Randomly pick a unit, change its state (from -1 to +1 or vice versa) just in case doing so will lower the energy". Thus the state of the network changes until it enters a state of minimal energy in the sense that no change in any one of the variables S will lower the value of energy E.

In general, in a Hopfield network, certain units are designated as inputs; their values are clamped so that the Updating rule is not applied to them. The Updating Rule is repeatedly applied to all other units until the network settles; and then read the value of certain designated output units.

One of the weaknesses of the Hopfield model is that global minimization is not guaranteed. For example, if Figure 1.16 represents the energy surface of a network, the network might move to the local minimum rather than the global minimum location. Secondly, weights in a Hopfield network are pre-calculated and set in advance so it does not learn. Furthermore, the number of patterns that can be recalled accurately is limited by the number of units in the network.

In spite of the limitations, Hopfield networks are good at association. They can recognize patterns by matching new inputs with the closest previously stored patterns and are especially good for finding the best answer out of many possibilities.

1.13 The Boltzmann Machine

The Boltzmann Machine [Hinton,1985] is another example of supervised models. It is a multi-layer model and is an extension of the Hopfield model described previously. It was developed to overcome the "local minima" problem of the Hopfield like net.

The state of units in the Boltzmann Machine is probabilistically determined. The probability for the ith unit to be in state 1 is defined as :-

$$P_i = P(\Delta E_i) = \frac{1}{1 + \exp(-\frac{\Delta E_i}{T})} \qquad (1.21)$$

$P(\Delta E_i)$ is a sigmoidal probability function, T is a parameter analogous to temperature and measures the noise introduced and ΔE_i is the total input to the i-th unit, (i.e. $E_i = \Sigma W_{ij} S_j$).

In much the same way as the Hopfield model, after presentation of the input values, if the network is left, it will settle into a minimum just as if it was being used to recall stored data. The problem is that this might not be the global minimum. In order to force the system to settle into the global minimum a technique called "Simulated Annealing" [Hinton,1985] is used. The basic principle of this technique, taken from statistical mechanics, is that if a sufficient random element is added to each unit's choice of state, then the network can escape from local minima. Furthermore, if this randomization is allowed to persist for long enough time, the system will reach an equilibrium state. Within this equilibrium state the network will occupy minima in proportion to the size of random element, so will spend more time in the global minimum.

The Boltzmann Machine learning algorithm consists of learning cycles of two phases and can be summarized as follows :-

1) The supervised phase. The inputs and outputs are held at the equilibrium state at a low non-zero temperature. For the next short period of time the weights are modified as follow: For each unit of time, the weights between

two active units are incremented by a small amount; in much the same way as in the Hebbian Rule.

2) In the second phase, the inputs are clamped and the outputs are calculated. The same inputs are used. The network is again brought to equilibrium and for the same short time as before, but now decrement the weights between active units by a small amount.

This process is continued until the average change of weights becomes zero. The major disadvantage of the Boltzmann Machine is that because of its probabilistic character it takes a long time for a network to settle down to a global minimum.

1.14 Competitive Learning

The models and algorithms that have been described previously except the Hebbian Rule are supervised learning techniques. Network models employ supervised learning technique are common in one point: they all assume that data for both inputs and outputs of a network are available. This however, may not necessarily be the case as with real biological systems. It is true that human beings or animals learn to react through experience. On the other hand, there is no indication that the brain is functioning as a supervised learning process. This is perhaps one of the most important reasons that motivated the introduction of the Competitive learning models.

The basic architecture of a Competitive learning model is illustrated in Figure 1.17. It consists of a set of hierarchically layered units in which each layer connects via excitatory connections, with the layer immediately above it, and has inhibitory connections to all units in its own layer. Within a layer, units are broken into a set of inhibitory non-overlapping clusters. The clusters behave in "winner-takes-all" fashion, such that the unit receiving the largest input is turned on and the other units in the cluster are turned off.

Each unit has a fixed amount of weight that is distributed among its input lines. The weight on the line connecting unit j to unit i is designated W_{ij} and the fixed total amount of weight for unit j is designated $\Sigma W_{ij}=1$. A unit learns by shifting weights from its inactive to its active input and wins the competition, then each of its input lines gives up some portion, g, of its weight and that

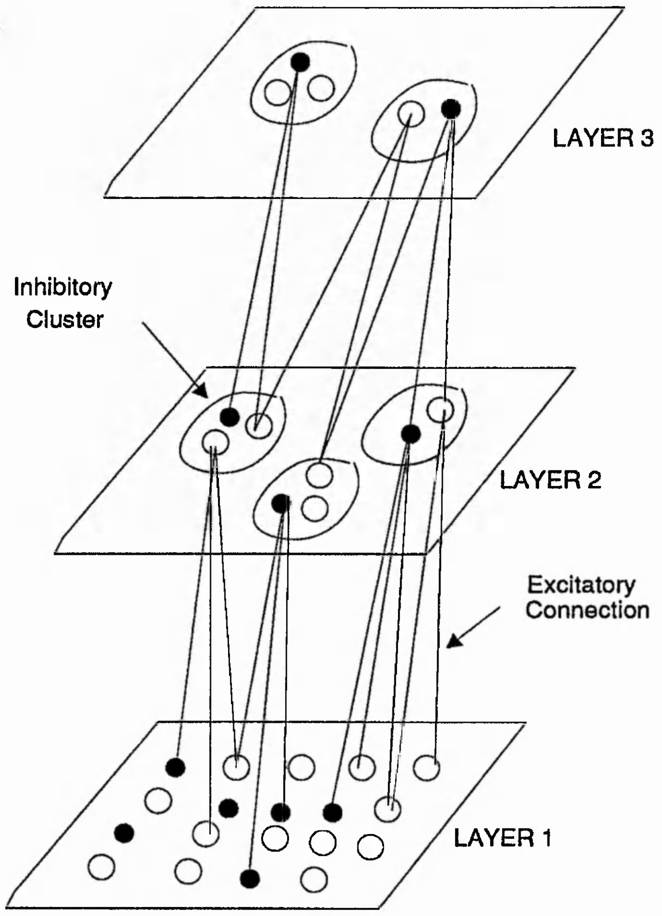


Figure 1.17 Competitive Architecture

weight is then distributed equally among the active input lines:-

$$\Delta W_{ij} = g \frac{c_{ik}}{n_k} - g W_{ij} \qquad (1.22)$$

where c_{ik} is 1 if in stimulus pattern S_k unit i in the lower layer is active and zero otherwise, and n_k is the number of active units in pattern S_k.

There are many variations on the competitive learning theme. A number of researchers have developed variants of competitive learning mechanisms. [Fukushi,1980], [Grossbe, 1987], [Kohonen,1984] among others have developed models which are competitive learning models or which have many properties in common with the competitive learning.

CHAPTER 2 NEURAL NETS FOR DYNAMICAL SYSTEMS

CHAPTER 2 NEURAL NETS FOR DYNAMICAL SYSTEMS

Neural networks have been applied to the fields of vision, speech and control. The ability to interact with environments and to solve problems in real time is fundamental to the implementation of more advanced control systems. It is important to investigate how neural nets can acquire these abilities by means of interaction of many simple processing elements. This chapter presents a selected survey of neural net implementation in the field of control/dynamical system applications.

2.0 Historical Perspective

Research in the field of neural-style computing systems (neural networks) can be traced back to the early forties [McCullo,1943]; the same period as the first electronic special purpose computer [Ashurst,1983]. With the success of, and competition from von Neumann computers, interest in neural networks decreased. Even after the Cybernetic philosophy (the study of control and communication in animal and machine) was established many researchers abandoned the neural-style approach and moved towards the modelling of human mental processes. It is only relatively recently that research in neural networks has become of great interest. In part, this is due to the increased understanding of brain functions, and, in part, due to the increased availability of computing power for the evaluation of theoretical models. Another factor stimulating research in this area, as already mentioned in chapter 1, is the disappointment with the performance of current computer technology when trying to solve problems that humans do well.

Although there is a much better understanding of the architecture and function of the biological brain, knowledge of the underlying dynamics that govern the intrinsic learning mechanisms are still very vague. In order to uncover this missing piece and increase understanding of the brain's dynamics, various architectures of neural-style models have been built and studied.

The fundamental building block of a neural-type system is a nerve cell like device, often referee to as artificial neuron or processing element or unit. In 1943, a model of such device was proposed by McCulloh and Pitts [McCullo,1943]. Under the influence of this model and in the years following the discovery of the Perceptron in 1962 [Rosenb,1962], many new models and sophisticated techniques were developed. Each offers a different approach to the problem of learning and adaptation: some are good at generalization, some can associate better than others. Currently, there are more than 50 models available.

2.1 Neural Nets For Control

Applications of neural nets to control of dynamical systems include: tracking of moving objects, robot arm control, pole balancing and the estimation of forces acting on reentry vehicles.

2.1.1 Tracking Of Moving objects

The tracking of a moving object belongs to the class of so called 'difficult' problems for conventional computer technology [Dobnika,1989]. A system proposed by Dobnika demonstrates that with the use of a multilayered artificial neural network the same quality of tracking results could be achieved as with classical methods.

Figure 2.1 shows the organisation of a system for visual tracking. The system consists of an Input Array (IA) where the camera's signals are stored and a multilayered neural network for different types of processing. The neural network accepts input information from the Input Array, records it topologically correctly in the first layer, performs filtering in the second and finds the centre of mass in the third layer. Each succeeding layer of the neural network performs specialised operations that represents higher or more abstract degree of processing. Its operation is synchronized in such a way that all processing elements on the same layer act simultaneously, while succeeding layers are sequentially enabled. The output of the processing elements of the first layer follows the well known McCulloch and Pitts equation in order to make a weighted record of input image from the input array :-

$$\eta_j = f\left(\sum_{i=1}^{n} W_{ij}\zeta_i \right), \quad j = 1 \ldots n \tag{2.1}$$

where W_{ij} denotes the weight from Input Array ζ to the first layer η. f() is the output nonlinear function.

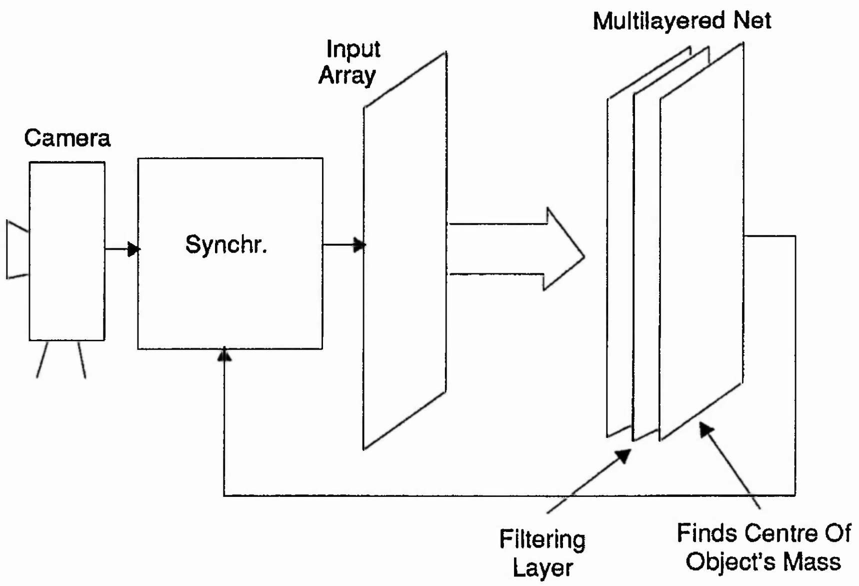


Figure 2.1 Neural Net Object Tracking System

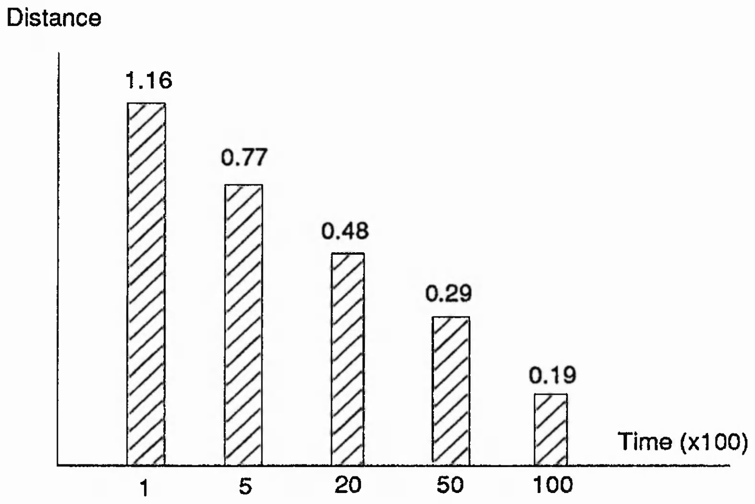


Figure 2.2 Convergence Of Learning

Processing elements in the second layer μ perform filtering operation by following the equation :-

$$\mu_i(t+1) = f(\mu_i(t) + k\,(\eta_i(t) - \mu_i(t))) \qquad (2.2)$$

where k is the gain factor of a dynamic filter of the first order and is changed according to the equation k = 1/processing steps. The third layer is responsible for modification of the second layer in order to achieve better selectivity between elements, which facilitates detection of the centre point. The difference between the centre of the objects's mass and the centre of the layers corresponds to the velocity of the object in the Input Array. According to that difference, the change of visual data in IA is activated, which in turn causes registration of new data into IA, that again starts the processing of all layers. The function of the third layer is achieved by the following processing equation :-

$$X_i = f(\sum_{j=1}^{Ni} \alpha_{ij}\,(\mu_j + \sum_{d} \delta_j(d))), \quad i=1...Ni \qquad (2.3)$$

α_{ij} denotes the weight between i-th element in the third layer and its neighbourhood Ni, $\delta_j(d)$ is increment to μ_j in the object, obtained by linear extrapolation of the object's edges in d direction, dE $\in$ (NS, NE-SW, EW, SE-

NW). The centre point is found by locating the maximum value of processing elements in the third layer. This multilayer neural network model uses an algorithm similar to the one described in [Kohonen,1990]. The results, illustrated in Figure 2.2 show that the average distance between the chosen centre of moving objects and the processed one considerably decreases with increasing number of learning steps.

2.1.2 Dynamic Neural Controller Model

The basic objective of a controller is to provide the appropriate input parameters to a physical process in order to obtain the desired output. In conventional approaches to process control, a significant amount of time and effort is spent developing control laws that describe how a process works and how it can be controlled. It would be interesting and very useful if the controller could learn to control the process in an interactive and autonomous way by observing the behaviour of the system, and by continuous adaption to the process. [Saevens,1989] propose a specialized learning method that allows a neural net to learn to control a process in an autonomous way, without specific learning stage.

Specialized learning means that the controller learns from a direct evaluation of the network accuracy with respect to the output of the plant. Figure 2.3 shows the learning architecture of this method. The network uses the difference between the actual and desired output of the plant to change the weight of connections. Moreover, the network learns continually and can therefore be used with processes having time varying characteristics. The algorithm consists of the following steps :-

(1) The controller receives the actual output state of the plant and the desired output parameters that have to be provided by the plant.

(2) The network outputs control parameters λ_j associated with μ_{dj}.

(3) Those parameters λ_j are input to the plant at time t.

(4) The plant outputs $\mu_i \neq \mu_{di}$ at time $(t+\Delta t)$.

(5) The error is evaluated and back propagated into the network.

The proposed back-propagation based learning algorithm has several interesting properties. There is no specific learning stage, - the system is self tuning. For static targets, the controller is immediately operational but requires several steps to reach the target. The system is

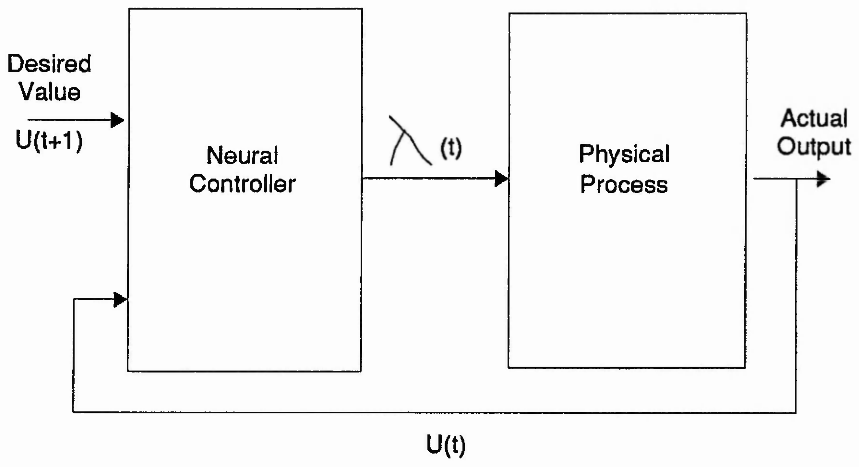


Figure 2.3 Specialized Learning Architecture

able to perform autonomous on-line learning on dynamic targets. It differs from conventional back-propagation learning by the fact that the system learns "by doing" and not "by example".

This specialized learning control algorithm has been applied to on-line learning on a dynamic target, namely robot arm control and the pole-balancing problem. In both experiments, the controller is a network with four layers (two hidden layers). Every processing element of each layer is connected with elements of adjacent layers, and the controller learns using the back-propagation algorithm with the learning rate and momentum term fixed to 0.1 and 0.2 respectively.

2.1.2.1 Robot Arm Control

The first application of the specialized learning scheme involves a robot arm, with two degrees of freedom which has to follow a moving target confined in a 2-d space, see Figure 2.4 for illustration. The control parameters of the arm are the two angles β and α. A camera transmits both the coordinates of the tip of the arm (x_d , y_d) and the object's position (x_o, y_o) to the controller. The

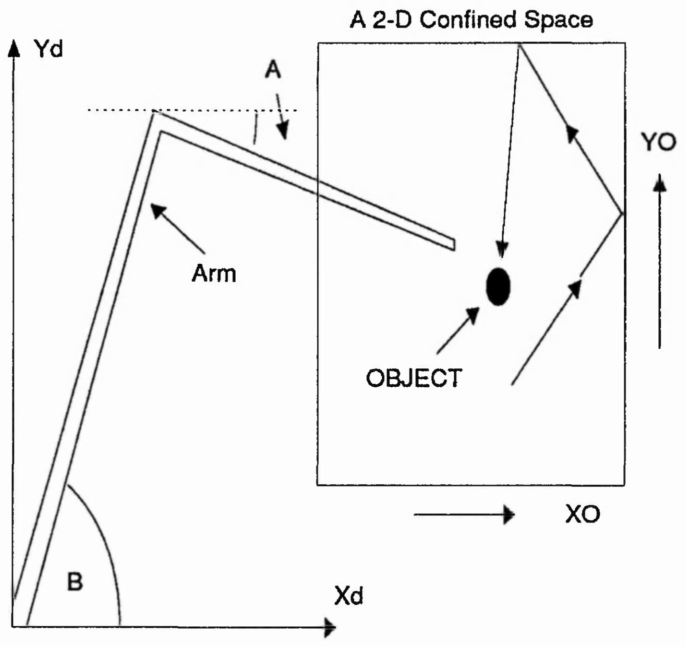


Figure 2.4 Robot Arm With 2 Degrees Of Freedom Following An Object

controller has to supply angles that permit the arm to reach the target.

Training is performed while the target is moving. With the network organisation as described above, the reaction time of the controller is recorded as 0.15 seconds, during which the target moves.

Two series of simulations were recorded. In the first, the neural network was given the coordinates of the target alone. The controller was unable to anticipate the movement of the target because the controller has no idea of the target's speed and direction. According to [Saevens,1989] the average distance between the arm and the target converges towards the product of arm reaction time and target velocity; the controller learns to move to the previous position of the target. During the second series of experiments, the network was given the difference between previous position and actual position of the target in addition to actual position. In this case, the network learns to anticipate the movement of the target after 8 mins.

2.1.2.2 The Pole Balancing Problem

The specialized learning scheme has also been applied to the Pole Balancing problem. A two-dimensional pole and wheeled-cart system is show in Figure 2.5. The pole is free to move only in the vertical plane of the cart and track. The cart can travel along the track. The goal is to produce a sequence of forces upon the cart's centre of mass such that the pole is balanced for as long as possible and the cart does not hit the end of the track.

Knowledge of the desired equation of motion of the cart-pole system was not used during learning. The same network organisation was used as in the case of robot arm application [Saevens,1989]. Four parameters were given to the network: the horizontal position of the cart relative to the track x, the horizontal velocity of the cart, x', the angle between the pole and vertical, θ, and the angular velocity of the pole, θ'. The learning algorithm can be summarized as follows :-

(1) The network receives the actual state variables (x, x', θ, θ').

(2) The controller back-propagates the error and outputs a force associated to the state variables. Saevens

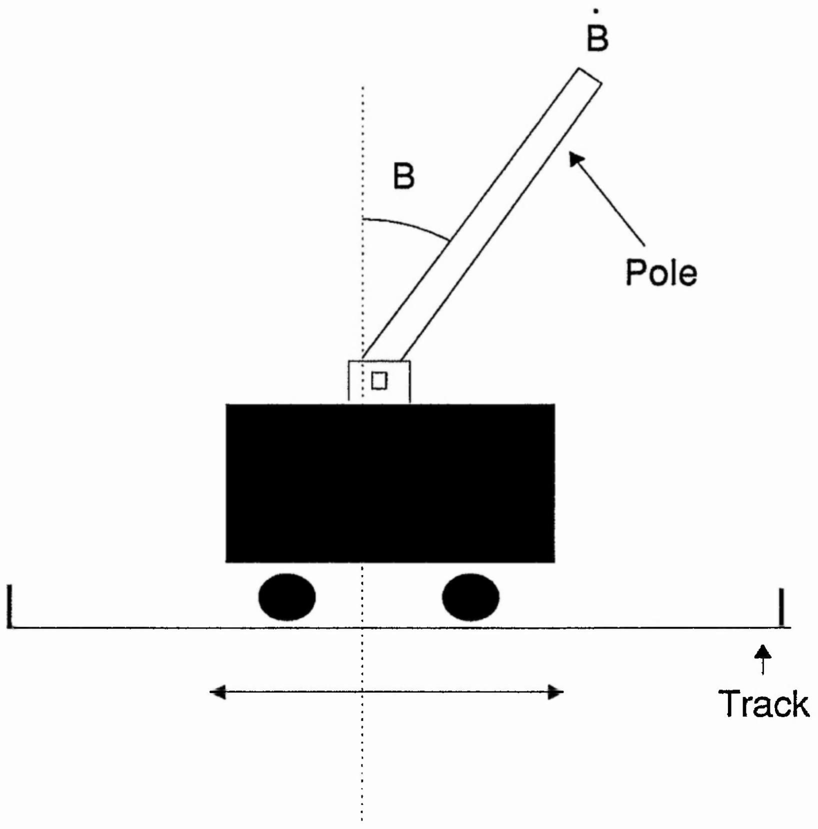


Figure 2.5 The Pole-Cart System

& Soquet have estimated that the transmission of information and processing takes 50 ms to control the cart.

(3) State variables are observed and return to step (1).

With the application of this learning algorithm, after several trials, the controller is able to balance the pole for more than 15 minutes.

2.1.3 State Estimation Of Unknown Forces Acting On Reentry Vehicles

The estimation of unknown forces acting on maneuvering reentry vehicle, MRV, can be thought of as finding the inverse dynamics model that has as input the observation and as output the forces. In the past, neural networks have been successfully used in the estimation of an inverse dynamics model of systems in robotics, [Jenhwa,1989] [Miyamoto,1988].

Conventional approaches to developing neural networks that act as inverse dynamics models can be characterized into two categories: off-line and on-line training.

In the off-line training category, the neural network is first taught how the forces acting on an MRV are generated. In the training phase, the network weights are changed to minimize the error between the true known forces and the forces estimated from the network. The inputs to the network are the pre-processed observations and their history as illustrated in Figure 2.6. After training has been accomplished with a finite set of data, and weights have converged, the network now represents the inverse dynamics model and it is ready to be used with real data.

In the on-line training category, the network is taught as the data is coming in and being processed. In this approach, no time is wasted on training, as illustrated in Figure 2.7. The pre-processed observation is the desired output and it drives a conventional neural net. The output of the neural net represents the estimated forces, which is added to a scaled value of the error to form the input to the target dynamics. The error signal is used to modify the estimated weights of the neural net, using, for example, Back-propagation. [Abutale,1991] points out that a major problem with these networks is that the form of nonlinearities are assumed. This issue is important because this form may not be the best to be used in the task. Also the size of the network, for any realistic problem, is sometimes prohibitory large if reasonable results are to be

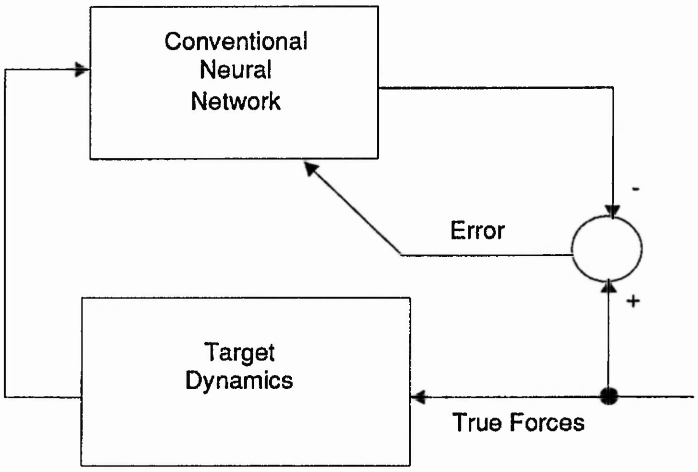


Figure 2.6 Off-Line Learning

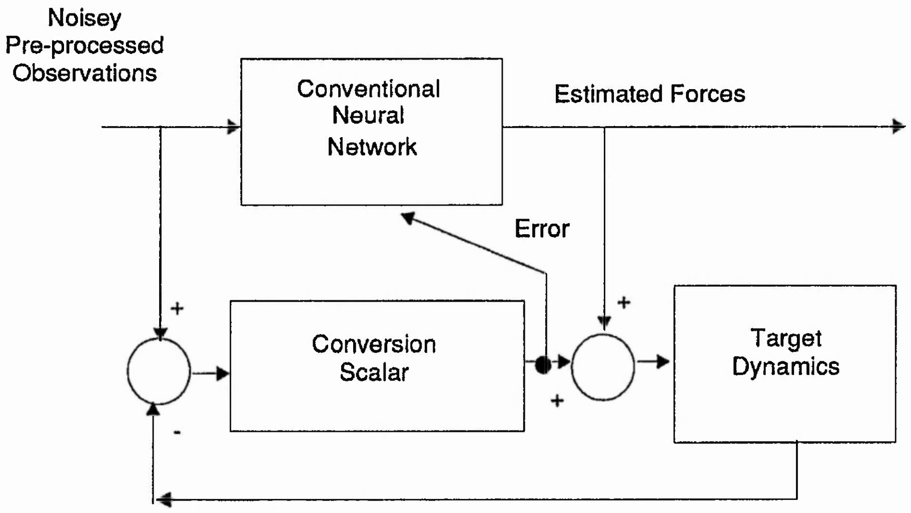


Figure 2.7 On-Line Learning

obtained. This is due to the fact that network weights are assumed to be constant parameters. Furthermore, it is known that a large number of parameters are needed to represent any useful primitive, and since the goal is to estimate the values of the parameters, the computational task becomes enormous. All these problems motivated [Abutale,1991] to develop an alternative network.

[Abutale,1991] proposed a hierarchical approach to estimate the unknown forces acting on a radar target, see Figure 2.8. The neural net based procedure can be summarized as follows :-

(1) Obtain an estimate of the target state, position and velocity, using an extended Kalman Filter method.
(2) Calculate the acceleration or the derivative of the state using a polynomial fit to the estimated states.
(3) Develop a neural net to estimate the unknown forces.

The proposed network architecture is shown in Figure 2.9. It is similar to the one described by [Miyamoto,1988] except for one fundamental difference; the network nonlinearities are not assumed, they are estimated implicity.

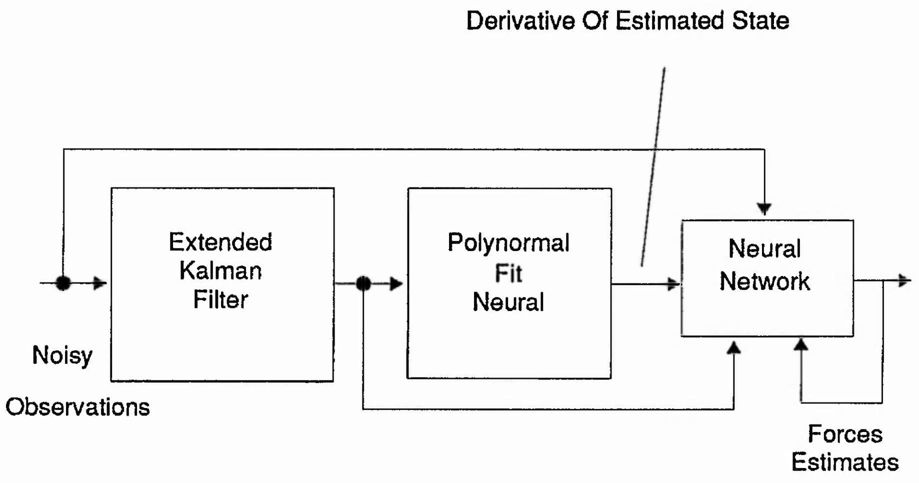


Figure 2.8 Neural Net Based System To Estimate Unknown Forces

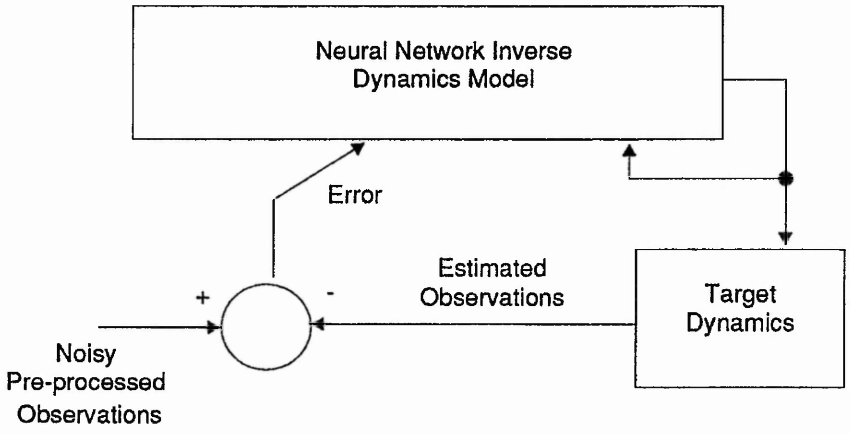

Neural Network Inverse Dynamics Model
Error
Estimated Observations
+
-
Target Dynamics
Noisy Pre-processed Observations

Figure 2.9 Dynamic System Architecture
Proposed By [Abutaleb]

The architecture has the advantage that no off-line teaching is needed. The network learns from the observations and at the same time generates the estimates of the unknown forces. The network equations are much simpler and easier to work with, and its convergence is fast when proper tuning parameters are used. However, the usefulness of the present algorithm is limited to application where system dynamics and the observation equation are known.

2.2 Limitations

The present understanding of real biological neural networks in respect to learning dynamics is very sparse. Neurobiologists may be able to reveal certain properties of certain type of neurons and the actual physical structures that these neurons are embedded in but the dynamics which govern these structures largely remains unknown. Obviously, it is one thing to model neurons to show that they have sufficient logical power to perform some computations, it is quite another to understand how the neurons in actual biological systems perform their tasks.

With limited knowledge, a precise detailed model is hard to achieve, therefore assumptions have to be made when modelling neurons and networks. One crude approach is to assume that the defined models do correspond to real systems, but only to a subset of them. In fact, no modelling approach is automatically appropriate, a model can be regarded valid even when elements of the model network details are not directly identified with real biological systems, for example the McCulloch-Pitts neurons used by many later network models. The most direct approach seems to be to design the simplest model adequate to address a given problem and then work backward to justify the model with real systems.

2.3 Feedback

In controlling interactions of a system with its environment, it is usually important that information be continually fed back from receptors/sensors to tell the controller how effective it is in controlling the interactions. Feedback is the comparison of actual performance with some desired performance. It plays an essential role in the control of an organism or artificial robot, e.g. movement of four limbs and eye co-ordination.

In feedback control, Figure 2.10(a), the actual output is continuously compared with the desired output to provide compensation signals so that the output is maintained near to its desired value.

When a single integrator is used direct feedback from output to input will provide an estimate of the derivative of a function. This type of arrangement is commonly used in electronic equipment for the detection of weak signals buried in noise. The integral over a time interval will indicate the percentage of time that the pattern lies along the trajectory. Thus for noisy data, the magnitude of the integral of a signal over some interval is much more reliable than the instantaneous value of the signal itself. As shown in Figure 2.10(b), with feedback from output added to the input the new output will follow the input and the by-product will be the higher state, time derivative, of the input displacement trajectory, F(t). To adopt this scheme, a neural network integrator is desired.

2.4 The Pictorial Integration Process

The main component of the neural module is a neural net which computes the integral of the input displacement-time

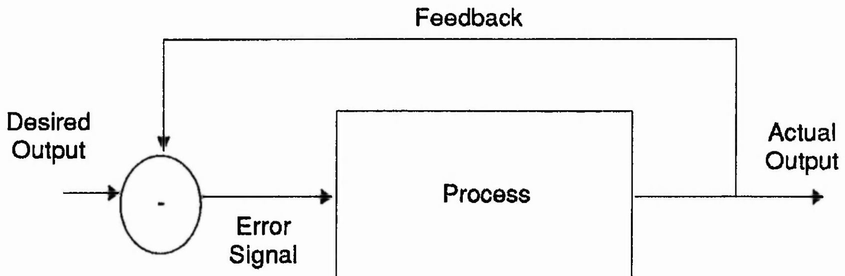


Figure 2.10(a) A General Scheme Of Feedback Control

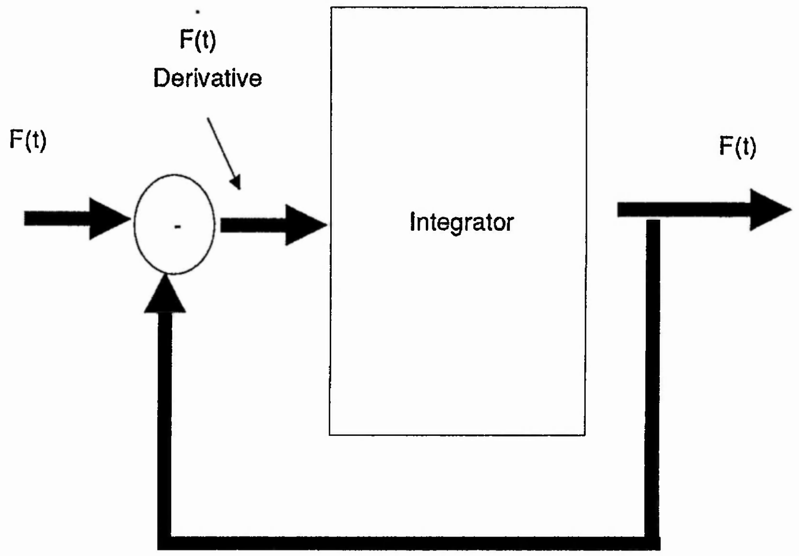


Figure 2.10(b) Integrator To Estimate Time Derivative

pattern. To aid the description of the neural integrator, the graphical integration process of a pictorial pattern is described in the following manner.

