THE DESIGN OF A NEURAL NETWORK COMPILER

by

MD. NASIR SULAIMAN

B.Sc. Ed. (Hons.), M.Sc. (Computing)

A Doctoral Thesis

Submitted in partial fulfilment of the requirements

for the award of Doctor of Philosophy

of the Loughborough University of Technology

October, 1994

Supervisor

PROFESSOR D. J. EVANS, Ph.D., D.Sc.

Parallel Algorithms Research Centre

Department of Computer Studies

ACKNOWLEDGEMENTS

I would like to express my thanks and gratitude to my Supervisor, Professor D. J. Evans for his invaluable help, guidance, supervision and support throughout my research.

I also would like to thank the Public Service Department, Malaysia for providing me with financial support during my research for the last three years.

I also would like to thank Universiti Pertanian Malaysia for giving me study leave and providing my family with financial support.

I also would like to thank friends who have helped me in one way or another and wish to extend my sincere appreciation and best wishes.

Finally, I would like to thank my wife, children and family for their constant love, devotion and patience.

ABSTRACT

Computer simulation is a flexible and economical way for rapid prototyping and concept evaluation with Neural Network (NN) models. Increasing research on NNs has led to the development of several simulation programs. Not all simulations have the same scope. Some simulations allow only a fixed network model and some are more general. Designing a simulation program for general purpose NN models has become a current trend nowadays because of its flexibility and efficiency. A proper programming language specifically for NN models is preferred since the existing high-level languages such as C are for NN designers from a strong computer background. The program translations for NN languages come from combinations which are either interpreter and/or compiler. There are also various styles of programming languages such as a procedural, functional, descriptive and object-oriented.

The main focus of this thesis is to study the feasibility of using a compiler method for the development of a general-purpose simulator - NEUCOMP that compiles the program written as a list of mathematical specifications of the particular NN model and translates it into a chosen target program. The language supported by NEUCOMP is based on a procedural style. Information regarding the list of mathematical statements required by the NN models are written in the program. The mathematical statements used are represented by scalar, vector and matrix assignments. NEUCOMP translates these expressions into actual program loops.

NEUCOMP enables compilation of a simulation program written in the NEUCOMP language for any NN model, contains graphical facilities such as portraying the NN architecture and displaying a graph of the result during training and finally to have a program that can run on a parallel shared memory multi-processor system.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Recently, usage of Neural network (NN) models has grown rapidly in solving applications involving massive parallelism such as image processing, pattern recognition and combinatorial problems where the traditional programming method is not suitable. Due to its self-organising and adaptive nature, the model potentially offers a new parallel processing paradigm that could be more robust [Lippman (1987), Kung (1993)].

A NN model is a structured distributed information processing system consisting of processing elements or nodes interconnected together with unidirectional signal channels called connections. A connection has a strength of type inhibitory or excitatory. This strength is called a weight. Each node has a single output connection which branches into as many collateral connections as desired. It can process local information and carry out localised information processing [Rumelhart *et al.* (1986), Simpson (1990)].

NN models are also mathematical models that can abstract parallel information handling features of biological systems. They are made up of many relatively simple elements called neurons and closely related to the physiology of the brain [Korn (1991b)]. The human brain contains more than 10^{11} neurons and 10^{14} synapses or connection weights in the human nervous system [Forrest *et al.* (1988), Männer (1989), Kung (1993)]. Each neuron can have from 1000 to 100000 interconnections with other neurons. They send excitatory and inhibitory messages to each other and update their weights on the basis of these simple messages. In the NN models the term neurons is represented by processing elements or nodes. The brain is able to operate easily in parallel to solve problems such as pattern recognition but the computer is a high-speed serial machine which is unable to solve simple recognition. Therefore in modelling the brain's basic system, the problem to be solved must suit the parallel model.

NN models can be implemented in various ways. These can range from a very complex hardware VLSI design to software simulators on a digital computer. Hardware implementations are faster than software simulators but they are confined to special purpose NNs. Computer simulation is more flexible and economical for rapid prototyping and problem solving [Feldman *et al.* (1988), Nijhuis *et al.* (1989), Almassy *et al.* (1990), Nelson *et al.* (1991), Shumsheruddin (1992)]. Increasing research on NNs has led to the development of several simulation programs. All the simulation tools have a different scope of design and implementation. Some simulations allow only a fixed network model and some are more general. Designing a simulation program for general purpose NN models has become a current trend nowadays because of its flexibility and efficiency [Shumsheruddin (1992)].

1.1 THE ROLE OF NEURAL NETWORK MODELS

NNs and their computational properties have attracted the interest of researchers in the area of machine perception by presenting an exciting, complementary alternative to symbolic processing paradigms. They hold with them the promise of exceedingly fast implementations, coupled with flexibility through self-organisation or learning rather than computer programming.

Modelling the NN can be divided into 2 categories [Kung (1993), Korn (1991b)] - Biological-type and Application-driven NN. For the Biological-type NN, the model mimics biological neural systems such as audio/vision functions like motion field, binocular stereo and edge detection. The Application-driven NN model is not closely tied to biological realities. For these models, the architectures are largely influenced by the application because of the following reasons [Kung (1993)] :-

(1) Adaptiveness and self-organisation
It offers robust and adaptive processing capabilities by adopting adaptive learning and self-organisation rules. This allows the network to improve with experience.

(2) Non-linear network processing
It enhances the network's approximation, classification and noise-immunity capabilities.

(3) Parallel processing
It usually employs a large number of processing nodes enhanced by extensive interconnectivity - massively parallelism which provides high speed performance.

NNs have found many successful applications in computer vision, signal or image processing, speech or character recognition, expert systems, remote sensing, robotics processing, industrial inspection and scientific exploration [Maren *et al.* (1990), Simpson (1990), Kung (1993)]. The application domains of NNs can be roughly divided into the following categories :-

(1) association
(2) classification
(3) pattern completion
(4) approximation/generalisation
(5) optimisation

Association can be of two types, namely auto-association and hetero-association. In auto-association, a NN is required to store a set of patterns by repeatedly presenting them to the network. The problem is to retrieve the complete pattern from a partial description or distorted part of the desired pattern. Hetero-association involves pairing an arbitrary set of input patterns with another arbitrary set of output patterns. The problem is to retrieve a corresponding pattern from a given input pattern.

There are two types of classifications. The first classification involves a fixed number of categories alongside a set of input patterns that are repeatedly presented to the network. When a new pattern is presented, the network is able to identify which category this pattern belongs to. The second classification involves a situation where there is no prior knowledge of the categories into which the input patterns are to be classified. In this case a network performs adaptive feature attraction or clustering during training.

Pattern completion is also known as information completion, where the original pattern is recovered from a given partial information. The process of completion takes place for many iterations. A process reaches a stable state when there is no change of state.

Approximation involves the following task. Suppose that a non-linear input-output mapping is described by the function

$$y = f(\mathbf{x})$$

where x is an input vector and y is the scalar output. The function f is assumed unknown. The requirement is to design a NN that approximates the unknown vector x from f after the input-output pairs (x_1,y_1), (x_2,y_2), ..., (x_n,y_n) have been repeatedly presented. A network is considered successful if it can closely approximate the actual values for the trained data set and can provide smooth interpolations for the untrained data set. The objective of generalisation is to yield a correct output response to an input pattern for which it has not been trained before.

Optimisation applications usually involve finding a global minimum of an energy function. Once the energy function is defined, the determination of the connection weights is relatively straightforward. In some applications, the energy function is directly available. In some others, the energy function must be derived from

the given cost criterion and special constraints. The difficulty associated with the optimisation problem is the large possibility of a solution converging to a local minimum instead of the global minimum. To tackle the problem, several statistical techniques are proposed, such as stochastic simulated annealing and mean-field annealing.

1.2 THE ROLE OF NEURAL NETWORK SIMULATION TOOLS

Among the many uses of computers is simulation. Modelling the real-world phenomena to see the affect of varying the conditions on the behaviour of the system can be done using the computer. If the real world is adequately described in the computer program, the results of the program should predict what happens in the real-world situation [Springer *et al.* (1989)].

NN simulation software is a computer-aided experimentation for NN models, typically implemented on a computer [Korn (1991a&b)]. NN models can also be implemented on specialised hardware. Hardware implementations currently come in several species such as computer emulations. They involve special boards or other special hardware, integrated circuit chips, optical or holographic devices.

Hardware implementations are faster than software simulations. However, they are for special purpose NN models, expensive and require a substantial commitment to the use of the system. Software simulations are more attractive because they can evaluated easily and the commitment is not as restrictive as that of the first class. The ease of making software changes has been a significant advantage. Redesigning chips take time and money whereas simulation can help avoid costly mistakes.

Computer simulations are ideal for research in NNs as they can be developed very quickly and cheaply. They are very flexible and these makes it easy to experiment

with alternative networks structures, activation functions and learning algorithms. They also allow easy collection and analysis of data on the behaviour and performance of the networks. However, NNs are computationally expensive because of the following reasons :-

(1) they contain a large number of nodes and interconnections. The number of interconnections are directly proportional to the complexity of the model that can be implemented.
(2) the learning algorithm involves many iterations in order to converge or reach a stable solution.

To improve the speed of the software implementations, several parallel simulator strategies have recently appeared. The reasons being that the parallel computers can offer faster execution time than the sequential machines.

Software simulation can be obtained either from commercially available packages (i.e. BrainMaker, ExploreNet, NeuroShell, etc. [Turban (1993)]) or writing a simulation program using conventional high-level languages such as C. Due to the interdisciplinary nature of the NN study, researchers are not always computer software experts and thus must rely heavily on commercial products. Many packages run on a PC. They provide good user interfaces and debugging tools for network simulations. Many include thoroughly tested and debugged library routines for simulating common types of network such as the backpropagation network, etc. Hence, packages do not allow extensive model development without a strong programming background. Writing a simulation program to simulate a network allows greater flexibility and enables simulation of arbitrary designs of networks. Specially written programs can be optimised for a particular type of network which may run faster than simulations developed using packages.

There are two approaches when writing a NN simulation program. The first approach is to design a simulation program for specific NN models. This approach has been used in practice for quite some time. The second approach is to design a simulation tool for any NN model [Feldman *et al.* (1988), Shumsheruddin (1992)]. There are two methods of designing this approach. They can be classified as, the user interface method [McClelland *et al.* (1988), Tarr *et al.* (1992)] and programming language method. The first method is still restricted to certain NN models. For a general-purpose NN simulation, a proper language specifically for NN model is preferred. This language is called a special-purpose language. A special-purpose language is used to avoid using the complexity of the existing high-level languages such as C or FORTRAN.

From the NN simulation language, the user can write a program for any NN model or combine these models to suit their applications. A graphical command is also available in the program which depicts the NN architecture and the graphs of the results.

The program translations for NN languages come from either compiler method [Almassy *et al.* (1990), Panetsos *et al.* (1993)] or a combination of both interpreter and compiler methods [Korn (1989, 1991a&b)]. The compiler method has been proved to produce a high performance result [Bennett (1990), Ford (1990)]. However, a general-purpose simulator that allows platform portability, ease of use and extensive model design freedom with minimal usage training may be considered as an effective simulation tool [Myler *et al.* (1992)].

The programming style based on the compiler method is further classified as a procedural, functional, declarative or object-oriented method [Ford (1990), Maeder (1991)].

1.3 OBJECTIVE OF NEURAL NETWORK COMPILER - NEUCOMP

The main focus of this thesis is to study the feasibility of using a compiler method for the development of a general purpose simulation program - NEUCOMP, that compiles the procedural style of programs known as the NEUCOMP language. The NEUCOMP language is a high level language. It is specially designed to cater for NN models with the complexity of the commands from the existing high level C-like language being simplified. This idea is based on Korn's work (i.e. DESIRE/NEUNET) [Korn (1989, 1991a&b)]. The translated program is based on a combination of interpreter and compiler methods. However, NEUCOMP is designed using the compiler method. A NEUCOMP program is written as a list of mathematical specifications of the particular NN model. The mathematical statements can be written as scalar, vector or matrix assignment required by the NN models.

NEUCOMP enables the compilation of a simulation program written in the NEUCOMP language for any NN model. It contains graphical facilities such as portraying the NN architecture and displaying a graph of the results during training. Finally a parallel version of the compiler (NEUCOMP2) generates a program that can run on a parallel shared-memory multi-processor system.

1.4 ORGANISATION OF THE THESIS

The research presented in this thesis covers the design of a general-purpose simulation tool. Specifically, its main focus is to study the feasibility of using a compiler method for the development of a general purpose simulation program. It includes a detailed study of the design of the NN compiler known as NEUCOMP and its language called the NEUCOMP language. They are implemented on a UNIX machine. A design of a parallel NN compiler (NEUCOMP2) is also introduced. NEUCOMP2 is implemented on a shared-memory parallel machine, SEQUENT Balance machine at the PARC (Parallel Algorithms Research

Centre, Loughborough University of Technology). Basic graphical facilities such as displaying NN architectures and various kinds of XY-graphs that depend on the type of applications are also included. Since the machine that supports NEUCOMP does not support graphical facilities, all graphical displays are done on a PC using 'Mathematica' [Wolfram (1991)]. This kind of graphical activities can give a flexible design to the user.

The thesis is organised as follows. **Chapter 2** gives a brief discussion on the taxonomy of the NNs, some examples of popular NN models and a survey of a general-purpose simulation tool. In **chapter 3**, a brief discussion of the compiler design and programming technique are presented. Various programming methods for the high-level programming languages and the compiler-construction tools such as 'Lex' and 'Yacc' are explained. **Chapter 4** gives a detailed design of NEUCOMP and its language, the NEUCOMP language. In that chapter the design of graphical displays on PC using 'Mathematica' is also discussed. Its purpose is to display graphical results computed by a NN program. **Chapter 5** discusses briefly the concepts of parallel computer architectures and parallel programming and gives a detailed design of the parallel NN compiler (NEUCOMP2) and its language. Comparisons of the speedup on selected NN models running using NEUCOMP2 are shown. **Chapter 6** contains an application using selected NN models for solving three different problems such as character recognition, intertwined spirals and travelling salesman problems. Sequential and parallel results are presented and their execution times and speedups are measured. Various graphical results are also shown which depend on the type of applications and type of graph to be shown. Finally **chapter 7** summarises and gives conclusions on the topics discussed throughout the thesis. Discussions on further research work conclude the final chapter of the thesis.

CHAPTER 2

A SURVEY OF NEURAL NETWORK MODELS AND SIMULATION TOOLS

This chapter covers the discussion of NN models which include the taxonomy of NNs, examples of some popular NN models based on their taxonomy and a survey of some currently general-purpose NN simulation tools.

The taxonomy of the NN models are identified by their network architectures, node characteristics and training or learning rules. A network architecture contains the nodes that are grouped in terms of layers such as the input layer, hidden layer (if multilayer network) and output layer. The node characteristics include the type of a node and the use of a transfer function for the activation node. The training or learning rules are based on supervised or unsupervised learning algorithms. These rules specify an initial set of weights which should be adapted during training.

Examples of NN models which represent different classes of the network are discussed. The backpropagation network is a multilayer feedforward network. The Counterpropagation is a multilayer network in which one of the layers is the competitive layer network. The Kohonen Self-organising network is a topological network. The Adaptive Resonance Theory (ART) network is a two layer network with feedforward and feedback connections. The Hopfield network is a single layer network with feedback connection. The backpropagation and Counterpropagation networks use supervised learning algorithms whilst the Kohonen, ART and Hopfield networks use unsupervised learning.

Some current general-purpose NN simulation tools are presented in the last section of this chapter. They cover the programming style of defining and simulating the NNs [DasGupta *et al.* (1990), Korn (1989, 1991a&b), Hu (1991)]. These programming languages are known as special-purpose languages. In the non-specific languages [Nijhuis *et al.* (1989), Koopman *et al.* (1990), Myler *et al.* (1992)], an existing high-level language such as C is used to support the tools. The built-in functions such as NN structures and their learning algorithms are called

interactively using a graphical user-interface. However this approach is not flexible enough to allow the designer to try several NN models. In order to run the new NN structures or learning algorithms, a high-level programming language like C-program code and its function calls are defined. These routines are then compiled and linked with the simulation tools.

2.1 TAXONOMY OF THE NEURAL NETWORK MODELS

Since the beginning of the 1980's the interest in NNs has greatly increased and a large range of models have been developed for different purposes [Dayhoff (1990), Maren *et al.* (1990), Simpson (1990), Kung (1993), Haykin (1994)]. However they can be specifically based on the following characteristics:-

(1) network architectures
(2) node characteristics, and
(3) training or learning rules

Their details are now explained in this section.

2.1.1 Architecture of Neural Networks

NN architectures or topologies are formed by organising nodes into layers and linking them with weighted interconnections. The following characteristics to describe the NN architectures are :-

(1) The number of layers in a network such as a single layer, two layers or multilayer.
(2) The type of connections are 'feedforward', 'feedback' and 'lateral'. Feedforward means data from nodes of a lower layer propagate forward to nodes of an upper layer via feedforward connection network. Feedback allows data from nodes of an upper layer to be fed back

to a lower layer via feedback connections. For lateral, there are connections between nodes in the same layer of nodes or local feedback to themselves.

(3) The connection maybe fully or locally connected.

(4) The connections can be excitatory (positive weights) or inhibitory (negative weights).

Based on the above distinctions, six different architectures related to the classes of networks can be identified, as shown in **figure 2.1.**

2.1.1.1 Multilayer feedforward networks

The multilayer feedforward networks as shown in **figure 2.1a**, propagate data from the previous layer to the next layer. They range from simple two-layer perceptron to feedforward networks with multiple hidden layers. With suitable supervised training algorithms, i.e. the backpropagation [Rumelhart *et al.* (1986)], such networks map input patterns on to desirable output patterns. The feedforward networks of one or more hidden layers are capable of doing generalisation and pattern recognition.

2.1.1.2 Single layer networks

The fully connected or laterally connected single layer networks or Hopfield-type networks have only one layer as shown in **figure 2.1b.** A one layer network can only activate one pattern at a time. The lateral or recurrent connections cause different patterns to appear in the single layer with each iteration of operation. Laterally connected networks are typically used for pattern autoassociation. Autoassociative networks can store many patterns, but can only manifest one at a time. They are good for generating clean versions of patterns they have learned when given a noisy or incomplete pattern as a starting point.

The Hopfield network [Aleksander *et al.* (1990), Beale *et al.* (1990), Lippmann (1987)] and Brain-State-in-a-Box [Maren *et al.* (1990), Simpson (1990)] are examples of a single layer network.

2.1.1.3 Topological networks

The topological networks are two layer networks. The second layer is based on topological-ordered vectors where the nodes are laterally connected, **figure 2.1c.** This layer acts as a competitive layer, fires selective output nodes (i.e. winner node) if an input pattern minimises or maximises corresponding functions. During learning, a measure of the vector distance between the different vector nodes is used to adjust their relative position in the vector. The use of topological-ordered vectors is to cluster different classes of input patterns.

This class of network includes the Learning Vector Quantisation and Kohonen Self-organising networks [Beale *et al.* (1990), Dayhoff (1990)].

2.1.1.4 Two layer feedforward/feedback networks

The two layer feedforward and feedback networks function like a Hopfield-type network with symmetrical connections. They can be seen as a two layer non-linear feedforward/feedback network, as shown in **figure 2.1d.** Patterns sweep from one node layer to the next, and then back again, slowly relaxing into a stable state that represents the network's association of the two patterns. This type of network structure is particularly good for associating a pattern in the first layer with another pattern in the second layer, which is called pattern heteroassociation. They can also be used for pattern classification.

The two most popular two-layer feedforward and feedback networks are the Adaptive Resonance Theory (ART) [Beale *et al.* (1990)] and Bidirectional Associative Memory (BAM) type of network [Simpson (1990), Wasserman (1989)].

2.1.1.5 Multilayer competitive networks

The multilayer network that contain one layer with lateral connections are specifically designed for competitive learning purposes (**figure 2.1e**). These connections contain excitatory (positive) connections and inhibitory (negative) connections which balance each other in a certain way. The output of the network is determined by the combination of the connection weights between the output layer and the winner node of the competitive layers.

The Counterpropagation network belongs to these type of networks [Hecht-Nielsen (1987, 1988, 1989), Dayhoff (1990)].

2.1.1.6 Cascading the networks

The possibility of cascading different structures, **figure 2.1f**, open up a sixth type of network structure known as 'hybrid network' [Maren *et al.* (1990)] or 'sequential network' [Korn (1991b)]. The basic variables are not individual node activations, but the input and output patterns of node layers or subnetworks. Feedforward and feedback connections relating such vector variables can form an interesting and powerful vector state machines.

2.1.1.7 Network with local feedback

The network with local feedback is also known as 'dynamic neural network' [Korn (1991b), Hush *et al.* (1993)] or

'temporal model' [Kung (1993)]. The network structures discussed earlier known as static network [Korn (1991b), Hush *et al.* (1993), Kung (1993)]. Static network are categorised by node equations that are memoryless. That is, their output is a function only of the current input, not of past or future inputs or outputs.

Dynamic networks, on the other hand, are systems with memory. They are more suitable for temporal pattern recognitions. Their node equations are typically described by differential or difference equations. They can be categorised into three different groups namely networks with feedforward dynamics, networks with output feedback and networks with state feedback.

However, the models of cascading the networks and dynamic networks are not discussed in this thesis.

2.1.2 Node characteristics

All NNs have a set of processing nodes which represent the neurons or nodes. These nodes operate in parallel, either synchronously, as in the case of most computer-simulated networks, or asynchronously, like biological NNs. Each node, receives input from one or more of its neighbours, computes an output value (it's activation state), and sends it to one or more of its neighbours. Input nodes receive signals from the environment and output nodes send signals to the environment.

The input from the environment may be analogue or digital. If the selected network design is optimised for bi-state nodes, some preprocessing will be necessary to represent the input data in binary format. The output to the environment requires the activation of one or more nodes, either for a single iteration operation or for several iterations. This output may be interpreted as a pattern classification, pattern associated with the

input, completed or noise-cleaned version of the input pattern.

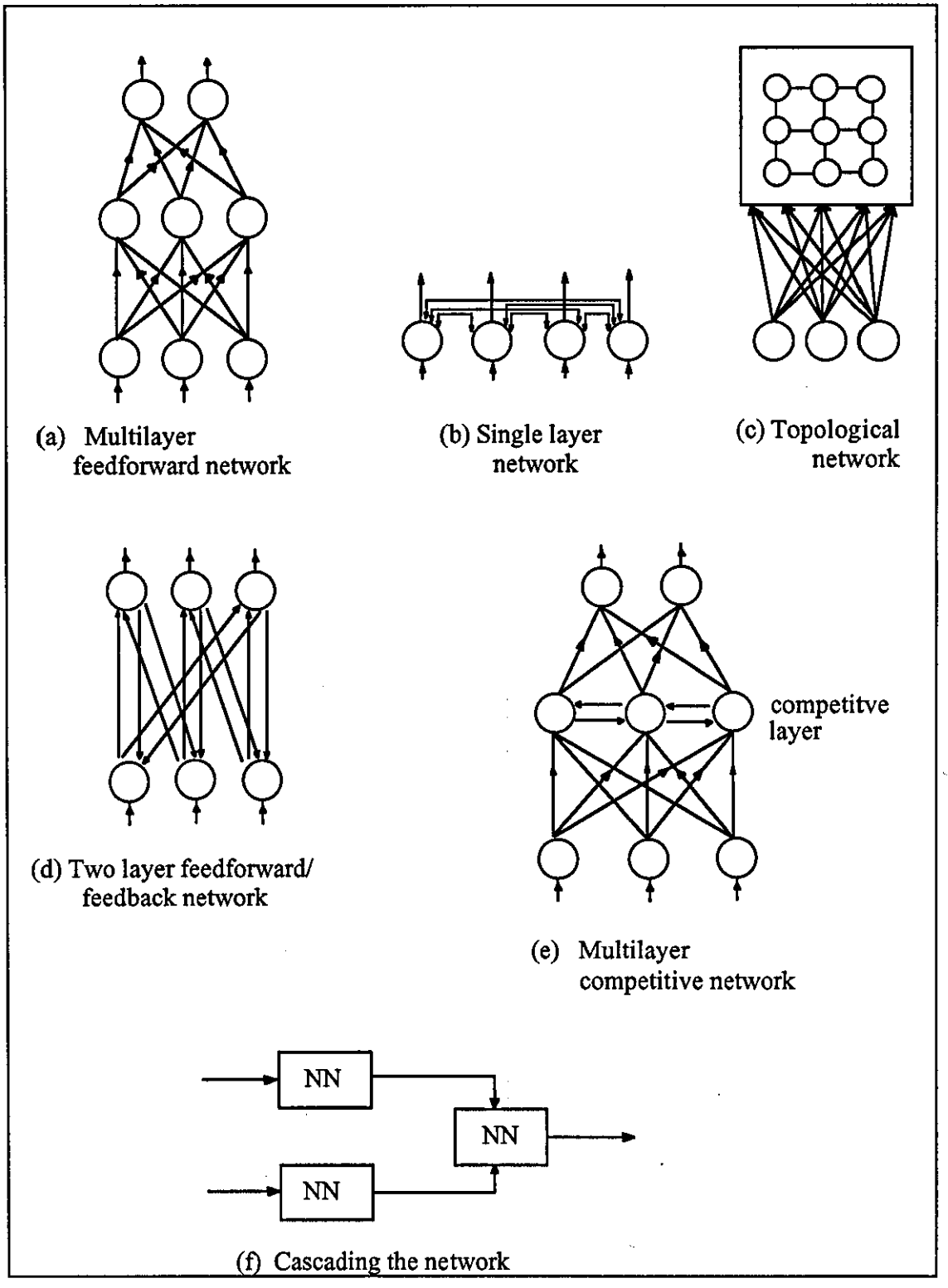


Fig. 2.1: Six NN structures

The activation state of a node, varies with time. Different networks allow different sets of activation values for their nodes. The activation level may be discrete or continuous, bounded or unbounded. Many networks employ a binary node having 0 and 1 as the only two possible levels of activation. Other networks allow integer-valued or real-valued activation levels.

Activated nodes can be the sum of the products of its inputs and the weights of their connections or rely on some defined threshold function known as the transfer function or activation function (**figure 2.2**) This value may change for every iteration until a stable pattern or convergence state has been reached. Although a single node may send out only one signal value from its end, the values which are received by the connected nodes may differ. This is because the strength of the signal which is sent out by a node is modified by the weights.

The nodes of the network are connected together by a set of links called connections. The connections are usually unidirectional, as in the case of biological systems, but may be bi-directional. Weights are the connection strengths between the node and its neighbours. They can have a positive value or negative value. The negative value connection is known as an inhibitory connection and the positive value connection is known as an excitatory connection.

Figure 2.3 is a simple example of a node with the following information :-

(1) $a_0, a_1, ..., a_{n-1}$ are the activation values of nodes $0, 1, ..., n-1$ and the input values to node i,
(2) net_i is the sum of products of weights between node i and output nodes of its neighbours,
(3) w_{ij} is the connection weight from node j to node i,
(4) a_i is the activation value or output value of node i,
(5) f is one of the functions as shown in **figure 2.2.**

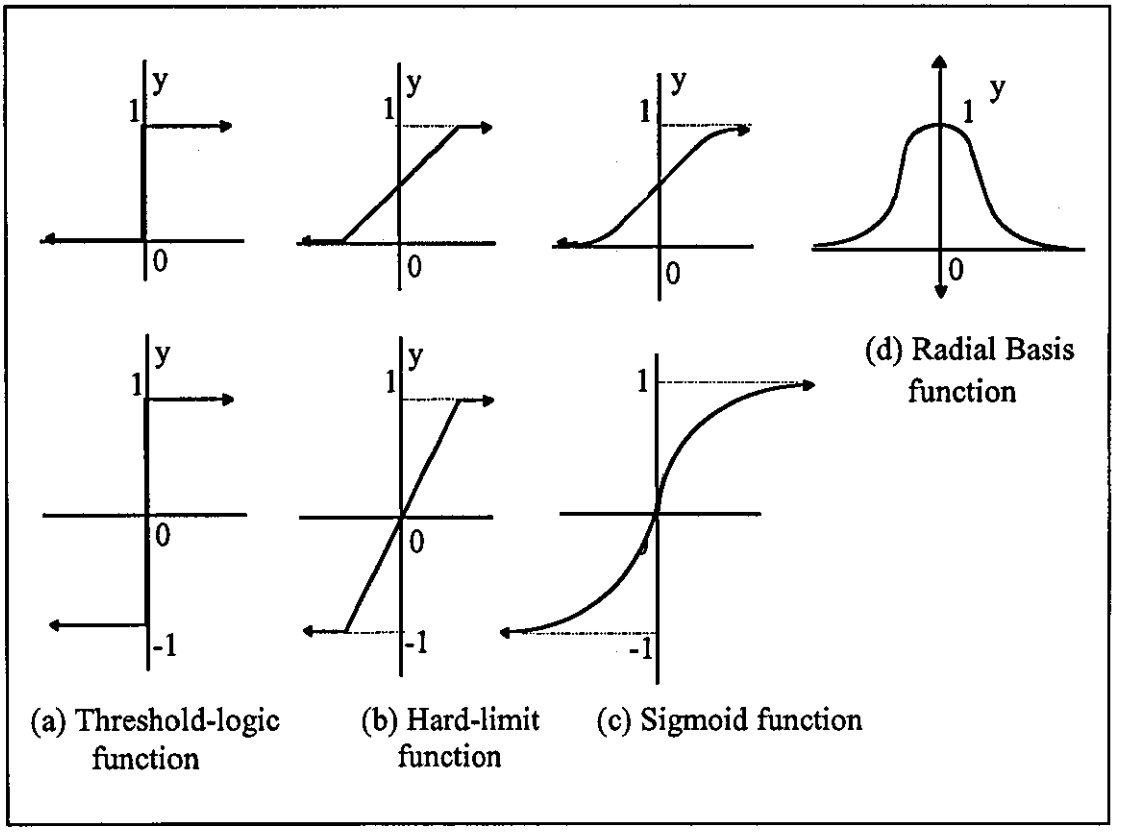


Fig. 2.2: Threshold-function

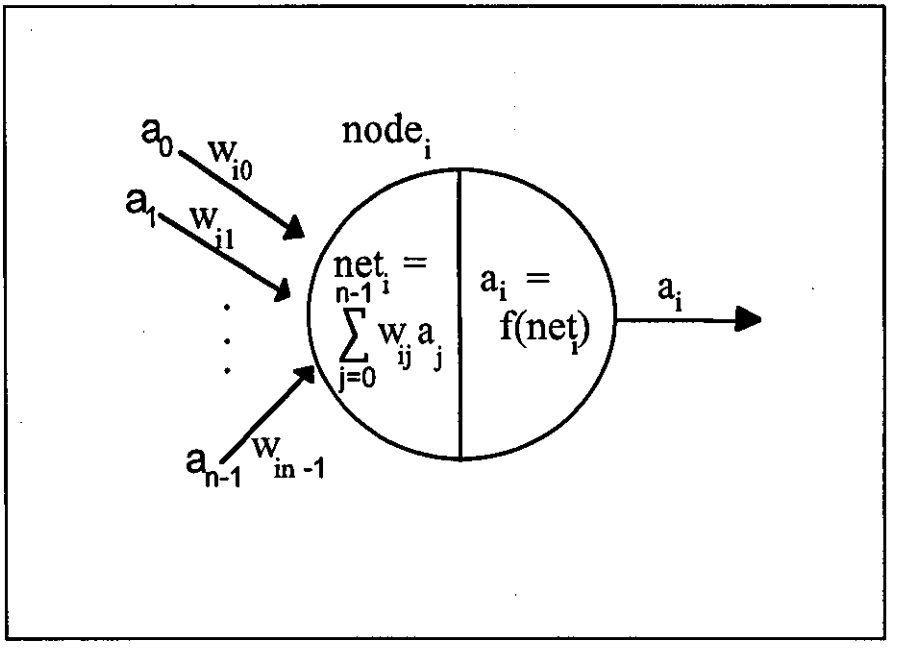


Fig. 2.3: A Simple *i*th. node

2.1.3 Learning rules

A neural network model has to be trained before it can be useful for various applications. The learning rules specify how the weights of the connections in the network are to be adjusted during the learning process or training. During learning, the weights are usually adjusted in a large number of small steps.

Learning denotes changes in the NNs that are adaptive in the sense that the NNs can do the same tasks drawn from the same population more efficiently next time.

Many learning algorithms have been introduced with the objective to allow the network to produce the correct output at a specific period of time. **Figure 2.4** shows the NNs commonly categorised in terms of their corresponding learning algorithms - supervised network and unsupervised network [Wasserman (1989), Beale (1990)].

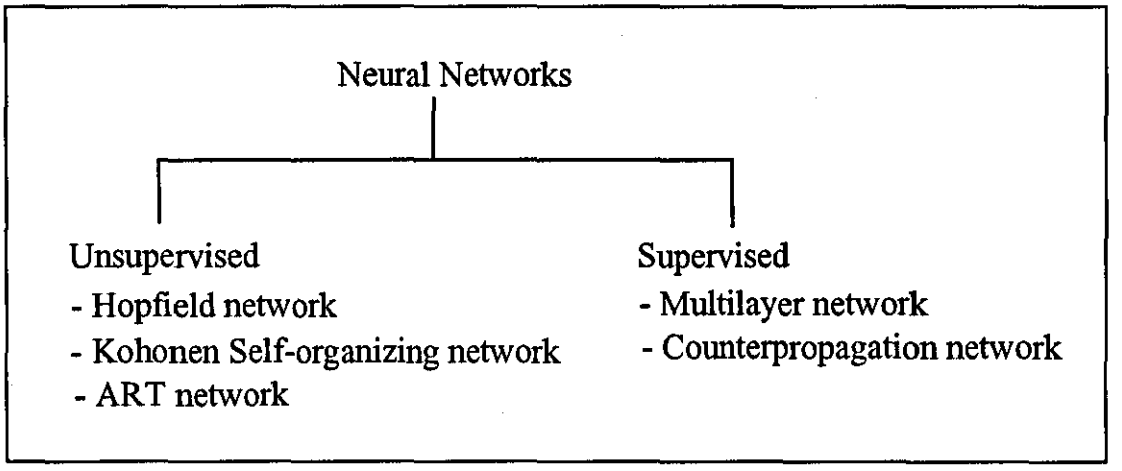


Fig. 2.4: Neural Network's learning algorithms

2.1.3.1 Supervised learning rule

Supervised learning requires the training data to be consist of a pairs of input patterns with a target patterns representing the desired output. These training patterns are called vector-pairs. The weights are adjusted

during training until the input patterns approach the output patterns. Therefore, the learning will benefit from the existence of a teacher. As an example, **figure 2.5** illustrates how the weight vectors, i.e. w, represented by the linear hyperplanes are gradually adjusted to separate one class of patterns from another. These weights can be adjusted by using the following update rule :-

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} + \Delta w_{ij}^{(t)}$$

where the amount of adjustment is proportional to the difference between the teacher response and the actual value.

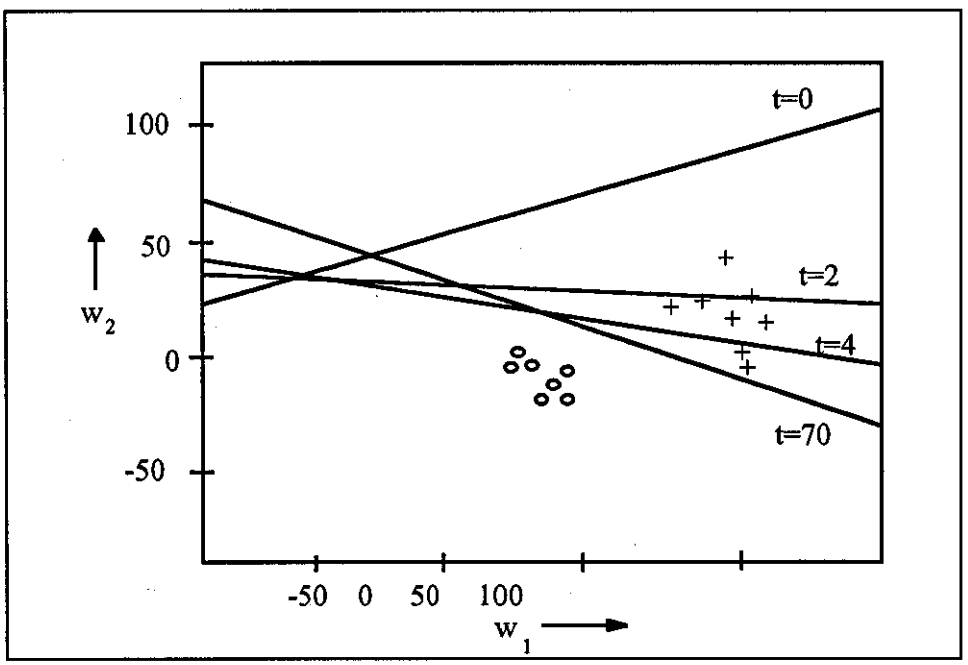


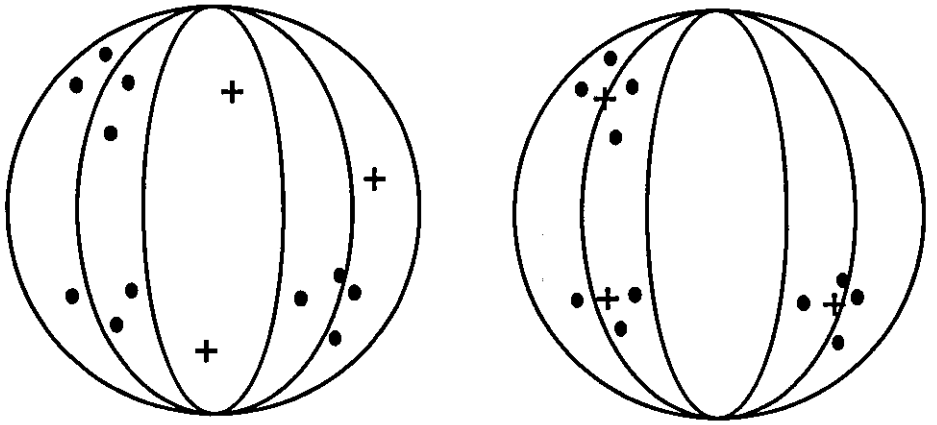
Fig. 2.5: Classification of two groups of patterns during training

2.1.3.2 Unsupervised learning rule

For an unsupervised learning, the training set consists of an input training pattern only. Therefore, the network is trained without the benefit of any target value. The

network learns to adapt based on the experiences collected through the previous training patterns. Typical examples are the Hebbian learning rule and the competitive learning rule. A simple version of Hebbian learning is that when nodes i and j are simultaneously excited, the strength of the connection between them increases in proportion to the product of their activations.

In competitive learning, a node learns by shifting connection weights from its inactive to active input nodes. If a node does not respond to a particular input vector, no learning takes place in that node. If a particular node wins the competition, then each input to that node gives up some proportion of its weights and these weights are equally distributed among the active inputs of the node. **Figure 2.6a** illustrates three natural groupings or clusters of the input patterns and a possible initial state of the weights that may exist before training. **Figure 2.6b** illustrates a typical final state of the weights that results from the use of the competitive learning rule. In particular, each of the output node has discovered a cluster of inputs by moving its connection weight vectors to the centre of gravity of the discovered cluster.