As illustrated in Figure 2.11 (a)(b), the definite integral of a function, f(t), within the limits of a & b, is equivalent to the area bounded by the graph of the function f(t), the horizontal axis and the lines parallel to the vertical axis at "a" and "b". Using rectangular, or Euler, integration it is possible to divide the interval into small sub-intervals of equal width δt_i and select from each sub-interval a value for the variable $f(t_i)$ as shown in Figure 2.11(b). The total area may be calculated by summing the area of the rectangles, $f(t_i)\delta t_i$.

Alternatively, if the function f(t) is plotted on an equally spaced/grid area, see Figure 2.12(a), a more appropriate way for the neural integrator can be determinated. Firstly there exists only two possible shapes: a single rectangle or a combination of a rectangle and a triangle. Secondly, as the function is plotted on a 1:1 scale, the area of any shape can easily be calculated by the following simple equation: Half The Distance Separating The Two Dots Plus The Hight Of The Rectangle, see Figure 3.12(b).

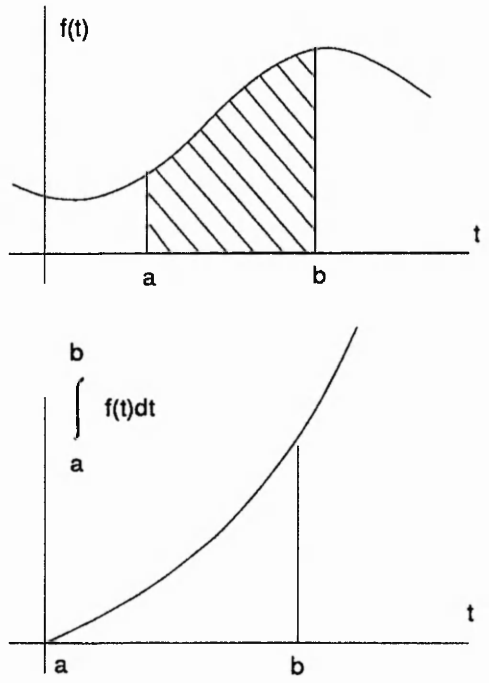


Figure 2.11(a) Graphical Representation Of Integral

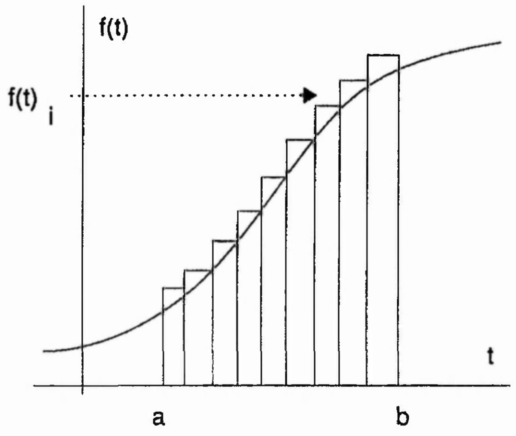


Figure 2.11(b) Approximation Method Of Integral

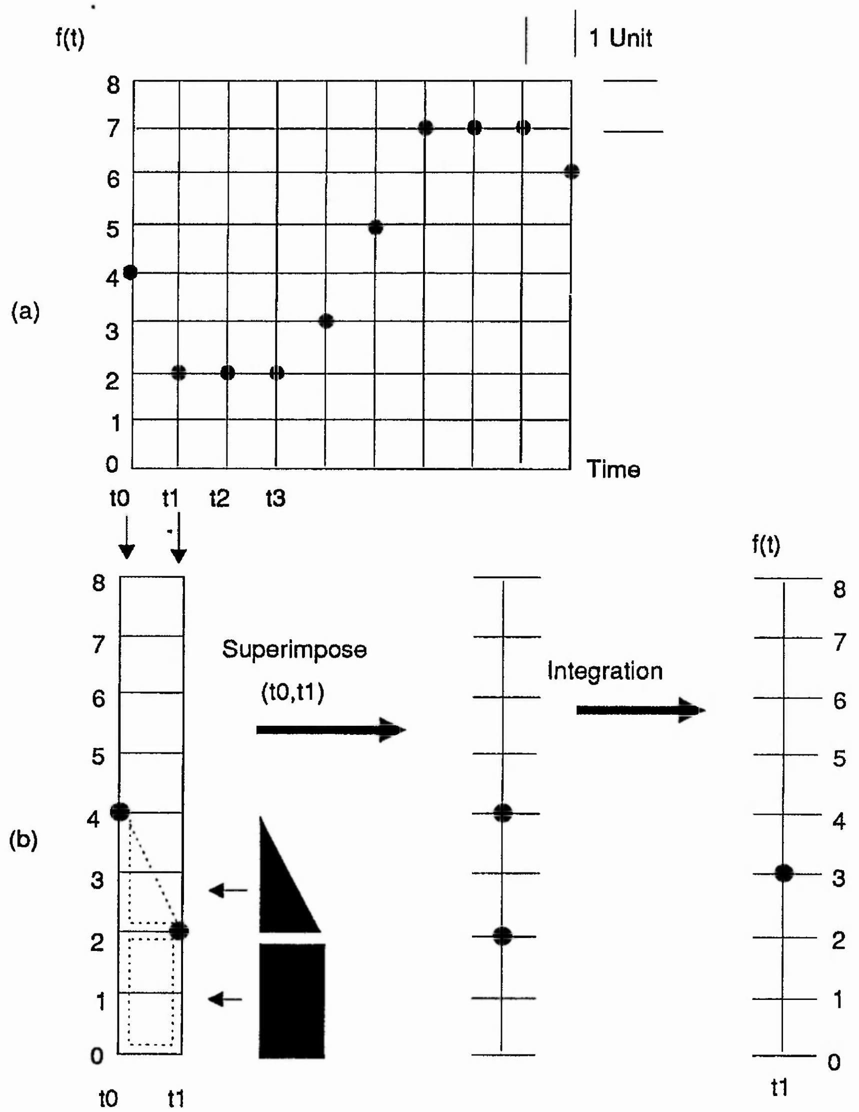


Figure 2.12 Estimation Of Area Of An Unknown Function

It is clear with this representation that regardless of the shape of the rectangle strip, the corresponding value for the area is the midpoint of the two dots. As an example, consider a 2-D displacement-time trajectory image. For each pair of successive input samples to the neural integrator at t_n and t_{n+1}, the resultant pattern is the pattern at t_n superimposed on to the pattern at t_{n+1}. The neural integrator takes in the superimposed pattern and produces an output representing the centroid of the two points, (area or integral between t_n and t_{n+1}).

Although this calculation is trivial using digital arithmetic units, the problem of scale emerges when repeating the process millions of times over high resolution 2-D images. It is therefore justifiable to explore a non-numeric alternative.

2.5 Vertebrate Retina Structure

As an essential background to the integrator model, a brief summary of the features of the vertebrate retina is included. This summary is by no means complete, and it will necessarily be an oversimplification of many aspects of the real model and only serves as supportive section. A more detailed description of the retina can be found in [Dowling,1987].

Living organisms perform visual processing by perception of light through eyes, recording it onto the retina, and distributing the signals to different areas of the neural system in the brain where it is believed that processing such as object recognition is actually performed. From the complex anatomical structure of the vertebrate retina, it is apparent that a great deal of processing of the visual image must take place inside the neural networks of the retina.

Figure 2.13 is a schematic drawing of the synaptic contacts observed in many vertebrate retina. The retina consists of five types of neurons: photoreceptors, horizontal cells, bipolar cells, amacrine cells and ganglion cells.

Light absorbed by **photoreceptors** is converted to electrical signals. These signals are then transmitted through the output synaptic cells: **bipolar cells** which transmit excitatory information directly from photoreceptors to the ganglion cells immediately beneath them in the inner plexiform section of the retina and the **horizontal cells** which mediate local lateral interaction in the outer

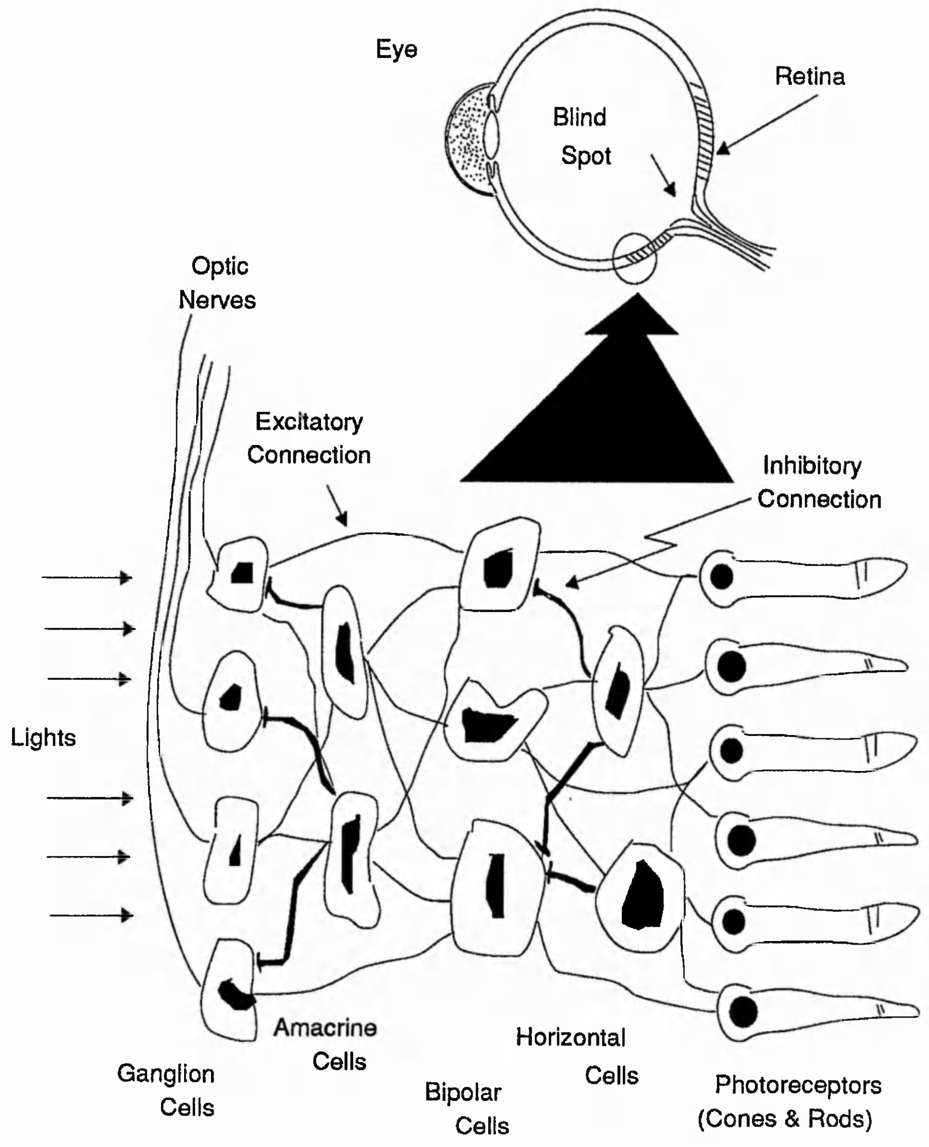


Figure 2.13 Vertebrate Retina Structure

plexiform section of the retina.

The **amacrine cells** are also inhibitory cells connected to bipolar and ganglion cells. Amacrine cells provide inhibitory input to ganglion cells similar to the inhibition imparted by the horizontal cells onto the bipolar cells in the outer plexiform section of the retina. The amacrine cells therefore can contribute to the centre-surround response of the ganglion cells. Their response is transitory in contrast to the continuous response of the bipolar and horizontal cells. When the photoreceptors are first stimulated, the amacrine response is intense, but this response dies away very quickly. This transient response produces a sensitivity to changing light intensities.

Ganglion cells receive inputs from bipolar cells and amacrine cells and send their axons to the brain via the optic nerves for further processing. The several levels of processing which culminate in an output ganglion cell can be best illustrated by the following example.

Figure 2.14 shows the portions of the so called visual 'receptive field' which affect the activity of a typical output ganglion cell type. This ganglion cell receives

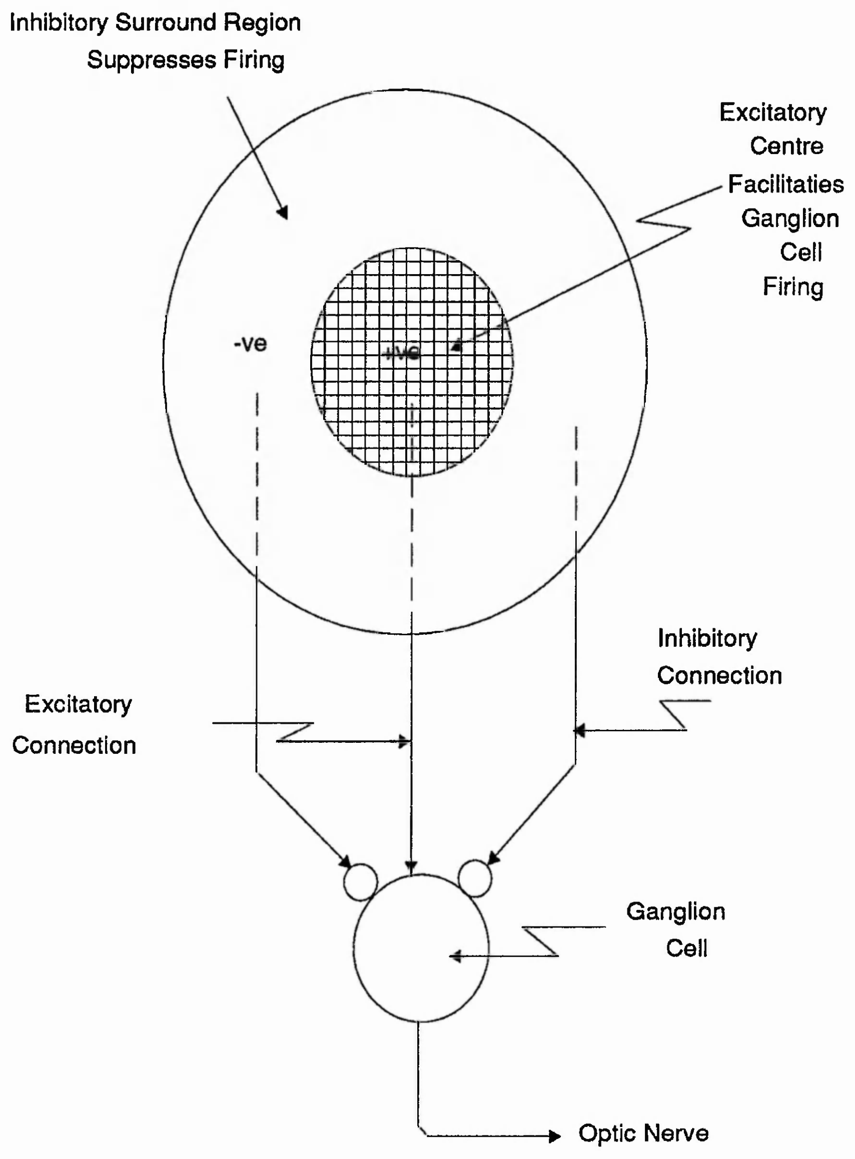


Figure 2.14 Receptive Regions Influence Firing Of Ganglion Cell

excitatory synapses from a small group of photoreceptors in a central spot (+ve region) and inhibitory synapses from photoreceptors in a ring surrounding the central spot (-ve region). When the central spot receives light, it increases the firing rate of the ganglion cell. When the inhibitory surround is illuminated, it decreases the output of the ganglion cell. If the entire 'receptive field' is illuminated, the excitatory and inhibitory effects tend to cancel.

The receptive field of ganglion cells are not fixed. In some of the ganglion cells it is the inverse of the one above. A structurally similar but functionally inverse set of interconnections produces 'centre-off surround-on' response.

Ganglion cells receive direct input from bipolar and amacrine cells. [Dowling,1987] suggests that the ganglion cell responses strongly reflect the properties of the input neurons. Bipolar cells give sustained response to retinal illumination, whereas many amacrine cells respond with transient potentials. Some ganglion cells receive most of their input from bipolar cells; their responses are sustained and reflect primarily the processing of information occurring in the outer plexiform section of the retina. Other ganglion cells receive most of their input from amacrine cells; their responses are often more transient and reflect inner plexiform section processing.

Although the synaptic interconnections between different layers of cells of the retina are still not yet fully understood, the centre-surround receptive field characteristic has provided the inspiration for the design of the neural integrator model.

CHAPTER 3 A NEURAL NET INTEGRATOR MODEL

CHAPTER 3 A NEURAL NET INTEGRATOR MODEL

A model for a neural net integrator is put forward as an alternative to traditional analogue/numerical integration. The execution of the resulting model on single and multiprocessor systems is considered.

The model is inspired by the vertebrate retina structure outlined in chapter 2. Architecture, functionalities and adaption method of the model are treated in detail.

The mapping of the resulting neural net models onto single and multiprocessor system is examined. A general framework is formulated to permit arbitrary network definition and easy alterations of network parameters. A parallel processing technique is devised to compute the resulting algorithms on distributed memory multiprocessor systems.

3.0 Basic Objectives Of The Neural Integrator Model

The neural module has to satisfy four objectives :-

(1) Provide Integration, e.g. linear ramp output for constant input, quadratic output for linear input, etc.

(2) Provide an automatic matching/filtering function to the input trajectory profile when used in a closed loop feedback arrangement, see Figure 3.1, i.e. the output of the neural module should be the same or a close approximation of the input.

(2) As a by product of achieving the above two objectives, the higher state/derivative of the input trajectory should also be obtained.

(3) Neural modules should be cascadable, as shown in Figure 3.1, such that higher states/derivatives can be estimated.

3.1 Neural Integrator Model

Consider the neural integrator (NI) model shown in Figure 3.2. It consists of four layers formed by photoreceptors, horizontal cell, bipolar cells and ganglion cells. Amacrine cells will be used with this configuration when expanded horizontally. The input layer consists of two photoreceptors with output feeding a horizontal and two bipolar cells in the second and third layers respectively. Connection between these cells are excitatory connections.

Bipolar cells in the third layer receive direct excitatory

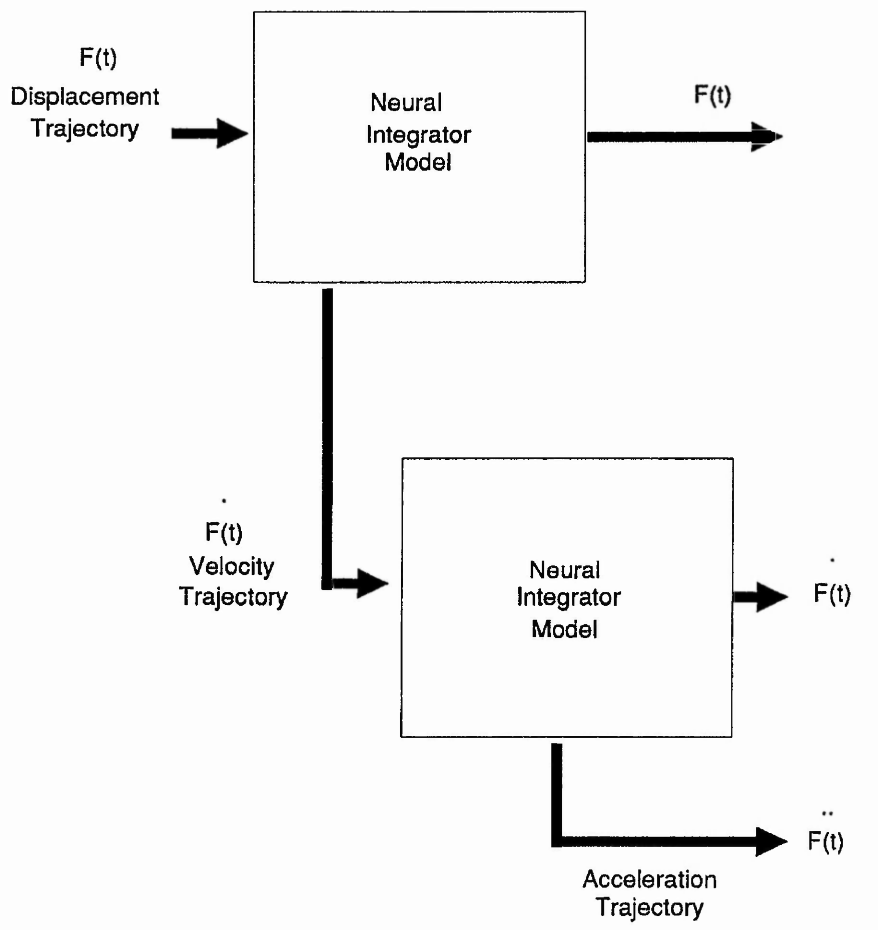


Figure 3.1 Cascaded Neural Module

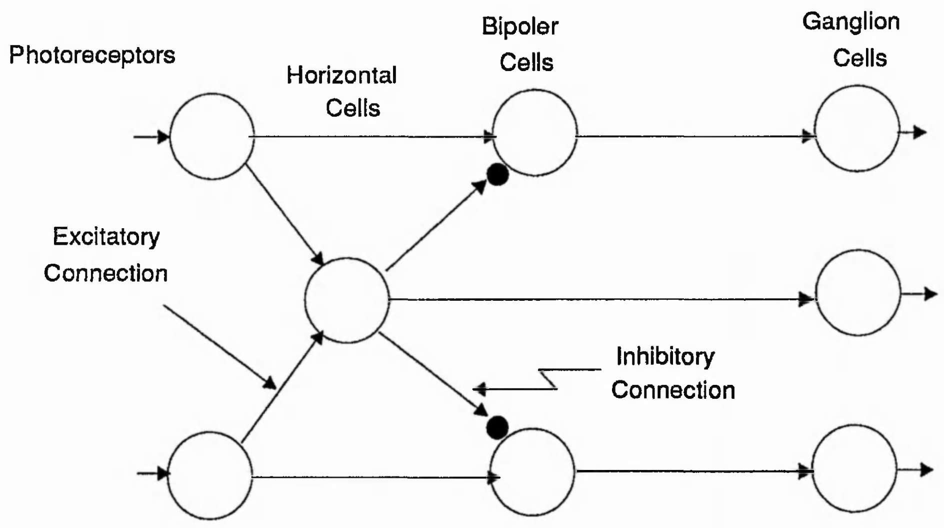


Figure 3.2 Basic Network Model

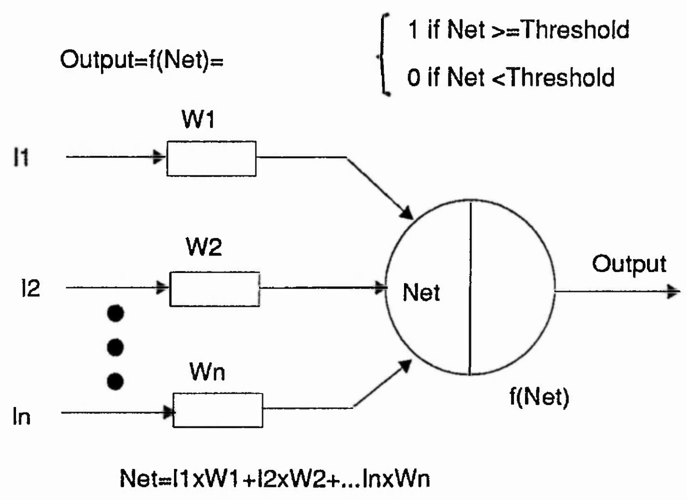


Figure 3.3 The Neuron Model

connections from photoreceptors as well as inhibitory connections from the horizontal cell. The horizontal cells mediate the lateral inhibition, when sufficient inputs are applied to a horizontal cell it will fire causing inhibition of the bipolar cells near to it. The output layer is formed by three ganglion cells with connections coming from bipolar and horizontal cells directly above.

Neurons in this **basic model** are identical, having the same neuron transfer characteristics, see Figure 3.3. A neuron generates an action potential if the weighted sum of the inputs is above the neuron threshold value. Mathematically this can simply be expressed by the following equations:-

$$Net = I_1W_1 + I_2W_2 + \ldots\ldots I_nW_n \tag{3.1}$$

$$Output = \begin{cases} 1 & if\ NET \geq 1 \\ 0 & if\ NET < 1 \end{cases} \tag{3.2}$$

The choice of the above neuronal properties is set by 'two constraints'. The 'first' was to mimic as closely as possible the real nature of neurons. The photoreceptors, react chemically to light and these chemical reactions in turn lead to generating potential in the neighbouring neurons. If the light falling upon an array of receptors is sufficiently strong, then their potential will induce potential changes sufficiently strong to activate other cells.

The 'second' constraint is to synthesis the model to enable the computational steps to be as small as possible; this is important for simulating networks of a large number of neurons with limited computer power in a practical execution times.

3.2 Functionality Of The Basic Net Model

To illustrate the functionality of this basic net model, consider the weight setting as shown in Figure 3.4 :-

a: As neurons are binary On/Off devices, four possible input states can exist, 00, 01, 10 and 11.

b: With no light illumination, the 00 condition; none of the cells are excited by external incident light.

c: In 01 or 10 conditions, the bipolar and ganglion cells directly beneath will be excited and the rest of the cells are off.

d: When both photoreceptors are on, the 11 condition, the horizontal and bipolar cells are initially turned on. As soon as the output of the horizontal cell is established (settled) the bipolar cells are

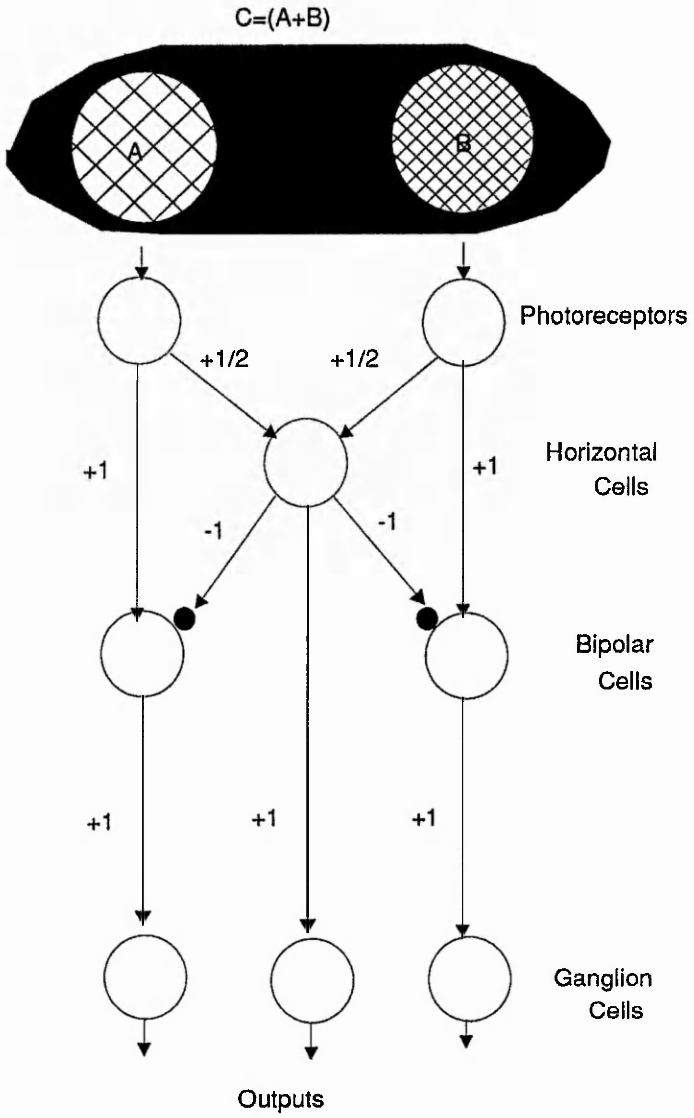


Figure 3.4 Weighted Basic Network Model

subsequently turned off due to the lateral inhibitory connection from the horizontal cell hence turning on the centre ganglion cell (centroid on).

e: Unlike the biological vertebrate retina where the stimuli for ganglion cells may be grouped into 'centre-on/surround-off' or 'centre-off/surround-on', the basic net model has only overlapping centre-on 'receptive field' regions. As shown in Figure 3.4, each region of the 'receptive field' is dedicated to a photorecptor/ganglion cell. Ganglion cells A and C are affected by illuminations on region A and C respectively and region B facilitates ganglion cell B firing.

This structure forms the basic building block of the neural integrator model. Depending on the required number of input and output neurons resolution the basic net model can be expanded sideways, see Figure 3.5, to form larger networks. When more than one basic structure is required an extra layer of amacrine cells is used to maintain the centre-on response. For example, with 3 photoreceptors (size of two basic nets) an amacrine cell is needed which receives input excitatory connection from the farthest bipolar cells, see Figure 3.5 and have inhibitory and exhibitory connections feeding the ganglion cells.

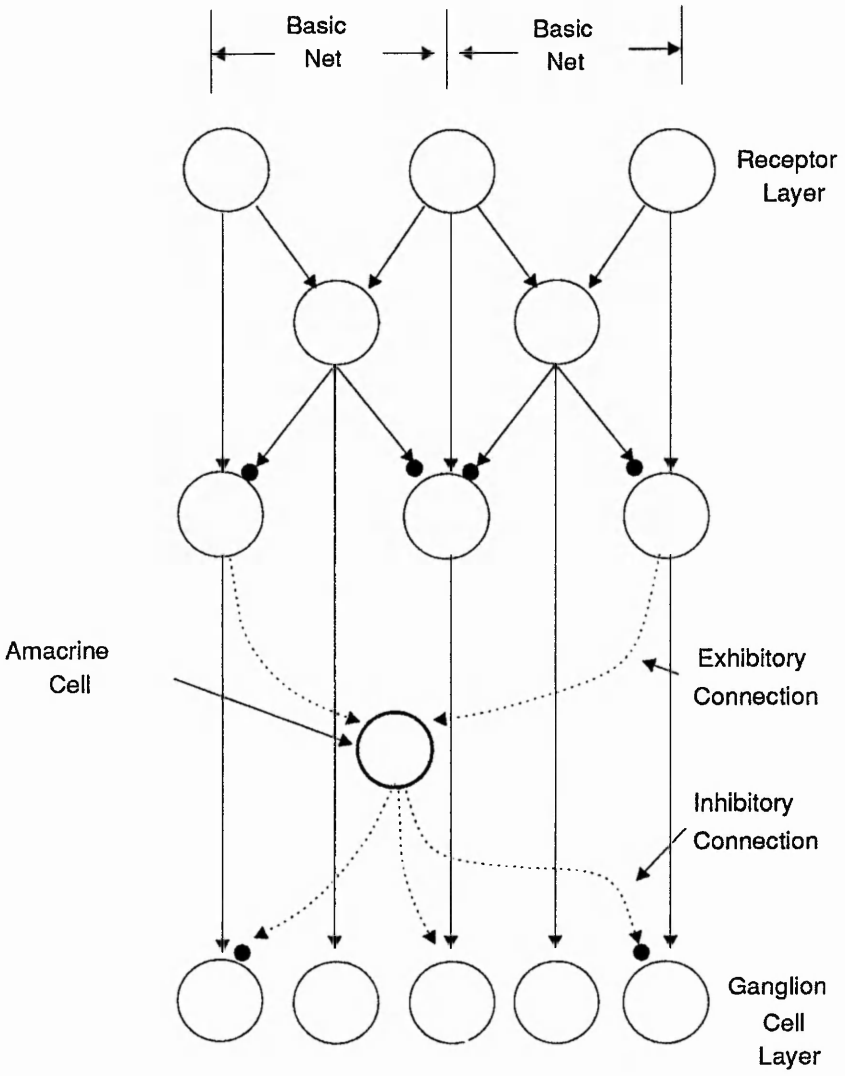


Figure 3.5 Network Construction From Basic Model

3.3 Weight Adjustment

Large networks can be realized from the basic net. Two phases of weight adjustment take place if more than one basic net is used: implicit weight setting of the outer section of the network (photoreceptors - horizontal - bipolar- ganglion), and synaptic weight adjustment of the inner section (bipolar - amacrine - ganglion).

The relationship between input and output of the basic network is that the basic network outputs the centroid of the inputs. If only one of the photoreceptor is ON then only the ganglion cell directly beneath will be turned ON.

3.3.1 Bipolar Cells

Each bipolar cell has one excitatory and one inhibitory input connection. To ensure the above functionality, the weights of the bipolar connections should be the same in ratio and in opposite sign to each other so that if the horizontal cell is active the bipolar cell will stay inactive. The horizontal cell also has two input connections. It receives inputs from both photoreceptors and is only active if both photoreceptors are turned ON.

Its connection carry weights such that if the weighted sum exceed the preset threshold the horizontal cell is ON.

Once the connection weights of the basic net are properly established (e.g Figure 3.4), the implicit regular weight setting can be used to define large network by replicating the structure. When expanding from the basic net structure a layer of amacrine cells is needed to maintain the centre-ON operation.

3.3.2 Amacrine Cells

Each amacrine cell receives many inputs from the bipolar cells layer and has excitatory as well as inhibitory output connections to the ganglion cell layer. The input and output weights of the amacrine layer are adjustable. The amacrine cell layer can be thought of as an unsupervised layer. When signals are propagated through the amacrine layer the amacrine neurons compete with each other. The one with the weights closest to the amacrine layer's input wins the competition. The connection weight leading to the winning amacrine neuron can then be adjusted by the following equation (3.3) : -

$$W_{BA}(t+1) = W_{BA}(t) + \zeta \, (B - W_{BA}(t)) A \tag{3.3}$$

where B and A denote the activation values of the input bipolar cell and the winning amacrine neuron respectively and ζ is the rate of weight change between 0 and 1.

Both input and output of the amacrine layer are fully connected and the weights of these connections are initially set to small random values thus any one of the amacrine neuron can win the input.

The number of input patterns that each amacrine cell can associate is also controllable. The activation of the amacrine layer is controlled by an additional Selection Criteria. If an amacrine neuron has been winning more input than it is allowed its winner status will be disqualified. When this happen the next neuron satisfying the suppression condition will take over the association.

3.3.3 Output Of Amacrine Cells

The output of the amacrine neuron can be expressed by equation (3.4) : -

$$\textit{Amacrine Cell Output} = \begin{cases} 1 & \textit{if Winner}, \ \frac{k}{c} < 1 \\ 0 & \textit{Otherwise} \end{cases} \qquad (3.4)$$

$$\textit{Winner} = \textit{Minimum} \left[\sum (B - W_{BA})^2 \right] \qquad (3.5)$$

where c denote the number of pattern the neuron is allowed to be associated with and k is the number of times the neuron has won; i.e. as k increases toward c suppression increases, when k=c suppression is at maximum and association of this neuron is saturated.

At the output side of the amacrine layer, weights between the winning amacrine neuron and the neurons in the ganglion layer are adjusted so as to provide the desired association pattern. These connection weights are adjusted according to equation (3.6) : -

$$W_{AG}(t+1) = W_{AG} + \zeta \ (G - W_{AG}(t)) A \qquad (3.6)$$

where G is the desired ganglion cell output, connections are strengthened between active amacrine and ganglion cells only, otherwise weights will be suppressed.