(a) before training (b) after training

Fig. 2.6: Geometrical interpretation of the input vectors (dots) and weight vectors (crosses) before and after training

2.2 EXAMPLES OF THE NEURAL NETWORK MODELS

In the following section some important NN models are presented. They are the backpropagation, Hopfield, Kohonen, Adaptive Resonance Theory (ART) and Counterpropagation networks.

2.2.1 The backpropagation network

The backpropagation network is a multilayer network. It has an input layer, one or more hidden layers and an output layer. Its network architecture is shown in **figure 2.1a**. This network is used to solve problems such as pattern recognition and function approximation.

The backpropagation learning algorithm is a systematic method for training a multilayer network since a two layer network fails to solve hard-learning problems [Rumelhart *et al.* (1986)]. In addition, it is a supervised learning algorithm.

An extended version of the backpropagation method called Gradient Range-Based Heuristic (GRBH) [Sanossian *et al.* (1991)] for accelerating the learning is used. The weights are adjusted during training using the following formula :-

$$w_{ij}(t+1) = w_{ij}(t) + \alpha_k \frac{\partial E(w)}{\partial w_{ij}} + \beta \Delta w_{ij}(t)$$

where

w_{ij} is the weight from node j to node i,

$\frac{\partial E(w)}{\partial w_{ij}} = \sum_p (\delta_{pi} o_{pj})$,

$\delta_{pj} = f'_j(net_{pj})(t_{pj} - o_{pj})$, for the output node,

$\delta_{pj} = f'_j(net_{pj}) \sum_k w_{jk} \delta_{pk}$, for the hidden node,

$\Delta w_{ij}(t) = w_{ij}(t) - w_{ij}(t-1)$,

β is the momentum factor

and α_k, (k=1, ..., n), depend on $\left|\frac{\partial E(w)}{\partial w_{ij}}\right|$. A small value of α_k is chosen when $\left|\frac{\partial E(w)}{\partial w_{ij}}\right|$ is large and large α_k for small $\left|\frac{\partial E(w)}{\partial w_{ij}}\right|$.

The algorithm aims to minimise the error function E by adjusting the weights in the network so that they correspond to those at which the error surface is the lowest. The error function is

$$E_p = \frac{1}{2}\sum_i (t_{pi} - o_{pi})^2$$

The variable t_{pi} is the target value for output node i and input pattern p, and $o_{pi} = f\left(\sum_{j=0}^{n-1} w_{ij} o_{pj} + bias_i\right)$. The global error is calculated as $E = \sum_p E_p$.

2.2.2 *The Hopfield network*

The Hopfield network is a single layer unsupervised network. Each node is connected to every other node as shown in **figure 2.1b**. All of these nodes are input nodes as well as output nodes. Their connections are symmetric, that is, $w_{ij} = w_{ji}$ and $w_{ii} = 0$.

The Hopfield network was introduced by Hopfield (1982) based on the physical models of materials with magnetic properties. Hopfield used this network as an associative memory with binary input and output vectors. Later this network [Hopfield *et al.* (1985)] was improved

to accept continuous input values and used to solve the combinatorial optimisation problem.

This section discusses only the Continuous Hopfield network. A larger range of information can be stored using analogue nodes. These nodes use a Sigmoid function rather than a Hard-limit (**figure 2.2**). Hopfield applied such a network to the TSP (Travelling Salesman Problem) [Hopfield *et al.* (1985)]. This is a difficult optimisation problem that belongs to the *NP*-complete class of problems. The task of the salesman is to visit all the N cities on his list once and only once, returning to his starting point after travelling the minimum possible distance.

Hopfield and Tank [Hopfield *et al.* (1985)] map the *N*-city TSP onto a network with $N \times N$ matrix. The element of the matrix (node) is written as $n_{i\alpha}$. The row i, corresponds to the city number and column α, corresponds to the station of the tour in which this city is visited. A valid tour is characterised by an activation pattern with exactly N nodes active and $N(N-1)$ nodes inactive. There must be exactly one entry of one in each row and column of the matrix, n.

The task of the TSP is to find the tour which has the shortest total length among the valid tours. To allow the network to compute a solution to the problem, Hopfield and Tank represents the following energy function to be minimised,

$$E = E_0 + \lambda_1 E_1 + \lambda_2 E_2 + \lambda_3 E_3$$

E_0 is the total length of a tour and the other terms are intended to ensure constraint satisfaction, the constant λ_n being Lagrange parameters [Müller *et al.* (1990)]. The total length of a tour is written as follows :-

$$E_0 = \frac{1}{2} \sum_{i,j} \sum_{\alpha} d_{ij} n_{i\alpha} \left(n_{j,\alpha+1} + n_{j,\alpha-1} \right)$$

where d_{ij} denotes the distance between the cities i and j. The matrix d_{ij} is the Euclidean distance in a two-dimensional plane which is written as :-

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

The first additional terms of the energy function,

$$E_1 = \frac{1}{2} \sum_i \sum_{\alpha, \beta}^{\alpha \neq \beta} n_{i\alpha} \, n_{i\beta}$$

is chosen such that it vanish if each row corresponding to a city contains a single one with all the other values being zero. The second additional term,

$$E_2 = \frac{1}{2} \sum_{i,j}^{i \neq j} \sum_{\alpha} n_{i\alpha} \, n_{j\alpha}$$

is zero if each column corresponding to a position in the tour contains a single one. The last constraint,

$$E_3 = \frac{1}{2} \left(\sum_i \sum_\alpha n_{i\alpha} - N \right)^2$$

is used to enforce the presence of N entries of magnitude one such that they will be zero only when the total number of one's in the network is N, the number of cities.

The approach to a solution of minimal energy $E[n]$ can be described by a differential equation in time t [Müller *et al.* (1990)] as shown below :-

$$\frac{du_{i\alpha}}{dt} = -\frac{u_{i\alpha}}{\tau} - \frac{\partial E_0}{\partial n_{i\alpha}} - \sum_{n=1}^{3} \lambda_n \frac{\partial E_n}{\partial n_{i\alpha}} .$$

where

$u_{i\alpha}$ is the local field of node $n_{i\alpha}$,

$n_{i\alpha} = \frac{1}{1+e^{-2u_{i\alpha}/T}}$, T is the temperature which determines the slope of the Sigmoid function, and τ representing the typical time constant of the nodes.

The differential equation of $u_{i\alpha}$ has to be solved numerically. The Euler's method is used to replace this differential quotient by the quotient of forward differences. It gives the following results,

$$u_{i\alpha}(t+1) = u_{i\alpha}(t) + \Delta t\left[-\frac{1}{\tau}u_{i\alpha} - \sum d_{ij}(n_{j,\alpha+1} + n_{j,\alpha-1})\right.$$

$$\left. -\lambda_1 \sum_{\beta \neq \alpha} n_{i\beta} - \lambda_2 \sum_{j \neq i} n_{j\alpha} - \lambda_3\left(\sum_{j,\beta} n_{j\beta} - N\right)\right].$$

2.2.3 The Kohonen network

The Kohonen network is a two layer feedforward network as shown in **figure 2.1c**. The first layer is an input layer and the second layer is a grid or map arranged in a one or two dimensional array. These layers are fully interconnected, as all input nodes connect to all nodes in the second layer. The network is trained by unsupervised learning.

The second layer is known as a competitive layer. Incoming patterns are classified by the nodes that they activate in the competitive layer. Similarities among patterns are mapped into closeness relationships on the competitive layer. After training is complete, the pattern relationships and groupings are observed from the competitive layer.

The first step in operating a Kohonen network is to compute a matching value for each unit in the competitive layer by using the following equation,

$$d_i = \sqrt{\sum_{j=0}^{n-1}(x_j - w_{ij})^2}$$

where d_i is the distance between the input pattern and the competitive node i, x_j is an input value for node j and w_{ij} is the weight between the input node j and the competitive node i.

The closest matching unit to a training input is computed as the minimum distance of d_i which is given by

$$d_c = \min(d_i)$$

where c is the winner node.

After the winning node is identified, the next step is to identify the neighbourhood consisting of those nodes that are close to the winner in the competitive layer. The neighbourhood consist of the units that are within a square centred on the winning node c.

The weights are updated for all nodes that are in the neighbourhood and the winning node c based on the following equation :-

$$w_{ij}(t+1) = w_{ij}(t) + \alpha(t)\left(x_j - w_{ij}(t)\right)$$

where α is the learning rate and i is the competitive node in the neighbourhood. The learning rate, α, begins initially at a relatively large value. During the learning process, α is decreased over the span of many iterations. The suggested rate [Dayhoff (1990)] is:-

$$\alpha(t) = \alpha_0\left(1 - \tfrac{t}{T}\right)$$

where α_0 is the initial value set by choice, i.e. 0.2 - 0.5. T is the total number of training iterations and t is the current iteration.

The size of the neighbourhood is also adjusted according to

$$n(t) = n_0\left(1 - \frac{t}{T}\right)$$

where n_0 is initially the neighbourhood size chosen either one-half or one-third of the width of the competitive layer. The neighbourhood is for all (x,y) such that, $c-n<x<c+n$ and $c-n<y<c+n$. Sometimes this calculated neighbourhood goes outside the grid of units in the competitive layer; in this case the actual neighbourhood is cut off at the edge of the grid.

2.2.4 Adaptive Resonance Theory (ART)

The ART is an unsupervised, competitive learning algorithm. It is a two layer network arranged in feedback and feedforward connections as shown in **figure 2.1d**. The layers have different functions - unlike the backpropagation or Kohonen networks. The first layer can be an input layer or a comparison layer and the second layer can be an output layer or a recognition layer. Both are interchangeable during training.

There are three models of ART called ART1, ART2 and ART3 [Beale *et al.* (1990)]. The ART1 is for a binary input value and ART2 is for real value. Both have similar architectures. ART3 uses equations that model the dynamics of chemical neurotransmitters. This section discusses only the ART1 network.

The major feature of ART1 is the ability to switch modes between plastic (the learning state where the internal parameters of the network can be modified) and stable (a fixed classification set), without detriment to any previous learning. The network also displays many behavioural type properties, such as sensitivity to

context, that enables the network to discriminate irrelevant information or information that is repeatedly shown to the network.

An input vector is presented to the input layer and passed to the second layer using the following equation

$$s_i = \sum_{j=0}^{n-1} w_{ij} x_j$$

where s_i is the output of node i in the competitive layer, w_{ij} is the feedforward weight between the input node j and the output node i and x_j is an input value for node j. The second layer is now in recognition mode where the best matching exemplar is calculated as :-

$$s_c = \max(s_i)$$

where c is the winner node to represent one class of pattern.

At the comparison stage - when a new pattern is presented to the first layer, this pattern is compared with the already learned pattern against a vigilance threshold, ρ. The vigilance parameter controls the resolution of the classification process. A low choice of threshold ($\rho < 0.4$) will produce a low resolution classification process, creating fewer type. A high vigilance threshold ($\rho = 1$) will produce a very fine resolution classification.

When $\dfrac{\sum_{j=0}^{n-1} v_{cj}(t)\, x_j}{\sum_{j=0}^{n-1} x_j} > \rho$, where v_{cj} is the feedback weight between the input node j and the competitive node c, then the classification is complete. All weights are then refined as follows :-

(1) $v_{cj}(t+1) = v_{cj}(t)\, x_j$, for the feedback weight.

(2) $w_{cj}(t+1) = \dfrac{v_{cj}(t)\, x_j}{0.5 + \sum_{j=0}^{n-1} v_{cj}(t)\, x_j}$, for the feedforward weight.

When $\dfrac{\sum_{j=0}^{n-1} v_{cj}(t)\, x_j}{\sum_{j=0}^{n-1} x_j} < \rho$, the next best-template matching node of the recognition layer is considered using the equation $s_c = \max(s_i)$.

The learning time for this network is much faster than the iterative convergence procedures proposed for most other NNs such as the backpropagation method. This is because the weights are set to the optimum values in very few learning cycles.

2.2.5 The Counterpropagation network

The Counterpropagation network is a combination of two well-known networks: the Self-organising map of Kohonen and the Grossberg networks [Hecht-Nielsen (1987, 1988, 1989), Dayhoff (1990)]. The basic architecture of the network is shown in **figure 2.1e.**

The first layer is an input layer and the third layer is an output layer. In between these layers is the competitive layer. The competitive layer performs a competitive classification to group the patterns. The learning algorithm at the Kohonen layer is based on unsupervised learning and the learning algorithm at the Grossberg layer is based on supervised learning.

When the input vector is presented to the network, the competitive layer then performs the weighted sum,

$$s_i = \sum_{j=0}^{n-1} w_{ij} x_j$$

Both x and w are normalised first so that their vector lies on a unit radius. w_{ij} is the weight between the input node j and the competitive node i. The node, c with the highest sum calculated as :-

$$s_c = \max(s_i)$$

is considered as the winner node at the competitive layer.

Only the weights of the interconnections that go to the winning node c are adjusted using

$$w_{cj}(t+1) = w_{cj}(t) + \alpha(x_j - w_{cj}(t))$$

where α is the learning rate. The network output values are then compared to the target pattern, and the output layer of weights is updated as

$$v_{jc}(t+1) = v_{jc}(t) + \beta(t_j - v_{jc})$$

where v_{jc} is the feedforward weight between the competitive node c and the output node j, t_j is the target value of output node j and β is the learning rate for the output layer.

2.3 CURRENT NEURAL NETWORK SIMULATORS

In order to define any of the NN simulation programs, a computer programming language is the most suitable. Through the language, the designer has the freedom to explore various simulation programs.

There are several methods to define networks by a computer program. One is to use an already existing high-level programming language such C or Pascal but many NN researchers do not come from a strong computer background. However, many programming languages exist specifically for the NN models [Paik *et al.* (1987), Almassy *et al.* (1990), DasGupta *et al.* (1990), Hu (1991), Korn (1989, 1991a&b), Zell *et al.* (1991), Vellacott (1991), Panetsos *et al.* (1993)]. These programming languages are known as the special-purpose languages. They cover many methods of programming styles such as declarative (descriptive), procedural and object-oriented.

The main objective of providing various styles of programming languages is that they are easy to use. Discussion of how these programming styles have been implemented in a high-level language are explained in the next chapter. A brief view of some NN languages are now given in this section.

2.3.1 Procedural language for Neural Network

The procedural language approach follows the algorithmic step of computation (**section 3.4.1**).

The DESIRE/NEUNET is a new environment for interactive experiments with NNs developed by Korn (1989, 1991a&b). It is for personal computers run under the PC-DOS/MS-DOS operating system. Special versions of DESIRE/NEUNET can generate ANSI C source code and then can be inserted in other user programs, including embedded-computer applications. The DESIRE/NEUNET

language is a high-level language specifically designed for general-purpose NN models. Its program is based on a simple vector/matrix notation for NN models. One-dimensional arrays represent vectors such as node activations and signal patterns, and two-dimensional arrays represent matrices which holds connection strengths or weights.

The structure of the DESIRE/NEUNET language is divided into two parts. The first part is called the interpreter program segment namely 'experiment protocol' and the second part is called compiled program segment namely 'simulation run'. Hence, the translation of the DESIRE/NEUNET program thus involves two separate tasks of interpretation and compilation.

The experiment protocol initially sets all array definitions to zero or can fill any desired array by using data/read assignments. It also involves program loops, modify model variables, simple assignment, setting scales and scalar parameters for graphical display, and then calls a simulation run or runs. A statement to call this simulation run is *drun*. The compiled code or simulation run does 'exercise' the simulation model through a sequence of time steps or trial. It can also manipulate complete arrays and solves difference equations to produce time histories, node activations, performance measures and display or list such results in a variety of ways. This program segment is written under a program named, *DYNAMIC*. The DYNAMIC program segment is separated from the interpreter program by the *DYNAMIC* statement, which must be the only statement on its line.

An example of a DESIRE/NEUNET program is shown in **figure 2.7**. It shows that successive rows of the *INPUT* and *TARGET* pattern matrices serve as input and target vectors for training. A simple least-mean-squares algorithm or Widrow-Hoff LMS algorithm minimises the error measure. Further details, examples and user-manual can be found in Korn (1991b).

```
N = 3
ARRAY layer1[4], layer2[2], bias[4],weight[2,4]
ARRAY INPUT[N,4], TARGET[N,2]
DATA 1,2,3,4;0,0,1,1;-1,0,-1,0 | read INPUT
DATA 10,20;50,60;0,0 | read TARGET
gain = 0.2
min = 0 | max = 1
NN = 30
t = 1 | TMAX = NN - 1 | drun | stop
-----------------------------------------------
DYNAMIC
-----------------------------------------------
iROW = t
VECTOR layer1 = INPUT# + bias
VECTOR layer2 = weight*layer1; min,max
VECTOR error = layer2 - TARGET#
LEARN weight = gain*error*layer1 + moment*weight
DOT enormsqr = error*error
```

Fig. 2.7: A simulation program for two-layer network

2.3.2 Declarative language for Neural Network

A declarative language (also known as descriptive language) for NN is a non-procedural language which allows the user to describe the network topologies and lets the computer undergo algorithmic steps of computation by itself (**section 3.4.2**). A brief discussion of a declarative language for NNs based on Leighton *et al.* (1992) is now presented.

Aspirin/MIGRAINES (version 6.0) is a NN environment developed by the MITRE Corporation. The Aspirin/MIGRAINES system is written for a UNIX environment. Aspirin is a high-level declarative language used to describe arbitrarily complex NNs and their learning algorithms. It includes the definition of the type of network, the size and topology of the network and descriptions of the network's input and output. It may also include items such as user defined function and the user manual for the MIGRAINES system. Aspirin supports the backpropagation learning techniques and topological variations.

The Aspirin program is then compiled by its code generator and generates a C program to simulate the network. It is further compiled using a standard compiler and linked either to the MIGRAINES interface or used with other application-specific systems. MIGRAINES is a terminal-based interactive interface that allows the user to export data from the NN simulation program to graphical packages such as 'Mathematica' [Wolfram (1991)] via UNIX pipes. MIGRAINES was intentionally kept separate from Aspirin so that the limitations of MIGRAINES do not restrict the performance of Aspirin.

Aspirin is organised around the concept of a 'black box' description of NN. A black box NN is an abstract unit which receives external input and produces some output. A complete NN is one example of a black box. A black box can also be a subnetwork of a larger, complex NN system.

2.3.3 Object-oriented language for Neural Network

Object-oriented concepts will be discussed in **section 3.4.3** of the following chapter. A brief example of an object-oriented language for NN based on Hu (1991) is now presented. Hu gave an informal overview of a general-purpose NN simulation language called an Object-Oriented Neural Network Language (OONNL). OONNL follows many characteristics of object-oriented methodology and embodies the features of NN models. The compiler for OONNL was implemented on a SUN3 workstation.

The structure of OONNL is divided into two parts. The first part is the description of the NN model and the second part is the description of information flowing amongst the processing units (nodes or neurons) and information processing in the units.

The description of the NN model involves defining the NN topology such as specifying the processing units and connections as well as the specification of the value of units such as activation function and connection strength. OONNL defines a processing unit as an object which includes the unit name, class name, its connection with other units and their weights, and name of procedures such as learning rule and activation function. OONNL implements inheritance by class. A class is an abstract of objects. Units can be classified as input, hidden and output units and share common attributes such as activation function. OONNL provides a set of statements to define a class that includes a number of units sharing common attributes.