3.3.4 Adaptation Algorithm

The adaptation algorithm consists of the following steps :-

1) Initialize the network by defining the regular weights of the basic net.
2) Expand from the basic net to the required input and output neuron organisation.
3) Set up the amacrine layer with size according to: (a) the number of patterns each amacrine neuron is allowed to associate, and (b) the number of patterns to be learned, i.e. Amacrine Layer Size=(integer)[(b)/(a)]+1.
4) Randomize the input and output weight connections of the amacrine layer to small random values.
5) Apply training data to the input of the network and activate the ganglion layer according to the desired output pattern.
6) Calculate the outputs of photoreceptors, horizontal cells, bipolar cells and amacrine cells layers.
7) Locate the next amacrine winner.
8) Run through the Suppression process to the winning amacrine neuron; if the winner satisfies the activation conditions go to step (9) otherwise repeat (7) and (8).
9) Adjust the input and output weights connections of the amacrine layer by equation (3.3) and (3.6) respectively.

10) Repeat from step (5) for another training pattern.

11) Repeat from step (5) for next cycle.

3.3.5 Example

As an example, Figure 3.6 is a pictorial pattern of a velocity-time trajectory profile plotted on an equally spaced area where the velocity and time quantities are represented by the horizontal and vertical grids. This graph is divided into equal rectangular stripes and each stripe is composed of two successive velocities which will be the input of the NI.

The velocities at t_0 and t_1 are graphically merged (superimposing the patterns at t_0 and t_1) and fed to the input of the integrator, see Figure 3.6, where each grid value is clamped to on photoreceptor only. Upon receiving external input, the neurons then perform computation and produce output according to their connections weights and neuron transfer characteristics.

At the output, each ganglion cell is clamped to a grid point on the output plane, separated by 1/2 unit in the vertical axis, see Figure 3.6. The new active neuron will

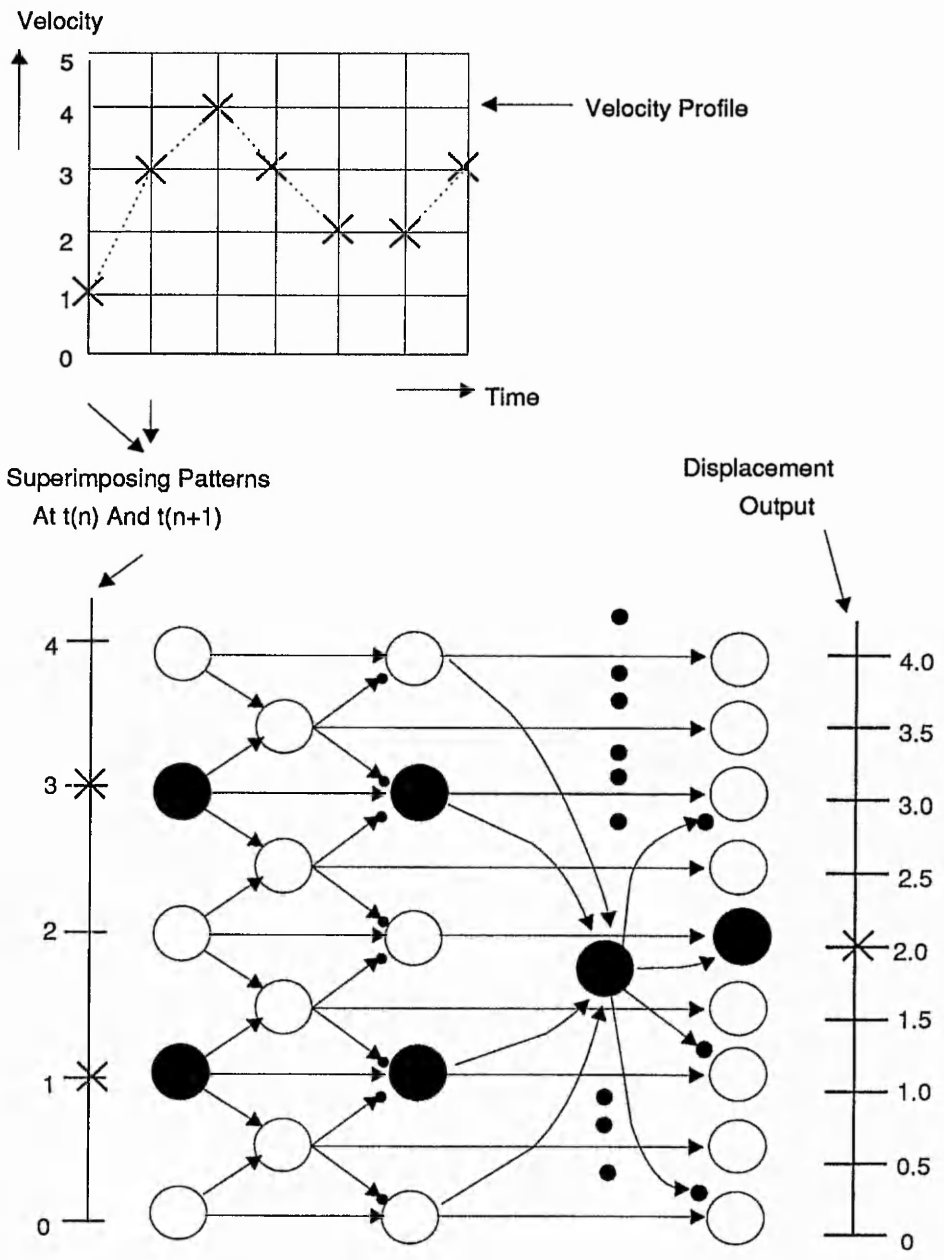


Figure 3.6 The Pictorial Integration Process

produce a "1" at its clamped position which represents the area (the integral) under the curve between t_0 and t_1.

By applying the above process to each pair of successive columns of the input plane, a set of 'local' integrals can be achieved; the global output can be obtained by accumulating the successive 'local' output/integrals.

3.4 Mapping Of Neural Nets On von Neumann Computer Systems

It is evident that current technology is not flexible and mature enough to allow the implementation of neural net directly in hardware. For this reason, digital computer simulations remain the primary method of implementing, experimenting with and validating neural net models.

One of the most discouraging aspects of simulating neural networks is that there is no programming language that generally supports this kind of application. A wide range of neural network models can be adopted when attempting to solve a particular problem. Using traditional methods to assemble networks by manually writing a collection of software routines is somewhat cumbersome as a change in the architecture of a network usually necessitates reprogramming to reflect the new architecture and its simulation actions. This reprogramming is tedious and time-consuming.

One possible way of avoiding this would be to adopt a common method or language which can be used to support and provide easy means of specifying any network configuration composed of many different neuron types and interconnections. However, designing a new language from scratch requires substantial effort. A better approach is to extend a widely used language to provide some common tools to simplify the assembly of networks thus reducing the overheads of developing a new neural network simulation.

Furthermore, simulating a neural network is extremely computationally intensive as a high degree of inherent parallelism is a key feature of efficient neural networks. Large amounts of computer resources are often required to teach a moderate-sized network using the standard sequential and pipeline computer structures.

During the last few years special purpose hardware using VLSI technology, so called neuro-hardware, to accelerate the processing of neurons and synapses, has been proposed

[Graff,1986] [Lambe,1986] [Sage,1986]. However, these special purpose hardware designs are specifically built for certain neural network model thus are deficient in flexibility for exploring different pattern of connectivities, neuron models and learning algorithms.

One natural way to overcome the speed limitation of neural network simulation is to explore the parallel processing approach. There have been several research efforts, [Deprit,1989] [Hicklin,1988] [Pomerle,1988], suggesting processing techniques for commercially available parallel machines. In the following sections, a general framework for mapping neural network models onto conventional and multiprocessor computer systems is described.

3.4.1 Framework For Programming Neural Nets

The underlying design primitive for programming neural networks can be described by three basic components:-

1) A set of simple processing elements which can be defined by a set of transfer functions representing the processing of inputs to a single output operation.
2) A connectivity pattern which defines the flow of data

between processing elements.

3) A set of learning equations which defines the adaption dynamics of the network.

From the computational points of view, the modelling of neurons and learning functions in software are relatively simple when compared to the associated connectivity pattern.

Connectivity patterns can be classified into two categories, regular and irregular. With a regular connectivity pattern, connections between neurons in different layers effectively have the same form, that is a network is formed by layers of neurons with each neuron connects to every neuron in the next layer. The representation of this type of network in software is very straight forward, (e.g. 2-dimensional array of connection).

Irregular connectivity patterns, however, are more complex and requires greater attention to the data structures and software arrangement. This suggests the need of some common tools to support the definition of arbitrary network topologies.

With standard computer hardware (i.e. serial uniprocessor systems) these demands can be satisfied with a framework of data structures and library functions. The framework of data structures provide a representation of the physical structure of a network which are manipulated by the library functions (descriptions of the library functions are given in chapter 4).

3.4.2 Data Structures

The representation of networks in software may be based on three fundamental data structures: NEURON, SYNAPSE, and GROUP.

NEURON and SYNAPSE types correspond to processing elements (neuron) and weighted connection links (synapse) between them , GROUP type defines a group of neurons. A complete network may consist of many different kinds of neurons, grouped according to their function in the net and the connections between them.

3.4.2.1 Neuron Type

In essence, a Neuron, see Figure 3.7, is defined by five status parameters, and a synapse pointer namely Neuron Net Input Value (Net_Val), Activation Value (Act_Val), Output Value (Out_Val), Error Value (Error_Val) which have special reserved meanings, and a free status parameter (Reg) which can be used when necessary (e.g. the desired target value of the neuron). Each neuron in a network may be given an identity. Depending on the network model, the neuron identity may be ignored, however, it is sometime useful in situations where specific coordination of neurons or synapse connections are desired and plays an important part in supporting the debugging processes. Synapse connections between neurons are assigned to the receiving neuron. The connection components (synapse - see below) leading into each receiving neuron are held in a link list which in turn is pointed to by the synapse Pointer (Syn_ptr).

3.4.2.2 Synapse Type

A Synapse has three status parameters and one neuron type pointer, see Figure 3.8, of which the Weight is reserved to hold the synapse connection strength, the other two being

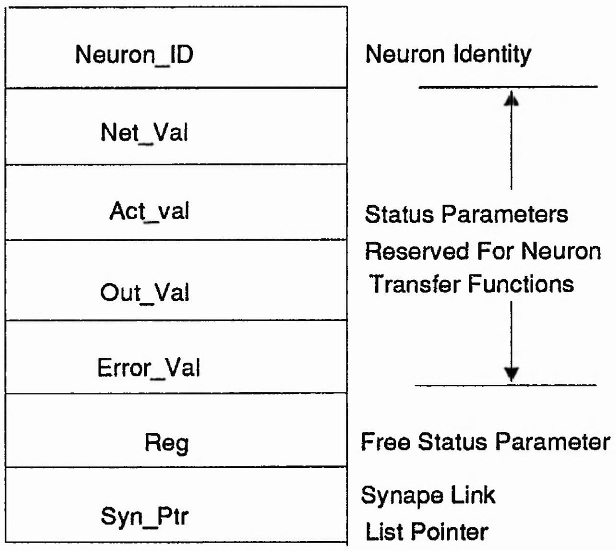


Figure 3.7 NEURON Data Structure

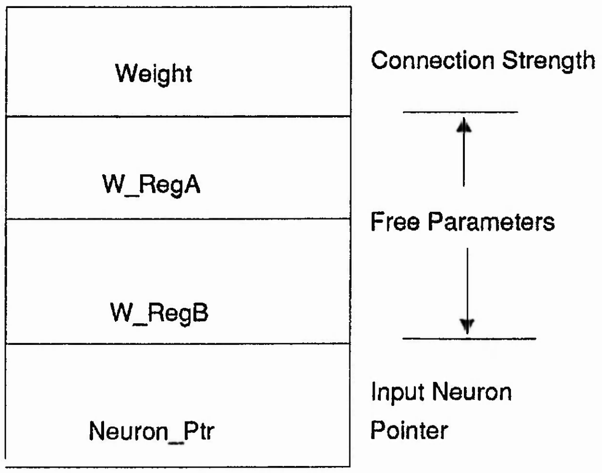


Figure 3.8 SYNAPSE Data Structure

free parameters, similarly as in the case of neurons, they may be used when necessary. Each Synapse also points to the neuron delivering the signal by (Neuron_Ptr).

3.4.2.3 Group Type

The Group concept is used to define groups of neurons sharing the same input to output transfer characteristics and which have to be connected to each other or which may be updated in parallel. In essence, a Group has a Group Identity (Grpid), see Figure 3.9, three function pointers, namely neuron Activation Function Pointer (ActFcn_Ptr), Output Function Pointer (OutFcn_Ptr) and Net Input Function Pointer (NetFcn_Ptr). These three function pointers are reserved for the transfer functions of neurons pointed (i.e. Grouped) to by the Neuron List Pointer (NeuronListPtr).

Each simulation model consists of a list of groups, Figure 3.10, which contains all of the existing neurons. Each Group points to a list of neurons and each neuron points to a list of synapses and each synapse points to a neuron record. It is by this mechanism arbitrary network configuration can be dynamically created and altered.

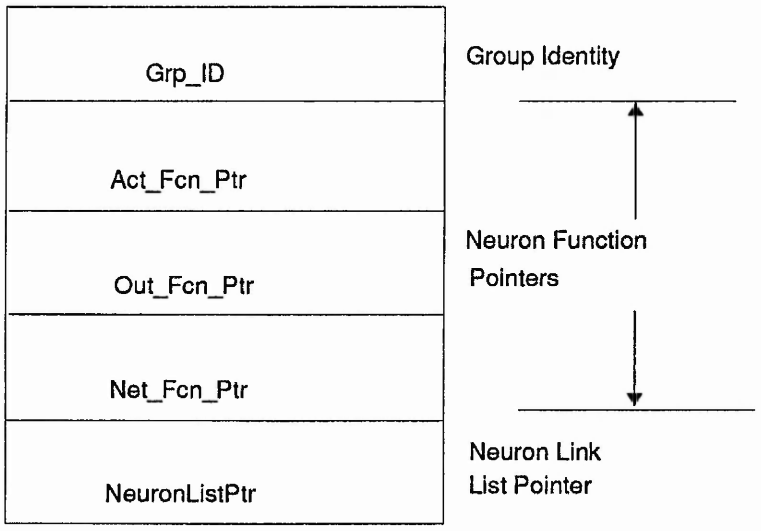


Figure 3.9 GROUP Data Structure

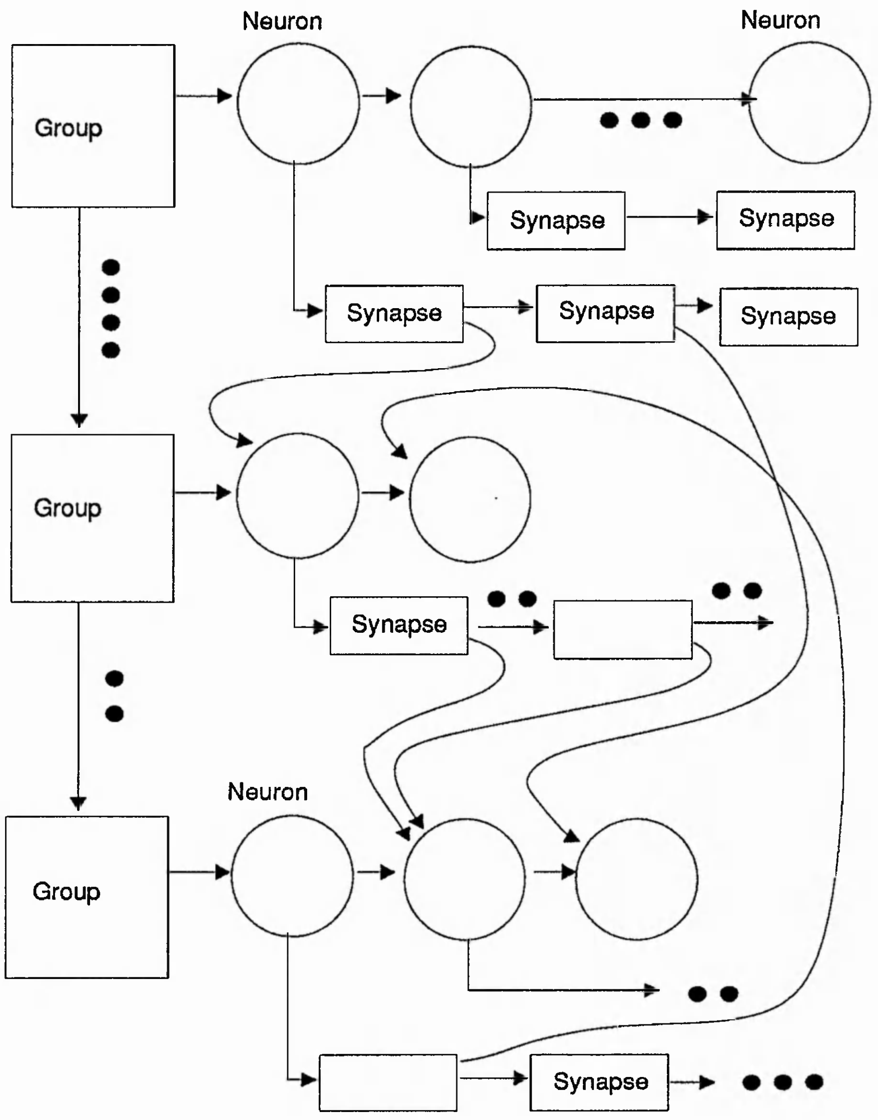


Figure 3.10 Network Representation in Software

Based on this grouping concept, a network with highly regular connectivity pattern can easily be defined by grouping neurons by layer and creating connection links simply by linking the appropriate groups, see Figure 3.11(a), any unwanted connection may be defined by 'ZERO' weight setting. For irregular connectivity patterns, such as the one shown in Figure 3.11(b), neurons, alternatively, may be grouped together according to their connection destinations. Neurons in a layer connected to the same neuron in a successive layer may be grouped together. With this second approach, no memory is wasted on any unwanted connection as they simply do not exist. In effect, neurons are grouped by sub-dividing the layer of neurons. Both approaches are supported by the same data structures, and are explicitly adaptable.

3.5 Neural Nets Simulation Using Multiprocessor Systems

The virtue of the simulation framework described in the last section is that it frees the user/programmer from the need to write code to assemble neural networks and set up simulation actions. The same mechanism can also be used to map any network onto multiprocessor systems, without major alterations and program complication, to achieve processing

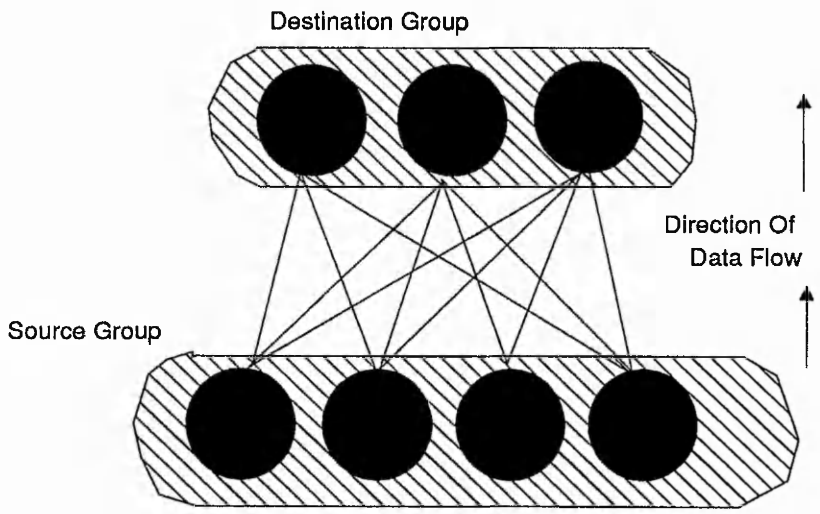


Figure 3.11(a) Group By Layer

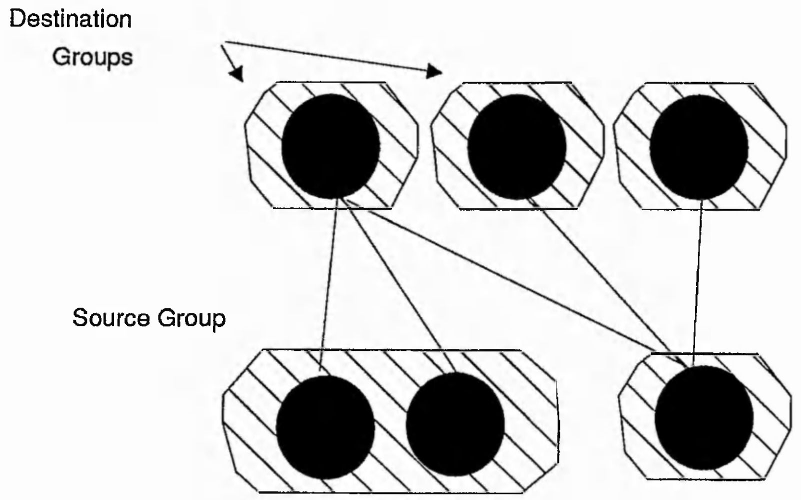


Figure 3.11(b) Group By Sub-dividing Layer

speeds which cannot be met by single processors.

The descriptions below concentrate on a category of multiprocessor systems proposed by [Hammes,1989]. The target multiprocessor system is a distributed memory type. The reason behind this choice is that a distributed memory multiprocessor system can be viewed as having general purpose parallel architecture with the potential of being a versatile, multi-purpose machine instead of a highly specialized ones.

3.5.1 Architecture Of A Target System

T target system falls into the MIMD (multiple instruction, multiple data) category in which each processor node executes its own instruction stream on its own data. Each processor node consists of a processor, local memory and an inter-processor communication interface chip, see Figure 3.12. Each node stores its program and data in its private memory so that it is in itself a complete computer. Processor nodes are connected to communicated with a global bus and the mode of communication chosen for the system is that of Associative Addressing.

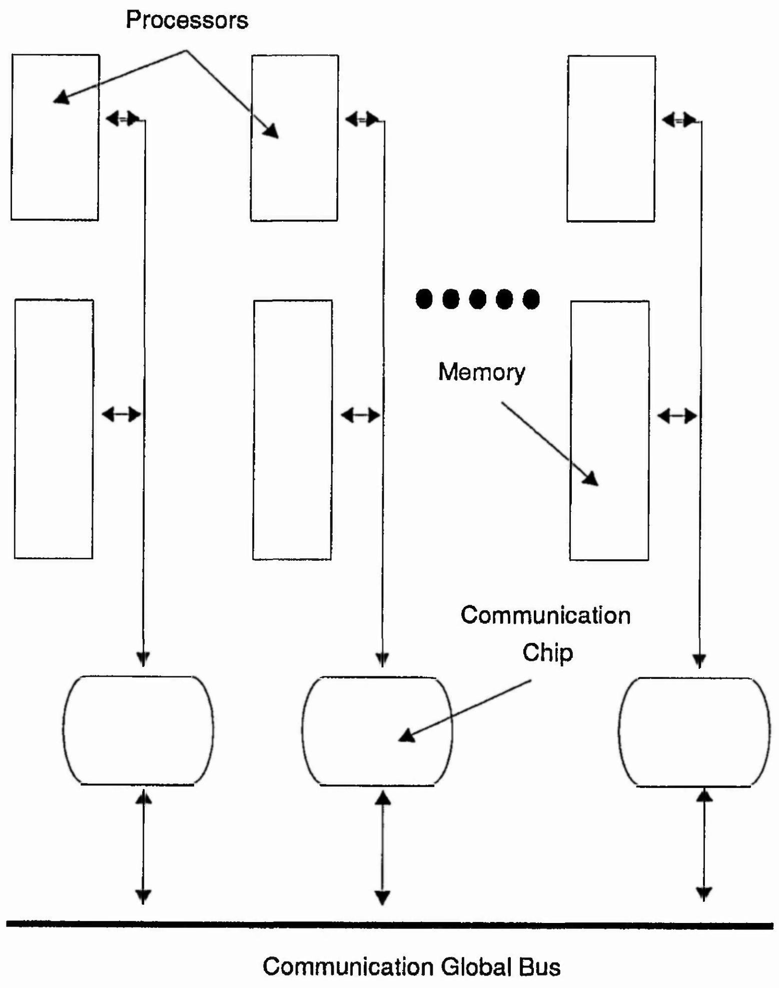


Figure 3.12 A Target Multiprocessor System

In Associative Addressing, each piece of data is placed onto the global bus with an address to indicate where the data is coming from. Each node monitors the global bus constantly, any node recognizing the address will copy the data simultaneously. The system accomplishes parallelism by this means of communication technique.

Processor nodes are equipped with a special interface chip to facilitate the associative addressing communication mechanism. The interface unit consists of input and output (data and address) register pairs. The address of each register is programmable so any one of these inputs can be programmable to receive data from any of the other nodes within the system. In addition, the global bus as well as processor nodes are all controlled by a master node which performs the management and bus allocation operations. A node requesting the bus will notify the allocation circuit, when the bus is free bus control is handed over depending on the location of the allocation signals. There are two levels of allocation circuit, thus two levels of allocation signal, see Figure 3.13. The complete multiprocessor system consists of 64 nodes. Processor nodes are grouped into clusters of eight in which each cluster has an allocation circuit linked to the higher level allocation signal.

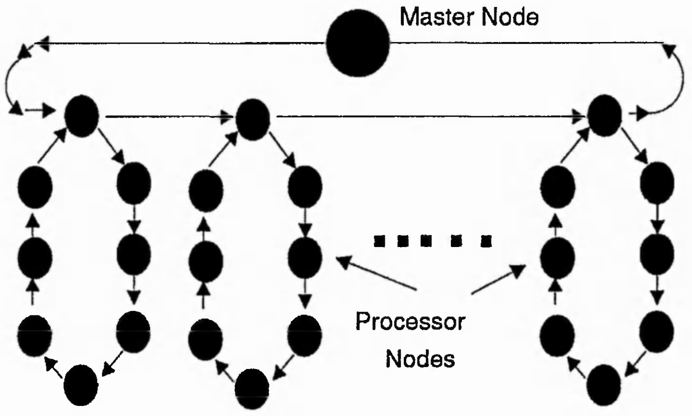


Figure 3.13

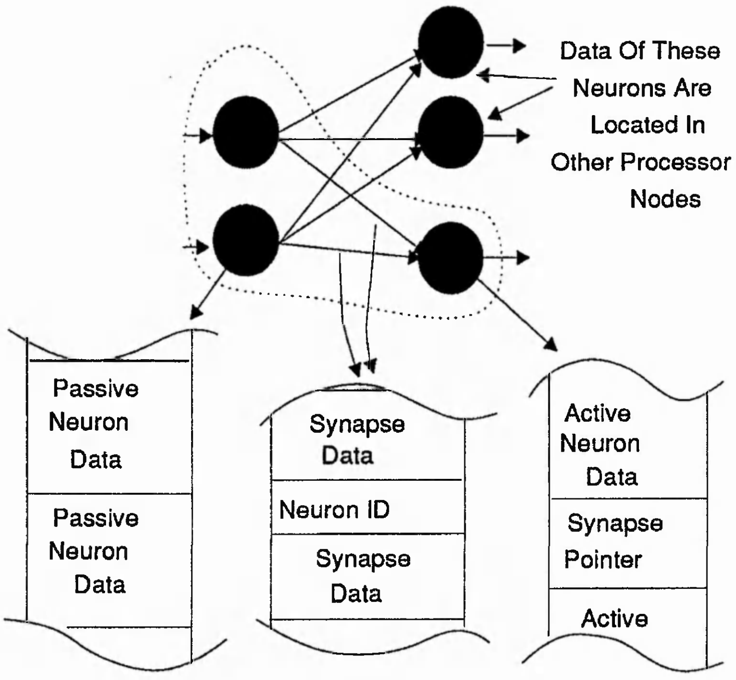


Figure 3.14 Data Distribution In Each Node

3.5.2 Simulation Primitives For Multiprocessor Systems

The most important factors when considering the construction of neural network simulation on multiprocessor systems are: the optimal distribution or allocation of neurons into processor nodes, and the communication between neurons allocated to different processor nodes.

Inspection of the most commonly used neural network architectures and the way they perform computations and learning reveals several parallel features. Networks are usually made up of neurons which are grouped in layer(s). Neurons belonging to the same layer can compute in parallel their outputs in the normal forward processing phase (i.e. propagation of signals from input to output). While in the backward learning phase they compute in parallel the weight modification process. However, the output of a neuron or the computation of new weights usually depends on the value computed by many other neurons. Without careful design, an implementation of such algorithms on parallel distributed memory systems can easily spend the majority of its running time in communication, i.e. sending data where they are needed rather than performing actual computations.

3.5.3 A Distribution Technique

Neurons in the same layer can be partitioned onto different processors, see Figure 3.14. As the amount of memory required by a neuron is usually small when compared to that used by the synapses leading to a neuron, neuron information exists both in a "Full" and "Redundant" fashions on each processor node, as described below.

Each processor node carries data for neurons created locally, the input weights associated with these neurons and the output values of neurons created in other processors and connected through these weights. Neurons assigned to the specific nodes are called active neurons and are created only on it's "home" processor node so "full" neuron information is stored. This also applies to synapse information leading to the active neurons. Neurons allocated to other nodes that are linked by the synapses are called passive neurons and only their output status values are stored.

The memory in each processor node is divided into four sections: support code area including neuron's and learning functions, active neurons information section, synapses and passive neurons sections. This can be best illustrated with

Figure 3.15. With this partition strategy active neurons partitioned on different processors can be updated in parallel without the need to communicate frequently.

Though this partition scheme results in the duplication of neuron values, it avoids complex communication requirements; as the amount of memory required by neurons is small when compare to that used by weights thus communication and storage requirements are minimal. Communication is only necessary to ensure the availability of updated neuron values. This is done by transferring the desired neuron parameters from the active neuron area into the respective passive neuron area whenever an update of a layer of neurons is made, see Figure 3.15 for illustration.

3.5.4 Data structures For Multiprocessor Systems

Data structures for the multiprocessor system simulation are similar to those described in section 3.4.2 (for uni-processor systems). The Neuron Pointer field of the Synapse is replaced by a Neuron Identity so that each Synapse contains a weight, two free status parameters and a neuron identity to indicate the neuron that the synapse is connected to. Each neuron has a uni-identity, whereby

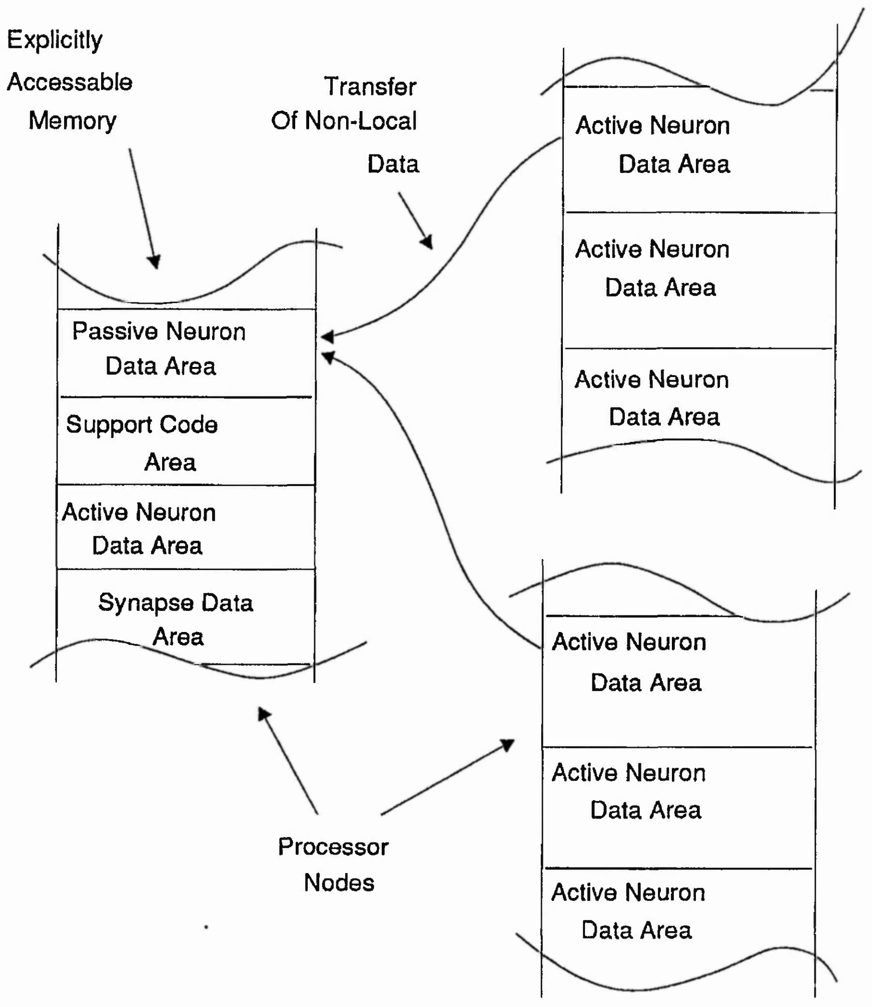


Figure 3.15 Deviation Of Memory And Transfer Of Non Local Data

following each complete update cycle the output value of each active neuron is transmitted together with their identity. Receiving processor nodes will place these data in their passive neuron areas so that they are ready for the next cycle. In each update processing, the passive neuron memory area in each node will be searched and matched to the associated synapse weight so that active neuron information can be updated.

CHAPTER 4 SOFTWARE IMPLEMENTATION

CHAPTER 4 SOFTWARE IMPLEMENTATION

Computer simulations need to be employed to study the essential differences in performance of existing neural nets and for experimenting with and validating the new retinal model for the dynamical system integrator.

The simulation framework proposed in chapter 3 will be refereed to here as toolbox. It is developed in this chapter to provide the foundation for the implementation of simulation programs to handle the process of generating arbitrary network definitions thus simplifying the tedious task of putting synthesising neural networks. The simulation toolbox consists of a set of simulation specific data structures, detailed in section 3.4, to provide a representation of the physical structure of a network and a set of procedural functions for the manipulation of the network data structures.

The simulation toolbox and all simulation programs are implemented in the 'C' language. The design is based on the use of records, structures, lists, mapping functions and dynamic memory operations which are well supported by the C language.

The implementations of the library functions are described.

In the absence of fully operational target hardware, the target multiprocessor system has been emulated by software. The implementation of this emulation system is also described.

4.0 Components Of The Simulation Toolbox

Building a neural network simulation program involves two main tasks: the **Construction** of a network including initialization of network activities and inter-connection weights, and the **Execution** of network functions including normal and learning processes.

The construction of a network in software in each simulation is based on three fundamental structures, the NEURON, the SYNAPSE and the GROUP, see section 3.4. These data structures are used to hold network information i.e. topology and knowledge (weights).

The data structures are manipulated by a set of functions in which the construction of a network and the simulation actions can be defined by function calls to the toolbox. This makes it possible to specify neural networks without having to deal with the actual representation of the network architecture. This approach provides flexibility as the functionality of the simulation toolbox can be improved incrementally by adding new functions to the procedural interface to gradually eliminate functionality limitations of artificial neurons.

A simulation can be created by writing a program in which the procedural function calls refer to manipulations which perform the functions in the toolbox. This process is summarized in Figure 4.1.