Data flow and control flow as well as object-oriented methodology in OONNL is realised through the processes of information flowing and information processing in the unit. Data flow involves processes such as get data from the outside world, feed data forwards or backwards, and control flow involves processes such as how to stop iterations when the desired learning is satisfied.

CHAPTER 3

BASIC COMPILER CONCEPTS

This chapter covers the basic concepts of compiler design which comprises its definition and comparison with an interpreter; the process of compilations; high-level programming methods; a brief discussion of some popular compiler-construction tools or compiler generators; and the general structure of a NN compiler called NEUCOMP and its language (the NEUCOMP language). A high-level programming language is a computer language that a human can understand whereas a low-level language is the machine language.

An interpreter is another technique of translating a high-level programming language. The comparisons of the compiler approach over an interpreter and its advantages in terms of execution speed will be discussed briefly.

The third section of discussion is about the process of compilation. It shows the technique of how the design of a compiler is broken down into many phases or modules. From the understanding of this technique, we can then understand how a NN compiler (NEUCOMP) can be developed. Its explanation can be found in the next chapter.

The fourth section of discussion explains how high-level programming languages are designed into various methods such as procedural, declarative, object-oriented and functional programming methods. A brief discussion of each is given in this section. A clear understanding of which programming method is easier to program and convenient to use leads to a preferred choice of a language method for NEUCOMP.

Since designing a compiler is a complicated task, which may involve frequent changing of program syntax, the use of compiler-construction tools or compiler generators are recommended. Popular tools such as 'Lex' and 'Yacc' are available under the UNIX operating system can help the designer to reduce programming maintenance. These are explained in the fifth section.

In the last section, the structure of NEUCOMP and the NEUCOMP language are introduced.

3.1 COMPILER

A compiler is a program which translates a program written in one language, the source language, to an equivalent program in a second language, the target language [Aho *et al.* (1986), Bennett (1990)]. During this translation process, the compiler reports the presence of errors in the source program together with diagnostic information about the source program being compiled. A target language can be another programming language or the machine language that already exists in the computer being used.

Figure 3.1 shows the general structure of the compiler. Typically the source language will be in a high-level programming language such as FORTRAN or Pascal, and the target language will be the machine code for the computer being used or an assembly language.

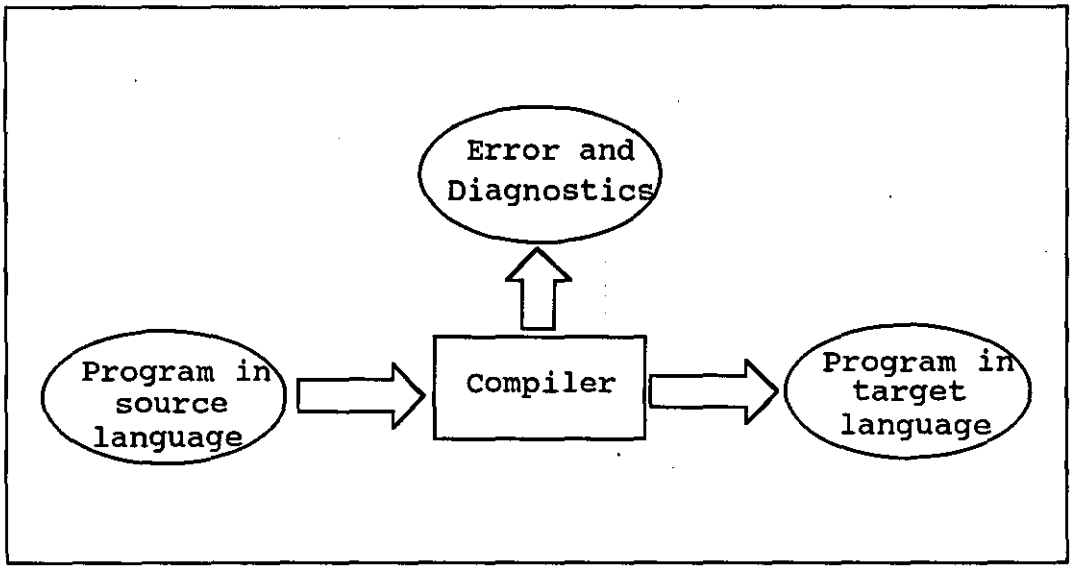


Fig. 3.1: Overall structure of a Compiler

3.2 INTERPRETER VERSUS COMPILER

Translation of the source language into the target language can be done by two approaches, namely, as the compiler and the interpreter [Aho *et al.* (1986), Bennett (1990), Ford (1990)].

The compiler reads the source code program and converts it into the target code program. This means that the entire program code is translated once and resaved as its target code. All errors of syntax or grammar found by the compiler during the translation process will be displayed together with the diagnostic information. This includes the type of errors that have occurred and the type of corrections that should be done. Recompilation is then required to the corrected source program. If there is no error after the end of translation, the translated code is then executed. The possibility of finding any errors of syntax during this execution process does not happen. There is no need to recompile the program once compiled when we want to execute the code program repeatedly.

An interpreter reads the program's statements one at a time. Each single statement is translated and if an error is found, the translation process is stopped. The user can then correct it immediately and if there is no error, the translated code is then executed. The process is then continued for the next statement. This means that the translation and execution phases occur together and not separately as in the case of the compiler. Doing the translation and execution at the same time has given a slight advantage to the interpreter. When error occurs we can immediately pinpoint its source from the original high-level language program. This is often a great help when developing and debugging programs. However, interpreters suffer from poorer execution speeds than their compiler competitors, particularly involving large scale repetition of code in loops. This is because the source code must be translated each time it is executed,

and then the repeated translation of the same code within a loop is clearly wasted. Ford (1990), has shown a comparison of Borland's Turbo BASIC which offers a compiled BASIC which is compatible with several interpreted versions based on the same hardware configuration. The result shows that the execution speed of a compiled version is two times faster than an interpreted version.

The choice of whether to compile or interpret is to a large extent influenced by the nature of the high-level language and the environment in which it is used [Bennett (1990), Ford (1990), Wilson *et al.* (1993)]. For example, FORTRAN is relatively simple and designed for translation to machine code. It is often used for solving big numerical problems on mainframe computers, where the speed of execution is essential. It is thus invariably compiled. BASIC, on the other hand, is mainly used on personal microcomputers where clearly error handling is important. The lack of processing power and memory could make compilation very difficult. However modern interpreters such as for the LISP language often use both interpretation and compilation. Programs are interpreted during program development to avoid time-consuming compilations each time the program is changed and to give clear error handling. When development is complete, compilation can begin.

Most language translation use a combination of compilation of high-level language into an intermediate low-level language which is then interpreted into a machine code [Bennett (1990)]. For example, the UCSD Pascal compiler, generates an intermediate code, PCODE, for interpretation. This is because compilation of high-level language into machine code is time consuming. If an error happens during execution it is difficult to relate the machine code that caused the error to its equivalent high-level code. It is easy to compile a high-level language into an intermediate language, i.e. an assembly language which is not too time consuming and

efficient to execute. When an error occurs during execution the intermediate code is easier to relate to the source language.

3.3 THE PROCESSES OF COMPILATION

The process of compilation is broken down into two main parts - analysis and synthesis [Aho *et al.* (1986)]. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. Synthesis requires more specialised techniques. The analysis is broken down into three phases - lexical analysis, syntax analysis and semantic analysis. The synthesis is also broken down into three phases - the intermediate code generation, code optimisation and code generation. Each phase transforms the source program from one representation to another. The process of compilation as shown in **figure 3.2** has become the standard routine in the development of the compiler [Aho *et al.* (1986), Bennett (1990)]. However in practice, some of the phases maybe grouped together and the intermediate representations between the group phases need not be so explicitly constructed.

Often, the phases are collected into a front-end and back-end [Aho *et al.* (1986), Bennett (1990)]. The front-end consists of phases that depend primarily on the source language and are largely independent of the target machine. These normally include lexical and syntax analysis, the creation of the symbol-table, semantic analysis, and the generation of intermediate code. The front-end also includes the error handling that goes along with each of these phases. The back-end includes those portions of the compiler that depends on the target machine. Generally, these portions do not depend on the source language, just the intermediate language. The

back-end takes the intermediate language representation as input. It undergoes the code optimisation phase, the code generation along with the necessary error handling and symbol-table operations.

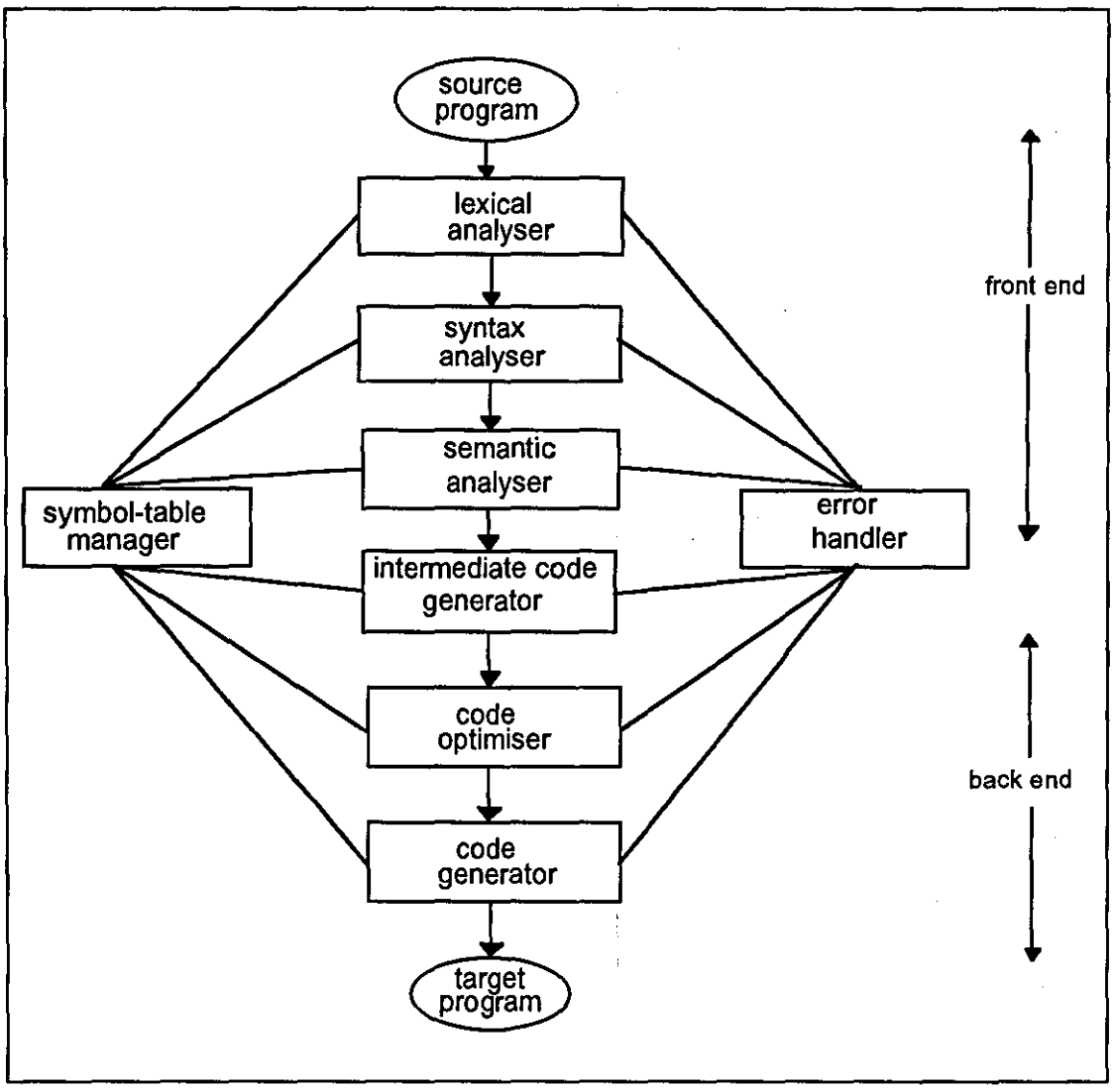


Fig. 3.2: Phases of compilation

Distinguishing the front-end with the back-end can help to produce compiler portability. For example, to produce the same language running on different machines, its associated back-end needs to be modified. To compile several different languages into the same intermediate

language, a common back-end running on the same machine can be used.

Discussions on phases of **figure 3.2** are now explained. The lexical analysis is the first stage in the process of compiling the source program. The stream of characters making up the source program is read from left-to-right and grouped into tokens. A token is a sequence of characters having collective meaning. The terminal symbols of the grammar are taken as complete entities, such as integer or variable names of complete keywords rather than the individual characters. The program that carries out this analysis is called a lexical analyser or scanner.

The syntax analysis phase groups the stream of tokens from the lexical analyser to form a valid sentence or grammatical phrases. Usually, the grammatical phrases of the source program are represented by a parse tree. The program that undergoes this analysis is called a syntax analyser or parser.

The semantic analysis phase checks the parse tree generated by the syntax analyser for semantic errors. It determines which variables are to hold integers, and which are to hold floating point numbers. It also checks that the size of all arrays are defined.

After the syntax and semantic analyses, the parse tree produced is converted into an intermediate representation. This representation can be the three-address code for a general-purpose assembly language which is still not dependent on the target machine. It serves as an interface between the front-end and back-end. The three-address code consists of a sequence of instructions, each of which has at most three operands. When generating these instructions, the compiler has to decide on the order in which the operations are to be done such as the multiplication precedes the addition operation in the source program.

The code optimisation phase attempts to improve the intermediate code into a more efficient equivalent, so

that when translated into a target code it runs faster. Basically the tasks in the code optimisation are to minimise the number of operations carried out in the source program by giving an alternative solution. It also minimise the number of memory accesses. The meaning of a program does not change. A significant fraction of time of the compiler is spent on this phase.

The final phase of the compiler is the generation of the target code, consisting of assembly code or machine code. Memory locations are selected for each of the identifiers used by the program. Intermediate instructions are then translated into a target assembly statements or a sequence of machine instructions that perform the same task.

Two other activities, symbol-table management and error handling, as shown in **figure 3.2**, interact with those six phases. The programs involved are called symbol-table manager and the error handler.

Symbol-table management is an essential module in a compiler. It builds up information about the identifiers used in the source program and collects information about various attributes of each identifier. These attributes may provide information about the storage allocated for an identifier and its types. In the case of procedure names, it provides information such as the number and types of its arguments, the method of parameter passing and the type returned. A 'symbol table' is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. For example, in the lexical analysis it may only hold the text of an identifier's name. During the syntax and semantic analyses, information about the identifier's type and scope will be added. When doing the semantic analysis and intermediate code generation, we need to know what the types of identifiers are, so that the source program uses them in a valid way and the

proper operations can be generated on them. During code generation we may wish to associate an address with an identifier.

The error handling involves detecting the error and reporting the type of error undergoing diagnosis. Every phase in the compiling process can encounter errors. However, after detecting an error, the phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. The syntax and semantic analyses phases usually handle a large fraction of the errors detectable by the compiler. The errors where the token stream violates the grammar rules of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved. This can be operations such as to add two identifiers, one of which is the name of an array and the other the name of a procedure. The intermediate code generator may detect an operator whose operands have incompatible types. The code optimiser, doing control flow analysis, may detect that certain statements can never be reached. The code generator may find a compiler-created constant that is too large to fit in a word of the target language.

These are all important considerations in order to obtain the true compilation.

3.4 PROGRAMMING LANGUAGES

A computer is a tool that solves problems by means of a programming language. Computers obey instructions which are issued to them. In order for the instructions to be understood both by the person who issues them and by the computer which obeys them, they must be issued in a particular form. The set of instructions and the rules by which they are to be issued and acted upon, forms the basis of the computer language. A computer program is a series of instructions which are executed in an appropriate order to perform a particular task [Ford (1990), Wilson *et al.* (1993)].

Programming languages are classified into low-level programming languages and high-level programming languages. A low-level language is one which is close to the machine's own language, and is therefore usually harder for humans to use. Such languages are assembly language and machine codes. Machine code languages are a series of computer instruction written in binary. An assembly language follows a series of machine codes' instructions but they are written using mnemonic codes [Ford (1990), Wilson *et al.* (1993)]. Programming at low level is a very difficult task. It requires a long job for a specialist computer expert who might take many hours to find any error. Because of this a high level language is designed to be easy for humans to learn and use. These languages are sometimes known as human-oriented languages.

All languages except machine code itself, need to be translated before they may be executed by the computer's processor. Assembly language is translated using an assembler. An assembler is a program written in machine code which is able to translate the assembly language instructions into the machine code which they represent. High level languages are translated using a compiler or an interpreter (**section 3.2**).

Nowadays there are many high level languages. Their existence has an objective at a higher level than others. These will be the languages which make a particular attempt to provide ease of programming for human beings [Ford (1990)]. However they can be grouped into a variety of styles. The most commonly identified programming styles are procedural, declarative, object-oriented and functional programming [Maeder (1991)]. The existence of the many programming styles are normally suited to specific applications. For example, FORTRAN is designed to suit numerical and scientific computation, COBOL is for business-data processing and PROLOG is suitable for logic programming such as an expert system and natural languages processing [Ford (1990), Wilson *et al.* (1993)].

3.4.1 Procedural languages

Conventional programming languages such as C, Pascal and FORTRAN support the procedural style. It involves frequent use of assignment statements to change the state of the computation and solve problem algorithmically [Springer *et al.* (1989)].

The following steps describe how to solve a problem algorithmically :-

(1) An understanding of the problem.
(2) A solution to the problem is designed and this solution is broken down into a series of distinct tasks of development of the algorithm.
(3) The tasks required are then translated into a suitable series of programming statements.

The advantage in designing an algorithm is that the programmer has a free choice of choosing the best method for efficient execution [Ford (1990)].

A brief description of some procedural languages are then explained. BASIC - Beginners' All-purpose Symbolic

Instruction Code, was motivated by the desire to have available a programming language which would be simpler to learn than FORTRAN. It is an interpretative language. It has become particularly commonly used for programming smaller microcomputer systems.

FORTRAN, FORmula TRANslation, was based on the original purpose of the language - to solve numerical computations. It was the first high-level language introduced, in 1950. At that time the majority of programming was being undertaken by programmers working in assembly language or machine code. FORTRAN became popular because it served a realistic and desirable alternative to low-level language programming for mathematical and scientific applications.

Pascal is commonly taught in courses on programming. Pascal is usually preferred to BASIC as the introductory language for specialist computer science students because it is well structured. This helps to make a program more readable.

COBOL which is an acronym for COmmon Business Oriented Language is suitable for use in writing programs which could handle efficiently large amounts of data in file processing applications.

The C language has become popular recently for a wide variety of programming applications. C may be distinguished from other languages by its chief design goal to be a tool for working programmers. It is flexible, convenient, powerful, portable and efficient. It is designed by programmers for programmers, and has become one of the most popular and widely used programming languages for the development of applications. Amongst facilities provided are compact codes such as the actual statements which need to be written in order to perform a given operation in C are significantly shorter than the corresponding instructions in many other languages. C provides both high-level as well as low-level language support such as assembly language which is required by a system programming

environment. It is widely available on a variety of different computers including UNIX operating system.

3.4.2 Declarative languages

A declarative language provides an efficient way for programming in logic. It is mainly used in the field of artificial intelligence.

An advantage of the use of a declarative language is that much of the work in writing a program is undertaken by the computer whereas the procedural approach involves human decision processes in designing the suitable algorithm. When a declarative language is used to solve a problem, the programmer is relieved from the responsibility to define the method to be used, which is instead selected by the language [Ford (1990)].

The steps in writing a program in a declarative language are :-

(1) understand the problem. The programmer's tasks are restricted to reaching a clear understanding of the problem.
(2) code as a program that describes the problem to the computer using suitable language statements.