4.1 Procedural Interface

The fabrication of a network and simulation actions are carried out by a collection of functions. These function are classified into three categories:-

I) Construction including Initialization,

II) Execution including normal processing and learning,

III) Peripheral functions, including debugging and data dumping routines.

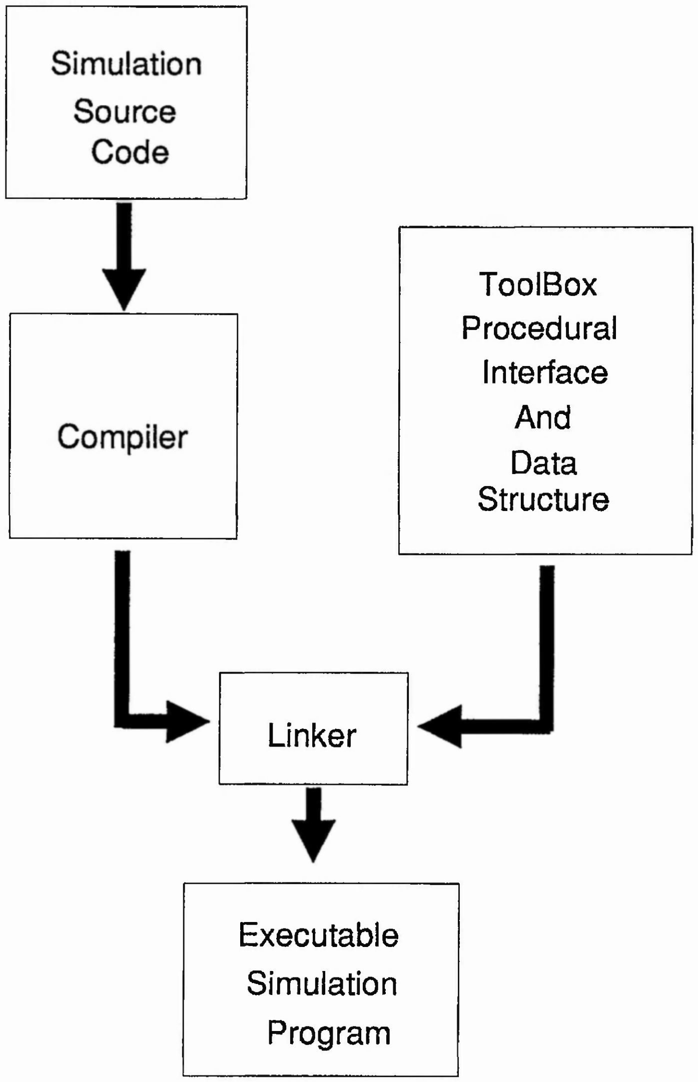

Simulation Source Code
Compiler
ToolBox Procedural Interface And Data Structure
Linker
Executable Simulation Program

Figure 4.1 Generation Of Simulation Program

4.1.1 Network Construction Functions

There are six standard functions in this category for the construction and initialization of arbitrary network architecture. **CreateGroup(NetFcnPtr, ActFcnPtr, OutFcnPtr, Nofneuron, GroupID)** is used to define a group of neurons having the same common neuron characteristic specified by three function pointer variables. 'NetFcnPtr' is the neuron input function pointer, 'ActFcnPtr' the neuron activation function pointer and 'OutFcnPtr' the neuron output function pointer.

A group may also be given a name/identity via 'GroupID' and the number of neurons created by this function will be according to the variable 'Nofneuron'.

Additional neurons can be added to an existing group by using **CreateUniNeuron(GroupID, NeuronID)**. This function takes in two variables: group identity 'GroupID' and the neuron identity 'NeuronId'. When this function is called, 'GroupId' is used for locating the group that the new neuron belongs to. Once the specified group is located, a new NEURON structure with the specified identity will be added to the end of the neuron link list.

Similarly, **LinkGroup()** and **CreateUniLink()** are used for creating multiple or single connections. **LinkGroup(SourceGrp, TargetGrp)** creates total connection between neurons belonging to the two specified groups while **CreatUniLink(SourceNeuronID, TargetNeuronID)** creates a Single connection between two specified neurons.

By the same notation, **InitWeight(ValueFcnPtr, GroupID)** initializes all the synapse connections belonging to neurons in the specified group with certain values. The connections are initialized according to the value of the generation function defined by the function pointer variable 'ValueFcnPtr'. **InitUniweight(ValueFcnPtr, SourceNeuronID, TargetNeuronID)** initializes a single connection between two neurons.

4.1.2 Execution Functions

There are three standard functions in this category for normal network processing and learning. **ActivateGroup(GroupID)** takes in the group identity. The output of each neuron belonging to 'GroupID' will be computed by performing operations to the connection weight and input signal pairs. This is performed according to the predefined neuron transfer functions specified during group/neuron creation by **CreateGroup()**.

CalculateError() and **AdjustWeight()** are both used during learning process. **CalculateError(GroupID, ErrorFcnPtr)** calculates the errors generated by each neuron within the group specified by 'GroupID' according to the given error calculation function 'ErrorFcnPtr'. **AdjustWeight(GroupID, WgtFcnPtr, LearningRate)** is used to adjust the connection weights leading to each neuron defined by the group identity according to the specified weight changing function 'WgtFcnPtr'.

4.1.3 Peripheral Functions

In addition to network Construction and Execution, a set of peripheral routines is also provided to perform searching and displaying of groups, neurons, connection weights, external input or output of whole network and for loading and saving of network information.

ShowGroup(), **ShowNeuron()** and **ShowWeight()** are network probing routines which display the status of neurons, the associated connection weights and input signal pairs. A call to the function **ShowGroup(GroupID)** will cause a screen dump of all the information contain in the specified group. **ShowNeuron(GroupID, NeuronID)** displays individual neuron and its associated weight values. Similarly **ShowWeight(SourceNeuronID, TargetNeuronID)** displays the weight of a single connection.

SearchNeuron(NeuronID) and SearchLink(SourceNeuronID, TargetNeuronID) are used for locating specific neuron and connection, both routines return the memory address of neuron or connection. **LoadNet(Filename) and SaveNet(Filename)** are used for loading and saving network to file so that retrieval or off-line inspection is made possible. The source code listing of the above procedures, including data structures, are enclosed in appendix (B.1) and (B.2).

4.2 Construction Of Simulation

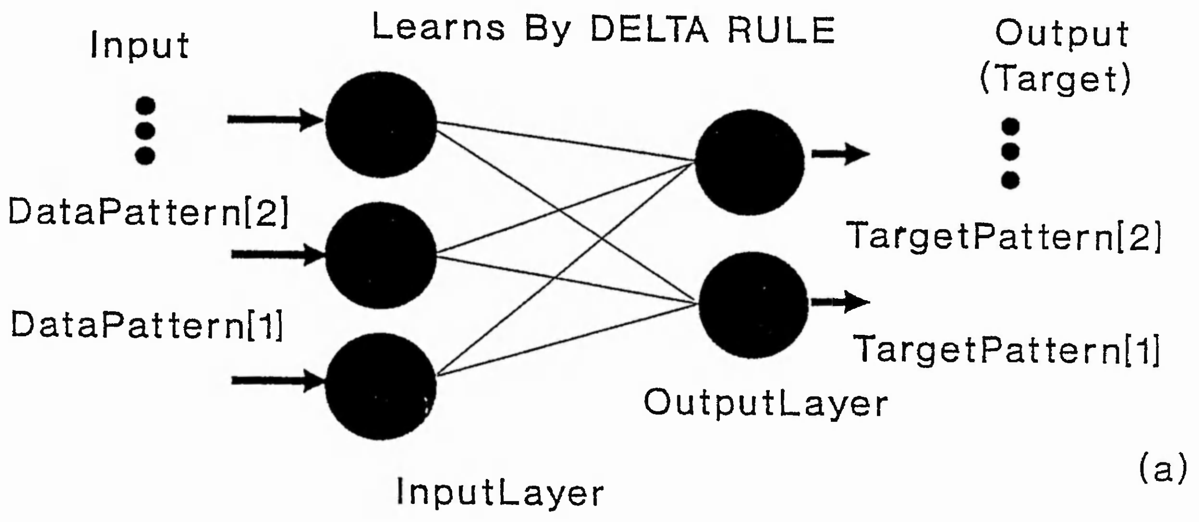
The data structures and the procedural interface are implemented in standard ANSI-C, transporting from machine to machine is not a problem as long as the C compiler supports standard ANSI specifications. The size of the support code has been deliberately kept as small as possible. The total size of the toolbox is less than 40 kbytes including all the neuron and learning function for simulating the various net models, i.e. Delta Rule, Backpropagation, Hopefield, Competitive, Kohonen, Counterpropagation, and the new retinal model. There are two alternative modes of operations for constructing a simulation: static and interactive.

In static mode, a simulation can be constructed by writing a C program which consists of a sequence of function calls from the toolbox. Any neuron transfer characteristic and learning functions can be added simply by writing routines and recompiling with the user program, see Figure 4.1. Static mode is more flexible and uses less memory than interactive mode but recompilation is necessary for each simulation. It is flexible as any new neuron and learning function can be added and removed.

Interactive mode is achieved by running a simulator program called **NNSim.exe**. This program, in addition to the standard toolbox functions, is also equipped with a collection of predefined neuron and learning functions. In interactive mode, a network can be dynamically created or altered on-line. Network construction and simulation is performed by means of a menu-driven user interface in which function calls are performed by selecting the appropriate item in a pop-up menu. In this mode of operation, simulation runs can be suspensed at any time so that network characteristics or topology can be changed and simulation resumed without recompilation. The major disadvantage of this mode of operation is that the size of the network that can be simulated is limited as all the necessary functions including unused routines are carried onboard. For this reason, static mode is more favourable when memory is a limitation; for example only 640 kbytes is available with an IBM compatible PC running in standard MSDOS environment (real mode). The structure chart of NNSim.exe is enclosed in appendix (C).

4.2.1 A Static Mode Simulation Example

The example code segments shown in Figure 4.2(b) for a simple Perceptron network simulation should give an idea of the programming style supported by the simulation toolbox. The network shown in Figure 4.2(a), consists of an input and output layers which the network learns using the Delta Rule, as has been described in chapter 1. In Figure 4.2(b), the first three lines are used for network construction including initialization and the rest are for teaching the network. **SetDataPattern(GroupID, DataFileName, DataType)**



```
/* Network Construction */

CreateGroup( Ramp,Ramp,Ramp,3,"Inputlayer");
CreateGroup(Soma,Binary,Ramp,2,"Outputlayer");
LinkGroup("Inputlayer","Outputlayer");
InitWeight(RamdomFcn,"Outputlayer");

/* Network Training Process */

for (i=0; i<nofcycle; i++)
 for ( j=0; j<nofpattern; j++)
 {
  SetDataPattern("Inputlayer",Datafile[j],INTYPE);
  SetDataPattern("Outputlayer",Targetfile[j],TARGETTYPE);
  ActivateGroup("Inputlayer");
  ActivateGroup("Outputlayer");
  CalculateError("Outputlayer",DeltaErrorFon);
  AdjustWeight("Outputlayer",DeltalearnFon,learnRate);
 }
```

<u>Note:-</u>

Ramp, Soma, Binary, DeltaErrorFon, DeltaLearnFon are neuron transfer functions and Delta Rule learning functions respectively.

(b)

Figure 4.2 Example Simulation Code Segment

is one of the peripheral functions in the toolbox. It loads data into the neurons within 'GroupID' from a data file specified by 'DataFileName'. The variable 'DataType' indicates where data should be loaded to; INTYPE indicates external network input, and TARGETTYPE indicates desired network output. Both INTYPE and TARGETTYPE can be found in the toolbox header file toolbox.h.

4.3 Neural Net Simulation Using Multiprocessor Systems

A prototype of the target multiprocessor system with eight processor nodes on board is available. However, as a prototype, the present system has several weaknesses: lack of basic operating system, limited node memory to 4 kbytes, and lack of language support. Due to these problems, the prototype can not provide a suitable environment for conducting experiments with the new concepts. In order to carry out performance evaluation the target hardware has been realized by software.

4.3.1 Emulation Of The Multiprocessor System

Processor nodes are emulated by blocks of memory. Each memory block is divided into two areas. One for network storage containing active neurons, weights and passive neurons data; and one for storing processor node data: containing input/output interface registers and global bus control and communication registers, see Figure 4.3. In real world situations each node would contain a set of support functions handling the creation, updating, and transferring of neurons or connections; however, for the purpose of simulation only one set is needed as the actual processing is still sequential.

As illustrated in Figure 4.4, memory blocks are chained together (as a Linked List). Working alongside with the memory blocks is a Data-Buffer register representing the global bus of the target system. Individual 'processor nodes' can read the contents of this register at any time, but loading of data from node to Data-Buffer depends on the individual local Bus-Request register; its boolean value indicates whether data is ready to be transmitted or not.

In order to maintain consistency of neuron values, all processors must finish their neuron updating process before communication can commence. After each complete neuron update (completing all neuron calculations within a layer), the Bus-Request register of each processor is

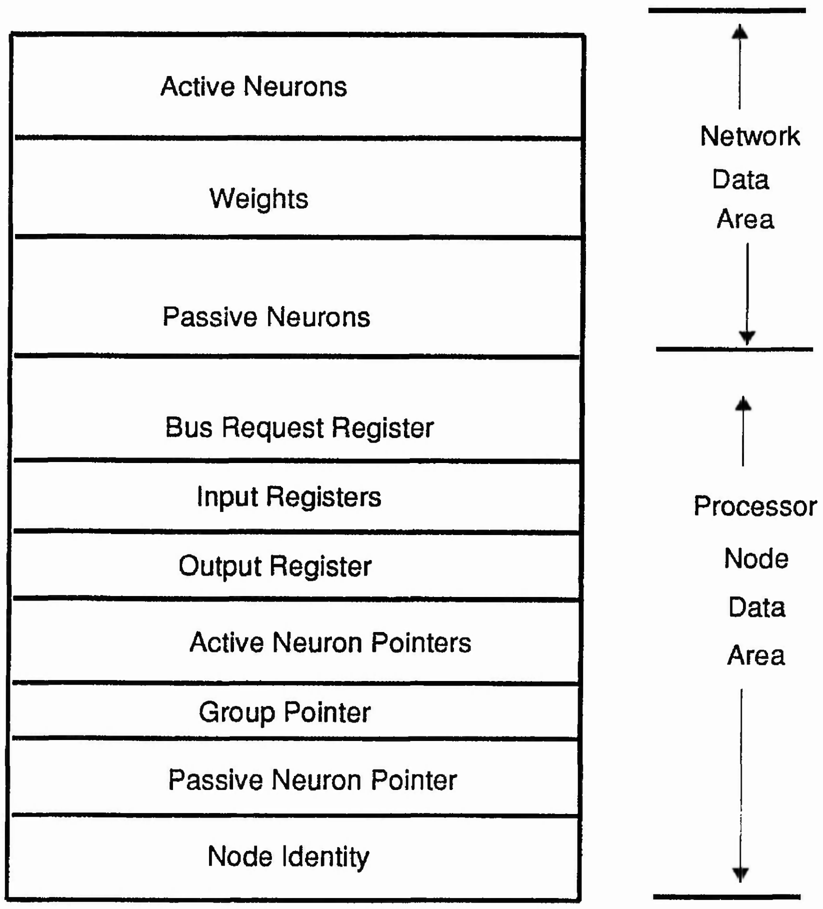


Figure 4.3 Memory Contents Of A Processor Node

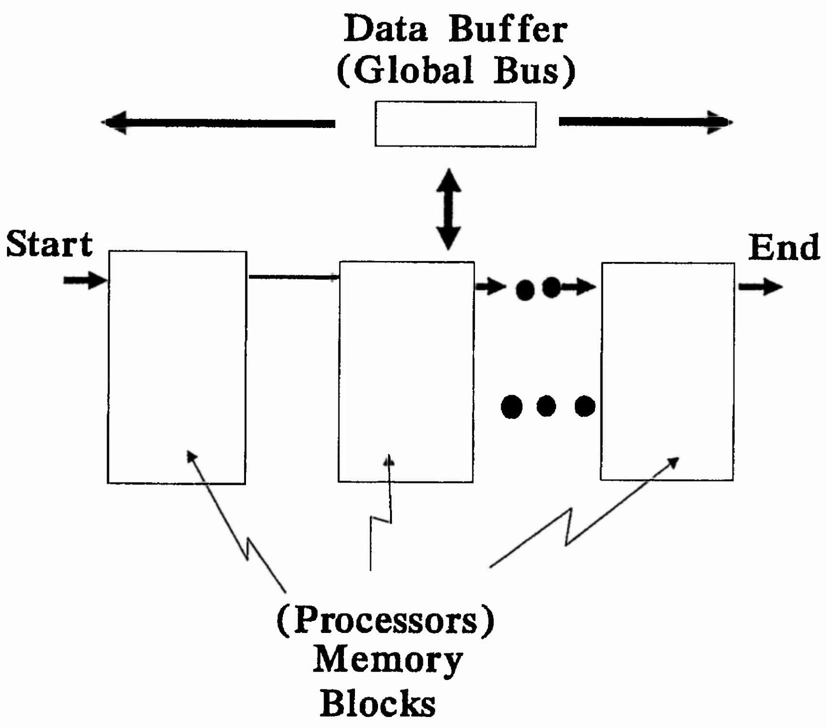


Figure 4.4 Emulation Of Multiprocessor System

compared one by one along the Linked List. If the test is positive, data will be transmitted and carried to other nodes by the Data-Buffer register.

The Data-Buffer carries two pieces of information: neuron value and neuron identity. The neuron identity in the Data-Buffer is compared by the passive neuron identity in each processor in turns. If a match is found the neuron value in the Data-Buffer will be copied to the corresponding memory location. This comparison process continues with the next processor block until all the processors have been visited. This 'communication' process will continue until all the Bus-Request registers are FALSE. The Bus-Request register will be set to FALSE if (1) the active neuron updating process has not been completed, or, (2) all the updated neuron values have been transmitted.

<u>4.3.2 Processor Node Emulation</u>

To enable the implementation of the distribution and communication concepts described in section 3.5, a set of mapping functions is provided. Similar to the case of uniprocessor systems these functions can be used by user programs not only to define arbitrary network architectures, neuron models and learning functions but also to automate the distribution and communication methods. The majority of support functions are very similar to those of uniprocessor machine, the prototype of these standard functions are shown in Table 4.1.

A new structure is used for emulating processor nodes, see Figure 4.3. It consists of node identity (NodeID), active neuron pointer (ActNeuronPtr), passive neuron pointer (PasNeuronPtr) and a Bus_Request register.

User programs use **EmulateNode(NodeID)** to initialize a processor node. For each processor node, the active neurons, the associated synapses and passive neurons are created by calling **CreateNeuron(NodeID, NeuronID, NeuronType)** and **CreateLink(NodeID, SourceNeuronID, TargetNeuronID)**. Depending on the emulated node number each processor node has the same number of input address register (containing the identity of other nodes) so for each transmition of data every node in the system makes a copy of it. Once data is loaded **SearchPasive(NodeID, NeuronID)** routine is used to check if the data belongs to any of the passive neurons defined within the node and places the data accordingly if identification is positive.

Neuron information is transported from node to node using **SendNeuron(NodeID, NeuronID, NeuronType)**. Each time this function is called the specified neuron will be located and its identity and output value loaded into the output registers. When the Data-Buffer is ready the content of the output register is transferred to the Data-Buffer and copied by other nodes.

The main simulation routine acts as the master control and handles the monitoring and control of communications by probing the Data-Buffer register and moving it from node to node, Figure 4.4. Any node needing the bus will enable the local Bus-Request line, and on the arrival of data it gains control of the register. To ensure proper operation, passive neuron transfers occur only after each complete cycle of all neuron updates.

The operation of the multiprocessor emulation program is illustrated by the flow chart in Figure 4.5.

```
EmulatedNode(NodeID)

CreateNeuron(NodeID, GroupID, Neuron ID, NeuronType)

CreateLink(NodeID,GroupID,SourceNeuronID,TargetNeuronID)

SearchPassive(NodeID, NeuronID)

SendNeuron(NodeID, GroupID, NeuronID, NeuronType)

ActivateNeuron(NodeID, GroupID)

CreateGroup(NodeID, NetFcnPtr, ActFcnPtr, OutFcnPtr,
              GroupID, NofNeuron, NeuronType)
```

Table 4.1 Sample Of Functions For multiprocessor Simulation

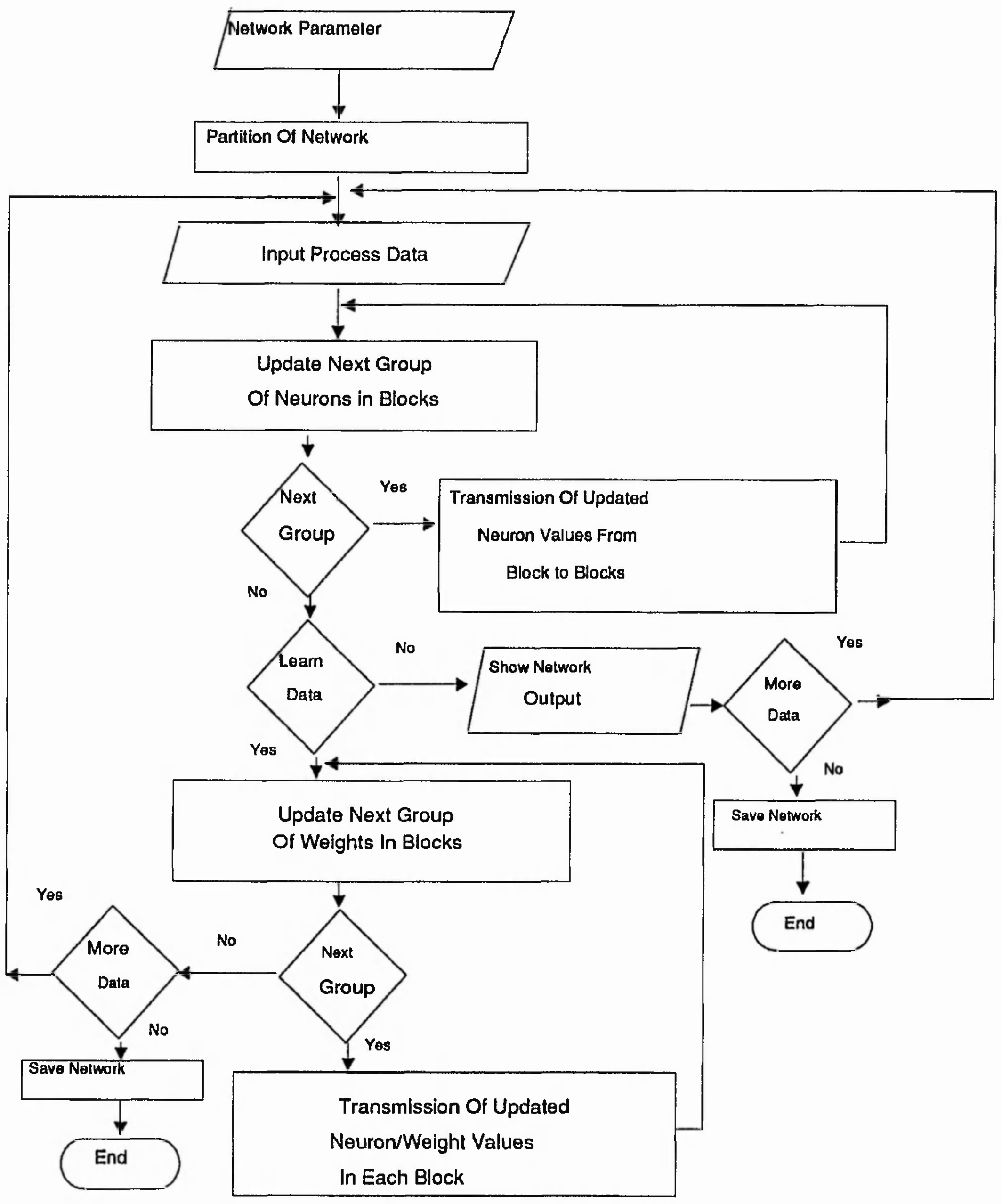

Network Parameter
Partition Of Network
Input Process Data
Update Next Group Of Neurons in Blocks
Next Group
Yes
No
Transmission Of Updated Neuron Values From Block to Blocks
Learn Data
No
Yes
Show Network Output
More Data
Yes
No
Save Network
End
Update Next Group Of Weights In Blocks
Next Group
No
Yes
More Data
Yes
No
Save Network
End
Transmission Of Updated Neuron/Weight Values In Each Block

Figure 4.5 Operation Of MultiSim.exe

CHAPTER 5 SIMULATION AND PERFORMANCE EVALUATION

CHAPTER 5 SIMULATION AND PERFORMANCE EVALUATION

The new network performance need to be compared with that of several current networks. Their performance in respect to the dynamics of computations, learning and response are compared and presented in the following sections.

All simulations are generated using the same simulation tools, described in chapter 4, and the same sets of learning data.

There are two main stages of simulations: open loop and closed loop. In the open loop case, networks are trained to behave as systems that compute the output representing the area bounded by two successive points on a trajectory image plane and the boundary of the time axis.

In the closed loop case, the output of a trained neural network are fed back and combined with the input vector to form a first order system so that any difference (error) generated is fed to the neural integrator and the by-product representing the derivative of the input is then extracted.

5.0 Training Data

In both open and closed loop stages the input and output neuron organisation of networks are fixed. This is because the training or learning data sets have a pre-defined format. Each set of training data consists of pairs of binary input and target output patterns; networks are trained as integrators and are expected to convert a distributed representation of patterns with (n) bits into a local representation over (2*n-1) bits.

5.0.1 Open Loop Training Data

A set of training data in the open loop case consists of pairs of binary pattern. Each input pattern has either one or two 'ON' bits representing the superimposed amplitude samples (representing position, velocity, acceleration, etc.) at two successive time instances. The length of each input pattern (n) is equal to the number of grid points used for the amplitude axis of the image plane.

The number of pattern pairs in each set of training data is varied according to the length of each input pattern. For an image plane with n*t grid positions, where t and n denote the horizontal time and vertical amplitude axis, the number of bits in each training pattern is (n) and the total number of pattern pairs within the training set is :-

$$\textit{Number Of Pattern Pairs} = \sum_{k=1}^{n} K \qquad (5.1)$$

Figure 5.1 shows that as the number of bits in each pattern increases so does rapidly the number of training pattern pairs. For this reason, the size of simulated nets have been kept to a minimum - typically 15 and 29 neurons in the input and output layers respectively.

Associated with each input training pattern is the target pattern. Each target pattern contains only one 'ON' bit. It represents the mid-point of the two 'ON' bits of an input pattern, this is illustrated by the example shown in Figure 5.2.

Table 5.1 is an example of a set of 4 bit training pairs. It is clear that the similarity structure of the distributed input pattern is simply not preserved in the output representation. No similar input patterns are associated with the same output and those input patterns sharing the same output response are non-orthogonal, this kind of problem has been regarded as 'difficult' for neural network models [Minsky,1969][Aleksander,1990].

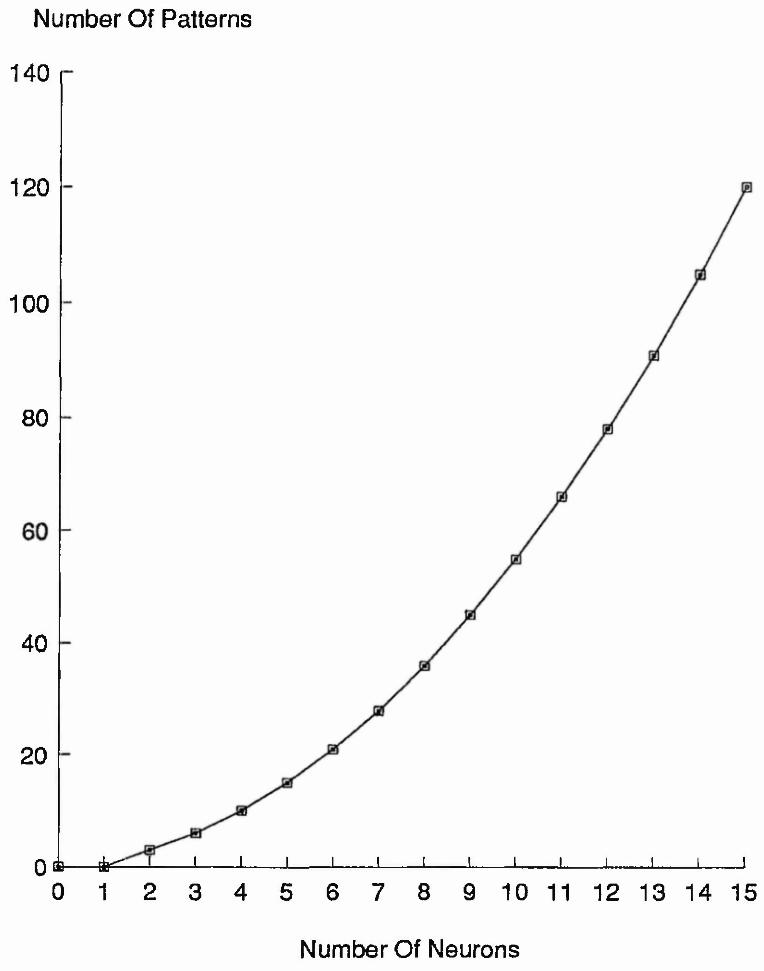


Figure 5.1 Number Of Patterns Versus Pattern Size

Table 5.1

Input	Output
0000	0000000
0001	0000001
0010	0000100
0100	0010000
1000	1000000
0011	0000010
0101	0000100
1001	0001000
0110	0001000
1010	0010000
1100	0100000

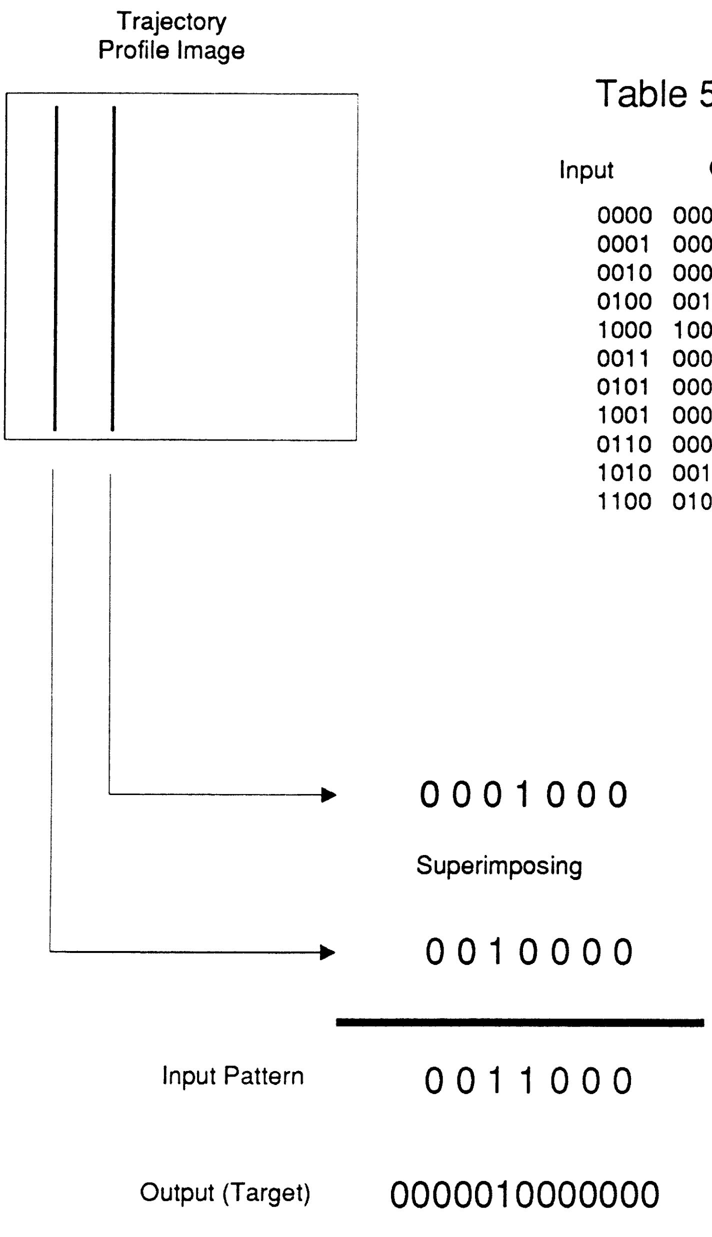


Figure 5.2 Example Of Training Pair

5.0.2 Closed Loop Simulation Data

Network models that have successfully learnt the open loop representation are subjected to the closed loop simulation tests. In this stage, each network is simply tested with several different functions: constant, linear, quadratic, exponential and trigonometric sine functions. In each test, the derivative of the input function is read off and compared to calculated values.

5.1 Open Loop Network Simulations

Current networks that have been simulated and used for comparison consists of Single Layer Perceptron with Delta Rule, Multi-layer Perceptrons with Backpropagation Algorithm, Associative Memory, Competitive, Kohonen, Counter-propagation and the new Retinal networks.

5.1.1 Single Layer Perceptron (Delta Rule)

Delta Rule or Window training scheme is a supervised training algorithm employed by networks with single layer of binary neurons and more commonly known as Single Layer Perceptron. Computation in the perceptron is synchronous and feedforward type so that all neurons within the single layer network are updated simultaneously. The activation rule of each neuron when it responds to an input pattern is :-

$$O_j = a_j(\sum_{i=1} W_{ij} X_i) \qquad (5.2)$$

where a() is binary with domain [0,1].

In the simulations an extra static layer has been introduced to the network to act as the fan-in of the network. During training each neuron in the static layer is clamped to a single bit of the input pattern and does not perform any computation except for holding and distributing the bit value. The simulated network is a fully connected type, weights between neurons are initially set to small random values between +1 and -1. Training is carried out as following :-

1) A training pattern is selected from the training set and the appropriate neuron in the static layer is set according to the clamped value.
2) The static input neurons activate the processing layer's neuron and the output of each neuron is calculated using equation (5.2).
3) The error for each neuron is computed by subtracting the actual output from the corresponding desired output pattern.

4) New weights are calculated with the following weight changing equation (5.3) : -

$$W_{ij}(t+1) = W_{ij}(t) + \delta_j X_i \zeta \qquad (5.3)$$

where ζ is the learning rate, δ_j the calculated error and X_i is the binary value of the i-th static input neuron.

5) Another pattern pair is selected from the training set and the procedure is repeated to reduce output errors.

6) The above steps are repeated for another cycle.

To produce representative performance a network with 15 neurons in the static layer and 29 neurons in the processing layer were used; the number of training pairs is thus 120. Various values of ζ, ranging from 0.1 to 0.5, are used during training. The graph shown in Figure 5.3 represents the average learning performance of the above network model. The result shows that none of the simulated single layer networks have been able to learn the open loop representation. Using larger training cycles and smaller values of ζ have no effect on the learning performance. In all cases, the high error level remained unchanged. The results show that networks of this type are simply not capable of representing integration.