Naturally the language cannot be expressed to display the same intuition which the programmer might display. Therefore, the efficiency of writing the actual program which is often far faster in a declarative language but not efficient in the execution of the result.

Declarative languages are suitable in the development of expert systems and other database programs. They involve logical decision-making programs many of which are particularly difficult to write in a procedural language. An example of logic programming is PROLOG - PROgramming in LOGic.

3.4.3 Object-oriented languages

A recent advance in programming languages has been the adoption of the object-oriented style of programming. This style is particularly suitable for simulating objects in the 'real world' and for structuring large systems in ways that allow recurrent patterns of computation to be shared by similar objects [Springer *et al.* (1989), Ford (1990) and Wilson *et al.* (1993)].

In this style of programming, certain objects are defined that respond to messages passed to them. We can think of an object as a computer dedicated to solving a particular type of problem. The input is the message passed to the object, the object does the computation, and the output is the value returned by the object. Objects provide a way of combining characteristics and operations to give a level of abstraction beyond that offered by records and procedures. Objects are described as some local data together with a set of procedures that operate on that data. All calculations are performed by sending messages to objects, and problems are solved by identifying real-world objects and modelling them by object-oriented programming.

Object-oriented languages support data abstraction and information hiding. An object has a hidden local state and exports operations that can act on this state. Meanwhile an object-oriented program consists of a set of objects that communicate with one another through calls of these exported operations. It also supports the concept of inheritance and dynamic binding. A central feature of object-oriented programming is that new classes are not created from scratch, but by inheriting information from existing classes which they can then modify or extend. The new class is said to be a subclass of the class from which it was derived, a superclass. Software reuse is central to the object-oriented approach. Object-oriented systems typically have a large

number of predefined classes from which new classes may be created.

An object of a subclass can always be used when an object of its superclass is expected. When an operation originally defined in a superclass is redefined in a subclass, the decision as to which operation is applicable in a given situation can be delayed until run time, that is, we have dynamic binding. Dynamic binding, where the decision about which version of an operation is to be used is delayed until run time, is then examined. Objects send messages to one another. On receipt of a message at run time, an object decides which method it will use in response.

Some examples of object-oriented languages are Smalltalk and C++. Smalltalk was designed to be used with powerful personal computers complete with windows, pop-up menus, icons and mouse pointing device, which has led the way in providing a user-friendly interface for both the expert and non-expert user. C++ is a hybrid between a traditional imperative language and an object-oriented language.

3.4.4 Functional languages

A functional language is mainly characterised by the replacement of assignment statements by calling functions. It involves the evaluation of an expression through a calling function instead of changing the value of variables through an assignment [Ford (1990), Wilson *et al.* (1993)]. It is closer in spirit to mathematics. Procedural languages use a sequence of commands to carry out the derived operation whereas in a functional language they are recursively executed. The problem with an assignment statement is that when it used in conjunction with reference parameters or non-local variables in subprograms it can lead to side effect and aliasing [Wilson *et al.* (1993)].

Functional language behaves like a mathematical function. Its program usually consists of a series of function definitions followed by an expression that involves the application of the function. It represents symbolic expressions and other information in the form of list structures in computer memory. Problem solving using a functional language involves symbolic manipulation that requires dealing with mathematical functions and formal mathematical reasoning such as Lambda calculus.

The functional language approach arose mainly because the designers were mainly mathematicians. They were particularly interested in applying computing to artificial intelligence problems such as game playing, theorem proving and natural language processing, and developing a mathematical theory of computation.

LISP which is a List Processing language, is an example of a functional language. It was the first functional language to be widely used. It was implemented during the period from 1959 - 1962 [Wilson *et al.* (1993)].

3.5 COMPILER GENERATORS

Designing a compiler is not an easy task. It involves a very large, complex programming project. Therefore the use of compiler-building tools can be a significant help to the compiler writer. Compiler generators are the software-development tools used to generate various phases of a compiler [Aho *et al.* (1986), Bennett (1990), Lemone (1992)]. Such tools are also referred to as 'compiler-compilers', or 'translator-writing systems'. **Figure 3.3** shows how the compiler generators generate a compiler based on rules to be defined by a compiler writer. Although this figure implies that an entire compiler can be created by a compiler generator, in fact, compiler generators cannot yet generate entire compilers automatically [Lemone (1992)].

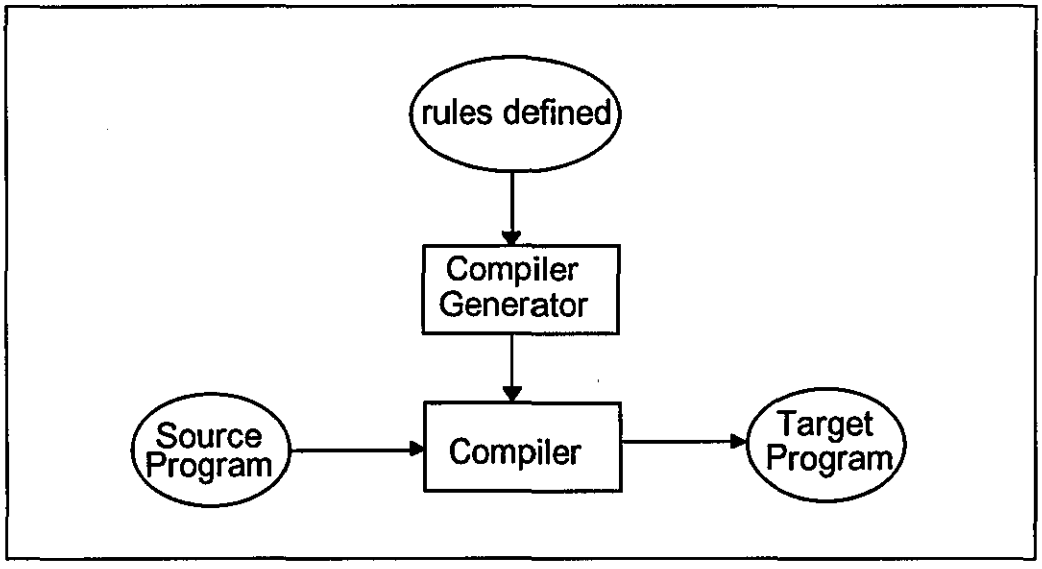


Fig. 3.3: An Overview of a Compiler Generator

Existing compiler generators are implemented in various phases [Aho *et al.* (1986), Lemone (1992)]. For the front-end of a compiler, the phases are often termed a lexical analyser generator, syntax analyser generator and semantic analyser generator. Generator phases for the back-end of compilers are still very much a research

topic although work has been done on code generator generators [Aho *et al.* (1986), Lemone (1992)].

The best-known compiler tool is 'Yacc - Yet Another Compiler-Compiler', written by Steve Johnson (1978). Yacc runs on the UNIX operating system and is associated with another tool called 'Lex - A Lexical Analyser Generator', written by Lesk *et al.* (1975) which generates a scanner.

3.5.1 Lex - A Lexical Analyser Generator

Lex is a scanner generator written in C. It takes a description of a set of sentences that make up the grammar for token to generate a scanner program called *yy.lex.c* [Lesk *et al.* (1975), Aho *et al.* (1986), Bennett (1990), Lemone (1992)]. It can be compiled and linked with other compiler modules. The program *yy.lex.c* contains an integer-valued function called, *yylex*() which returns the next token from the source program. An example of such tokens is a sentence "1","0","0", which represent a token for number 100.

The syntax of token is described in the form of a regular expression. The regular expressions and actions to be carried out are specified by the user in the Lex program. A Lex program contains a format that allows the user to define a type of string representing token and action code written in a C program to be carried out. For example, a list of characters that represent an identifier, a C routine *mkname*() is called to save an identifier name in the symbol table and return the token *IDENTIFIER* from this routine. Similarly, for a list of digits that represent number, a routine *mkval()* converts this string of digits into a numeric value and return the token *NUMBER*. The routine *mkname()* and *mkval*() are specified by a user in the Lex program. **Figure 3.4** shows how a Lex generates the scanner program written in C, *yy.lex.c*. It is then compiled to produce an object code

known as scanner. The output of scanner is the list of tokens.

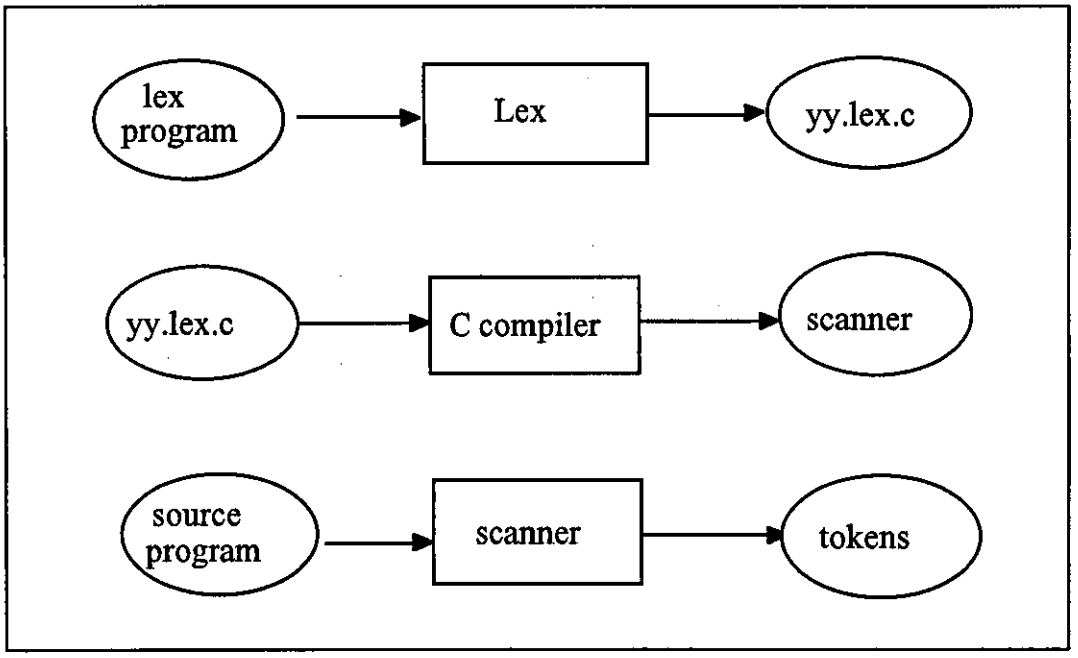


Fig. 3.4: An Overview of Lex

3.5.2 Yacc - Yet Another Compiler-Compiler

Yacc is a parser generator written in C program. It takes a specification of a programming language grammar and semantic actions, and generates LALR(1) parsing tables and a shift-reduce parser called *y.tab.c* [Johnson (1978), Aho *et al.* (1986), Bennett (1990), Lemone (1992)]. It can be compiled and linked with other compiler modules such as the scanner generated by Lex as shown in **figure 3.5.** The program *y.tab.c* contains a routine called *yyparse*() which is called to carry out parsing. The function *yyparse*() in term uses *yylex*() from program *yy.lex.c*, for the next token from the source program. *y.tab.c* is also compiled to produce an object code known as parser. Its input is the list of tokens produced by scanner and its output is the parsed tree.

The Yacc program contains a format that allows the user to define valid sentence or grammar rules for the

source language. For example a sentence *if* (*expression*) *then* *statement* is a grammar rule for 'if statement'. The string *if* and *then*, the symbol '(' and ')' are tokens produced by the scanner while *expression* and *statement* are other grammar rules. Further details of grammar rules, and the use of Yacc and Lex are discussed in **chapter 4** when designing and implementing NEUCOMP.

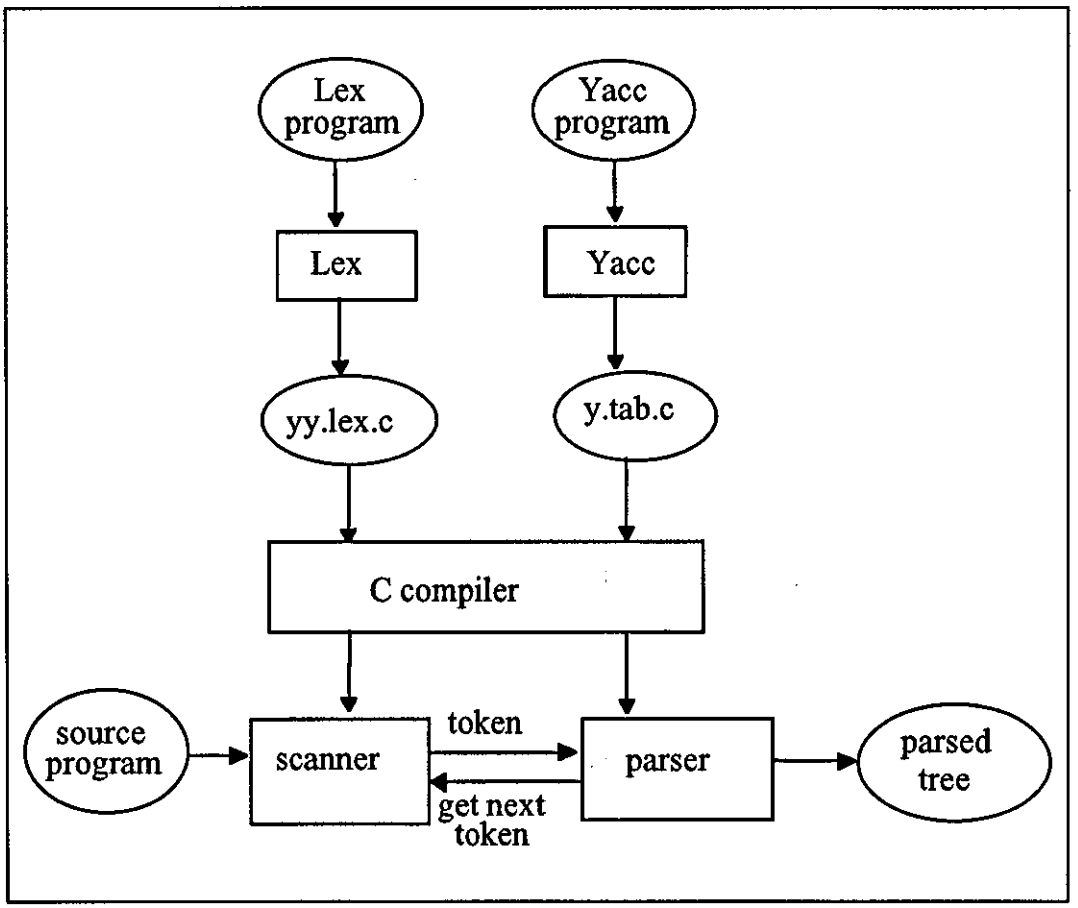


Fig. 3.5: Lex with Yacc

3.6 STRUCTURE OF A NEURAL NETWORK COMPILER

The general structure of a NN Compiler called NEUCOMP and its language (the NEUCOMP language) are now explained in this section.

3.6.1 The General Structure of NEUCOMP

The compilation of a programming language naturally breaks down into a number of logical phases (**section 3.3**). These phases may run simultaneously, or they may run consecutively.

Compilation of a high-level program has been proved to produce a high performance result [Bennett (1990), Ford (1990)]. However, to develop a true compiler is a difficult task. It involves a large, complex programming project [Aho *et al.* (1986), Lemone (1992)]. NEUCOMP takes a simpler approach as the objective here is to study the suitability of the NEUCOMP language to perform general implementations of NN models. The simplified phases of compilation are shown in **figure 3.6**. The reason is to provide an ad hoc and workable compiler at an early stage so that when it is successful a true compiler can be later developed. The C language is chosen as the target language because it is portable to any machine under the UNIX platform. It has a [illegible] structured data type, is machine independent and has more mathematical library routines provided by the UNIX operating system.

The semantic analysis for NEUCOMP is done during the syntax analysis. The output after the semantic analysis is the target program which is in C. The code optimisation and code generation (**section 3.3**) are not carried out on the C program because the C compiler has its own code optimiser. However, for the execution of an assignment statement involving a matrix/vector manipulation there may be repeated loops of the same matrix/vector size. This may affect the performance of the program. The loop optimiser is responsible for

combining the same vector or matrix loop of every matrix/vector assignment from the target program in order to remove the repeated loops.

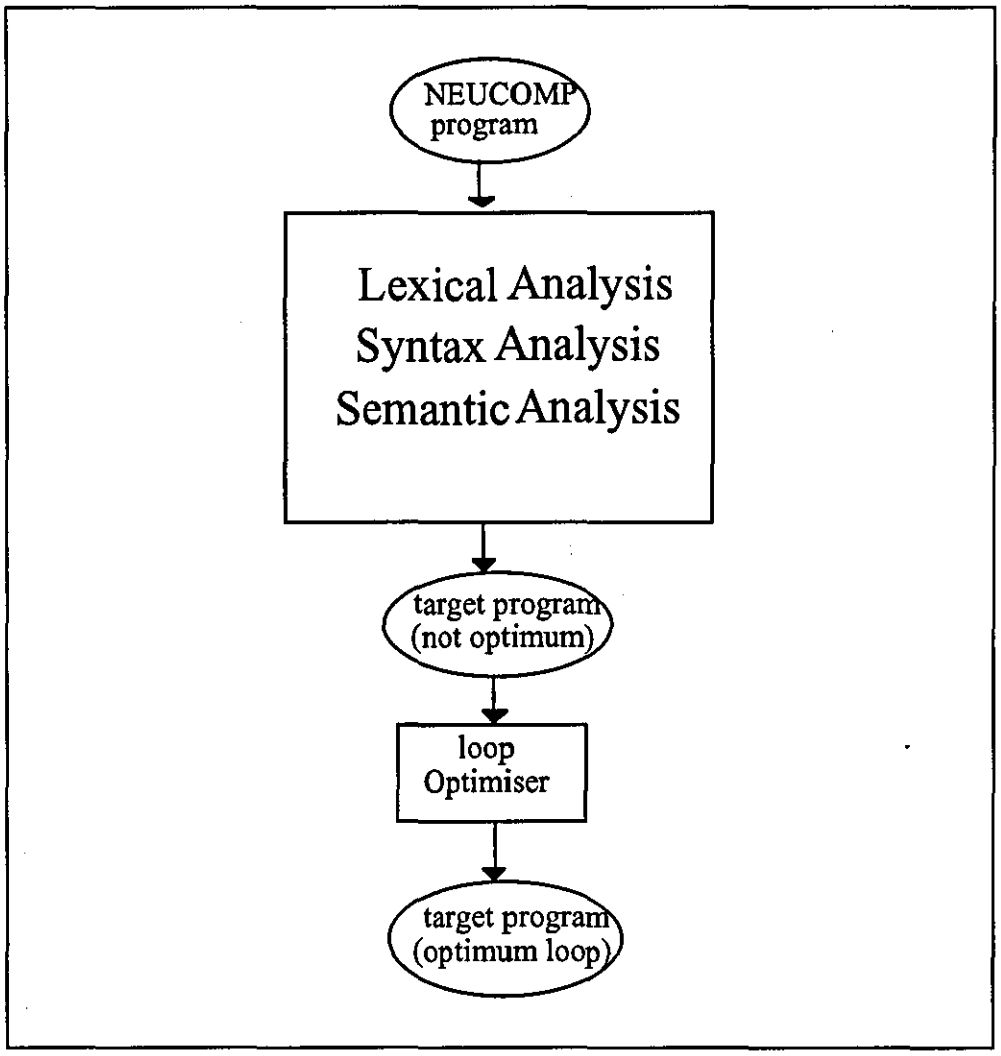


Fig. 3.6: Phases of compilation

3.6.2 The General Structure of the NEUCOMP language

Many of the properties of the NN models described are governed by the mathematics of linear algebra [Rumelhart *et al.* (1986)]. Vector and matrix analysis are a useful way to describe a pattern of numbers. In a NN model, many quantities are best represented by vectors and matrices. For example, activations of a node can be written as,

$$\mathbf{y} = f(\mathbf{W}*\mathbf{x}+\mathbf{b})$$

where y is a vector of a current node, x is a vector of the previous node impinging on the current node, W is a matrix which represents the connection strengths between nodes x and nodes y, $\mathbf{b}$ is a vector which represents a bias to node $\mathbf{y}$, and f is an activation function, i.e. Sigmoid function.

The NEUCOMP language is a procedural high level language which is designed for a user to write a simulation program specifically for any NN model. It contains information regarding the list of mathematical specifications required by the NN models as well as standard high-level programming statements such as *if..then..else* and *while..do* statements. The mathematical specifications used are represented by either a scalar, vector or matrix manipulation. The NEUCOMP will translate these expressions into the actual loop expressions.

The idea which brought about the development of the NEUCOMP language came from the mathematical representation of the DESIRE/NEUNET language (**section 2.3.1**). The DESIRE/NEUNET program is translated using a combination of both an interpreter and compiler whereas a NEUCOMP program is translated by a compiler. DESIRE/NEUNET uses an interpreter on a program segment which contains variables definition, simple assignments and loop initialisation on matrix/vector variables. Ford (1990) has shown experimentally that interpreters suffer from poor execution speed particularly those involving large scale repetition of code in loops (**section 3.2**). Also as explained in **section 3.2**, an interpreter is useful for program development because it avoids time-consuming operations during compilation each time the source program is changed. Furthermore, most language translations use a combination of compilation of high-level language into an intermediate low-level language which is then interpreted into a machine code.

The Procedural approach is chosen because traditionally this approach has been established since the evolvement of the FORTRAN language. Furthermore, the procedural approach allows a list of mathematical specifications to be easily organised or written algorithmically. Other approaches as mentioned earlier which are declarative (descriptive), functional and object-oriented are not suitable for writing a series of mathematical specifications. An example of the declarative approach is Nessila [Korb *et al.* (1989)]. This language performs poorly because it is hard to implement and takes too long to generate a large NN. However, its new version, Nessus, which is based on the procedural approach is more efficient [Zell *et al.* (1991)]. To my knowledge, there has been no implementation of a NN language based on the functional approach. Even though a functional language such as LISP has been established for AI research however for NN research, it is more difficult to understand and implement [Myler *et al.* (1992)]. The object-oriented approach is a new field for a NN language. Hence, only those people with a good background in computing prefer to use an object-oriented language.

The NEUCOMP language structure (**figure 3.7**) follows the general structure of the C-language [Kernighan *et al.* (1980)]. It shows that *program_name* under *NEURALNET* is the name that must be given as a program heading for a simulation program. The *identifier-declarations* is a section where one or more variables for scalar, vector or matrix operations are declared either as global or local. All variables must be of type real, integer, string or file. The content within *MAINPROGRAM ... END* is the body of a program or main program where one or more statements or *statement-list* can be written to carry out the simulation. The *statement-list* such as 'assignment-statement', 'if-statement', and calling 'subprogram' are parts of NEUCOMP language statements.

The *subprogram_declarations* are declarations of one or more subprograms of types procedure or function. The structure of a subprogram (**figure 3.8**) follows the same structure of the main program. A subprogram allows the reference by a name to a collection of statements which perform a clearly defined purpose. The procedure statement serves like a function in C of type *void* and the function statement serves like a function in C which returns a value. However the type of a return value is based on a type of a variable after statement *RETURN*. Provision of procedure and function can make the program well structured and help to improve program readability. The *argument* in a subprogram contains one or more variables depending on the argument of its calling subprogram. The argument in the calling subprogram acts as a passing parameter to the declared subprogram.

The capital letter words such as *NEURALNET*, *MAINPROGRAM*, *END*, *PROC*, *FUNC* and *RETURN* as shown in **figure 3.7** and **figure 3.8**, are the reserved words.

```
NEURALNET program_name
identifier_declarations (global use)

MAINPROGRAM
identifier_declarations (local use)
statement_list
END;

subprogram_declarations
```

Fig. 3.7: Structure of the NEUCOMP language

```
PROC procedure_name argument
identifier_declarations (local use)
statement_list
END;

FUNC function_name argument
identifier_declarations (local use)
statement_list
RETURN variable;
```

Fig. 3.8: Structure of subprograms

CHAPTER 4

A NEURAL NETWORK COMPILER

This chapter discusses the design and implementation of a NN compiler called NEUCOMP which includes the mathematical specifications on scalar, vector and matrix operations; designing and implementing the compiler; compilation of the compiler modules; some NNs simulation programs; and developing the required graphical displays.

The design and implementation of NEUCOMP includes the definition of the NEUCOMP language which is based on grammar rules or productions and the process of compilation which involves the lexical analysis, syntax analysis, semantic analysis and optimising the loops or loop improvement. The lexical and syntax analyses phases are implemented using the compiler tools, Lex and Yacc (**section 3.5**) which both generate the C program modules. The semantic analysis is implemented in conjunction with the syntax analysis. These compiler modules are compiled with other C program modules, i.e. user-support routines and loop optimiser. The executable compiler program is called NEUCOMP.

The NN simulation programs for some NN models are developed and compiled by NEUCOMP. The chosen models are the backpropagation, Kohonen, ART1 and Counterpropagation networks. They are chosen based on the differences of their structures and learning algorithms (**section 2.2**).

The graphical features are used for viewing and analysing the simulation results. However, the NEUCOMP language does not provide a statement to display the results. An existing graphical software is recommended for such purposes. It provides facilities to allow us to write the graphical program easily. The graphical features that are explained in this chapter are the programs to display the NN structure, XY-graph and plotting points for data clustering. The simulation results from the NEUCOMP program can then be transferred to a graphical software package.

4.1 MATHEMATICAL SPECIFICATIONS

An assignment statement is presented as follows:-

variable assigntype expression (4.1)

where *variable* can be a scalar, vector or matrix variable, *assigntype* is a mathematical operator of types '=','+=', '-=', '*=' or '/=' and *expression* can be a variable or variables in a mathematical expression. A scalar variable holds a single value. A vector variable is a one-dimensional array and matrix variable is a two-dimensional array.

For example, a mathematical specification for the activation function (4.2) and the modification weights during training (4.3) in the backpropagation algorithm are expressed in the matrix-vector forms as follows:-

layer2 = f(weight*layer1 + bias2) (4.2)

weight += alpha*dweight + beta*cweight (4.3)

where *alpha* and *beta* are scalar variables, *layer1* and *layer2* are vectors which represent the nodes in the first and second layers, *bias2* is a vector variable which represents 'bias' on the second layer, *cweight* is a matrix which contains the change of weight, *weight* is the connection strength between the first and second layers, *dweight* is the matrix derivative of *weight* and f is the activation function such as the Sigmoid function (**figure 2.2c**). The operator '+=' is equivalent to

weight = weight + ...

The compiler translates (4.2) and (4.3) into the following algorithms :-

```
for (i = 0, m-1) {

  layer2[i]= f( Σ_{k=0}^{n-1} (weight[i][k]*layer1[k]) + bias2[i]);
  for (j = 0,n-1)
    weight[i][j]+=alpha*dweight[i][j]+beta*cweight[i][j];
}
```

where *weight*[*i*][*k*] means the connection weight from node *k* to node *i*.

An assignment statement is divided into 3 types. The first is a scalar assignment, second is a vector assignment and third is a matrix assignment. In a scalar assignment, the left hand-side (4.1) is a scalar variable and its right-hand side must give a scalar result.

In the following section, vector and matrix assignments will be discussed. The vector and matrix variables are named as, vector and matrix, and a scalar variable is named as scalar.

4.1.1 Vector assignment

In a vector assignment, the left hand-side of (4.1) is a vector variable and its right hand-side must contain the following expressions :-

4.1.1.1 Scalar expression

Scalar expression can be a number, a scalar variable or mathematical expression that gives a scalar result. For example :-

```
vector = scalar + 3
```

This means that the calculated scalar expression is assigned to all vector components. The equivalent translated statement is as follows :-

```
for (i = 0, m-1)
     vector[i] = scalar + 3;
```

4.1.1.2 Vector expression

A vector expression can be a vector variable or mathematical expression that gives a vector result. The size of the vector must be equal to the vector size of the left hand-side of (4.1). For example:-

```
vector = vector1 + vector2
vector = scalar1*vector1 - scalar2
```

Each component of a vector expression is assigned to each component of a vector variable. The equivalent translated statement is as follows:-

```
for (i = 0, m-1) {
   vector[i] = vector1[i] + vector2[i];
   vector[i] = scalar1*vector1[i] - scalar2;
}
```

4.1.1.3 Matrix-Vector multiplication

Matrix-vector multiplication is a product between a matrix and vector variables. The column size of a matrix must be equal to the size of the vector variable. The result is a vector of size equal to the row size of the matrix and the vector size of the left-hand side of (4.1). For example:-

```
vector = matrix * vector1 + vector2;
```

Each component of the multiplication results is assigned to each component of the vector variable. The equivalent translated algorithm is as follows :-

```
for (i = 0, m-1)
```

$$\text{vector}[i] = \sum_{k=0}^{n-1}(\text{matrix}[i][k]*\text{vector1}[k]) + \text{vector2}[i];$$

4.1.1.4 Function expression

A function expression can be a built-in function or a user-defined function. The function type depends on its argument. If the argument is of scalar type, the function type is a scalar. If its argument is a vector, the function type is a vector with size must be the same as its left-hand side of (4.1). For example:-

```
vector = SIGMOID(matrix*vector1+vector2);
vector = SQR(scalar);
```

where *SIGMOID* and *SQR* are built-in functions. The equivalent translated algorithm is as follows:-

```
for (i = 0, m-1) {
```

$$\text{vector}[i] = \text{SIGMOID}\left(\sum_{k=0}^{n-1}(\text{matrix}[i][k]*\text{vector1}[k]) + \text{vector2}[i]\right);$$

```
  vector[i] = SQR(scalar);
}
```

4.1.1.5 Vector-Matrix assignment

A vector-matrix assignment is involved when an expression contains a matrix variable. There are two ways of representing this matrix variable as a vector type. These are :-

(1) to get all values of a matrix on a specified row, the following statement is used :-

```
vector = matrix@;
```

(2) to get all values of a matrix on a specified column, the following statement is used :-

```
vector = matrix#;
```

The specified row or column depends on the status of a reserved word ROW. Its use will be explained later.

4.1.1.6 Recursive Vector assignment

The operator used in the previous assignment is only '='. However, other operators defined under *assigntype* of (4.1) can be applied. They serve as a recursive assignment for a vector variable on the left-hand side of (4.1). For example, the update operator is written as follows :-

```
vector += vector_expression
```

It is equivalent to the following statement,

```
vector = vector + vector_expression
```

The equivalent translated algorithm is as follows:-

```
for (i = 0, m-1)
   vector[i] += vector_expression;
```

The operator '+=' and other recursive operators are also valid operators for C language.

4.1.2 Matrix assignment

In a matrix assignment, the left hand-side of (4.1) is a matrix variable and its right hand-side must contain the following expressions :-

4.1.2.1 Scalar expression

A scalar expression can be a number, a scalar variable or a mathematical expression that gives a scalar result. For example :-

```
matrix = scalar + 3;
```

This means that the calculated scalar expression is assigned to all matrix components. The equivalent translated statement is as follows :-

```
for (i = 0, m-1)
  for (j = 0,n-1)
     matrix[i][j] = scalar + 3;
```

4.1.2.2 Matrix expression

A matrix expression can be a matrix variable or mathematical expression that gives a matrix result. The row and column size of this matrix must be equal to the size of the matrix variable of the left hand-side of (4.1). For example :-

```
matrix = matrix1 + matrix2;
matrix = scalar1*matrix1 - scalar2;
```

Each component of a matrix expression is assigned to each component of a matrix variable. The equivalent translated statement is as follows :-

```
for (i = 0, m-1)
   for (j = 0, n-1) {
      matrix[i][j] = matrix1[i][j] + matrix2[i][j];
      matrix[i][j] = scalar1*matrix1[i][j] - scalar2;
   }
```

4.1.2.3 Function expression

A function expression can be a built-in function or a user-defined function. The function type depends on its argument. If the argument is of scalar type, the function type is a scalar. If its argument is a matrix, the function type is a matrix with size must be the same as the left-hand side of (4.1). For example :-

```
matrix = SQRT(matrix1);
```

where *SQRT* is a built-in function. The equivalent translated statement is as follows :-

```
for (i = 0, m-1)
   for (j = 0, n-1)
      matrix[i][j] = SQRT(matrix[i][j]);
```

4.1.2.4 Outer-Product of two vectors

The outer-product of two vector yields a matrix with its row size equal to the size of the first vector and its column size is equal to the size of the second vector. For example :-

```
matrix = vector1*vector2;
```

The equivalent translated statement is as follows :-

```
for (i = 0, m-1)
   for (j = 0, n-1)
      matrix[i][j] = vector1[i]*vector2[j];
```

4.1.2.5 Matrix-Vector assignment

A matrix-vector assignment follows a similar concept to that of vector-matrix assignment. However, an expression contains a vector expression and a matrix variable of the left-hand side of (4.1) acts as a vector type. There are two types that represent a vector for a matrix variable. These are :-

(1) to get all values of a matrix on a specified row is written as follows :-

```
matrix@ = vector;
```

(2) to get all values of a matrix on a specified column is written as follows :-

```
matrix# = vector;
```

The specified row or column depends on the status of a reserved word ROW. Its uses will be explained later.

4.1.2.6 Recursive Matrix assignment

A recursive matrix assignment follows the same concept as a recursive vector assignment. For example, the update operator is written as follows :-

```
matrix += matrix_expression
```

It is equivalent to the following statement,

```
matrix = matrix + matrix_expression
```

The equivalent translated algorithm is as follows:-

```
for (i = 0, m-1)
     for (i = 0, n-1)
       matrix[i][j] += matrix_expression;
```

4.1.2.7 Matrix transpose

The matrix transpose is written as matrix&.

4.1.3 Vector Dot Product

The dot product symbol is '.' to distinguish it from operator *. A dot-product is a multiplication of two vectors of the same size. It produces a scalar value. For example :-

```
scalar = vector 1.vector 2
```

The equivalent translated algorithm is as follows :-

$$\text{scalar} = \sum_{k=0}^{n-1} (\text{vector1}[k] * \text{vector2}[k])$$

4.2 THE DESIGN AND IMPLEMENTATION OF COMPILER MODULES

This section contains a detailed discussion of how the NN compiler (NEUCOMP) is designed and implemented. It begins with the design of its language (NEUCOMP language). The NEUCOMP language is the high-level programming language specifically designed for any NN simulation.

4.2.1 Defining Formal Grammar

Formal grammars are used to define the syntax of a language [Aho *et al.* (1986), Bennett (1990)]. This syntax is specified in a top-down fashion. Grammar rules or productions are used to define each component of the language from a simpler component, i.e. individual characters, into a sentence.

The general form of a production used in the definition of a programming language is :-

$$A ::= B_1B_2B_3 \ \ldots \ B_n$$

where entity A is made up of the string of simpler $B_1B_2B_3 \ldots B_n$ which could be a character or a string defined elsewhere. It means that A will be replaced with $B_1B_2B_3 \ldots B_n$ when we find A anywhere in the definition of the program. When a string cannot be expanded further, it is called a sentence. Hence, syntactically correct programs are sentences derived using the formal grammar defining the syntax of the programming language.

A typical example of a production of defining the 'if-statement' for the NEUCOMP language is as follows :-

```
if_statement ::= IF '(' logical_expression ')'
                                    statement_list ENDIF
```

where it means that *if_statement* consists of a reserved word *IF* followed by a symbol '(', followed by *logical_expression*,

followed by a symbol ')', followed by *statement_list* and a reserved word *ENDIF*.

The syntax of the NEUCOMP language begins with a single entity from which all syntactically correct programs are derived. It is written as follows :-

```
program  ::=  program_heading
              identifier_declarations
              main_declaration
              subprogram_declarations
```

where *program* is known as the 'sentence symbol' which contains a production of *program_heading*, *identifier_declarations*, *main_declaration* and *subprogram_declarations*.

There is more than one type of production that can be written. These are :-

(1) Alternative definition which is written as :-

$$A ::= B_1B_2B_3 \ldots B_n$$
$$A ::= C_1C_2C_3 \ldots C_m$$

which can be written as follows :-

$$A ::= B_1B_2B_3 \ldots B_n \mid C_1C_2C_3 \ldots C_m$$

(2) Self-referential or recursive definition which is written as follows :-

$$A ::= Ax \mid y$$

(3) The null symbol is written as follows :-

$$A ::= \varepsilon \mid B$$

where ε is a null symbol which means A is either null or made up of B.

Some examples of the NEUCOMP grammar rules are as follows:-

```
(1) subprogram_heading  ::= PROC  |  FUNC
(2) identifier_list ::= identifier |
                        identifier_list ',' identifier
(3) identifier_declarations  ::=  ε
                                |  declarations_list
```

where production (1) shows two alternatives in the definition of a symbol *subprogram_heading*, production (2) illustrates recursion on a symbol *identifier_list*, and production (3) makes use of the null symbol, ε which means there is no declaration or declarations defined in the form of symbol *declarations_list*.

The use of all types of productions when specifying the syntax of the programming languages is known as the 'Backus-Naur form' or more commonly BNF after its inventors [Aho *et al.* (1986), Bennett (1990)]. The complete BNF specification of the NEUCOMP language is shown in **Appendix A.**

4.2.2 Defining the Symbol Table

A symbol table plays an important role throughout the compilation process because it provides information about the names used in the source program. The usage of the symbol table is explained. The lexical analyser looks up for a name in the symbol table. If it does not exist then its name is inserted in the table. The syntax analyser looks up for the name and adds information such as the type of variable in the symbol table. The semantic analyser looks up the name in the symbol table that has the type used in accordance to its role in the program, i.e. the procedure name cannot be used as an expression.

Hence, the symbol table contains information about the names and their type that are used in the program. Their declaration are as follow :-

```
struct symb {
  char *name;
  int type;
  union {
    char *dim1, *dim2;
    int status;
    char *rval;
    int intORreal; /* integer or real */
    int scope; /* local or global */
  } val;
};
```

where *name* is a string that holds a variable name, *type* is used to specify a variable type which could be a file, string, function or procedure, scalar, vector or matrix, *union* is the C code used to allow a variable type to contain additional information such as a scalar can have field types *intORreal*, *rval* and *scope*, a vector or matrix can have field types *dim1*, *dim2*, *status*, *scope* and *intORreal*, and a variable of type file does not require any additional information. The field types *dim1* and *dim2* are used to hold the size of an array in the form of a variable name or number. The field type *status* is used by a vector/matrix variable to represent the current status of this variable, i.e. a matrix variable is used as a vector variable (**section 4.1**). The variable *rval* is used by a scalar variable to hold an integer or a real constant. The field type *intORreal* is used to show that the variable is of type integer or real. The field type *scope* signifies whether a variable is global or local.

NEUCOMP uses an open hash table for efficient look-ups of names in the symbol table. For example, in checking if the name of a variable used in an assignment statement has been declared. A linear search is not

efficient [Bennett (1990)]. In an open hash table, variable names with the same hash index are linked in the same list. Therefore, the data structure of an open hash table follows the following convention :-

```
#define HASHSIZE 999

struct symb {
  struct symb *next;
     ...
};

struct symb *symbtab[HASHSIZE];
```

where *HASHSIZE* is the size of the symbol table and *symbtab* is an array of size *HASHSIZE*, each pointing to *struct symb*.