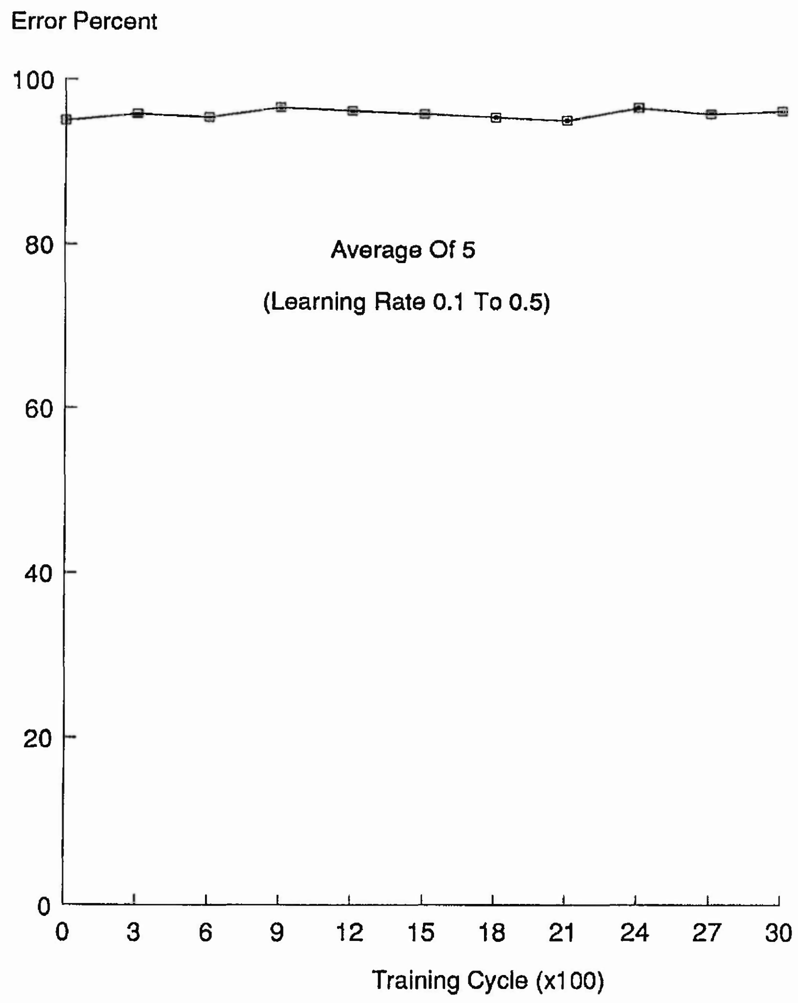


Figure 5.3 Error Percent Versus Training

5.1.2 Associative Memory Model

An associative memory model stores and recalls association of patterns learned by summing correlation matrices.

Hopfield and Bidirectional Associative Memory (BAM) are examples of associative memory models. In associative memory neural networks, each neuron is connected to other neurons through a connection strength matrix T. Each neuron is a simple non-linear function and transfers the sum of weighted input signals into a single output signal. The main difference between Hopfield and BAM is that Hopfield nets are auto-associative memory type while BAMs are hetero-associative memory models, see appendix (A.1) for more detailed description.

A network of associative memory can be constructed by specifying the connection strength matrix T. If a pattern Y is to be associated with pattern X, T is constructed by finding the outer product of these two vectors :-

$$T = Y^t X \qquad (5.4)$$

Where 't' denotes the transpose of a column vector. If there are K patterns to be stored in the memory, then T_{ij}

can be derived by :-

$$T_{ij} = \sum_{p=1}^{k} (Y_i^p)^t (X_j^p) \qquad (5.5)$$

that is by adding the outer product of all the individual memory vectors.

The simulated network models were hetero-associative memory type (BAM) because for representing integration Y and X are different and have different dimensions, (fields), so the connection matrices are asymmetric, $Y^p \neq X^p$.

The following procedure was used to simulate the hetero-associative memory (BAM) model :-

To construct the network, step (1) and (2) are used:-

1) For each pair of patterns in the training set, convert the binary patterns to bipolar format such that 1 to +1 and 0 to -1.
2) Calculate the correlation matrix for each association pattern pair and then add up the correlation matrices as given by equation (5.5).

To activate the network, the following steps are used :-

3) Neurons in the input field are forced to the values of the input pattern; either ON or OFF according to whether the corresponding binary values are 1 or 0.

4) Calculate the neurons outputs in the output field according to equation (5.6):-

$$O_i(t+1) = f(\sum_{j=1}^{n} T_{ij}O_j(t)) \qquad (5.6)$$

where f() is a binary function; if the inputs sum of a neuron is larger than 0 set it to 1; if the inputs sum is below 0 set it to 0; and state of neuron is unchanged if the inputs sum is equal to zero, O(t) and O(t+1) represent the current and next state of the network.

5) Calculate the outputs of neurons in the input field.

6) Repeat (4) to (6) until the outputs of neurons in the input and output fields stabilize (i.e. stop changing).

Following procedures (1) and (2), networks of BAM models with an input size rising up to 15 neurons have been constructed. Simulation results have shown that the reliability of this type of model is very low. The average reliable retrieval of associations is less than 15%. The only exceptional case is a small 2 by 3 network where all the stored associations were successfully retrieved. As the network size is increased so does the number of associations. For a 3 by 5 network there are 6 associations, and in this case less than 72% of association were successful.

Single pattern simulations were also carried out. Using this approach all the simulated networks can retrieve the single association reliably. The results indicated that, for any network size, as more patterns were added to a connection matrix the network's association degraded. Fewer patterns were able to be recalled as more matrices were added. Such misclassification reflects that the connection matrix can be overcrowded.

For representing integration, it is difficult to determine the relationship between the network reliability and the number of associations. This is because the number of patterns needed to be stored is always larger than the size of the input field (i.e. neurons clamped to input) and the degradation of reliability is not constant. From the simulation results it is clear that high reliability can only be obtained if the number of associations is less than or equal to the higher of the two-pattern dimensions.

5.1.3 Multi-Layer Networks Using Backpropagation

Both Associative Memory and Single Layer Perceptron have serious storage limitations; when more data is stored in the networks adaptation degrades and unlearning start to occur. The limitations of these networks can be overcomed by Multi-Layer networks using a nonlinear neuron activation rule.

In this section, the simulation results of three multi-layer networks using the Error Backpropagation algorithm are described. The simulated networks architectures include: a four layer network with two hidden layers of neurons, and two three-layer networks with a single layer of hidden neurons.

The two three-layer networks have different connections arrangements; one with restricted connections between successive layers only and the other with additional direct connections between input and output layers. All multi-layer networks have the same 15 input and 29 output neuron organisations.

The simulation procedures are the same for all the multi-layer networks and are as follows :-

1) Initialize all weights to small random values within the range of +1 and -1.
2) Select a pair of training patterns and feed the input pattern to the network.
3) Calculate the outputs of neuron in each layer according to equation (5.7).

$$O_j(t+1) = \frac{1}{1 + \exp(-\sum_{i=1}^{n} W_{ij}O_i(t))} \qquad (5.7)$$

4) Calculate the error of neurons in each layer except the input layer as follows :-

   I) Calculate neurons error in the output layer with equation (5.8).

$$\delta_j = O_j(1 - O_j)(T_j - I_j) \qquad (5.8)$$

   where T_j and O_j represent the desired target and the actual output values.

   II) Calculate neuron errors in the hidden layer(s) with equation (5.9) :-

$$\delta_i = O_i(1 - O_i)\sum_{j=1}^{n}\delta_j W_{ij} \qquad (5.9)$$

5) Adjust all the weights leading to each neuron with equation (5.10) :-

$$W_{ij}(t+1) = W_{ij}(t) + \zeta\delta_j O_i(t) \qquad (5.10)$$

6) Repeat (2) - (5) for other training pattern pairs.

7) Repeat (6) for another cycle until error levels are acceptable.

Each simulated network was trained with >3000 cycles. Though it may seem more reasonable to allow the network to run freely and make the stopping criterion controlled by error level, however, as can be seen in Figure 5.4, in some cases particularly with small size of hidden layer learning can take quit a long time. Where the network fails to achieve the required error level, the learning process is forced to stop.

5.1.3.1 A 3 Layer Net With Restricted Connections

Simulation results for this case are as summarized in Figure 5.4(a). The input and output layers consist of 15 and 29 neurons respectively. During simulation ζ=0.1 was used to enable fine adjustment of weights. The number of hidden neuron was varied from 5 to 40. From Figure 5.4(a)(b) it is clear that as the number of hidden neurons is increased the corresponding percentage error decreases rapidly. With smaller sizes of the hidden layer the network learned much slower and became unreliable when the hidden layer size was too small. Increasing the size of the hidden layer did not improve performance. In fact in this particular case the percentage error declined much slower as the hidden layer size was increased from 19. The best result (the fastest decline in error) was recorded with 19 hidden neurons where the error reaches 5% after 990 training cycles, no further improvement was observed, see Figure 5.4(b).

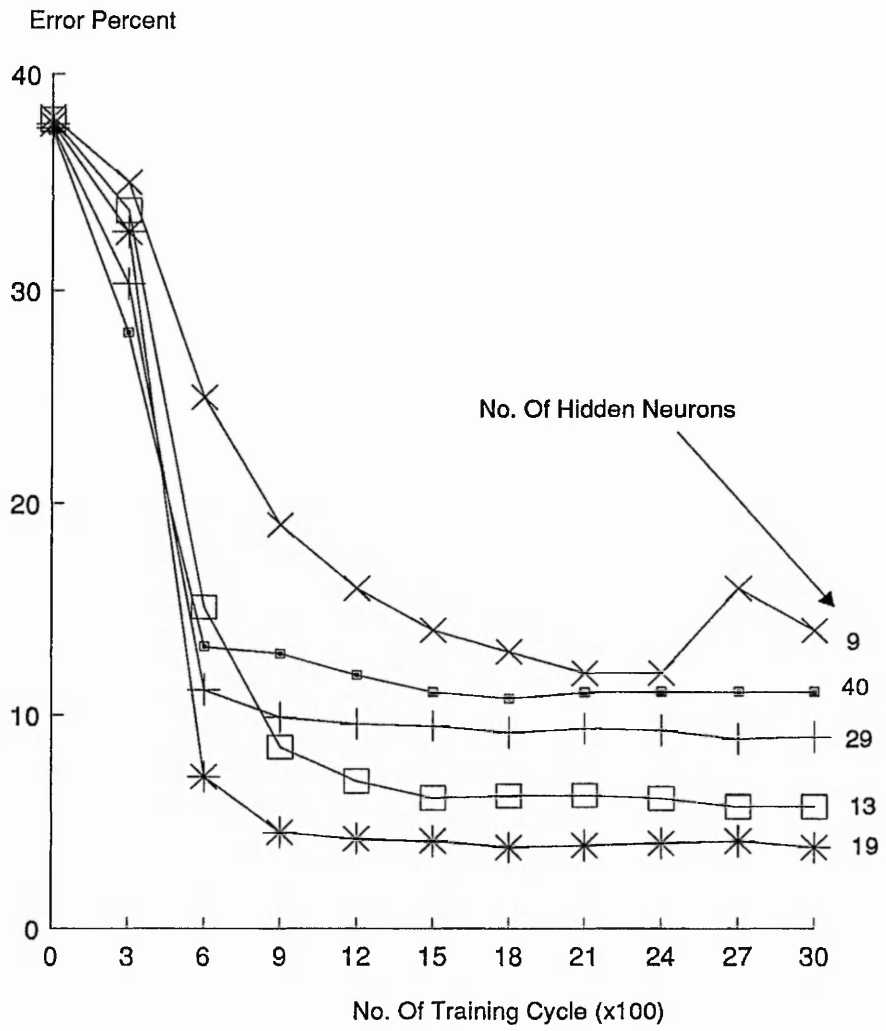


Figure 5.4(a) Error Percent Versus Training Cycle Number

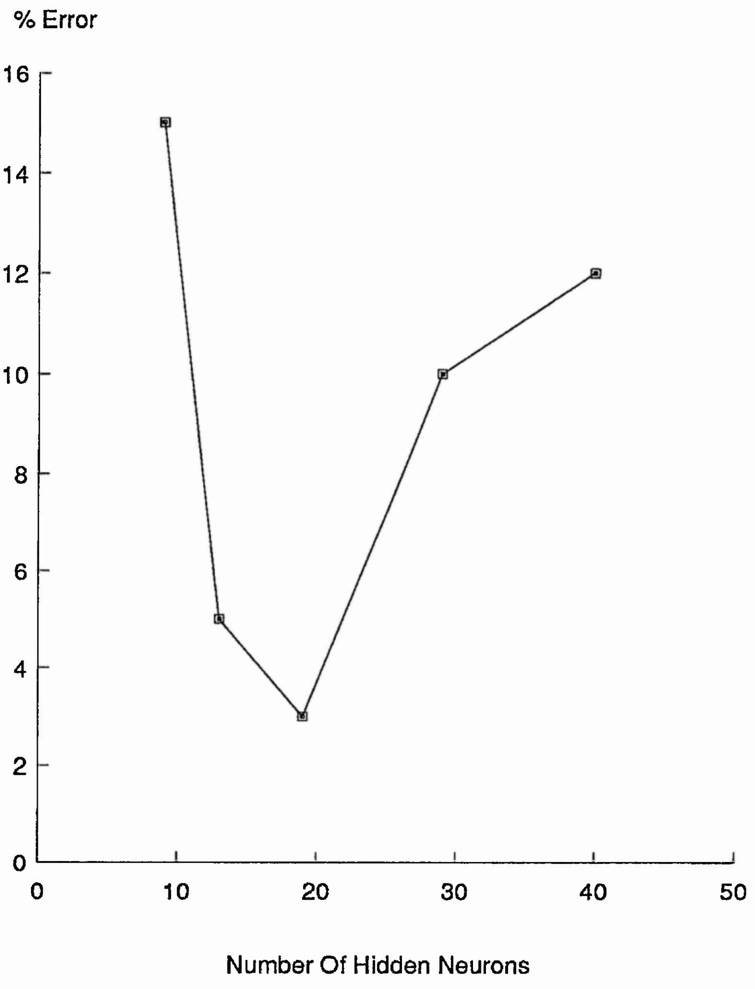


Figure 5.4(b) Optimum Hidden Layer Size

5.1.3.2 A 3 Layer Net With Direct Input-Output Connections

The set of simulation results for this case are summarized in Figure 5.5(a). The same procedures were used to simulate this network. With this architecture the network performance is degraded. The time it takes the network to learn is longer. Slight improvement in terms of the rate of learning has been observed. The best result gives an error percentage of <10% after 1980 training cycles with 19 hidden neurons. A degradation of performance is noticeable, see Figure 5.5(b).

5.1.3.3 A Four Layer Net with Restricted Connections

To improve the classification capability, simulations were also carried out with a four-layer architecture. The input and output neuron organisations are the same as before (15 and 29 respectively) but two hidden layers with various sizes were used. Results are presented according to the sizes of hidden layers, see Figure 5.6 to Figure 5.8.

Figure 5.6 shows error percentage graphs for networks with fixed 30 neurons in the first hidden layer. Using 40 neurons in the second layer the network reached <1% error in 693 cycles. Experiments have shown that decreasing the size in the second hidden layer will lead to a longer

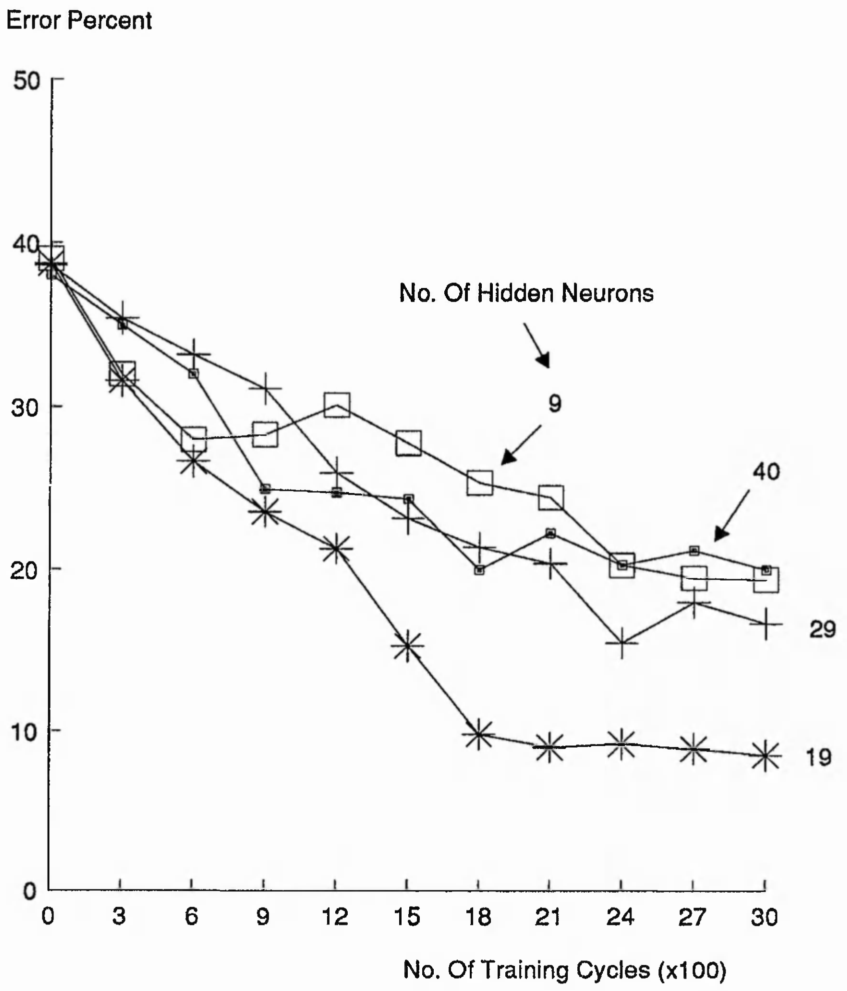


Figure 5.5(a) Error Percent Versus Training Cycle Number

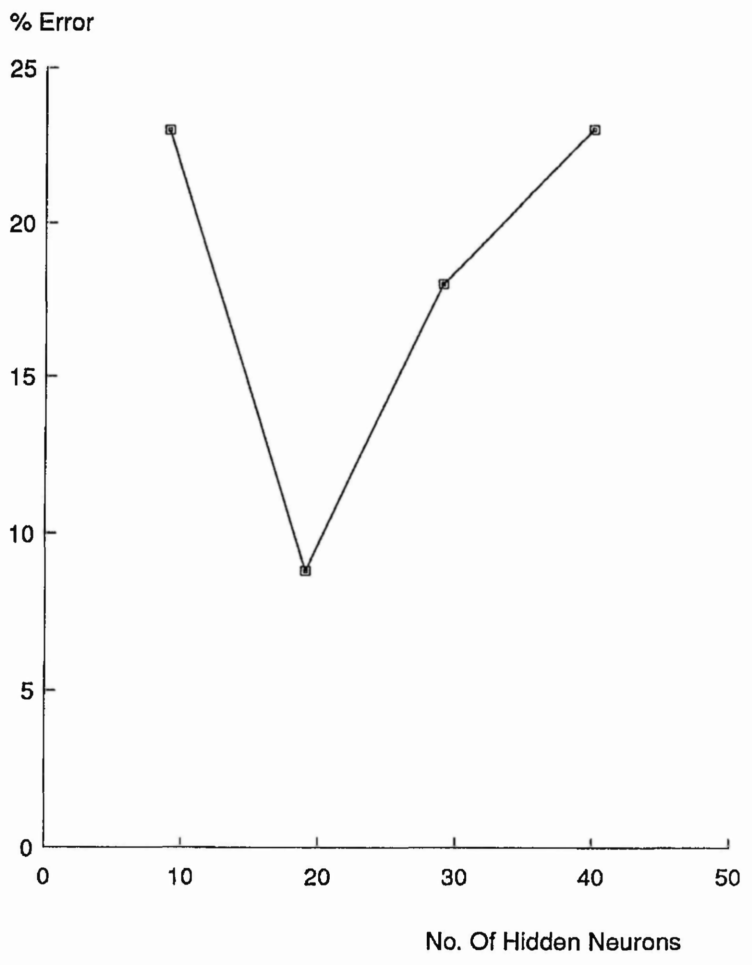


Figure 5.5(b) Optimum Hidden Layer Size

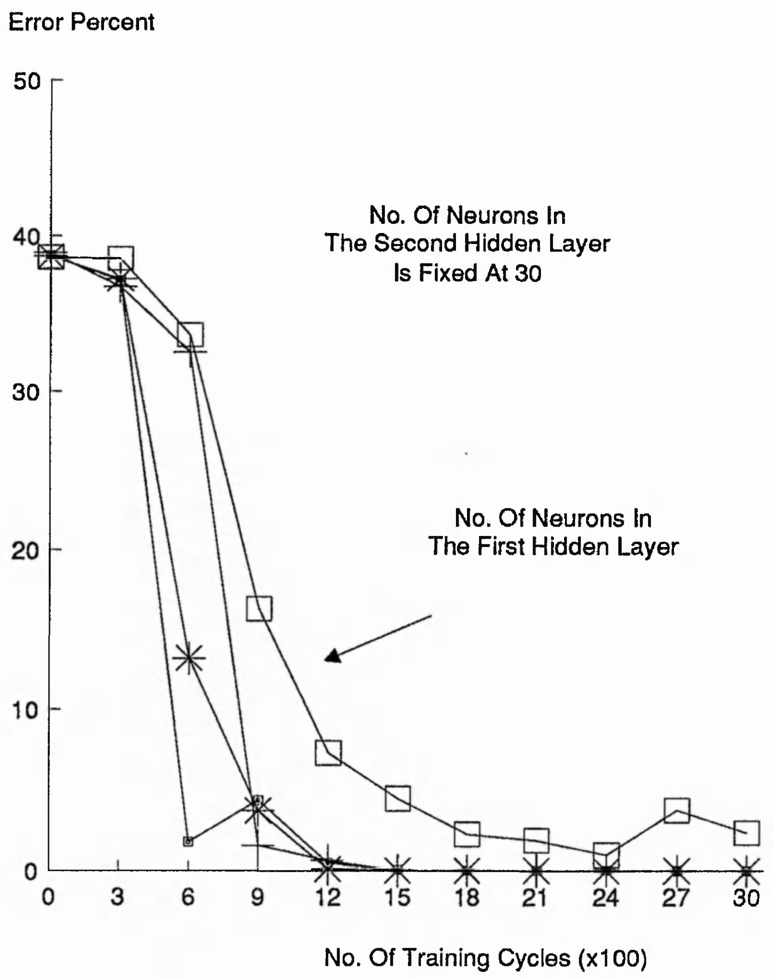


Figure 5.6 Error Percent Versus Number Of Training Cycles

training time for the network to reach the same results. With 10 neurons in the second hidden layer it took the network 2475 cycles to reach 1% error.

Very similar results were also obtained with fixed sizes, 20 and 10, of neurons in the first hidden layer, presented in Figure 5.7 and figure 5.8 respectively. On average, as long as the size of the second layer is not less than 15 (the size of the input layer) the network will learn all the associations. The only draw back is that longer training time is needed when using small sizes of hidden layers.

From this results it is deduced that for the integrator model, with n input and 2*n-1 output neurons, the minimum number of neurons required for the two hidden layers are 2*n/3 and 4*n/3 respectively. Using this result a network with n=20 was tested (the size of the hidden layers were 15 and 20 respectively). The network learned all the associations within 1089 cycles with a learning rate of 0.2, (a small learning rate enables fine weight adjustment).

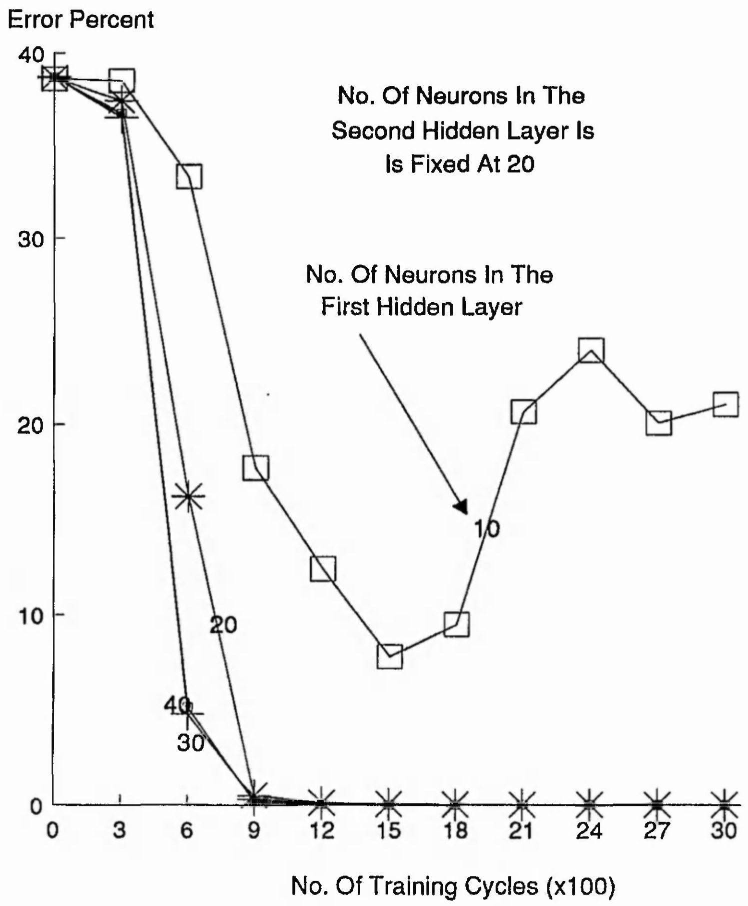


Figure 5.7 Error Percent Versus Training Cycle Number

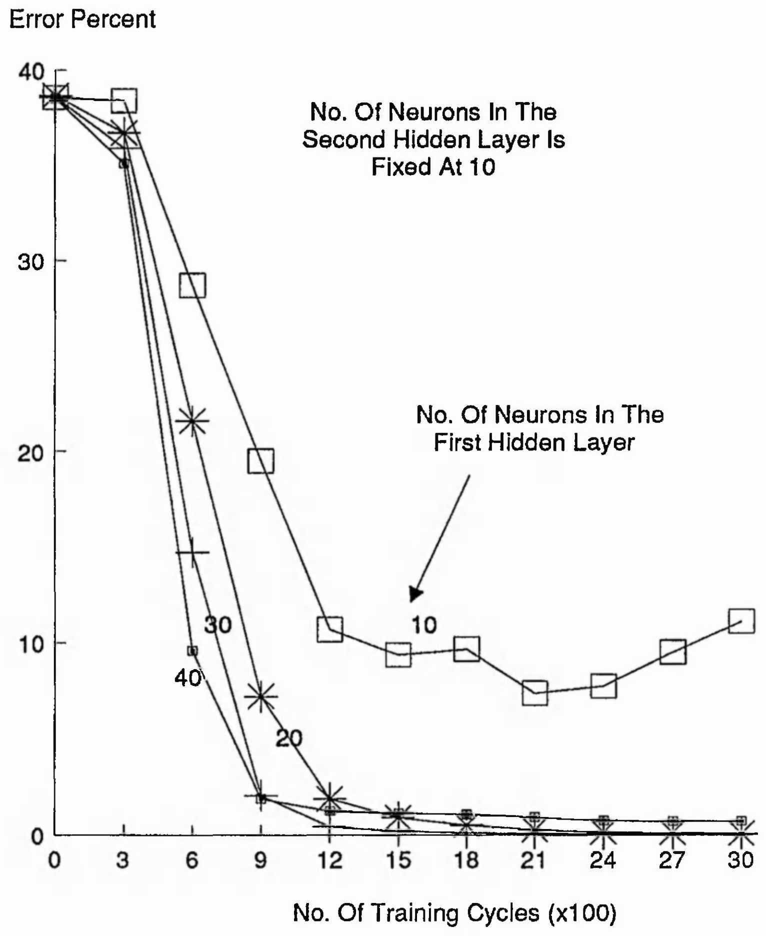


Figure 5.8 Error Percent Versus Training Cycle Number

5.1.4 The Competitive Model

The previously simulated networks were of the supervised type, which assumes that inputs and desired outputs are available. In this section the simulations of unsupervised networks are described. In the simplest, the weights are adjusted according to the inputs such that only one population of neurons, may be only one or a group of neurons, will respond to a particular pattern, or a class of patterns.

The simulated networks have two layers of neurons, a layer of static (input) neurons and a layer of competition neurons. Similar to the case of the Perceptron model the static layer does not actually perform any computation, it is used primarily for holding and distributing the inputs.

The competition layer consists of identical neurons with connections from the static layer and inhibitory connections from neighbour neurons in the competition layer.

The computation of a competitive network is very straight forward. Each neuron in the competition layer calculates the sum of the weighted inputs. Neurons within a predefined population then compete with each other, and the one with the largest input sum is activated and the others suppressed.

Weights leading to the active neuron(s) are adjusted by shifting weight from inactive input connections to the active connections. The rest of the weights leading to inactive neurons remain unchanged. The procedure for simulating such networks is as follows :-

1) Initially randomize all weights so that the sum of weights of a neuron is equal to 1.
2) Select a pattern from the training set and feed to the static input layer. The sum of weighted inputs of neurons in the competition layer is then calculated.
3) Activate (set output to 1) neurons with the largest sum of inputs within a predefined population in the competition layer and set the outputs of the rest to 0.
4) Adjust the weights leading to the active neuron with equation (5.11) :-

$$W_{ij}(t+1) = W_{ij}(t) + [\frac{gc}{n} - gW_{ij}(t)] \qquad (5.11)$$

   where n is the number of 'ON' neurons in the static input layer; c is the value of the input static neuron so that if c is 'Zero' the weight is reduced by the gain factor g, and if c is 'one' a small amount proportional to g will be shifted to the active connections, i.e. moving weights from the inactive to active input paths.

5) Repeat (2) to (5) for another pattern.

6) Repeat (2) to (5) until the output stabilizes.

In the first set of simulations, each simulated network has a layer of 15 input neurons and a single competition cluster of various sizes.

Using the competitive learning algorithm it is expected that the competition layer will tend to group similar patterns together. However, the integrator application requires each competition neuron to group non-similar patterns. Simulations have shown that this is very difficult with this model. In many cases, with learning gain of 0.5, the outputs of the simulated networks stabilized within a few hundred cycles, using smaller learning gain would merely increase the time it takes the networks to stabilize and has no effect on the final output.

Depending on the size of the single competition cluster the size of the divided, or grouped, input patterns, changed inversely; better isolation can usually be obtained when a larger competition cluster is used.

With 15 components in each input pattern there are 120 patterns to be isolated. Various sizes of the single competition cluster ranging from 120 to 1000 were used. A base size of 120 was used to ensure that the network has sufficient neurons to choose from. Table 5.2 summarizes the simulation results.

The results show a slight increase in isolation with a large increase in the competition cluster size. With 1000 neurons in the single competition cluster approximately 26% isolation was obtained. Larger competition layer size was not simulated as it would take a very impractical processing time; it took nearly 40 hours to train the 15 by 1000 network on the VAX with learning gain of 0.3!

In the second set of simulations a fixed size of 200 neurons with various number of clusters were used. Networks with a number of clusters ranging from 2 to 20 were simulated. In all cases the use of a new number of clusters would change the output of the competition layer; changing the network dimension would change the activation of neurons. However, no significant improvement in terms of input isolation was observed. The results are effectively the same as when using a single competition cluster.

5.1.5 The Kohonen Network

The architecture of Kohonen networks is exactly the same as the competitive model and consists of two layers of

Size Of Cluster	Average Number Of Patterns Associated With a Neuron
120	~31
200	~31
300	~31
500	~28
750	~28
1000	~26

Table 5.2 Average Number Of Pattern Grouped By Neuron

neurons: a static and a Kohonen layer. Neurons in the Kohonen layer are competition neurons. During adaptation the neuron with the largest output and its lateral neighbouring neurons are declared as the winner population and weights leading to neurons in the winning population are adjusted.

There are two type of connections in a Kohonen network: the external weighted input connections (the static layer connections), and the lateral feedback connections from other neighbouring neuron. The external weighted input connections are adjustable while the weights of the lateral feedback connections are determined by the lateral interaction function. The lateral interaction function is a function of the lateral physical distance between neurons. A typical lateral distance function is shown in Figure 5.9, commonly called the "Mexican Hat" function. The updating of the Kohonen neuron can be described by equation (5.12) :-

$$O_k(t+1) = G(F(W_{kk-1}, O_{k-1}(t))) \qquad (5.12)$$

where F() is a measure of distance between the connection weights W_{ij} and the signals O_i from the static layer. The activation function G() may be of the logistic type or

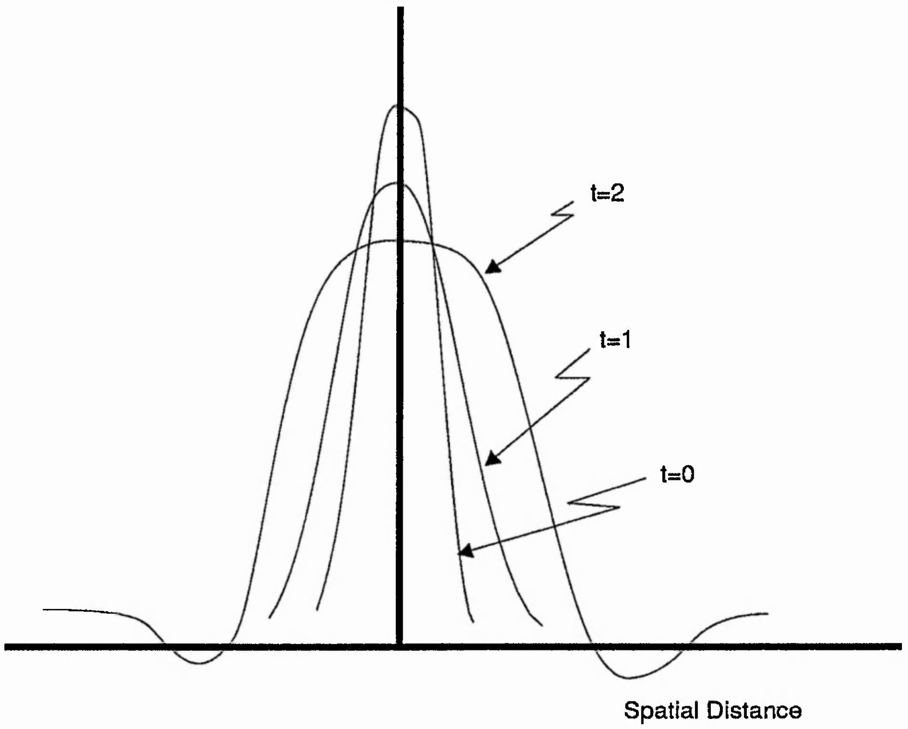


Figure 5.9 Typical Shape Of Mexican Hat Function

linear or binary activation function, thus : -

$$O_k(t+1) = \begin{cases} 1 \ if \ O_k(t) = \min[\sum (W_{kk-1} - O_{k-1})^2] \\ 0 \qquad otherwise \end{cases} \tag{5.13}$$

Computation of the simulated Kohonen network differ slightly from other neural network models (but functionally the same). When an external input is applied to the network, the complete input is presented to each Kohonen neuron through the static layer. Each Kohonen neuron sets its output according to the sum of it's measure of distance between the input signals and it's corresponding connection weight given by :-

$$\sum_{k=1}^{p} (W_{kk-1} - O_{k-1})^2 \tag{5.14}$$

where p is the length of the external input or the number of neurons in the static layer.