Discussions on designing and implementing compiler modules, references to each field in the symbol table are based on the following definitions :-

```
#define MN           symb.name
#define MT           symb.type
#define MR       symb.val.rval
#define ML1      symb.val.dim1
#define ML2      symb.val.dim2
#define IR  symb.val.intORreal
#define SC     symb.val.scope
#define MLT   symb.val.status
```

4.2.3 Implementing the Lexical Analyser

The lexical analyser can be designed by hand. However, to achieve better program maintenance, using a compiler generator is recommended. A tool called Lex (**section 3.5.1**) is used to generate a lexical analyser.

4.2.3.1 Lex - A Tool for Building the Lexical Analyser

Lex is a lexical analyser generator available under the UNIX operating system [Lesk *et al.* (1975)]. It generates a stream of tokens useful for syntax analysis. A token is a group of individual characters of the source language.

Lex uses regular expression [Aho *et al.* (1986), Bennett (1990), Lemone (1992)] instead of the BNF grammars to describe the syntax of each token. Regular expressions make use of the following basic operations :-

concatenation	xy	x followed by y
Alternation	x | y	either x or y
Arbitrary repetition	x*	string x repeated zero or more times

For example, to represent a number the regular expression can be written as below :-

```
(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*
```

means a digit, or a digit followed by one or more digits.

The structure of the Lex language is as given below:-

```
Lex definitions section
%%
Lex rules section
%%
User-support routines written in C
```

4.2.3.2 Lex Definitions section

The strings that will be used in the rules section are defined in the definitions section. The definition is written as a name being defined on the left and its definition on the right. For example :-

```
comment         "//".*
lc_letter       [a - z]
digit           [0 - 9]
identifier      {lc_letter}({lc_letter}|{digit})*
```

where *comment* consists of a symbol // (must be quoted) followed by an arbitrary number of characters before reaching end of line, *lc_letter* consists of any lower-case alphabet, *digit* consists of any number between 0 and 9, and *identifier* consists of the first character which must be *lc_letter* then followed by none or more combinations of *lc_letter* or *digit*.

The definitions section may contain a variable definition written in C code enclosed within %{ and %}. This declared variable will be used in Lex rules (C code section) or user-support routine.

4.2.3.3 Lex Rules section

The rules section contains the name of a token on the left and the right contains some C program code within { and } to obey if that match succeeds. For a token name enclosed with { and } means its definition is available in the Lex definitions section. An example of the Lex rules section for the NEUCOMP language are as follows :-

```
{identifier}    { mkname(); return IDENTIFIER; }
NEURALNET       { return NEURALNET; }
```

where *identifier* is the name of the token defined in the Lex definitions section. It is written within { and }. If the input that represents this token is an identifier, Lex calls the function *mkname*() and returns an integer variable, *IDENTIFIER* to represent the token identifier. The name *NEURALNET* is a token to represent NEUCOMP's reserved word.

Every token is assigned with a different integer value when the 'Yacc program' (**section 4.2.4**) is executed with -d option. They are defined in a header file, *y.tab.h* which is produced by the Yacc program. This header file is included in the Lex definitions.

4.2.3.4 Lex User-support routines

The Lex User-support routines are required when we want to include a subroutine to support the Lex program. It is written in the C language. An example of the Lex user-support routines are *mkname*() and *mkval*(). These C-code routines are defined by the user. When the Lex analyser recognises that a token is an identifier a routine *mkname*() is called. It will look up this identifier in a symbol table using function *lookup*() which is declared as an external variable in the Lex definitions. If an identifier is not found in a symbol table, its name is inserted in a symbol table using the function *insert*(). It is also declared as external variable in the Lex definitions. Functions *lookup*() and *insert*() are defined in other C files. The reason these functions are put separate from the Lex program is that these functions are also required by other compiler modules such as during syntax and semantic analysis. The function *mkval*() converts a number previously defined as a string into a number.

A Lex program is executed using the Lex command code. The output is a file called *lex.yy.c* written as a C program. It contains an integer-valued function called *yylex*(). When this function is called by the syntax analyser, it returns the next token from the input language (NEUCOMP program). The file, *lex.yy.c* will be compiled with other compiler modules in order to produce an executable compiler program.

4.2.4 Implementing the Syntax Analyser

There are many methods of designing a syntax analyser or parser. Most of them are table driven [Aho *et al.* (1986), Bennett (1990)]. This is a tedious process when done by hand. However, such tables can be generated automatically by using a software tool. A popular tool called Yacc is used to generate a syntax analyser (**section 3.5.2**).

4.2.4.1 Yacc - A Tool for Building the Syntax Analyser

Yacc (stands for Yet Another Compiler-Compiler) is a parser generator which is widely available under UNIX.

A YACC program takes a specification of a NEUCOMP grammar and its semantic actions, and produces LALR(1) parsing tables and a shift-reduce parser [Aho *et al.* (1986), Bennett (1990), Lemone (1992)]. The source program (NEUCOMP program) is read as a stream of tokens provided by a separate compiler module called the lexical analyser. The output is the C-program and kept in a file, *y.tab.c*. It contains a routine called *yyparse*() that is responsible for carrying out the parsing.

The general form of the Yacc language is as follows:-

```
Yacc definitions section
%%
Yacc rules section
%%
User-support routines
```

4.2.4.2 Yacc Definitions section

The list of tokens for the NEUCOMP language is presented in the Yacc definitions section. These tokens are returned by the lexical analyser.

All tokens are represented by a name which is written as :-

```
%token <name>
```

For example, the tokens for the NEUCOMP language are specified below :-

```
%token NEURALNET
%token IDENTIFIER
```

where *NEURALNET* is a token for program heading and *IDENTIFIER* is a token for a variable.

When the Yacc program that contains the above specifications is executed with -d option, a header file called *y.tab.c* is produced. It contains the list of an internal representation represented by a small integer, starting from 257 (numbers up to 255 are used for the single ASCII characters, 256 is used as an error token). Examples of the above tokens are written in the header file as :-

```
#define NEURALNET  287
#define IDENTIFIER 290
```

These representation are useful for the lexical analyser to return internal representations of tokens when they are recognised. Therefore, before we can run the Lex program, we have to run the Yacc program with -d option.

In the Yacc definitions section, any ambiguity that may occur in an arithmetic expression for operators '+', '-', '*' and '/' can be overcome by specifying their precedences as below :-

```
%left      '+' '-'
%left      '*' '/'
```

where operators that have the same precedence appear in the same declaration. The arrangement of higher precedences is based on the given order.

The sentence symbol (**section 4.2.1**) for the NEUCOMP language can also be defined in the Yacc definitions section as :-

```
%start program
```

The Yacc definitions section may contain more than one variable definition written in C code for the support routine. It is written within %{ and }%.

4.2.4.3 Yacc Rules section

The Yacc rules section defines the grammar rules and semantic actions of the NEUCOMP language. A grammar rule or production in Yacc has the following form :-

```
non-terminal : right hand side { actions };
```

where *non-terminal* is a string that has its definition on the right. Terminal is another word for token. Typical rules for the NEUCOMP language are as follows :-

```
expression :  expression  PLUS  term
              { $$ = build_tree("+",T_OP,$1,$3); }
           |  expression  MINUS term
              { $$ = build_tree("-",T_OP,$1,$3); }
           |  term  { $$ = $1; }
           ;
```

The right-hand side may include terminals (tokens) such as *PLUS* and *MINUS*, and non-terminals which are *expression* and *term*. The non-terminal for *expression* is defined recursively and *term* is defined elsewhere. The vertical line '|' means alternative definition (**section 4.2.1**).

The *actions* contain C codes which are required to perform semantic actions. Each production is given semantic rules which describe how to compute the attribute value associated with each variable (terminal or non-terminal) in the production. This attribute value is passed up the parse tree to be used by other productions. The variable associated with the attribute value has attribute type to describe type of variable. This type can be an integer, a character or structure type. The attribute value has the form $n or $$ where $n is the attribute value associated with the *n*th. item in the right-hand side production. From the example given, the $1 is an attribute value associated with an expression or term and $3 is an attribute value associated with a term (first and second production). The $$ is the attribute being synthesised or a synthesised attribute. It associates with the non-terminal of the left-hand side of the production being derived from the attribute values on the right-hand side. For example, *$$ = build_tree("+",T_OP,$1,$3)* defined from the previous production means an expression tree is built from a function *build_tree* and the result is returned to an expression which is in the left-hand side of the production. If there is no action to be specified, the default is written as $$=$1.

The attribute type that is associated with the variables must be of the same type. Their types must be declared in the Yacc definitions section as follows :-

```
%type <expr_tree> expression
%type <expr_tree>       term
```

The type of token can also be defined if we want to do semantic action on this token. For example, if we want to store or retrieve any information about an identifier in the symbol table, the type definition for token IDENTIFIER is as follows :-

```
%token <symb> IDENTIFIER
```

The type for *expr_tree* and *symb* must be specified by the user as a C union of all the types that attributes may have in the Yacc program. Examples of attribute types used by NEUCOMP are written as :-

```
union
{
   struct symb *symb;
   struct treenode *expr_tree;
   char *chr;
}
```

The first and second fields are data structures to build up the symbol table and an expression tree respectively.

When run under Yacc with -d option the above definition will appear in *y.tab.h* as shown below :-

```
typedef union {
   struct symb *symb;
   struct treenode *expr_tree;
   char *chr;
} YYSTYPE
```

Any associated attribute is passed back using the global variable *yylval*. This has the type *YYSTYPE* so that the appropriate member of the union must be used.

4.2.4.4 Yacc User-support routines

C code is placed in the user-support routines to support the semantic actions defined in the rules section. An example of a C code written in Yacc rules taken from the previous production (**section 4.2.4.3**) is the function *build_tree("+",T_OP,$1,$3)*. It is defined as follows:-

```
struct treenode *build_tree(operand,
                    op_type,left_tree,right_tree)
char *operand;
int op_type;
struct treenode *left_tree,*right_tree;
{
     struct treenode *tree;
     tree = gettree();
     tree ->MN = operand;
     tree -> left = left_tree;
     tree -> right = right_tree;
     tree -> MT = op_type;
     return tree;
}
```

4.2.5 Implementing the Semantic Analyser

The semantic analyser for NEUCOMP performs semantic analysis during syntax analysis. When the syntax analyser recognises the NEUCOMP program construct (production) it calls a semantic routine which takes the construct and checks for semantic correctness.

The semantic analyser also translates the NEUCOMP program into an equivalent C program. This is done after the checking for semantic correctness.

4.2.5.1 Implementing Semantic checking

The NEUCOMP's semantic analyser implements four types of semantic checking which are :-

(1) checking that an identifier is declared once.
(2) checking that an identifier used has been declared.
(3) checking that a variable and value are compatible.
(4) checking the scope of a variable.

The semantic checking is performed on a variable. A variable written by the programmer is called an identifier. A variable written in capital letters is a reserved word, i.e. *CYCLE* and *NPATTERN* serve as a specific purpose (**section 4.4**).

The following section shows how the semantic checking for NEUCOMP are implemented. While undergoing semantic analysis, the identifier in the symbol table is changed accordingly.

Checking that an identifier is declared once

All identifiers used in a NEUCOMP program are declared in the declaration section. Each identifier is declared only once.

The production rule for an identifier-declarations is defined as follows :-

```
identifier_declarations : type identifier_list ';'
```

where the type supported by NEUCOMP is a simple type which is an integer, a real, string or file. The structure type like record or pointer is not implemented. The production rule for *identifier_list* is further defined as:-

```
identifier-list : identifier
                |identifier_list ',' identifier
```

where *identifier* can be a scalar or an array variable. The one-dimensional array variable is known as a vector variable and the two-dimensional array variable is known as a matrix variable.

The production rule for a scalar variable is defined as follows :-

```
identifier : IDENTIFIER | IDENTIFIER '=' NUMBER
```

where the alternative production allows a variable to be declared with an initial value.

The production rule for a vector variable is defined as follows :-

```
identifier : IDENTIFIER '[' numORid ']'
```

and the production rule for a matrix variable is defined as follows :-

```
identifier : IDENTIFIER '[' numORid ',' numORid ']'
```

where *numORid* can be an integer constant or an identifier of type integer. This identifier need not be declared because the compiler will declare that identifier with type scalar integer and insert it in the symbol table.

To implement declaration checking, the field *type* in the symbol table declared earlier (**section 4.2.2**) is used to determine the type of the identifier. A small integer is represented by a variable type as shown below :-

```
#define T_UNDEF    0  /* undefined type */
#define T_SCALAR   1  /* scalar type    */
#define T_VECTOR   2  /* vector type    */
#define T_MATRIX   3  /* matrix type    */
#define T_INT      4  /* integer type   */
#define T_REAL     5  /* real type      */
#define T_STRING   6  /* string type    */
#define T_FILE     7  /* file type      */
#define T_FUNC     8  /* function type  */
```

The name of an identifier would have been entered into the symbol table by the lexical analyser, with type *T-UNDEF* using the function *mkname*() (**section 4.2.3**). The function *lookup*() takes a name and yields a pointer to its symbol table entry, in which we can set the type field.

An algorithm to implement the declaration checking on a scalar is written as :-

```
identifier : IDENTIFIER
             { if ( $1->MT == T_UNDEF ) {
                 if ( type is T_STRING or T_FILE )
                    $1->MT = type;
                 else { /* scalar type */
                    $1->MT = T_SCALAR;
                    $1->IR = type;
                 }
               }
               else  error ("identifier declared
                             more than once");
             }
           | IDENTIFIER ' = ' NUMBER
             { if ( $1->MT == T_UNDEF ) {
                  if (type is T_INT or T_REAL) {
                    $1->MT = T_SCALAR;
                    $1->MR = $3;
                    $1->IR = type; /* int or real */
                  }
                  else error("not an integer or
                             a real type");
               }
                else  error ("identifier declared
                             more than once");
             }
```

where *T_UNDEF* means an identifier is not given any type. If the type has been given, the first if-statement will not allow the same identifier to be declared more than once. An identifier of type string or file, its field type in the symbol table, *$1->MT* is assigned to T_STRING or T_FILE. An identifier of type integer or real, its field type, *$1->MT* is assigned to T_SCALAR and its second field type, *$1->IR* is assigned to type T_INT or T_REAL. An identifier can be given a value and this is reflected in the symbol table as *$1->MR*. This is shown in the above alternative production.

An algorithm to implement declaration checking on a vector variable and its array size is written as :-

```
identifier : IDENTIFIER '[' numORid ']'
      {     if ( $1->MT == T_UNDEF ) {
               $1->MT = T_VECTOR;
               $1->IR = type;/*integer or real*/
               if ( $3 is an integer constant )
                     $1->ML1 = $3;
               else
               if ( $3 is an identifier
                     and $3->MT == T_UNDEF ) {
                  $3->MT = T_SCALAR;
                  $3->IR = T_INT;
               }
               else
                  error ("integer is expected");
            }
            else  error ("identifier declared
                           more than once");
         }
```

Similar to scalar type declaration, the field type, *$1->MT* is assigned to T_VECTOR and *$1->IR* is assigned to T_INT or T_REAL. Any information regarding *numORid* is also kept in the symbol table using the field, *$1->ML1* depending on how *numORid* is defined. It represents an array size of type integer constant or an identifier. If it is an identifier, no declaration is necessary because the compiler will change its type in the symbol table to integer scalar. Its size will be determined at run-time. This characterises a dynamic-like structure.

An algorithm to implement declaration checking on a matrix variable is similar to vector declaration provided that an array variable declaration has two *numORids* representing two-dimensional sizes. The field type in the symbol table, *$1->ML2* is used to store the second dimensional size.

Checking that an identifier used has been declared

All identifiers used in the NEUCOMP program must be declared before they can be used in the program's body. In order to know that an identifier has been declared, the field type in the symbol table for that identifier must not be T_UNDEF. This value should have been changed when that identifier was declared.

There are three types of functions that return TRUE or FALSE. These are used to check an identifier type, namely, *exist_id*, *exist_file* and *exist_func*. The function *exist_id* returns TRUE when the type of the identifier is either integer or real, otherwise it will return FALSE. Similarly for the other functions, provided that *exist_file* is used to check an identifier of type file and *exist_func* is used to check an identifier of type function or procedure. An error message showing that an identifier is not declared will be displayed.

The following shows an algorithm for the semantic checking on a production such as 'openfile-statement' and 'for-statement' :-

```
openfile_statement : OPENREAD '(' IDENTIFIER ',' ...
                   { if ( exist_file($3) == TRUE )
                     {  /* do other routine */  }
                     else error("undeclared file name");
                   }

for_statement : FOR IDENTIFIER '=' ...
                { if (exist_id($2) == TRUE)
                    {  /* do other routine */  }
                  else error("undeclared identifier");
                }
```

Checking that a variable and value are compatible

A variable refers to either a reserved word or declared variable (which is an identifier). The value for the

NEUCOMP language refers to an integer, a real or string constant.

The type rules involved are :-

(1) if a variable is of type file, it is only used in a statement such as open file, read from file, write into file and close file. The string constant is assigned to this variable to indicate the file name.
(2) if a variable is of type string, it accepts a string constant written within " " or assigned the string constant using a read statement.
(3) if a variable is of type procedure/function, it is used in the name of subprogram-heading, calling procedure or function.
(4) if a variable is of type integer or real, it is used in the assignment statement and other statements such as 'print-statement', 'read-statement', etc. This variable is declared as a scalar, vector or matrix variable.

The algorithms to implement semantic checking on (1) to (3) which use the functions such as *exist_file* and *exist_func* are considered as straight forward. However, the rule (4) is not really straight forward.

This section focuses on how semantic checking is implemented in the assignment statement. The production rule for an assignment statement is defined as follows :-

assignment_statement : variable assigntype expression

where *assigntype* is the symbol '=', '+=', '*=' or '/=', and *expression* can be a single item such as a variable, number or function, or consists of the following form :-

operand1 operator operand2

In order to do the semantic checking, the expression itself must have a type. For a single item such as a

variable, the expression type is based on the variable type. For a function, the expression type follows the type of function argument. If its argument is a scalar then the expression type is a scalar. If its argument is a vector/matrix then the expression type is a vector/matrix including its size.

If the expression is not a single item then the following shows an algorithm to assign a type to it :-

(1) assign either real or integer type :-

```
if (operand1->IR == T_INT && operand2->IR == T_INT)
     expression->IR = T_INT;
else
     expression->IR = T_REAL;
```

(2) assign the type scalar, vector or matrix :-

```
if (operand1->MT == T_SCALAR &&
          operand2->MT == T_SCALAR/T_NUM ) {
     expression->MT = T_SCALAR;
     /* do translate expression */
}
else
if (operand1->MT == T_VECTOR/T_MATRIX &&
          operand2->MT == T_VECTOR/T_MATRIX) {
  if (matrix/vector size are equal) {
     expression->MT = T_VECTOR/T_MATRIX;
     expression->ML1 = operand1->ML1;
     expression->ML2 = operand1->ML2; /*for matrix*/
     /* do translate expression */
  }
  else
     error("incompatible size");
}
else
if (operand1->MT == T_MATRIX/T_VECTOR
          AND operand2->MT == T_SCALAR ) {
```

```
        expression->MT = T_VECTOR/T_MATRIX;
        expression->ML1 = operand1->ML1;
        expression->ML2 = operand1->ML2; /*for matrix*/
        /* do translate expression */
    }
    else
    if (operand1->MT == T_SCALAR
        AND operand2->MT == T_MATRIX/T_VECTOR ) {
        expression->MT = T_MATRIX/T_VECTOR;
        expression->ML1 = operand2->ML1;
        expression->ML2 = operand2->ML2; /*for matrix*/
        /* do translate expression */
    }
```

The routine to perform *do translate expression* will be explained in **section 4.2.5.2.**

After the expression is given a type, the next step is to perform semantic checking on an assignment. The algorithm is as follows :-

(1) If the variable is of type integer then the expression must be an integer. For a variable of type real, the expression can be real or integer.