The output of the Kohonen neuron with connection weights closest to the external input values is activated. The size of the winning population is then determined by the lateral interaction function. During the learning process the size of the lateral "Mexican Hat" function varies and shrinks as learning progresses such that as the network learns the number of neurons associated with an input pattern is reduced. The size of the lateral function changes according to the learning interaction and eventually only one neuron is allowed to associate to an input or a group of inputs. The simulation procedure is summarized by the following steps :-

1) Initially randomize all weights to be within 0 and +1.
2) Normalize the weights by dividing each weight component by the length of the weight vector. The length is found by taking the square root of the sum of the squares of all the weight components leading to a neuron, i.e :-

$$X_i^{normalize} = \frac{X_i}{\sqrt{W_1^2 + W_2^2 + \ldots W_n^2}} \qquad (5.15)$$

3) Select a pattern from the input set, normalize the input as above and apply the normalized version to the static layer.
4) Calculate the output of each neuron in the Kohonen layer using equation (5.14).
5) Calculate the radius size of the winning population with equation (5.16) : -

$$Radius = \frac{Number\ Of\ Kohonen\ Neuron}{2} - \frac{TrainingCycle}{Step\ Size} \qquad (5.16)$$

6) Select the neuron that produces the smallest output and it's neighbouring neurons according to the radius size calculated in step (5) and set their output to 1 (activate them).
7) Alter the weights leading to the active neurons with equation (5.17) : -

$$W_{kj}(t+1) = W_{kj}(t) + \zeta(X_j - W_{kj}(t)) \quad (5.17)$$

8) Repeat from step (2) for the rest of the training data.

9) Repeat step (8) for the next training cycle.

Kohonen networks of various input and output dimensions were simulated. The performance of small networks suggested that the size of the Kohonen layer must be at least several times larger than the number of patterns to be learned if total isolation is desirable; each kohonen neuron points to a specific pattern.

Using an unsupervised network, it is expected that the network either groups similar patterns together or completely isolates all input patterns. For the integrator application complete isolation is desirable as Kohonen neurons are required to point (or to group) selectively non-orthogonal inputs together. In order to achieve this, the size of the Kohonen layer must be at least many times larger than the number of patterns.

As the size of the network grows so does the number of patterns, and it becomes increasingly difficult to predict the suitable Kohonen layer size. Experiments show that when a small layer of Kohonen neurons is used the network will always isolate uncorrelated inputs. When the size of the Kohonen layer is increased the isolation capability will follow; the correlated patterns will further be arranged into smaller groups. Experiments show no clear relationship between the size of the Kohonen layer and the number of patterns to be learned.

The outcome of the Kohonen and the competitive models are very similar. Both models will tend to group data according to their correlations. However, higher isolation is obtainable with the kohonen model using the lateral neighbouring interaction. Though, with the lateral interaction function it would seem that some of the weights may not be adjusted at all (weights leading to neurons which are never active) but inspections have shown that weights do all change in time. Some neurons may not be winners at all during the early stage of the learning process but as learning progresses the lateral interaction will alter the situation by distributing the weights evenly thereby increasing the degree of isolation.

5.1.6 The Counter-propagation network

This network is a hybrid combining Kohonen and Grossberg learning methods. During learning, pairs of vectors (say [X, Y]) are presented to the network, these vectors then propagate through the network in a counterflow manner to yield a pair of output vectors which are an approximation to the input vector pair. After learning, if only one of the learned vector or a pair having some components of both X and Y zeroed out is entered, the network will complete the vector pair and the output will be approximately the same as the best matching input vectors. Thus the network functions as a lookup table.

There are many variants of the counter-propagation network. The type described in this section is a three layer feedforward only version in which only transformation from one vector to another are of interest. The network architecture consists of three layers of neurons. Neurons in the first layer serve only as fan-in of the network and perform no computation. The second layer is the Kohonen layer where competition of neurons is performed. Neurons in this layer are fully connected to the input and the Grossberg layers. The network architecture is very much the same as the three layer perceptron described earlier, the differences lie in the processing performed by the Kohonen and Grossberg layers.

Neurons in the counter-propagation network perform the calculation of their sum of weighted inputs. When an input is applied to the network neurons in the Kohonen layer compete with each other; their weighted inputs are summed and the neuron with the largest sum is set to one, all others are set to zero. Neurons in the Grossberg layer then sum the weights connecting to the winning Kohonen neuron to yield the desired vector.

When learning is performed for a given input vector, each weight associated with the winning Kohonen neuron is changed by an amount proportional to the difference between its weight value and the value of the input to which it connects. Weights between a winning Kohonen neuron and Grossberg layer neurons are adjusted by an amount proportional to the difference between target and actual output vectors. Simulations were performed as follows :-

1) Initialize connection weights of the Grossberg layer to small random values.
2) Initialize connection weights of the Kohonen layer by one of the following methods : -
   (a) By **Convex Combination Method** - set all weights to the same value $1/\sqrt{n}$ where n is the number of input components.
   (b) By **Lateral Neighbouring Interaction Method** - set all weights to a small random value.
3) Normalize connection weights of the Kohonen layer by dividing each weight component by the length of the weight vector. The length is calculated by taking the square root of the sum of the square of all the weight components leading to a neuron.
4) Apply an external input vector to the network; if using

the convex combination method coincide each input component with the weight vectors by giving each input component, X, the value :-

$$X_i = \zeta X_i + [\frac{1}{\sqrt{n}} (1 - \zeta)] \qquad (5.18)$$

where n is the number of input components of the input vector. Initially, the convex combination method start off with ζ near zero. This forces all input vectors to be close to weight vectors. As time goes on the value of ζ is raised slowly to one. As this happens, the weight vectors are 'peeled' off and follow the input vector as they move away from 1/√n.

5) Calculate the output of each Kohonen neuron by summing the weighted inputs. If using lateral neighbouring interaction set the neuron with the largest value and it's neighbouring neurons (determinated by the lateral function, see the Kohonen network in section 5.1.5) to one and set all other neurons to zero. With the convex combination method simply set the neuron with the largest sum of inputs to one.

6) Adjust weights of the active Kohonen neurons with equation (5.19) :-

$$W_{ij}(t+1) = W_{ij}(t) + \zeta (O_j - W_{ij}(t)) \qquad (5.19)$$

where $W_{ij}(t+1)$ denotes the next weight value between the active i-th Kohonen neuron and the input O_j, ζ is the learning rate and should start out with a value close to

1 and reduced gradually as learning progresses.

7) Calculate the Grossberg layer output by summing the weighted input of each neuron.

8) Adjust the connection weights of the Grossberg layer by equation (5.20) :-

$$W_{gk}(t+1) = W_{gk}(t) + \zeta(T_g + O_g) O_k \qquad (5.20)$$

where W_{gk} is the weight between the active Kohonen and Grossberg neurons, ζ is the learning rate and starts out at around 0.1 and reduced as learning progresses, T_g is the target value of the Grossberg neuron and O_k is the output value of the Kohonen neuron such that if the Kohonen neuron is not active then the weight is unchanged.

9) Repeat from (3) to (8) for the rest of the training inputs.

Simulations show that the accuracy of the counter-propagation network depends on the performance of the Kohonen layer. The only action of each neuron in the Grossberg layer is to output the value of the weights that connects it to the single non-zero Kohonen neuron.

Networks trained by the lateral neighbouring interaction method and by the convex combination methods vary little in performance. Both techniques lead networks to separate dissimilar input patterns. When the number of patterns to be learned is small, < 21, it is possible to predict and start off with the correct size of the Kohonen layer and train the network to meet 100% input to output associations, Table 5.3. As the network size grows it becomes difficult to maintain this level of accuracy. The Grossberg layer has very little influence on the final network outcome. Among all the simulated network configurations the Kohonen layer always manage to isolate groups of patterns but fails to respond to single individual pattern. Due to this problem the output of the Grossberg layer is unstable; swinging from one desired output to another as training progresses.

Experiments have shown that the counterpropagation model is not suitable for representing integration. However, due to its flexible basic architecture, there is another learning method to ensure 100% associations for any size of input-output neuron organisation. An experiment was carried out using the same basic architecture but a new method as described below.

A New Method:

Each set of weights leading to each kohonen neuron were initialized to the input values so that there are equal number of Kohonen neurons and number of patterns to be

Size Of input Layer Neurons Organisation	Number Of Patterns	Size Of Output to Obtain 100% Isolation
4	10	20
5	15	50
6	21	160

Table 5.3

learned. With this arrangement all the input patterns are automatically isolated and all that remains is to train the weights of the Grossberg layer to match the desired output. Simulations have shown that if the learning equation (5.19) is used through out the training process and the size of the Kohonen layer is made equal to the number of training patterns the Kohonen layer can be trained with only one calculation per weight with a learning rate of 1. The advantages of this method include: short learning time, easily obtained total association, and predictability of Kohonen layer size. The main draw back is that if the number of patterns to be associated is large the size of the network would be unpractically large.

5.1.7 The New Integrator Model

The architecture of the new integrator model consists of five layers of neurons, see chapter 3. The regular synaptic connections of the outer section of the network contains four layers and are presetable in a manner that if two adjoined photoreceptors are both activated by external inputs then the active bipolar cell directly beneath the photoreceptors will be turned off by the inhibitory connections coming from the horizontal cell. Thus triggering the middle ganglion cell in between the signal pathway of the bipolar cells.

The amacrine cell layer mediates the lateral interaction of the outer section of the network and the input and output connections of the amacrine layer are adjustable. The learning process of the amacrine layer is similar to the counter-propagation model, weights leading into the amacrine cell layer are adjusted according to the competition process and weights connecting the output of the amacrine and the ganglion layers are adjusted by a supervised learning method.

When signals arrive to the amacrine layer neurons with weights closest to the input win the competition. A winning neuron then undergoes a suppression test; if it has been winning more input patterns than it is allowed it's output will be suppressed and the next runner up neuron will be the winner if it survives the suppression test. The weights leading to the winner are adjusted by moving weights closer to the input signals.

Weights between the output of the amacrine and ganglion layers are adjusted such that the weights connecting the active ganglion and amacrine neurons are strengthen and the rest of weights are unchanged, (i.e. the Hebb's law). The simulation procedure consists of the following steps :-

1) Initialize the weights of the basic net such that the middle ganglion cell will be activated if both adjoint

photoreceptors are turned 'ON'; if only one photoreceptor is turned 'ON' then the ganglion cell directly beneath the bipolar cell along the signal path will be turned 'ON'.

2) Initialize the input and output synaptic connections of the amacrine layer to small random values.
3) Normalize the input and output weights of the amacrine layer, see Kohonen or counter-propagation network simulations.
4) Apply an external input to the network.
5) Calculate the output of the horizontal and bipolar layers by equation (5.21) : -

$$Output = \begin{cases} +1 & if \sum W_{ij} X_i \geq 1 \\ 0 & Otherwise \end{cases} \qquad (5.21)$$

6) Calculate the output of the amacrine layer according to equation (5.22) : -

$$Output_{amacrine} = \sum_{i=1} (X_i - W_{amacrine\ i})^2 \qquad (5.22)$$

7) Locate the (next) amacrine neuron with the minimum value.
8) Calculate the suppression criteria by k/c where k is the number of times the winning neuron has been activated and c is the maximum number of times it is allowed; if the suppression criteria is <1 then activate it else repeat step (7).
9) Adjust the input weight leading to the active amacrine

neuron by equation (5.23) : -

$$W_{ab}(t+1) = W_{ab}(t) + \zeta(\ O_a\ O_b - W_{ab}(t)\ ) \qquad (5.23)$$

10) Activate the ganglion cell layer according to the desired output pattern.

11) Adjust the output weights of the amacrine layer according to equation (5.24) : -

$$W_{ag}(t+1) = W_{ag}(t) + \zeta(\ Target_g - W_{ag}(t)\ )O_a \qquad (5.24)$$

12) Repeat from step (3) for another data pattern pair from the training set.

13) Reset the parameters of the suppression criteria.

14) Repeat from step (3) for next learning cycle.

Several simulation runs of the retinal model were made with input dimension rising up to 15 neurons. In all cases the same set of weight ratios was used for the basic building nets (weights between receptors, horizontal cells, bipolar and ganglion cells, see chapter (3). And the input and output connections of the amacrine cell layer were adjusted according to equation (5.23) and (5.24). In all cases total associations were achieved.

Simulations also showed that altering the learning gain merely changes the time for a network to reach satisfactory response and has no effect on the final outcome. With a learning gain ζ (equation (5.23) and (5.24)) of 1, the amacrine layer achieves coincidence with inputs in one pattern per cycle. The 'c' parameter of the suppression criteria was set to 1 in all simulations, thus each amacrine neuron responds to an unique input pattern. These settings provided faster speed for networks to reach total associations.

5.2 Closed Loop Simulations

Neural networks that were successfully trained in the open loop stage were used for the closed loop simulation. In the closed loop test each network is placed inside the forward path of a feedback system as shown in Figure 5.10.

To process a complete time related trajectory profile the input profile image is segmented into equal strips to simulate the time axis. Before data is fed into the system, each pattern is first converted to binary. At each processing time step two successive binary patterns are first superimposed together before being applied to the system, see Figure 5.10 for illustration. At the output side, successive outputs are fed back to the input. The resultant effect is two profiles one produced at the output of the system and the other produced at the forward path representing the corresponding derivative.

Neural networks based on the backpropagation and the new

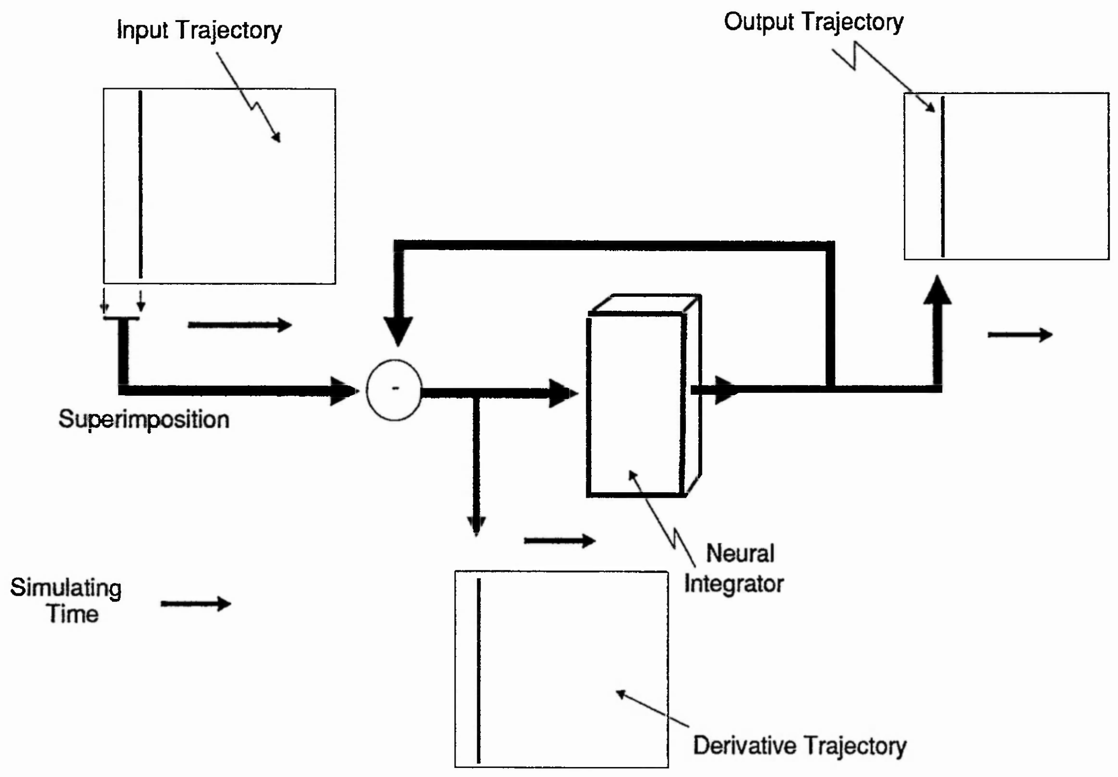


Figure 5.10 Close Loop Operation

retinal models were used for simulations. Both network models produced the same results as they were successfully trained to provide total associations to represent integration.

For each model, five different input functions (Constant, Linear, Quadratic, Exponential and Sine) were applied. Figures 5.11 to 5.15 show the responses of networks to these functions, including the corresponding estimated derivatives. The results show that the system is capable of: following the input profile, and producing the corresponding derivative shape of the input profile.

5.3 Speed Performance Of The Integrator Model

The pictorial integration process involves computation steps which are suitable for conventional programming.

With the conventional programming approach, the main computation involves scanning and comparing arrays of elements where data are stored. The computation speed thus depends on the size of the arrays (i.e. the size of the input patterns), the larger the arrays the longer the computation. In the ideal neural network approach, the computation speed would be constant as all inputs are processed in parallel. In reality, this depends on the

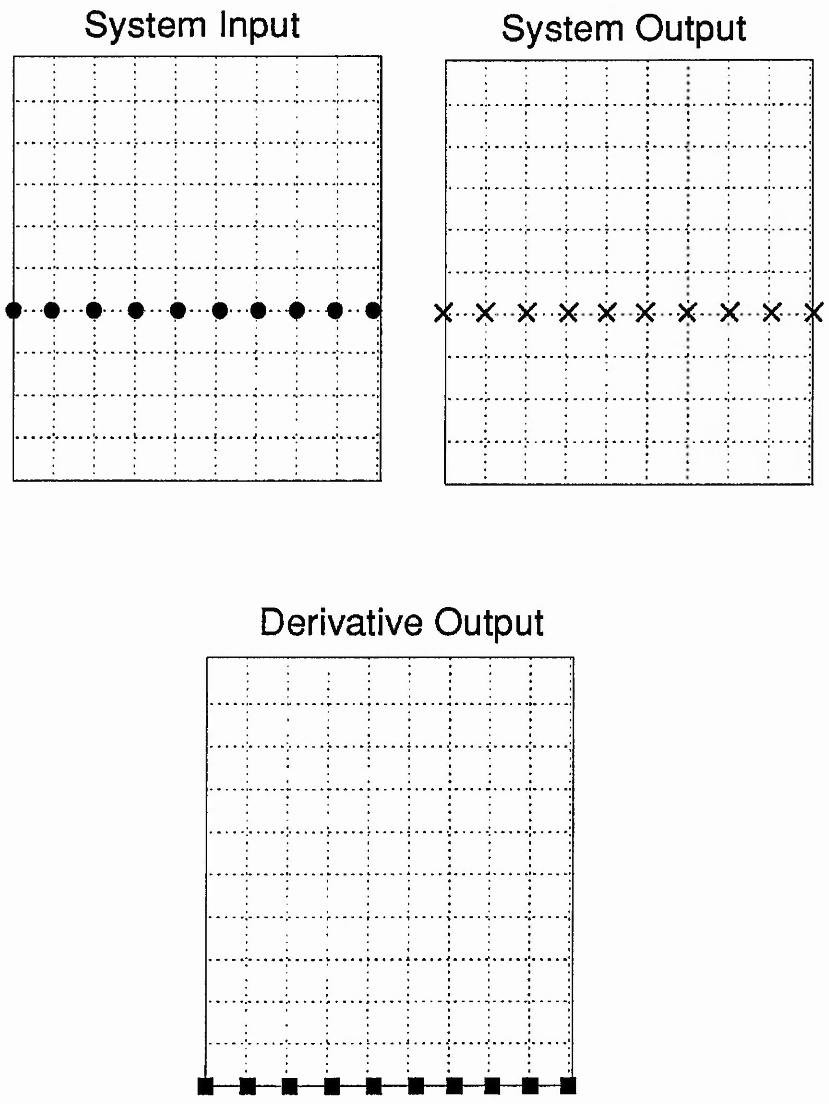


Figure 5.11 Processing Of A Constant Input

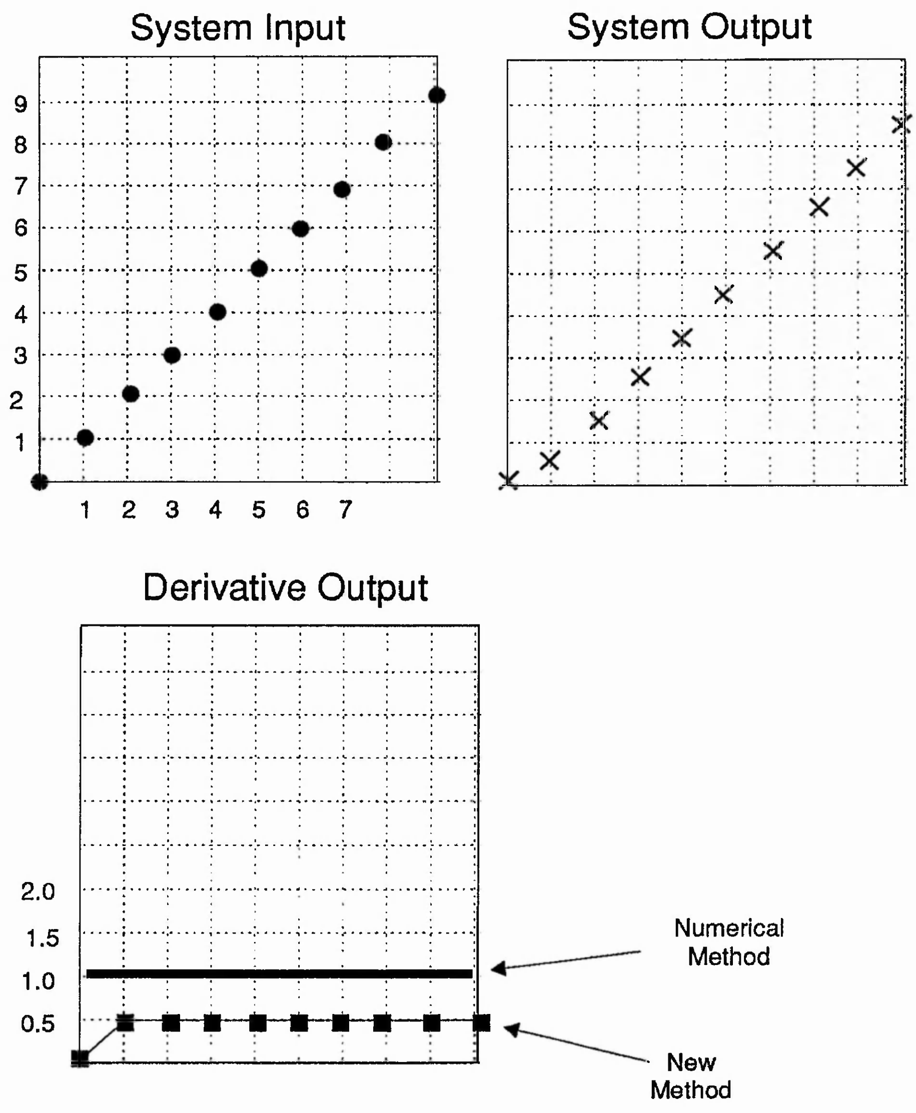


Figure 5.12 Processing Of A Linear Input

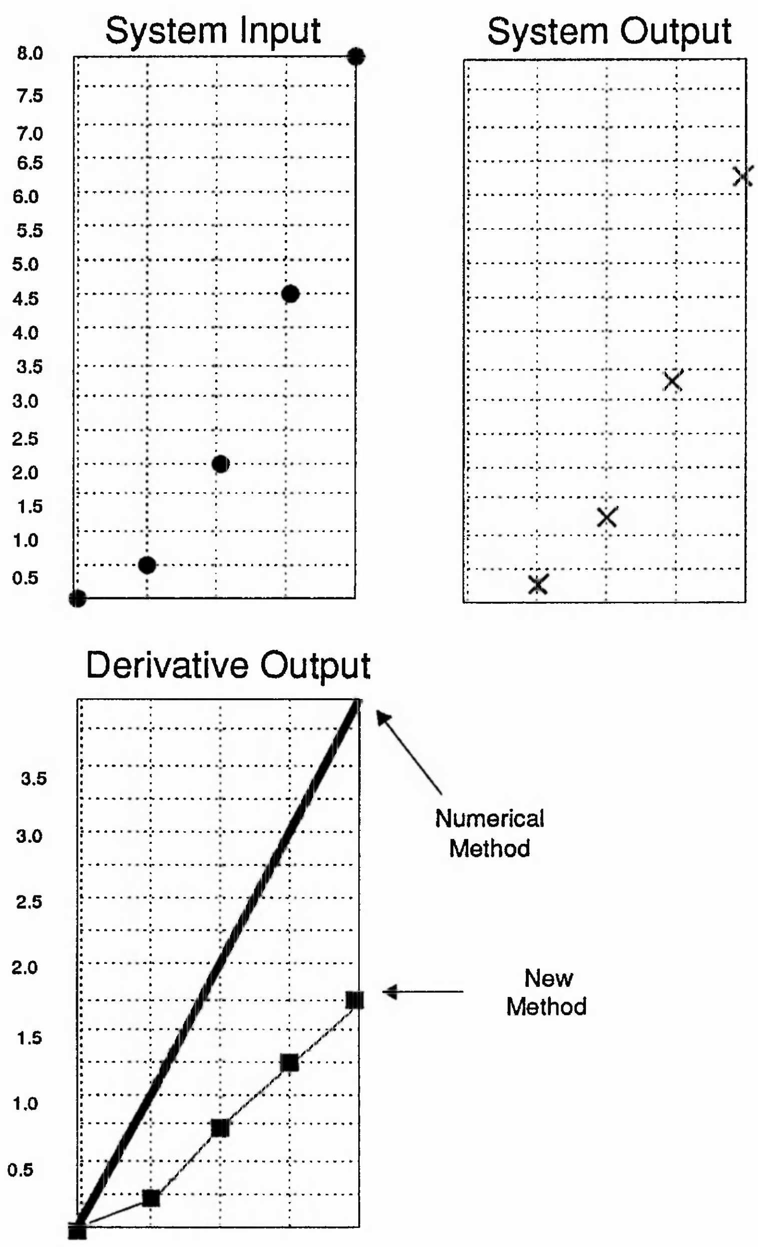
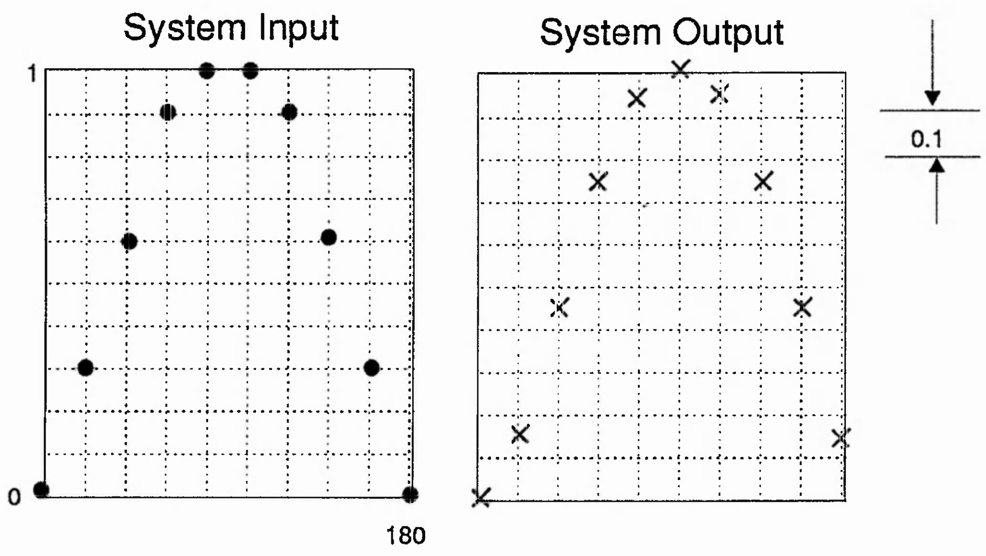


Figure 5.13 Processing Of A Quadratic Input



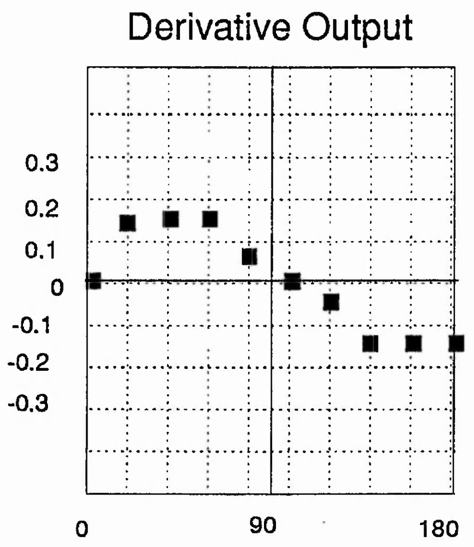


Figure 5.14 Processing Of A Sine Input

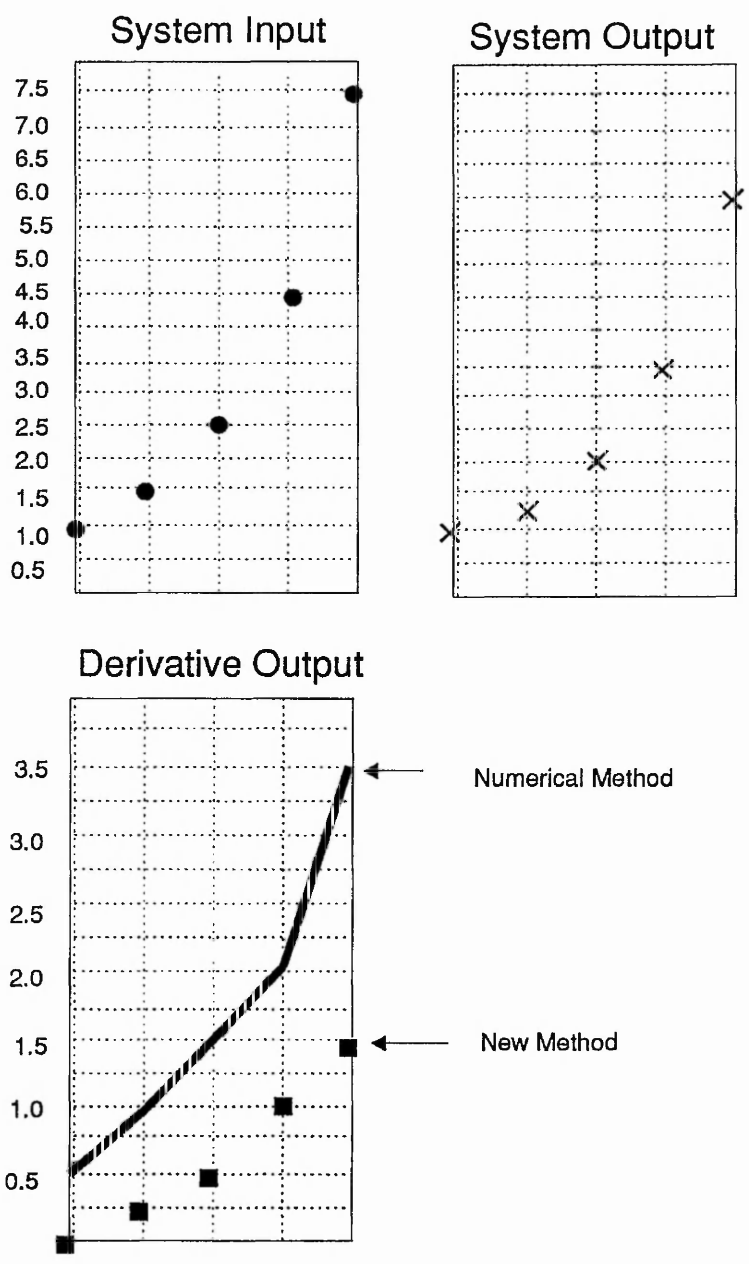


Figure 5.15 Processing Of An Exp(x/2) Input

hardware platform the neural network is running on.

Typical computation speeds of the pictorial integration process by these two approaches running on single processor machines are illustrated in Figure 5.16. For input pattern sizes less than 10 the neural network approach, on average, is roughly 160mS faster. As the pattern dimension is increased to 10 bits, the conventional approach gives a shorter computation speed.

Using a single processor the speed of the neural network approach is limited by the size of the input. This is due to the increasing number of neurons needed to be processed in serial. Figure 5.17 shows that the number of neuron increases non linearly with the dimension of the input patterns. In order to overcome this limitation, parallel processing of neural networks is desirable.

5.4 Performance Evaluation Of The Parallel Algorithm

Parallel processing and direct hardware implementation of neural networks can significantly reduce processing time. A parallel processing method for a distributed memory multiprocessor system was proposed in chapter 3. Software simulation of this algorithm is possible with a general purpose single processor machine. However, without actually

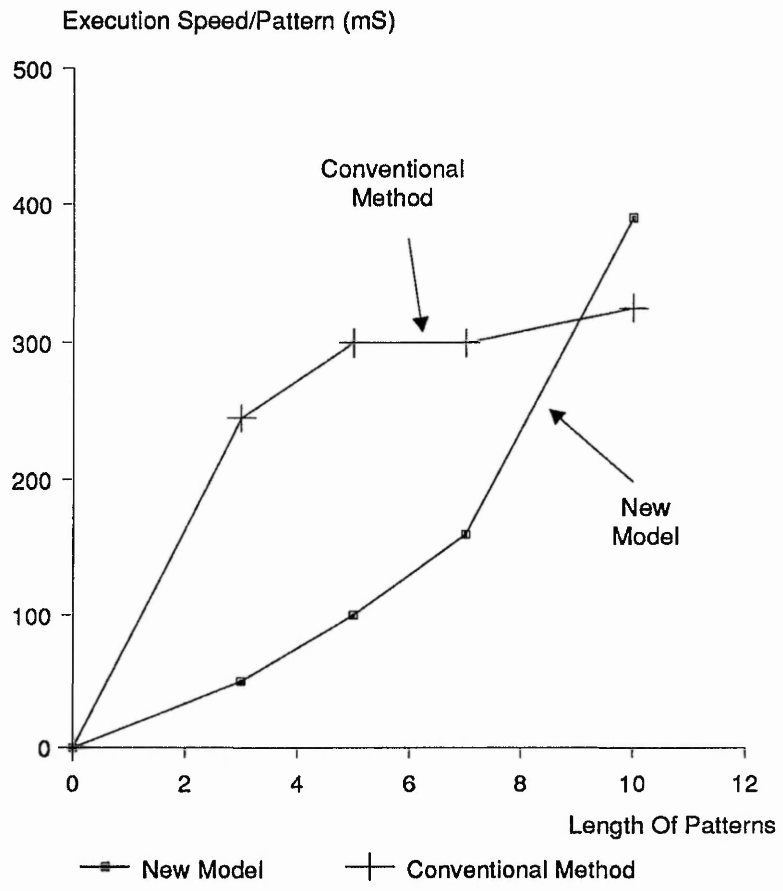


Figure 5.16 Computation Speed Of The Pictorial Integration Process

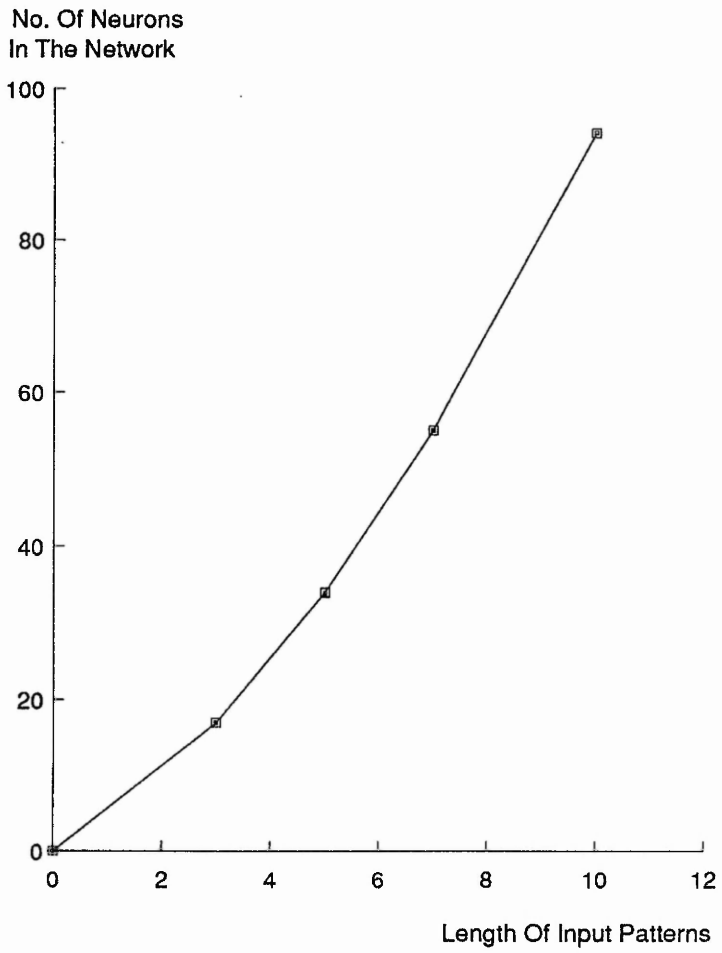


Figure 5.17 Size Of Net Versus Pattern Dimension

implementing the algorithm on the target hardware it is impractical to measure the performance of this algorithm by software simulation.

In order to provide a better method for evaluating the performance of the proposed technique a performance model is proposed. Developing a performance model yields several important benefits. Execution times for different topologies and learning algorithms can be estimated without having to execute the programs on the target hardware. The performance model can also predict the optimal number of processors to use for a given problem and reveal the importance of specific machine features. From such analysis it would be possible to identify the areas where further improvements could be made.

Using the proposed parallel processing algorithm the total time to compute all the outputs of a two layers network on the target multiprocessor system is calculated as follow :-

$$\begin{aligned} \textit{Total-Time} = {} & \textit{Total-Communication-Time} + \\ & \textit{MAX}[\ \textit{Node-Computation-Time}\ ] \qquad (5.29) \end{aligned}$$

The total-Communication-Time is equal to the time it takes to copy neuron outputs from active to passive nodes. The MAX[Node-Computation-Time] denotes the computation time on weights and on neurons of a processor node housing the maximum number of neurons.

Neurons within the same layer are partitioned into different processor nodes. If the number of neurons (n) is less than or equal to the number of processor nodes (p) then the total time to complete one updating process in each processor is the same. On the other hand, if n is larger than p and n/p is not a integer value then the number of neurons partitioned into p processor nodes will not be equal. One of the processor nodes will contain [integer(n/p) + 1] neurons while all other nodes contain [integer(n/p)] neurons.

In order to ensure correct operation, processor nodes must finish all their updating processes before communication is allowed, equation (5.29) should therefore either contain the term [integer(n/p)] or [integer(n/p) + 1] when the number of neuron is larger than the number of available processor nodes.

In the case of single processor systems the total time to compute all (n) neuron outputs of a two layer network may be expressed as :-

$$Total\text{-}Time = \sum_{1}^{n} (WeightxInput + Neuron\text{-}Time) \qquad (5.30)$$

As an example, for a two layer fully connected network with n_g and n_{g+1} neurons in each successive layer mapped on to p processors, n_g denotes neurons from the lower layer and n_{g+1} neurons from the higher layer. The total forward execution time can be expressed as :-

if $n_{g+1} <= p$

$$T_{total} = (n_g - 1)\,t_a + n_g t_m + t_f + n_g t_{comm} \qquad (5.31)$$

if $n_{g+1} > p$

$$T_{total} = [integer(\frac{n_{g+1}}{p}) + 1]\,[(n_g - 1)\,t_a + n_g t_m + t_f + n_g t_{comm}] \qquad (5.32)$$

where t_a denotes time to add two floating point numbers, t_m denotes time to multiply two numbers, t_f denotes time to compute a neuron output function(s) and t_{comm} denotes time needed to transfer one floating point number from one processor to another.

The performance model for the forward computation is applicable to most of the commonly used networks. The majority of commonly used neural network models consist of forward computation as well as backward learning phases.

The best way to demonstrate the general method to calculate the learning time is by an example. In the following example the backpropagation model is considered. The architecture of the network consists of 3 layers with n_g, n_{g+1}, n_{g+2} neurons in each successive layer. The layer n_g is the first input layer, neurons in this layer perform no computation and are used for holding input data only. p processors are available in the system.

Forward Computation Phase

$T_{forward}$ = *Total Neuron Computation Time Of Layer g+1, g+2*

Case I: $n_1 > p$

Let k = integer$(n_1/p)+1$

$$t_{g+1} = kt_f + n_g t_m k + (n_g-1)kt_a + n_g t_{comm}$$

$$t_{g+2} = kt_f + n_{g+1} t_m k + (n_{g+1}-1)kt_a + n_{g+1} t_{comm}$$

$$T_{forward}^{total} = 2k(t_f-t_a)+(n_g n_{g+1})(t_m k+t_{comm}+kt_a) \qquad (5.33)$$

Case II: $n_1 <= p$

$$t_{g+1} = t_f + n_g t_m + (n_g-1)t_a+n_g t_{comm}$$

$$t_{g+2} = t_f + n_{g+1} t_m + (n_{g+1}-1)t_a+n_{g+1} t_{comm}$$

$$T_{forward}^{total} = 2(t_f+t_a)+(n_g n_{g+1})(t_m+t_{comm}+t_a) \qquad (5.34)$$

Backward Computation Phase

$$Time^{total}_{backward} = \text{Time For Errors} + \text{Time For Weight Changes}$$

$$= t_e + t_w$$

Case III: $n_i > p$

$$t_e = n_{g+2} t_{comm} + (n_{g+1}+4) k t_m + t_a k (n_{g+2}+2)$$

$$t_w = (t_a + 2t_m) k (n_{g+2} + n_{g+1})$$

$$T^{total}_{backward} = t_m k (3n_{g+1}+4) + 2n_{g+2} k (t_m + t_a) + t_a k (2 + n_{g+1}) + n_{g+2} t_{comm} \quad (5.35)$$

Case IV: $n_i <= p$

$$t_e = n_{g+2} t_{comm} + (n_{g+1}+4) t_m + t_a (n_{g+2}+2)$$

$$t_w = (t_a + 2t_m)(n_{g+1} + n_{g+2})$$

$$T^{total}_{backward} = t_m (3n_{g+1}+4) + 2n_{g+2} (t_m + ta) + t_a (2 + n_{g+1}) + n_{g+2} t_{comm} \quad (5.36)$$

FOR UNIPROCESSOR SYSTEMS

Forward Computation Phase

$$T^{total}_{forward} = (n_{g+1} + n_{g+2}) t_f + (n_g + n_{g+2}) n_{g+1} (t_m + t_a) - 2t_a \quad (5.37)$$

__Backward Computation Phase__

$$T^{total}_{backward}=n_{g+2}(t_m(2+3n_{g+1})+2t_a(1+n_{g+1}))+2t_m+(2t_m+t_a)n_{g+1}n_g$$

(5.38)

Using the above performance equations for a given network topology, n_g, n_{g+1}, and n_{g+2}, the performance of the parallel processing algorithm can be evaluated, as the parameters t_m, t_a, t_f, and t_{comm}, of the target hardware are constant and can be measured independently. The theoretical forward and backward computation speed of various sizes of network are shown in Figure 5.18 to 5.20. In all cases the performances are calculated under the same assumptions, where n_g, n_{g+1}, and n_{g+2} are equal and their sum is represented by (n), and the target hardware is assumed to have the following computation time units: $t_a=1$, $t_m=2$, $t_f=3$, and $t_{comm}=4$.

The forward and backward computation speed performance of the parallel processing algorithm are shown separately, Figure 5.19 and 5.20. This is because once learning (backward computation) is complete for a specific application, the application can be hard-wired, and the network may execute only the forward computation.

For the distributed simulation of multilayered neural nets,

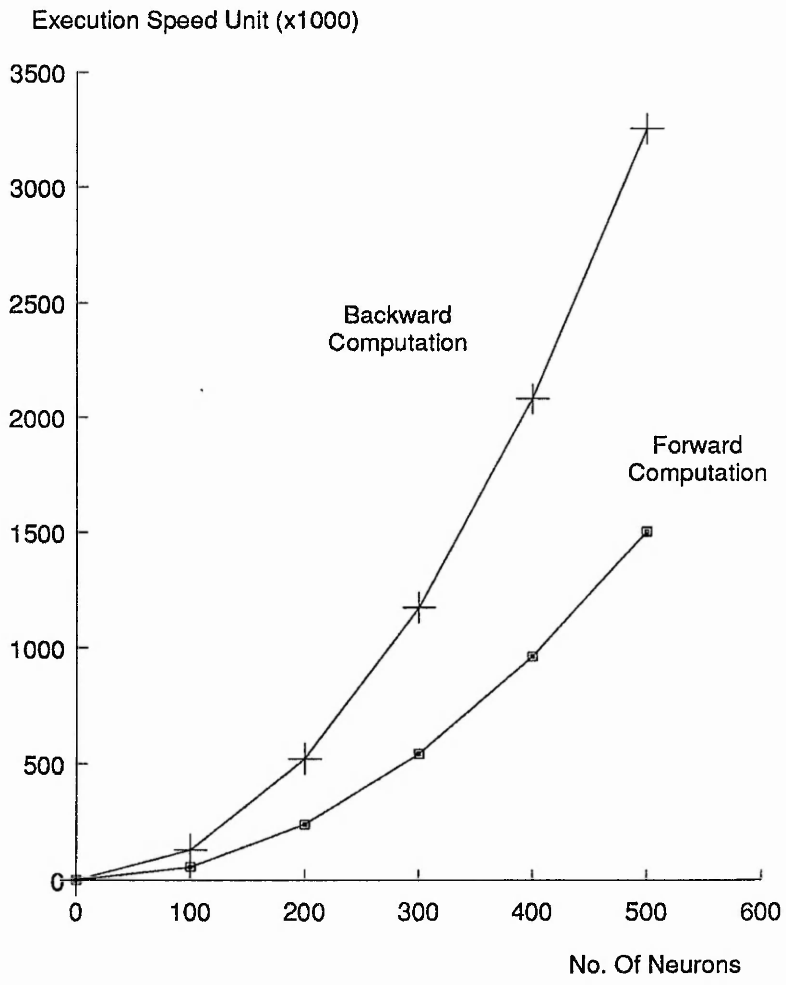


Figure 5.18 Speed Performance Of A Single Processor

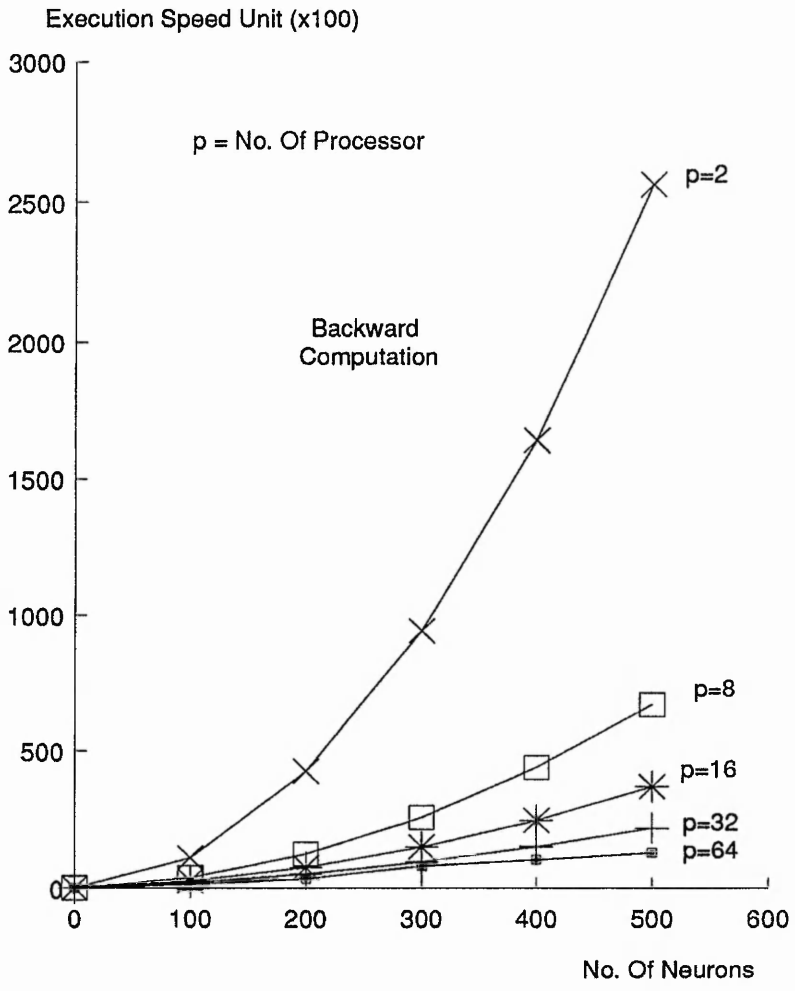


Figure 5.19 Speed Performance Of Parallel Processing On A Multiprocessor System

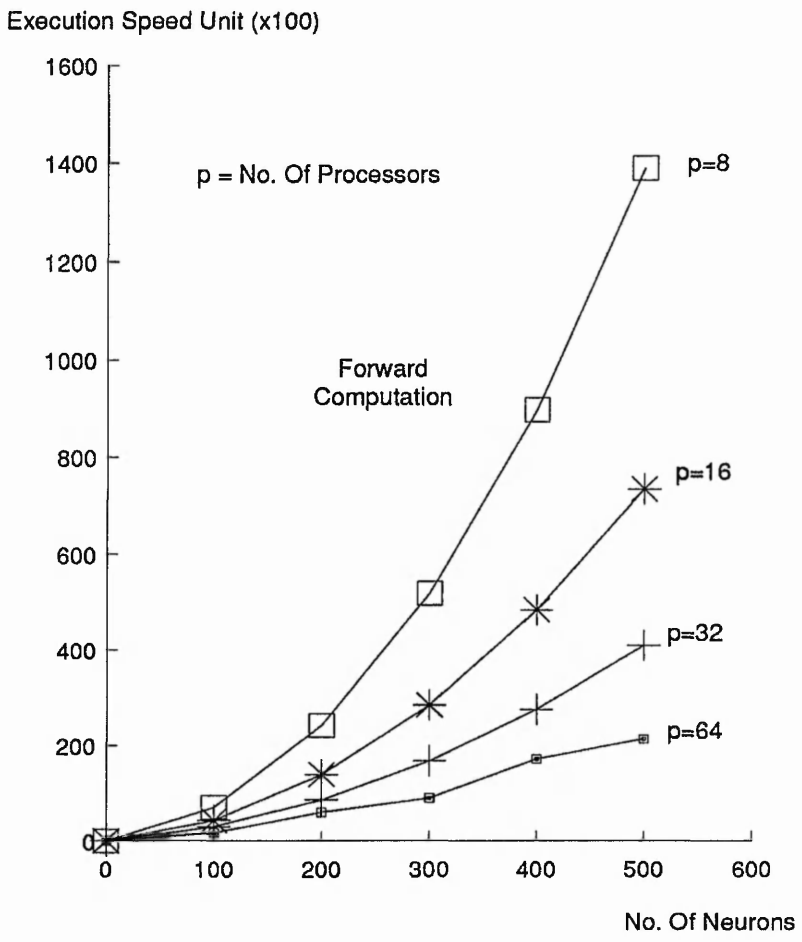


Figure 5.20 Speed Performance Of Parallel Processing On A Multiprocessor System

it is seen from Figure 5.19 and 5.20 that there is a 'diminishing return' effect as more processors are used due to inter-processor communication, n_i x t_{comm}, such that though more processors are added to the simulation, the computation speed performance is not improved by a great deal. The speed performance of a network with 500 neurons running on different hardware platform is summarized in Table 5.4 for illustration.

Different network models have different learning mechanisms, thus the performance evaluation method of the learning process may vary. Whichever network model is used the same approach can still be adopted. The parallel processing algorithm and the equations for the speed performance evaluations provide convenient means for studying the effect of different number of neurons in the network, the communication and computational costs.

p = No Of Processor Time Unit x 1000		
p	Forward Computation	Backward Computation
1	150	3250
8	140	70
16	80	30
32	40	25
64	20	15

Table 5.4 Speed Comparisions

CHAPTER 6 CONCLUSIONS AND FURTHER RESEARCH

CHAPTER 6 CONCLUSIONS AND FURTHER RESEARCH

6.0 Conclusions

A neural network based algorithm to perform integration was presented. The new algorithm consists of a neural net model inspired by the vertebrate retina neuron organisation. The task under consideration is that of generating the integral of a given time trajectory, where the neural network model behave as a pictorial integrator operating within a closed loop system. The architecture of the neural integrator model is that of a multilayer feedforward type. Adaption techniques employed in the model include a mixture of implicit weight setting, supervised and unsupervised learning processes.

Simulation results showed that this multilayer neural net algorithm is capable of estimating the derivative of a continuous input profile pictorially even when no explicit formulation of the input profile is known. The work also showed that although the same quality of integration results can also be achieved by networks employing the Backpropagation algorithm, the new retinal neural net model has an advantage in that network convergence is much faster. The proposed neural net algorithm seems to offer the following characteristics :-

(1) It's real time closed loop structure yields consistent estimates of the derivatives of unknown input profiles. It provides accurate estimation for constant and linear inputs. Quadratic and exponential inputs are also acceptable. Sinusoidal input, however, did not produce such accurate results.

(2) The network equations are much simpler than other networks and easier to work with.

(3) Its convergence is fast when compared to existing model producing compatible results.

(4) When a general purpose single processor machine is used, the execution speed of the new model is several times faster than conventional numerical methods but limited by the input/output neuron dimensions. This is due to the additional components (neurons and connections) of the network as its dimensions are increased.

In order to reduce processing time and maintain the speed advantage over numerical methods a parallel processing algorithm for executing neural nets on a distributed memory multiprocessor system was devised and its performance evaluated. The speed up factor of this parallel algorithm seems very favourable when compared to general purpose single processor machines.

All the algorithms described were implemented in software. A software neural network simulation tool was developed to automatically handle the process of generating arbitrary network definitions thus simplifying the tedious task of synthesising neural networks. The simulation toolbox consists of a set of simulation specific data structures to provide a representation of the physical structure of a network and a set of procedural functions for the manipulation of the network data structures. Due to its transparency it is also useful as a network debuger and as an educational tool.

6.1 Further Research

Suggestions for further research may be summarized as follows :-

(1) One of the limitations of the new neural network algorithm is that it consists of off-line supervised learning. There is therefore a need to investigate the formulation of unsupervised learning paradigms to eliminate the off-line processing burden.

(2) Higher order integration/quadrature formulae, e.g. 4th

order Runge-Kutta and 5th order Gaussian, need to be considered for the new neural integrator to provide high accuracy integration and derivative estimation.

(3) Extensions to the proposed neural architecture need to be evaluated for more accurate integration and derivative estimation of trigonometric and exponential functions.

(4) The speed performance of the parallel processing algorithm is mainly limited by the performance of the target hardware. Special neural net coprocessors [Vindlacher,1992] need to be assessed for executing the relevant algorithms.

(5) The developed software simulation tool can be enhanced by encoperating a graphical interface. This is desirable not only because it increases the efficiency of creating more complex topologies by graphical drawing but more importantly to provide a visual representation of activities in the network. Such a presentation of the functioning of a neural net permits visual diagnoses of the behaviour of large network which is particularly useful for network development, debugging, and may provide valuable contributions for educational and commercial purposes.

REFERENCES

[Aldabass,1976]; D Al-Dabass.
"Parallel Processors In The Design And Simulation Of Dynamical Systems", Ph.D. Thesis, Department Of Electrical Engineering And Electronics, North Staffordshire Polytechnic, Stafford, England, 1976.

[Arsenault,1989]; H H Arsenault.
"Neural Network Model For Fast Learning And Retrieval," Optical Engineering, May 1989, Vol 28 No5, 1989.

[Arbib,1989]; M A Arbib.
"The Metaphorical Brain 2 Neural Networks And Beyond," Wiley Interscience, 1989.

[Abu-Mostafa,1986]; Y S Abu-Mostafa.
"Neural Networks For Computing," American Institute Of Physics. 1986.

[Abutaleb,1991]; A S Abutaleb.
"A Neural Network For Estimation Of Forces Acting On Radar Targets," Neural Networks, Vol. 4 pp667-678, 1991.

[Aleksander,1990]; I Aleksander, H Morton
"An Introduction To Neural Computing", Chapman And Hall 1990.

[Ashurst,1983]; F G Ashurst.
"Pioneers Of Computing," Frederick Muller. 1983.

[Callatay,1989]; A de Callatay.
"Biological Aspects Of Neural Networks," pp1-15 1989.

[Deprit,1989]; E Deprit.
"Implementating Recurrent Backpropagation On The Connection Machine," Neural Network 2, 1989 pp 295-314.

[Dobnikar,1989]; A Dobnikar, D Podbregar,
"Optimal Visual Tracking With Artifical Neural Network,"IEEE International Conference On Neural Networks. Vol II San Diego, 1989 pp275-279.

[Dowling,1987]; J E Dowling.
"The Retina: An Approachable Part Of The Brain," Cambridge: Harvard University Press, 1987, Chapter 3.

[Gorman,1988]; R P Gormann And T J Sejnowski.
"Analysis Of Hidden Units In a Layered Network Trained To Classify Sonar Targets. Neural Networks 1, pp75-89 1988.

[Graff,1986]; H P Graf, L D Jackel, R E Howard, and B L Straughn, "VLSI Implementation Of A Neural Network Memory

With Several Hundred Neurons," Proc. Of Conf. On Neural Networks For Computing, 1986.

[Grossberg,1987]; S Grossberg, A Carpenter.
"Self-organization Of Stable Category Recognition Codes For Analog Input Patterns," Applied Optics, Vol 26, No 23, 1 Dec 1987.

[Hammes,1989]; M R Hames.
"A Node Interface For Parallel Processing,"Ph.D Thesis, Nottingham Polytechnic 1989.

[Hebb,1949];
The Organisation Of Behavior, Wiley, N.Y.,1949.

[Hicklin,1988]; J Hicklin And H Demuth.
"Modeling Neural Networks On The MPP," Proc. 2nd Symp. Frontiers Of Massively Parallel Computation 1988 pp39-42.

[Hinton,1986]; G E Hinton And T Sejnowski.
"Learning And Relearning in Boltzmann Machines," Chapter 7, Parallel Distributed Processing Cambridge, MA: MIT Press, 1986.

[Hopfield,1982]; J J Hopfield.
"Neural Network And Physical Systems With Emergent Collective Computational Abilities," Proc. Natl. Acad. Sci. USA, Vol.79, pp2554-2558, April 1982.

[Jenhwa,1989]; G Jenhwa, C Vladimir.
"A Solution To The Inverse Kinematic Problem In Robotics Using Neural Network Processing," IEEE International Conference On Neural Networks. Vol II San Diego, 1989.

[Kohonen,1990]; T Kohonen.
"The Self-organizing Map", Proc. Of The IEEE, Vol. 78. No.9 September 1990.

[Lambe,1986]; J Lambe, A. Moopenn, and A P Thakoor.
"Error Correction And Asymmetry In A Binary Memory Matrix," Proc. Of Conf. On Neural Networks For Computing, 1986.

[Lee,1988]; K Lee.
"Neural Network Applications In Handwritten Symbol Understanding," SPIE Vol. 1002 Intelligent Robots And Computer Vision: Seventh In A Series 1988.

[Lippmann,1987]; R P Lippmann.
"An Introduction To Computing With Neural Network," IEEE ASSP Magazine, Vol. 4, April 1987.

[McCulloch,1943]; W S McCulloch and W Pitts.
"A Logical Calculus Of The Ideas Immanent In Nervous Activity," Bull. Math. Biophys., 5, pp115-133 1943.

[Minsky,1968]; M Minsky and S Papert, Perceptrons

(Cambridge, MA: MIT Press, 1968).

[Miyamoto,1988]; H Miyamoto, M Kawato, T Setoyama, and R Suzuki, "Feedback Error Learning Neural Network For Trajectory Control Of a Robotic Manipulator," Neural Networks, 1(3), 251-265, 1988.

[Newman,1990]; W C Newman.
"Detecting Speech With An Adaptive Neural Network," Electronic Design pp79-89 March 22, 1990.

[Pomerleau,1988]; D A Pomerleau et al.
"Neural Network Simulation At Warp Speed: How We Got 17 Million Connections Per Second," Proc. IEEE 2nd Internat. Conf. Neural Networks II 1988 pp143-150.

[Rosenblatt,1962]; F Rosenblatt.
"Principle Of Neuro- Dynamics". Spartan, 1962.

[Rumelh,1986(a)]; D E Rumelhart, G E Hinton, And R J Williams, "Learning Internal Representation By Error Propagation" Parallel Distributed Processing: Exploration In The Microstructure Of Cognition. Vol. 1: Foundations. MIT Press, 1986.

[Rumelhart,1986(b)]; D E Rumelhart, G E Hinton, and R J Williams, "Learning Internal Representations By Back Propagating Errors," Nature, 323,533-536, 1988.

[Saerens,1989]; M Saerens, A Soquet.
"A Neural Controller," IEEE International Conference On Neural Networks. San Diego, 1989.

[Sage,1986]; J P Sage, K. Thompson, And R S Withers.
"A Neural Network Integrated Circuit With Synapses Based On CCD/NMOS Technology," Proc. Of Conf. OnNeural Networks For Computing, 1986.

[Sejnowski,1987]; T J Sejnowski, C R Rosenberg.
"NET talk: A Parallel Network That Learns To Read Aloud," Complex Systems Vol.1, pp145-168,1987.

[Strange,1989]; P G Strange.
"Basic Brain Mechanisms: A Biologist's View Of Neural Networks": IEE Colloquim 18 May 1989.

[Sutton, 1981]; R S Sutton, A G Barto.
"Toward A Modern Theory Of Adaptive Networks: Expectation And Prediction. Psychol. Rev 99: 135-170, 1981.

[Vindlachenuvu,1992]; P Vindlachenuvu.
"Simulation Of A Neural Node coprocessor,"Ph.D Thesis, Department Of Computing. Nottingham Polytechnic,1992.

[Widrow,1990]; B Widrow.
"30 Years Of Adaptive Neural Networks: Perceptron, Madaline, And Backpropagation," Proc. Of The IEEE, Vol. 78, No.9 September 1990.

[Yamada,1989]; K Yamada, H Kami, J Tsukumo, and T Temma, "Handwritten Neural Recognition By Multi-layered Neural Network With Improved Learning Algorithm," IEEE International Conference On Neural Networks. Vol II San Diego, 1989.

[Yoon,1989]; B L Yoon.
"Artifical Neural Network Technology" pp4-16 1989.

<u>APPENDIX</u>

Appendix (A.1) Associative Memory Models

The connection matrix of a typical associative memory model is derived from a set of vectors to be memorized. When given an input, such a network will evolve and become stable at the nearest memory vector from the input. Hopfield and Bidirectional Associative Memory (BAM) are models of associative memory.

The Hopfield model is a recurrent non-layer type in which the outputs of neurons are either +1 or -1 according to the threshold law. For example, if the input to a neuron is positive its output is +1 and if negative its output is -1. When the input equals zero (threshold) the neuron maintains its current state. In the case of BAM, neuron outputs are either +1 or 0. A BAM uses two fields of neurons. Neurons in both fields are both input and output neurons. The main difference between Hopfield and BAM models is that Hopfield Models are autoassociative memory type while BAMs are heteroassociative memory models.

The main evolution rule for a Hopfield memory is :-

$$O_i(t+1) = sgn\left(\sum_{j=1}^{N} T_{ij} O_j(t)\right) \qquad \text{(a.1)}$$

where T_{ij} is the N x N connection matrix.

The connection of T_{ij} is a learning process. If vector Y is to be associated with vector X, the connection matrix is constructed by finding the outer product of these two vectors :-

$$T = Y^t X \qquad \text{(a.2)}$$

where 't' denotes the transpose of a column vector. If X and Y are the same then it is an autoassociative memory model (Hopfield) and if X ≠ Y then it is a heteroassociative memory (BAM).

If there are k patterns (vectors) to be stored in the matrix, then T_{ij} can be derived by adding the outer product of all the individual memory vectors :-

$$T_{ij} = \sum_{p=1}^{k} (Y_i^p)^t \, (X_j^p) \qquad \text{(a.3)}$$

In both cases, once the connection matrix is determined, the network may be used to produce the desired output vector, even when given an input that may be partially correct. To do this, the outputs of the network are initially set to the values of an input vector. Next the input is removed and the network is allowed to run freely until there are no changes in the network (outputs stablized).

Appendix (A.2) Backpropagation Or Generalized Delta Rule

Backpropagation probably represents the most widely used learning algorithm. It applies to feedforward networks with three type of neurons: input neurons, hidden neurons carrying an internal representation, and output neurons. The description here follows the version given by [Rumelbh,1988]. The dynamics of a network is determined by a local update rule :-

$$S_i(t+1) = f(\sum_j W_{ij} S_j(t)) \qquad (a.4)$$

where S denotes the state of a neuron and f() is a nonlinear activation function.

Neural networks learn from examples which are presented many times, and learning procedure can be viewed as a strategy to minimize a suitably defined error function E. In this case it is a gradient decent method, each weight is changed by an amount proportional to the respective gradient of E :-

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} \qquad (a.5)$$

and the procedure is repeated for a new learning example until E is minimized to a satisfactory level.

In its original form :-

$$E = \frac{1}{2}\sum_{i}(T_i - O_i)^2 \qquad (a.6)$$

For a weight W_{ij} from a (input or hidden) neuron j to an output neurin i :-

$$-\frac{\partial E}{\partial W_{ij}} = (T_i - O_i)\, f'(\sum_{k} W_{ik}S_k)\, S_j \qquad (a.7)$$

where f'() is the derivative of the nonlinear activation function, and for weights which do not connect to an output neuron, the gradient can successively be determined by applying the chain rule of differentiation.

Thus for output neurons the error signal is :-

$$\delta_j = (t_j - o_j)\, f'_j(net_j) \qquad (a.8)$$

where $net_j = \Sigma W_{ij} O_i$.

Finally the error signal for hidden neurons for which there is no specified target is determined recursively in terms of the error signals of neurons to which it directly connects and the weights of those connections. For hidden neurons :-

$$\delta_j = f'_j(net_j) \sum_{k} \delta_k W_{kj} \qquad (a.9)$$

For nonlinear logistic neuron output function of the form:-

$$O_j = \frac{1}{1 + e^{-(\sum_i W_{ij} O_j)}} \tag{a.10}$$

$$\frac{\partial O_j}{\partial net_j} = O_j(1 - O_j)$$

The error signal for an output neuron is :-

$$\delta_j = (t_j - O_j) O_j (1 - O_j) \tag{a.11}$$

and the error signal for arbitrary hidden neuron is given by :-

$$\delta_j = O_j(1 - O_j) \sum_k \delta_k W_{kj} \tag{a.12}$$

Appendix (B.1) Data Structures For Mapping Of Arbitrary Network

```
struct SYNAPE{
       float weight;
       float reg;
       struct SYNAPE *nexsynape;
       struct NEURON *fromneuron;
       };

struct NEURON{
       int neuronid;
       float netval;
       float outval;
       float actval;
       float error;
       float reg;
       struct SYNAPE *firsynape;
       struct NEURON *nexneuron;
       };

struct GROUP{
       char grpid[10];
       int nofneuron;
       float (*actfcn)();
       float (*outfcn)();
       float (*netfcn)();
       struct NEURON *firsneuron;
       struct GROUP *nexgroup;
       };
```

Appendix (B.2) Source Code Listing Of Simulation Toolbox

```
#include "neurotoo.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <alloc.h>

float crate;
int nactive,nofinput;
struct GROUP *firsgroup=(struct GROUP *)NULL;

/* Standard Support Code Mapping Of Network */

void quit(void)
{
 printf("Not Enough Memory - Process Terminated !!!");
 exit(0);
}

void    CreateGroup(int    nofcell,char    *name,float
(*fcnact)(),float (*fcnout)(),float (*fcnnet)())
{
 struct GROUP *thisgroup;

 if (firsgroup==(struct GROUP *)NULL)
 {
  thisgroup=firsgroup=NewGroup(nofcell,name,fcnact,
  fcnout,fcnnet);
  thisgroup->firsneuron=NeuronList(thisgroup);
 }
 else
 {
  thisgroup=firsgroup;
  while (thisgroup->nexgroup!=(struct GROUP *)NULL)
      thisgroup=thisgroup->nexgroup;
  thisgroup->nexgroup=NewGroup(nofcell,name,fcnact,
  fcnout,fcnnet);
  thisgroup=thisgroup->nexgroup;
  thisgroup->firsneuron=NeuronList(thisgroup);
 }
}

struct   GROUP   *NewGroup(int   nofcell,char   *name,float
(*fcnact)(),float (*fcnout)(),float (*fcnnet)())
{
 struct GROUP *newgpptr;

 if ((newgpptr=(struct GROUP *)malloc(sizeof(struct
      GROUP)))!=NULL)
 {
   newgpptr->nofneuron=nofcell;
   newgpptr->actfcn=fcnact;
   newgpptr->outfcn=fcnout;
   newgpptr->netfcn=fcnnet;
   strcpy(newgpptr->grpid,name);
   newgpptr->firsneuron=(struct NEURON *)NULL;
   newgpptr->nexgroup=(struct GROUP *)NULL;
 }
 else
  quit();
}

struct NEURON *NeuronList(struct GROUP *gptr)
{
 int i;
 struct NEURON *thisneuron;

 for (i=0; i<gptr->nofneuron; i++)
 {
  if (gptr->firsneuron==(struct NEURON *)NULL)
  {
     thisneuron=gptr->firsneuron=CreateNeuron(i+1);
  }
  else
  {
   thisneuron=gptr->firsneuron;
   while (thisneuron->nexneuron!=(struct NEURON *)NULL)
      thisneuron=thisneuron->nexneuron;
   thisneuron->nexneuron=CreateNeuron(i+1);
  }
 }
 return((struct NEURON *)gptr->firsneuron);
}

struct NEURON *CreateNeuron(int id)
{
 float outval;
 struct NEURON *newcellptr;

 if ((newcellptr=(struct NEURON *)malloc(sizeof(struct
 NEURON)))!=NULL)
 {
   newcellptr->firsynape=(struct SYNAPE *)NULL;
   newcellptr->nexneuron=(struct NEURON *)NULL;
   newcellptr->actval=-1.0;
   newcellptr->outval=-1.0;
   newcellptr->netval=-1.0;
   newcellptr->error=0.0;
   newcellptr->reg=0.0;
   newcellptr->neuronid=id;
   return((struct NEURON *)newcellptr);
 }
 else
   quit();
}

struct SYNAPE *CreateSynape(void)
{
 struct SYNAPE *newsyptr;

 if ((newsyptr=(struct SYNAPE *)malloc( sizeof(struct
  SYNAPE)))!=NULL)
  {
   newsyptr->weight=0.0;
   newsyptr->reg=0.0;
   newsyptr->nexsynape=(struct SYNAPE *)NULL;
   newsyptr->fromneuron=(struct NEURON *)NULL;
   return((struct SYNAPE *)newsyptr);
  }
  else
   quit();
}

struct GROUP *SearchGrp(char *grpname)
{
 struct GROUP *thisgptr;

 thisgptr=firsgroup;
 while(thisgptr!=(struct GROUP *)NULL)
 {
  if (strcmp(thisgptr->grpid,grpname)==0)
     return((struct GROUP *)thisgptr);
  thisgptr=thisgptr->nexgroup;
 }
 return((struct GROUP *)NULL);
}

struct NEURON *SearchNeuron(char *grpname,int cellid)
{
 struct GROUP *grp;
 struct NEURON *thecell;

 if ((grp=SearchGrp(grpname))!=(struct GROUP *)NULL)
 {
   thecell=grp->firsneuron;
   while(thecell!=(struct NEURON *)NULL)
   {
    if (thecell->neuronid==cellid)
       return((struct NEURON *)thecell);
    thecell=thecell->nexneuron;
   }
   printf("\nCell Not Found \n");
   runterror();
 }
 else
  runterror();
}

void runterror(void)
{
 printf("Run Time Error !!");
 exit(0);
}

void LinkGroup(char *from,char *to)
{
 struct NEURON *fromneuron,*toneuron;
 struct GROUP *fromgrp,*togrp;
 struct SYNAPE *newsynape,*thissynape;

 randomize();
 if ((fromgrp=SearchGrp(from))==(struct GROUP *)NULL ||
 (togrp=SearchGrp(to))==(struct GROUP *)NULL)
    runterror();
 fromneuron=fromgrp->firsneuron;
 toneuron=togrp->firsneuron;
 if (toneuron==(struct NEURON *)NULL || fromneuron==(struct
 NEURON *)NULL)
    runterror();
 while(toneuron!=(struct NEURON *)NULL)
 {
  while(fromneuron!=(struct NEURON *)NULL)
  {
    if (toneuron->firsynape==(struct SYNAPE *)NULL)
       thissynape=toneuron->firsynape=CreateSynape();
    else
    {
     thissynape=toneuron->firsynape;
     while(thissynape->nexsynape!=(struct SYNAPE *)NULL)
       thissynape=thissynape->nexsynape;
     thissynape->nexsynape=CreateSynape();
     thissynape=thissynape->nexsynape;
    }
    thissynape->fromneuron=fromneuron;
    fromneuron=fromneuron->nexneuron;
  }
  fromneuron=fromgrp->firsneuron;
  toneuron=toneuron->nexneuron;
 }
}

void InitWeight(float (*initwgtfcn)(),char *grpname)
{
 struct SYNAPE *thesynap;
 struct GROUP *thisgroup;
 struct NEURON *thisneuron;

 if ((thisgroup=SearchGrp(grpname))!=(struct GROUP *)NULL
 && thisgroup->firsneuron!=(struct NEURON *)NULL)
 {
  thisneuron=thisgroup->firsneuron;
  while(thisneuron!=(struct NEURON *)NULL)
  {
   thesynap=thisneuron->firsynape;
   NofInputData(thesynap);
   while(thesynap!=(struct SYNAPE *)NULL)
   {
    thesynap->weight=(*initwgtfcn)(thisneuron,thesynap);
    thesynap=thesynap->nexsynape;
   }
   thisneuron=thisneuron->nexneuron;
  }
 }
 else
  runterror();
}

void ActivateGroup(char *grpname)
{
 struct GROUP *thisgroup;
 struct NEURON *thisneuron;

 if ((thisgroup=SearchGrp(grpname))!=(struct GROUP *)NULL
 && thisgroup->firsneuron!=(struct NEURON *)NULL)
 {
  thisneuron=thisgroup->firsneuron;
  while(thisneuron!=(struct NEURON *)NULL)
  {
   thisneuron->netval=(*thisgroup->netfcn)(thisneuron);
   thisneuron->actval=(*thisgroup->actfcn)(thisneuron->
   netval);
   thisneuron->outval=(*thisgroup->outfcn)(thisneuron->
   actval);
   thisneuron=thisneuron->nexneuron;
  }
 }
 else
  runterror();
}

void CalculateError(char *grpname,char *soursegrp,float
(*errfcn)())
{
 static struct GROUP *thisgroup;
 static struct NEURON *thisneuron;

 if ((thisgroup=SearchGrp(grpname))==(struct GROUP *)NULL
 || thisgroup->firsneuron==(struct NEURON *)NULL)
   runterror();
 thisneuron=thisgroup->firsneuron;
 while (thisneuron!=(struct NEURON *)NULL)
 {
  thisneuron->error=(*errfcn)(thisneuron,soursegrp);
  thisneuron=thisneuron->nexneuron;
 }
}

void AdjustWeight(float learnrate,char *grpname,float
(*learnfcn)(),int learntype)
{
 struct GROUP *thisgroup;
 struct NEURON *thisneuron;
 struct SYNAPE *thissynape;

 if ((thisgroup=SearchGrp(grpname))==(struct GROUP *)NULL
|| thisgroup->firsneuron==(struct NEURON *)NULL)
   runterror();
 thisneuron=thisgroup->firsneuron;
 while (thisneuron!=(struct NEURON *)NULL)
```

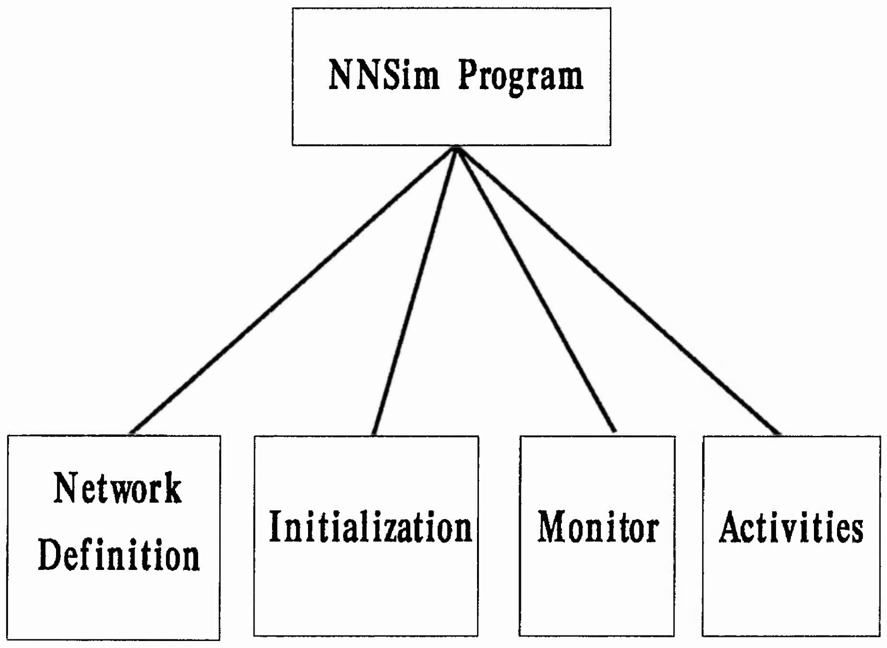
```
  {
   if (learntype==TEACHER)
   {
    if  ((thissynape=thisneuron->firsynape)==(struct  SYNAPE
    *)NULL)
        runterror();
    while(thissynape!=(struct SYNAPE *)NULL)
    {
     thissynape->weight=(*learnfcn)(&learnrate,thisneuron,
     thissynape);
     thissynape=thissynape->nexsynape;
    }}
   else
   {
    if (learntype==SELFORGANISE)
    {
     if (thisneuron->outval==1.0)
     {
      if ((thissynape=thisneuron->firsynape)==(struct SYNAPE
         *)NULL)
       runterror();
      {
       while(thissynape!=(struct SYNAPE *)NULL)
       {
        thissynape->weight=(*learnfcn)(&learnrate,
        thisneuron, thissynape);
        thissynape=thissynape->nexsynape;
       }
      }
      return;
     }
    }
   }
   thisneuron=thisneuron->nexneuron;
  }
 }


void  SetDataPattern(char  *grpname,char  *datafname,int
datatype)
{
 struct GROUP *thisgroup;
 struct NEURON *thisneuron;
 FILE *dfptr;
 int data;

 if ((thisgroup=SearchGrp(grpname))!=(struct GROUP *)NULL)
 {
   thisneuron=thisgroup->firsneuron;
   if ((dfptr=fopen(datafname,"rb"))!=NULL &&
   thisneuron!=(struct NEURON *)NULL)
   {
     while(thisneuron!=(struct NEURON *)NULL &&
     fread(&data,sizeof(int),1,dfptr)!=0)
     {
       if (datatype==INTYPE)
```

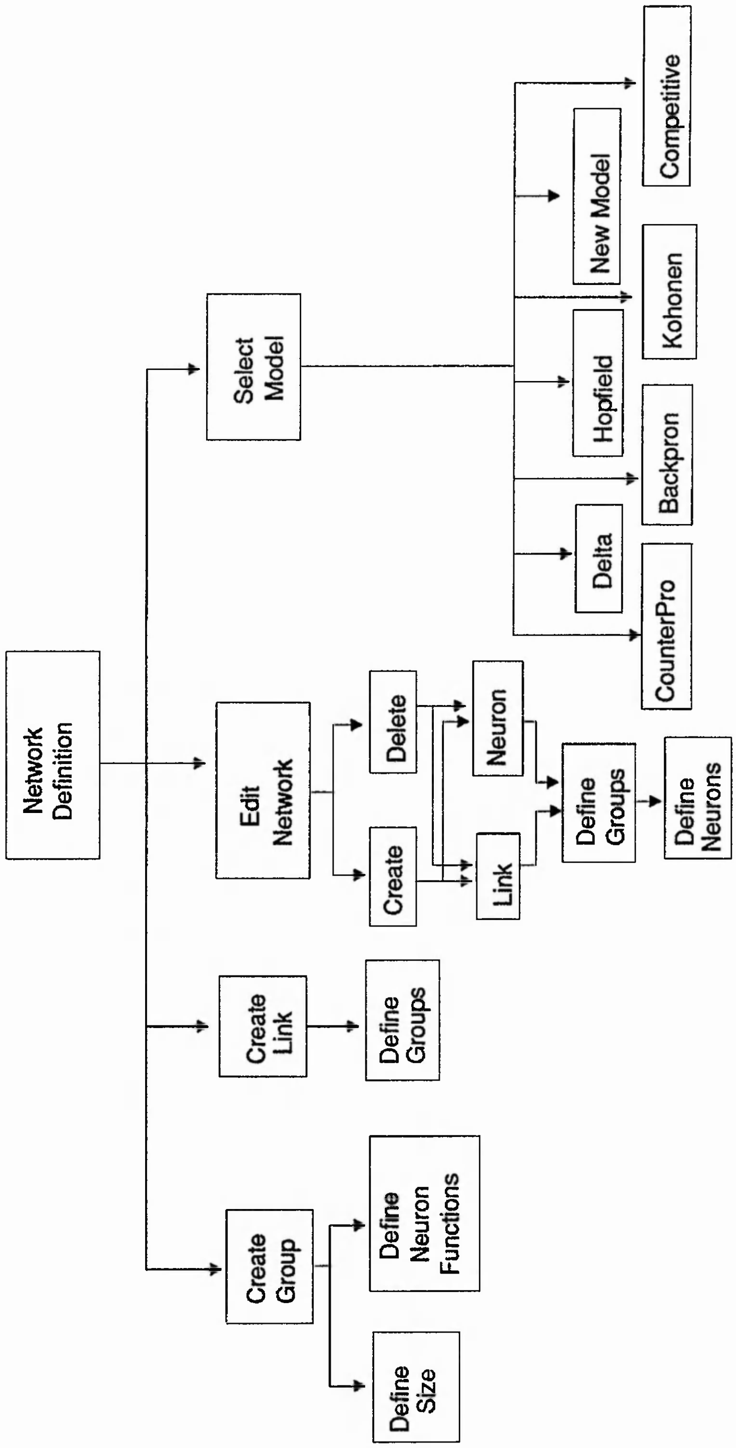
```
          thisneuron->outval=data;
       else
          thisneuron->reg=data;
       thisneuron=thisneuron->nexneuron;
     }
     fclose(dfptr);
   }
   else
      runterror();
 }
 else runterror();
}

void   AdjustNeuronWgt(float   learnrate,struct   NEURON
*fromneuron,struct NEURON *toneuron,float (*learnfcn)())
{
 struct SYNAPE *thissynape;
 while (fromneuron!=toneuron)
 {
   thissynape=fromneuron->firsynape;
   while(thissynape!=(struct SYNAPE *)NULL)
   {
    thissynape->weight=(*learnfcn)(&learnrate,
    fromneuron,thissynape);
    thissynape=thissynape->nexsynape;
   }
   fromneuron=fromneuron->nexneuron;
 }
 thissynape=toneuron->firsynape;
 while(thissynape!=(struct SYNAPE *)NULL)
 {
  thissynape->weight=(*learnfcn)(&learnrate,
  toneuron,thissynape);
  thissynape=thissynape->nexsynape;
 }
}

void NormalizeInput(char *grpname)
{
 struct GROUP *thisgroup;
 struct NEURON *thisneuron;
 float denomin;

 denomin=0.0;
 if ((thisgroup=SearchGrp(grpname))!=(struct GROUP *)NULL)
 {
  thisneuron=thisgroup->firsneuron;
  while(thisneuron!=(struct NEURON *)NULL)
  {
   denomin+=thisneuron->outval*thisneuron->outval;
   thisneuron=thisneuron->nexneuron;
  }
  denomin=(float)sqrt((double)denomin);
  thisneuron=thisgroup->firsneuron;
  while(thisneuron!=(struct NEURON *)NULL)
  {
    thisneuron->outval=thisneuron->outval/denomin;
    thisneuron=thisneuron->nexneuron;
   }
  }
 else
    runterror();
}


void NormalizeWgt(char *grpname)
{
 struct GROUP *thisgroup;
 struct NEURON *thisneuron;
 struct SYNAPE *thissynape;
 float denomin;

 if ((thisgroup=SearchGrp(grpname))!=(struct GROUP *)NULL)
 {
  thisneuron=thisgroup->firsneuron;
  while(thisneuron!=(struct NEURON *)NULL)
  {
   denomin=0.0;
   thissynape=thisneuron->firsynape;
   while(thissynape!=(struct SYNAPE *)NULL)
   {
    denomin+=thissynape->weight*thissynape->weight;
    thissynape=thissynape->nexsynape;
   }
   denomin=(float)sqrt((double)denomin);
   thissynape=thisneuron->firsynape;
   while(thissynape!=(struct SYNAPE *)NULL)
   {
    thissynape->weight=thissynape->weight/denomin;
    thissynape=thissynape->nexsynape;
   }
   thisneuron=thisneuron->nexneuron;
  }
 }
 else
    runterror();
}


float  RandomValue(struct  NEURON  *aneuron,struct  SYNAPE
*asynape)
{
 return((float)random(5)*0.1+(float)random(16)*0.01+
 (float)random(16)*0.001);
}

void SaveLoadNet(char *fname,int proctype)
{
 struct GROUP *grp;
 struct NEURON *cell;
 struct SYNAPE *synp;
 FILE *fptr;
 if (proctype==SAVE)
    if ((fptr=fopen(fname,"wb"))==NULL) runterror();
 if (proctype==LOAD)
    if ((fptr=fopen(fname,"rb"))==NULL) runterror();
 grp=firsgroup;
 while(grp!=(struct GROUP *)NULL)
 {
  cell=grp->firsneuron;
  while(cell!=(struct NEURON *)NULL)
  {
   synp=cell->firsynape;
   while(synp!=(struct SYNAPE *)NULL)
   {
    if (proctype==SAVE)
       fwrite(&synp->weight,sizeof(synp->weight),1,fptr);
    else
       fread(&synp->weight,sizeof(synp->weight),1,fptr);
    synp=synp->nexsynape;
   }
   cell=cell->nexneuron;
  }
  grp=grp->nexgroup;
 }
 fclose(fptr);
}

FILE  *LoadGrpData(struct  NEURON  *cell,FILE  *fptr,int
datatype)
{
 int dot;

 while(cell!=(struct NEURON *)NULL)
 {
  fread(&dot,sizeof(dot),1,fptr);
  if (datatype==INTYPE)
     cell->outval=dot;
  else
  {
     if (dot==1.0)
     cell->reg=1.0;
     else
     cell->reg=0.0;
  }
  cell=cell->nexneuron;
 }
 return(fptr);
}
void ShowGroup(char *gpname)
{
 struct NEURON *thisneuron;
 struct GROUP *thisgroup;
 struct SYNAPE *thissynp;

 if ((thisgroup=SearchGrp(gpname))==(struct GROUP *)NULL)
     runterror();
 printf("group %s nofneuron %d\n",thisgroup->grpid,
  thisgroup->nofneuron);
  thisneuron=thisgroup->firsneuron;
  while(thisneuron!=(struct NEURON *)NULL)
  {
   printf("neuron %d outval %f netval %f reg %f err
   %f\n",thisneuron->neuronid, thisneuron->outval,
   thisneuron->netval, thisneuron->reg,thisneuron->error);
   thissynp=thisneuron->firsynape;
   while (thissynp!=(struct SYNAPE *)NULL)
   {
    printf("->weight = %f from neuron outval %f\n",
    thissynp->weight,thissynp->fromneuron->outval);
    thissynp=thissynp->nexsynape;
   }
   thisneuron=thisneuron->nexneuron;
   getch();
  }
 }

void A_ShowGroup(char *gpname)
{
 struct NEURON *thisneuron;
 struct GROUP *thisgroup;

 if ((thisgroup=SearchGrp(gpname))==(struct GROUP *) NULL)
    runterror();
 thisneuron=thisgroup->firsneuron;
 if (strcmp(thisgroup->grpid,"INLAYER")!=0)
 while(thisneuron!=(struct NEURON *)NULL)
 {
   if (thisneuron->outval>0.69)
      printf("1");
   else
      printf("0");
   thisneuron=thisneuron->nexneuron;
 }
 else
 while(thisneuron!=(struct NEURON *)NULL)
 {
   if (thisneuron->outval>0.69)
      printf("1 ");
   else
      printf("0 ");
   thisneuron=thisneuron->nexneuron;
 }
 printf("\n");
}


/* Neuron Functions */

float InputNOptn(struct NEURON *cell)
{
 return(cell->outval);
}
```

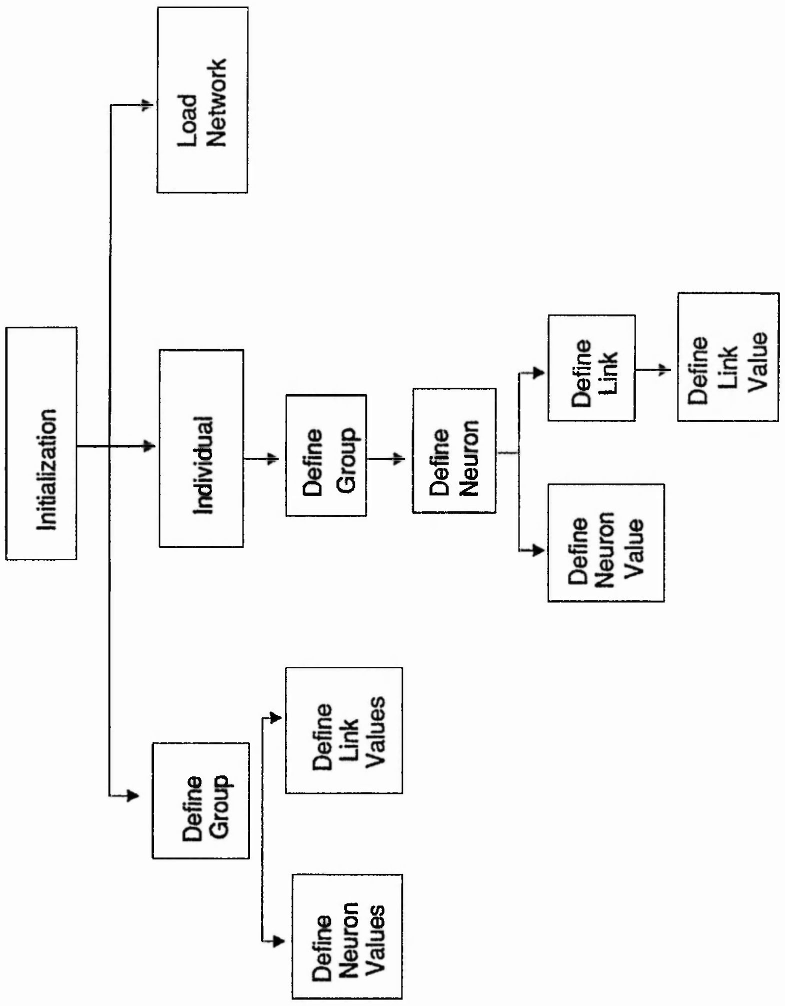
```
float Suma(struct NEURON *cell)
{
 struct SYNAPE *thislink;
 float sum;

 sum=0.0;
 thislink=cell->firsynape;
 while(thislink!=(struct SYNAPE *)NULL)
 {
  sum+=thislink->weight*thislink->fromneuron->outval;
  thislink=thislink->nexsynape;
 }
 return(sum);
}

float Logistic(float invalue)
{
 if (invalue>14 && invalue<-14)
 {
  if (invalue>14) return(0.9);
  else return(0.0);
 }
 else
 {
  invalue=(float)(1/(1+(float)exp(-(double)invalue)));
  if (invalue>0.8895) return(0.9);
  if (invalue<0.0000001) return(0.0);
  if (invalue>0.0000001 && invalue<0.8895) return(invalue);
 }
}

float Binary(float invalue)
{
 if (invalue>0.8)
   return(1.0);
 else
   return(0.0);
}

float Ramp(float invalue)
{
 return(invalue);
}



/* The procedures below are support code for simulating:
Single Layer Perceptrons, Multi-layer Nets, Hopfield Nets,
Competitive Nets, Kohonen Nets, and Counter-propagation
Nets. */



float InitHopeWgt(struct NEURON *aneuron,struct SYNAPE
*asynape)
{
 if (aneuron->neuronid!=asynape->fromneuron->neuronid)
    return(asynape->weight+asynape->fromneuron->outval*
    aneuron->reg);
 else
    return(0.0);
}

float DeltaError(struct NEURON *theneuron,char *soursegrp)
{
 if (theneuron!=(struct NEURON *)NULL)
    return(theneuron->reg-theneuron->outval);
 else
  runterror();
}

float    Back_O_Error(struct    NEURON    *theneuron,char
*soursegrp)
{
 if (theneuron!=(struct NEURON *)NULL)
    return(theneuron->outval*(theneuron->reg-
    theneuron->outval)*(1-theneuron->outval));
 else
  runterror();
}

float    Back_H_Error(struct    NEURON    *theneuron,char
*soursegrp)
{
 struct GROUP *thisgroup;
 struct SYNAPE *thesynape;
 struct NEURON *s_neuron;
 int synapflag;

 theneuron->error=0.0;
 if ((thisgroup=SearchGrp(soursegrp))!=(struct GROUP
 *)NULL)
 {
  s_neuron=thisgroup->firsneuron;
  while (s_neuron!=(struct NEURON *)NULL)
  {
   synapflag=0;
   thesynape=s_neuron->firsynape;
   while (thesynape!=(struct SYNAPE *)NULL || synapflag!=1)
   {
    if (thesynape->fromneuron==theneuron)
    {
     theneuron->error+=thesynape->weight*s_neuron->error;
     synapflag=1;
    }
    thesynape=thesynape->nexsynape;
   }
   s_neuron=s_neuron->nexneuron;
  }
  return(theneuron->outval*(1-theneuron->outval)*
  theneuron->error);
 }
 else
  runterror();
}

float    DeltaLearn(float    *rate,struct    NEURON
*theneuron,struct SYNAPE *thesynape)
{
 return(thesynape->weight+*rate*thesynape->fromneuron->
 outval*theneuron->error);
}

float    Norm_BackLearn(float    *rate,struct    NEURON
*theneuron,struct SYNAPE *thesynape)
{
 return(thesynape->weight+*rate*thesynape->fromneuron->
 outval*theneuron->error);
}

float    Mome_BackLearn(float    *rate,struct    NEURON
*theneuron,struct SYNAPE *thesynape)
{
 float t_weight;

 t_weight=crate*(thesynape->weight-thesynape->reg);
 thesynape->reg=thesynape->weight;
 return(thesynape->reg+*rate*thesynape->
 fromneuron->outval*theneuron->error+t_weight);
}

float   ProcessRate(int   ratetype,float   norminator,float
denorminator)
{

 if  (ratetype==KOH_IN)
    return(0.99*(norminator+1)/denorminator);
 if  (ratetype==GROSS)
    return(0.1*(1-norminator/denorminator));
 if  (ratetype==KOHON)
    return(0.7*(1-norminator/denorminator));
}

void FindDistance(char *grpname)
{
 struct SYNAPE *thislink;
 struct GROUP *thisgrp;
 struct NEURON *thiscell;
 float sum;

 if ((thisgrp=SearchGrp(grpname))!=(struct GROUP *)NULL)
 {
  thiscell=thisgrp->firsneuron;
  while (thiscell!=(struct NEURON *)NULL)
  {
   sum=0.0;
   thislink=thiscell->firsynape;
   while(thislink!=(struct SYNAPE *)NULL)
```

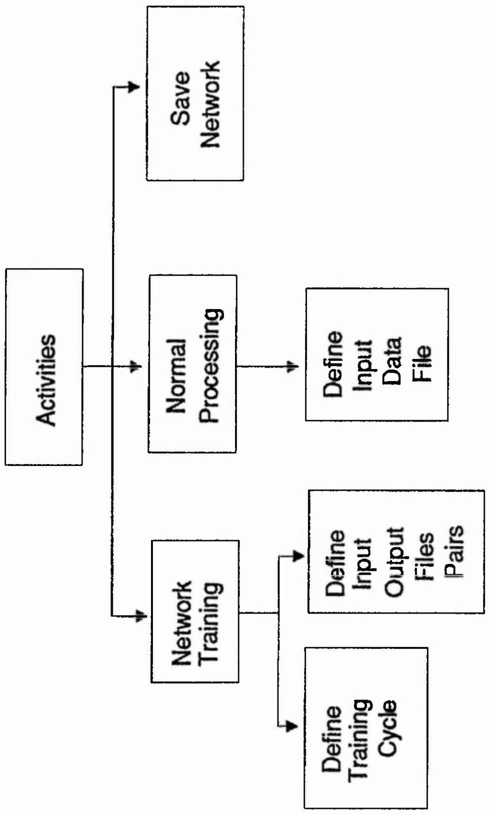
```
    {
     sum+=(thislink->fromneuron->outval-thislink->
     weight)*(thislink->fromneuron->outval-
     thislink->weight);
     thislink=thislink->nexsynape;
    }
    thiscell->reg=sum;
    thiscell=thiscell->nexneuron;
   }
  }
 }

int FindRadius(char *grpname,int cyclenum)
{
 struct GROUP *thisgroup;

 if ((thisgroup=SearchGrp(grpname))!=(struct GROUP *)NULL)
    return((int)(thisgroup->nofneuron/(2+cyclenum/500)));
 else
   runterror();
}

float   GrossLearn(float   *rate,struct   NEURON
*theneuron,struct SYNAPE *thesynape)
{
  return(thesynape->weight+*rate*(theneuron->reg-thesynape
  ->weight)*thesynape->fromneuron->outval);}

void SetKonPattern(char *grpname,float alpha)
{
 struct GROUP *thisgroup;
 struct NEURON *thisneuron;

 if ((thisgroup=SearchGrp(grpname))!=(struct GROUP *)NULL)
 {
  thisneuron=thisgroup->firsneuron;
  while(thisneuron!=(struct NEURON *)NULL)
  {
   thisneuron->outval=alpha*thisneuron->outval+
   ((1/(float)sqrt((double)nofinput))*(1-alpha));
   thisneuron=thisneuron->nexneuron;
  }
 }
 else
  runterror();
}


float   KohonLearn(float   *rate,struct   NEURON
*theneuron,struct SYNAPE *thesynape)
{
 return(thesynape->weight+*rate*(thesynape->fromneuron->
 outval-thesynape->weight));
}

float   KohoWgt(struct   NEURON   *aneuron,struct   SYNAPE
*asynape)
{
 return((float)random(16)*0.01+
 1/((float)sqrt((double)nofinput)));
}

void CompInitWgt(char *grpname)
{
 struct SYNAPE *thesynape;
 struct GROUP *thegroup;
 struct NEURON *theneuron;
 float nofsynape;
 float accwgt,basewgt;

 if ((thegroup=SearchGrp(grpname))!=(struct GROUP *) NULL
 && thegroup->firsneuron!=(struct NEURON *)NULL)
 {
  randomize();
  theneuron=thegroup->firsneuron;
  while(theneuron!=(struct NEURON *)NULL)
  {
   nofsynape=0; accwgt=0.0;
   thesynape=theneuron->firsynape;
   while (thesynape!=(struct SYNAPE *)NULL)
   { nofsynape++; thesynape=thesynape->nexsynape; }
   basewgt=1/(nofsynape);
   thesynape=theneuron->firsynape;
   while(thesynape->nexsynape!=(struct SYNAPE *)NULL)
   {
    thesynape->weight=basewgt-random(9)*
    (1/(100*nofsynape));
    accwgt+=thesynape->weight;
    thesynape=thesynape->nexsynape;
   }
   thesynape->weight=1-accwgt;
   theneuron=theneuron->nexneuron;
  }
 } else runterror();
}

struct NEURON *SetWinner(char *grpname,int nettype)
{
 struct GROUP *thegroup;
 struct NEURON *theneuron,*winner;

 if ((thegroup=SearchGrp(grpname))!=(struct GROUP *)NULL)
 {
  theneuron=thegroup->firsneuron;
  winner=theneuron;
  while (theneuron!=(struct NEURON *)NULL)
  {
   if (nettype==CNT_KON)
   {
    if (theneuron->netval>winner->netval ||
    theneuron->netval==winner->netval)
    {
      winner->outval=0.0;
      winner=theneuron;
      winner->outval=1.0;
     }
     else
       theneuron->outval=0.0;
    }
    else
    {
     if (theneuron->reg<winner->reg ||
     theneuron->reg==winner->reg)
     {
      winner->outval=0.0;
      winner=theneuron;
      winner->outval=1.0;
     }
     else
       theneuron->outval=0.0;
    }
    theneuron=theneuron->nexneuron;
   }
   theneuron=thegroup->firsneuron;
   while(theneuron!=(struct NEURON *)NULL)
   {
    if (theneuron->outval==1.0)
       return((struct NEURON *)theneuron);
    theneuron=theneuron->nexneuron;
   }
  }
  else runterror();
}

float    CompetLearn(float    *rate,struct    NEURON
*theneuron,struct SYNAPE *thesynape)
{
 return(thesynape->weight+*rate*(thesynape->fromneuron->
 outval/nactive)-*rate*thesynape->weight);
}

int NofActiveInput(struct SYNAPE *thesynape)
{
 int nofactive;

 nofactive=0;
 while (thesynape!=(struct SYNAPE *)NULL)
 {
  if (thesynape->fromneuron->outval==1.0)
     nofactive++;
  thesynape=thesynape->nexsynape;
 }
 return(nofactive);
}

void NofInputData(struct SYNAPE *thesynape)
{
 nofinput=0;
 while (thesynape!=(struct SYNAPE *)NULL)
  {
   nofinput++;
   thesynape=thesynape->nexsynape;
  }
}
```

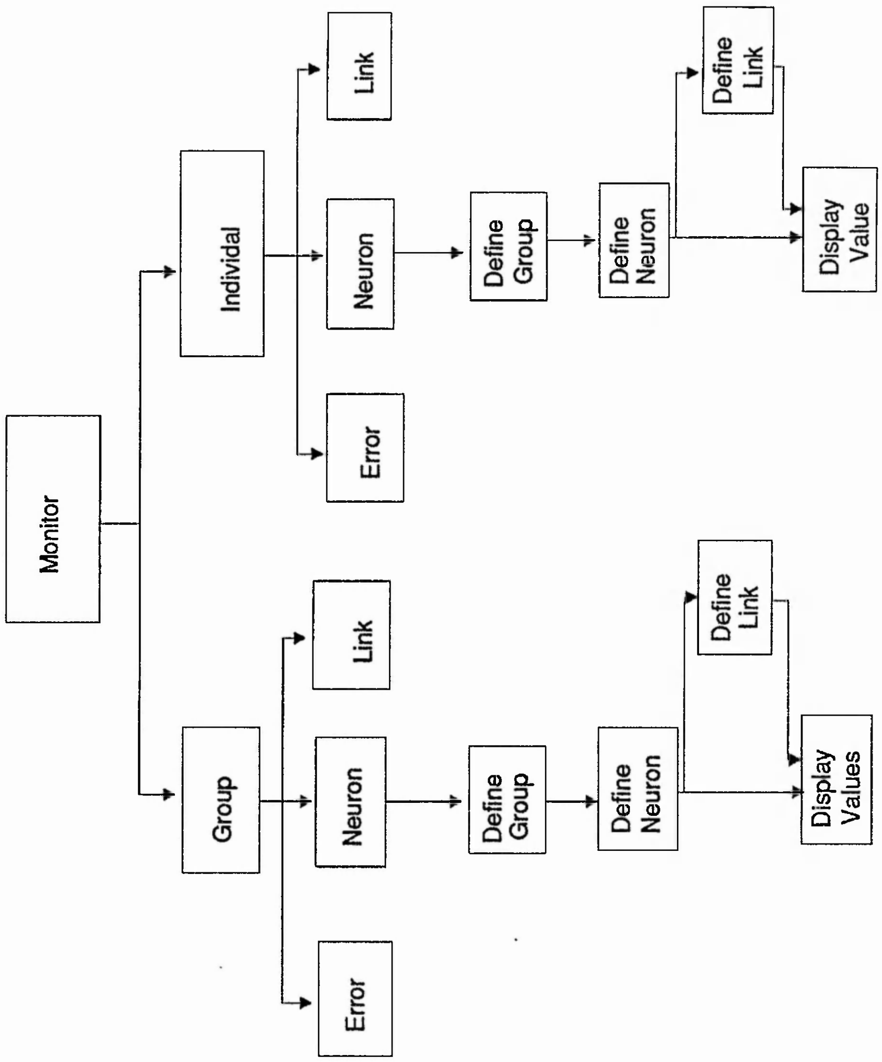
Appendix (c) Structure Chart Of NNSim.exe




Network Definition
Create Group
Create Link
Edit Network
Select Model
Define Size
Define Neuron Functions
Define Groups
Create
Delete
Link
Neuron
Define Groups
Define Neurons
CounterPro
Delta
Backpron
Hopfield
Kohonen
New Model
Competitive


Initialization
Load Network
Individual
Define Group
Define Neuron
Define Link
Define Link Value
Define Neuron Value
Define Group
Define Link Values
Define Neuron Values


Activities
Network Training
Normal Processing
Save Network
Define Training Cycle
Define Input Output Files Pairs
Define Input Data File


Monitor
Individal
Error
Neuron
Link
Define Group
Define Neuron
Define Link
Display Value
Group
Error
Neuron
Link
Define Group
Define Neuron
Define Link
Display Values