The following algorithm shows how to perform this rule :-

```
assignment_statement : variable assigntype expression
  {
   if ( ($1->IR == T_INT && $3->IR == T_INT)
    ||( $1->IR == T_REAL && ( $3->IR == T_INT
                    || $3->IR == T_REAL ) ) )
            { goto 2 }
    else
       error("variable and expression
                  have different types");
  }
```

(2) If the variable is a scalar then the expression must be a scalar. For a variable of type vector, the expression must be a scalar or vector with size equal to the size of vector variable. For a variable of type matrix, the expression must be a scalar or matrix with size equal to the size of matrix variable.

The algorithm to perform this rule is as follows :-

```
if ($1->MT == T_SCALAR/T_VECTOR/T_MATRIX &&
        $3->IR == T_SCALAR)
{ /* do translate assignment */ }
else
if ($1->IR == T_VECTOR && $3->IR == T_VECTOR) {
   if ( $1->ML1 == $3->ML1) /* check sizes */
        { /* do translate assignment */ }
   else
        error("incompatible vector size");
}
else
if ($1->IR == T_MATRIX && $3->IR == T_MATRIX) {
   if ( $1->ML1 == $3->ML1 && $1->ML2 == $3->ML2)
        { /* do translate assignment */ }
   else
        error("incompatible matrix size");
}
```

The routine to perform *do translate assignment* will also be explained.

Checking the scope of a variable

All variables used in a NEUCOMP program can be declared either as local or global. The variables which are reserved words, are declared as global by the compiler. A variable declared by the programmer in the declaration section, above the main program is called a global

variable. A variable declared in the declaration section within main/subprogram (called block) is a local variable. Its use is within the block in which it is declared.

The semantic checking for the scope of a variable is based on the following simple approaches which are :-

(1) NEUCOMP does not allow a global variable to be declared as local. An error message will be displayed specifying the same variable is declared more than once. The algorithm follows the semantic checking for a variable declared once as described previously.
(2) In the symbol table, the field scope is used to keep an integer value type, namely GLOBAL and LOCAL. All global variables will have their field scope in the symbol table are set to GLOBAL. The field scope for the local variables are set to LOCAL. However, at the end of the block where they are declared, they are removed from the symbol table. This is done by searching for all the names with field LOCAL and deleting them. In this way, a variable declared in two different subprograms, represents two different entities which are not related.

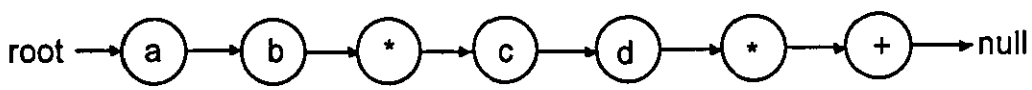
4.2.5.2 Translating into the Target program

A NEUCOMP program is translated into an equivalent C program after semantic checking. This section focuses on the process of translation that involves a vector/matrix variable such as in an assignment statement. Statements such as 'for-statement', 'if-statement', 'repeat-statement' and 'while-statement' involve only a scalar variable. Translation on a scalar variable is a straight forward process, however, a vector/matrix variable requires more effort.

There are three stages involved in the process of translating an assignment statement which are :-

(1) After the parsing of the expression, an expression tree is built. An expression tree is a binary tree data structure [Wirth (1976)] in which its root is an operator and its left and right children are an operand.

(2) An expression tree is then converted into a postfix expression. A postfix expression is an expression that consists of two operands followed by an operator. Evaluating a postfix expression is easy to program because an operator and operand have been arranged based on their precedences. An example of a postfix expression for the expression *a*b+c*d*, is written as,

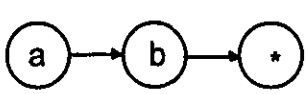


(3) The translation is carried out involving two stages. First, the postfix expression is translated and then an assignment statement which involves *variable*, *assigntype* and the translation code of the expression is translated. These are performed under the if condition of the semantic checking on the expression and assignment statement as discussed previously.

Translating the postfix expression

The process of translating the postfix expression is based on the following operations :-

(1) find the first 3 nodes from the root of the postfix expression that contains two operands and an operator. The example of the above expression is :-



The semantic checking was implemented on both operands and has been explained earlier.

(2) translate the expression based on the following conditions and find its type. Both are under the routine *do translate expression*.

(a) If the first and second operands are scalars then translate directly into an infix expression, i.e. *a*b*. A bracket is required if an operator is '+' or '-' in order to maintain precedences. The type of translated code is T_SCALAR.

(b) If the first operand is a scalar and the second is a vector then translate into an infix expression including "[*I*]" on the vector variable, i.e. *a*b*[*I*]. Similarly, if the second operand is a matrix, it is translated as *a*b*[*I*][*J*]. The use of *I* and *J* (both are reserved words) will be discussed when describing *translating an assignment*. The type of translated code is T_VECTOR/T_MATRIX and the size of an array is that of the vector/matrix size.

(c) If the first operand is a vector and the second is a scalar then the translated code and its type are similar to (b), i.e. *a*[*I*]*b*. Similarly, if the operand is a matrix then it is translated as *a*[*I*][*J*]*b*.

(d) If the first and second operands are vectors then there are three types of translation, which are :-

(d1) If the operator is '.', then this is a dot vector product which gives a scalar type result. The library routine for the translated code called *Dot_product* is used. It contains the argument of these two vectors and their array size, i.e. *Dot_product*(*a*,*b*,*n*) where *n* is the array size of vector *a*.

(d2) If the variable on the left-hand side of an assignment is a matrix, then this is an

outer product of two vectors which gives a matrix type result. The translated code is $a[I]*b[J]$ where the size of the vector a is the row size of the matrix and the size of vector b is the size of its column.

(d3) If the variable on the left-hand side of an assignment is a vector, then the translated code is $a[I]*b[I]$ and its type is a vector and the array size is the size of the vector a.

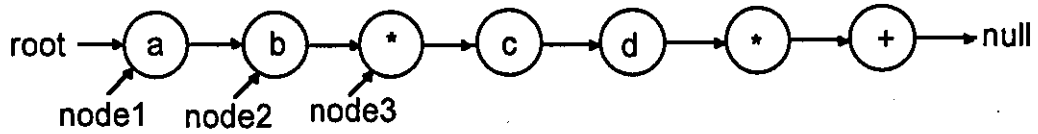
(e) If the first operand is a matrix and the second operand is a vector and the operator is '*' then this is the matrix-vector multiplication which gives a vector type result. The library routine for the translated code called *Mul_mat_vec* is used. It contains the arguments of these two operands, the size of the vector variable and I, i.e. *Mul_mat_vec*(a,b,n,I) where n is the array size of the vector b.

(f) If the first operand is a matrix and the second operand is a matrix then the translated code is i.e. $a[I][J]*b[I][J]$ and its type is also a matrix.

(3) After translating the two operands and operator, this translated code is then 'push' onto the stack. Stack is the First-in-First-out data structure [Wirth (1976)]. It contains a pointer that points to the translated code, the type of this code, and the size of array if the type is a vector/matrix. The type of expression in the stack is determined at stage (2).

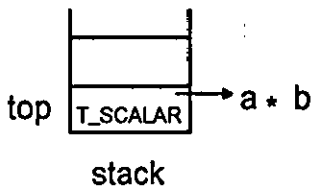
The following is an example of how the given expression is translated :-

(a) From the operation (1), the three pointers that point to two operands and operator are :-

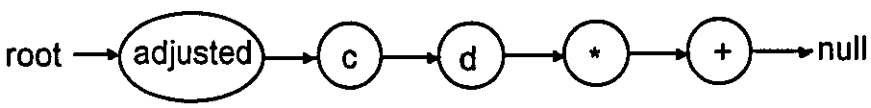


(b) From the operation (2), the translated code is a*b if both are scalars and its type is T_SCALAR.

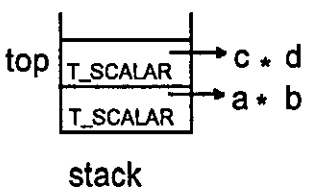
(c) From the operation (3), the translated code and its type are pushed onto the stack as shown below :-



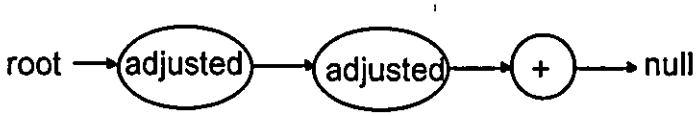
where *top* is an index that shows the content of the top of the stack. The postfix expression is adjusted accordingly as :-



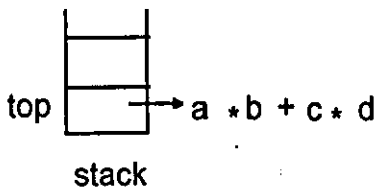
All nodes contain an integer variable called INSTACK. For the adjusted node, its INSTACK is set to TRUE which means that the operand is the translated code from the stack. The adjusted node is also an operand. By repeating the same step (c), the stack now contains the following translated codes :-



and the adjusted postfix expression is now



In this case both adjusted nodes have INSTACK value TRUE. The operands are taken or 'popped' out twice from the stack. They are then merged into one translated code. Its type is obtained as in rule (3). This translated code is then pushed back into the stack as shown below :-



Translating an assignment statement

The routine to translate an assignment statement is under the name *do translate assignment* that was discussed in the semantic checking of an assignment statement. An example of an assignment statement to carry out translation is as follows :-

```
x = a*b + c*d;
```

where the expression was translated previously. Its translated code is in the stack. The variable *x* is then translated according to the following rules :-

(1) if the variable is a scalar and the type in the stack is a scalar, then the routine *do translate assignment* is given by the example below :-

```
x = a*b + c*d;
```

or if *a* and *b* are vectors of size *n* then

```
x = Dot_product(a,b,n)+c*d;
```

(2) if the variable is a vector and the type in the stack is a scalar, then the routine *do translate assignment* is given by the example below :-

```
for (I = 0; I < n; I++)
        x[I] = a*b + c*d;
```

(3) if the variable is a matrix and the type in the stack is a scalar, then the routine *do translate assignment* is given by the example below :-

```
for (I = 0; I < m; I++)
    for (J = 0; J < n; J++)
        x[I][J] = a*b + c*d;
```

(4) if the variable is a vector and the type in the stack is a vector, then the routine *do translate assignment* is given by the example below :-

```
for (I = 0; I < n; I++)
        x[I] = a[I]*b[I] + c*d;
```

where a and b are vectors of size n or if a is a vector of size n and b is a matrix of size $m*n$ then we have :-

```
for (I = 0; I < m; I++)
    x[I] = Mul_mat_vec(a,b,n,I) + c*d;
```

(5) if the variable is a matrix and the type in the stack is a matrix, then the routine *do translate assignment* is given by the example below :-

```
for (I = 0; I < m; I++)
    for (J = 0; J < n; J++)
        x[I][J] = a[I]*b[J] + c*d;
```

where a and b are vector variables of size m and n.

The variables c and d are considered scalar variables.

4.2.6 Dynamic Allocation Memory

The size of the NEUCOMP's array declaration can be defined as static or dynamic. A static size means a positive integer constant is allocated to an array variable. Dynamic size means a scalar variable become the size of an array variable where it will be later allocated at run-time.

The size of the memory that will be allocated at run-time is called Dynamic Allocation Memory (DAM). Its advantage is that the size of the NN structures can be changed at run-time for the particular NN simulation programs. The disadvantage of using static allocation is that when changing the size of an array, re-compilation and re-execution of the NN program is necessary.

The following example shows how to declare DAM :-

```
REAL  layer1[n1], layer2[n2], weight[n2,n1];
```

The scalar variables *n1* and *n2* need not be declared. NEUCOMP will translate the above declaration into the following C-code :-

```
int n1, n2;
float **weight1,*layer1,*layer2;
extern float * setup_vector();
extern float **setup_matrix();

main() {
   printf("Type size of n1 = ");
   scanf("%d",&n1);
   printf("Type size of n2 = ");
   scanf("%d",&n2);
   layer1 = setup_vector(n1);
   layer2 = setup_vector(n2);
   weight1 = setup_matrix(n2,n1);
        . . .
}
```

The procedure *setup_vector*() and *setup_matrix*() are defined in the NEUCOMP library routine as follows :-

```
float *setup_vector(size)
int size;
{ int i;
  float *new;
  new = (float *) malloc(size*(sizeof(float)));
  return (new);
}

float **setup_matrix(row,col)
int row,col;
{ int i;
  float **new;
  new=(float **)malloc(row*(sizeof(float*)));
  new[0]=(float*)malloc(row*col*(sizeof(float)));
  for (i=1;i< row; i++)
    new[i] = new[0] + (col*i);
  return (new);
}
```

4.2.7 Implementing the Loop Optimiser

Improving the target program so that it can run faster or take less memory space or both is a difficult task [Aho *et al.* (1986), Bennett (1990)]. The improvement is done by program transformations that are traditionally called 'optimisation', although this term is a misnomer because there is a rarely any guarantee that the resulting code is the best possible.

Actually, optimising the target program generated by NEUCOMP is not really necessary because the target code is written in C and the C compiler has its own code optimiser. The only improvement that can be done on this target program is on the 'for loop' generated by NEUCOMP that involves vector and matrix variables as discussed

earlier. There may be many repeated loops containing many statements involving vector/matrix variables. For example, in the backpropagation algorithm (**section 2.2.1**), to update the weight and bias between the input layer and the first hidden layer we have the following matrix/vector operations :-

```
weight1 = alpha*dweight1 + beta*cweight;
cweight1 = weight1 - oweight1;
bias2 = alpha*ddelta2 + beta*cbias2;
cbias2 = bias2 - obias2;
```

where the first two equations involve matrix variables and the rest involve vector variables. The generated target code is as follows :-

```
for (I = 0; I < n2; I++)
for (J = 0; J < n1; J++)
  weight1[I][J]= alpha*dweight1[I][J]+beta*cweight[I][J];
for (I = 0; I < n2; I++)
for (J = 0; J < n1; J++)
  cweight1[I][J] = weight1[I][J] - oweight1[I][J];
for (I = 0; I < n2; I++)
  bias2[I] = alpha*ddelta2[I] + beta*cbias2[I];
for (I = 0; I < n2; I++)
  cbias2[I] = bias2[I] - obias2[I];
```

The above program code can then be improved by removing the repeated loops as shown below :-

```
for (I = 0; I < n2; I++) {
  for (J = 0; J < n1; J++) {
   weight1[I][J]=alpha*dweight1[I][J]+beta*cweight[I][J];
   cweight1[I][J]=weight1[I][J] - oweight1[I][J];
  }
  bias2[I] = alpha*ddelta2[I] + beta*cbias2[I];
  cbias2[I] = bias2[I] - obias2[I];
}
```

The loop optimiser for NEUCOMP was implemented by searching the statements that involve the loop *for* (*I=0; ...*) and the same array size within the program block. They are then combined into one loop. Within the loop *for* (*I=0; ...*), there can be many repeated loops on *for* (*J=0 ...*). This can also be done using the same technique.

4.3 COMPILING THE C PROGRAM

To compile the C program from the compiler modules and the C program from the translated code is now explained in this section.

4.3.1 Obtaining object code for Compiler modules

The compiler program to develop NEUCOMP contains seven files which are *main.c*, *lex.yy.c*, *y.tab.c*, *useroutine.c*, *translexpr.c*, *translassign.c* and *looptimiser.c*. The file *main.c* is the main routine of the compiler program. It contains the call function, *yyparse*() which is a routine defined in *y.tab.c* and *optimiser*() which is a routine in file *looptimiser.c*. It also contains the functions *lookup*() and *insert*() which serve as look up names and the insertion of new names in the symbol table respectively. The file *useroutine.c* contains a user-support routine for the Yacc program. The files *translexpr.c* and *translassign.c* are used for translating the expressions and assignment statements respectively.

The command to compile the compiler program is as follows :-

```
cc main.c y.tab.c lex.yy.c useroutine.c translexpr.c
translassign.c optimiser.c -o NEUCOMP -lm
```

where -o is used to include the name of the executable file called NEUCOMP and -lm is used to allow the C compiler to access the C library routine that contains

standard mathematical formula. However, the UNIX executable file, 'make' can be used to compile just the corrected source file. Those files that involve no correction and have been previously compiled will not be compiled again. To achieve the above, type *make* after the UNIX prompt.

Once the compilation is completed with no error, the executable file called NEUCOMP can then be used to compile any NEUCOMP program using the following command:-

```
NEUCOMP filename
```

where *filename* is the name of a file that contains a NEUCOMP program.

4.3.2 Compiling the Translated code

When successfully compiled by NEUCOMP, the file *neu.c* is generated. This file contains the translated version of a NEUCOMP program which is written in C.

However we have to compile the *neu.c* program together with the NEUCOMP's library routine defined in file *libfunc.c*, in order to start a NN simulation. This library function contains the functions to evaluate an assignment statement such as *Dot_product* and *Mul_mat_vec* (**section 4.2.5.2**). The command to compile these files is as follows :-

```
cc neu.c libfunc.c -o NET -lm
```

where *NET* is an executable file which can be used to execute the simulation program written in the NEUCOMP language. The 'make' facility is also used to compile the above command.

In order to simulate other NN models we can have other NEUCOMP programs for them. We can then compile and execute this program using NEUCOMP as described earlier.

4.4 SOME NEURAL NETWORK SIMULATION PROGRAMS

This section contains the discussion of how to write the NEUCOMP simulation program on the chosen NN models - the backpropagation, Kohonen, ART1 and Counterpropagation networks. The size of the problem can be changed since the simulation program is based on dynamic-like structure.

Each program has the heading name which is written as :-

```
NEURALNET name
```

where *name* is the name of valid identifier which need not be declared but its name cannot be used elsewhere.

4.4.1 The backpropagation simulation program

The backpropagation is the multilayer network (**section 2.2.1**). A three layer fully connected feedforward network is used to develop the backpropagation simulation. Its network structure is declared as global variables shown below :-

```
REAL  layer1[n1], layer2[n2], layer3[n3],
      weight1[n2,n1], weight2[n3,n2],
      bias2[n2], bias3[n3], pattern[n4,n1],
      target[n4,n3];
```

where *layer1* is a vector for the input layer, *layer2* is a vector for the hidden layer, *layer3* is a vector for the output layer, *weight1* is a matrix for the connection between the input layer and hidden layer, *weight2* is a matrix variable for the connection between the hidden layer and output layer, *bias2* is a threshold vector for *layer2*, *bias3* is a threshold vector for *layer3*, *pattern* and *target* are used to hold the set of input and desired patterns respectively. The row size of *pattern* and *target*

represented by *n4*, is used to hold the number of patterns in the training set. Their columns hold the sizes of input and output layers respectively. The values of *n1*, *n2*, *n3* and *n4* will be assigned at run-time.

To train the network, it can be done by using the following training loop :-

```
TRAINING
   ...
END;
```

where the statement *TRAINING* contains a reserved word variable of type integer called *CYCLE* which is initially set to 100. It means the number of iterations is 100. However, this value can be changed by assigning a new value to *CYCLE*. The training algorithm is within the loop.

To assign an input layer with a pattern, the following pattern loop is used :-

```
EPOCH
  layer1 = pattern@;
    ...
END;
```

where the statement *EPOCH ... END* contains the loop starting from zero to the pattern size minus one set by an integer variable called *NPATTERN*. Each iteration is assigned to the reserved word variable called ROW. The variable *pattern* with symbol '@' means its *row*th. represented by ROW is assigned to *layer1* **(section 4.1.1.5)**. The loop may contain other statements. The *NPATTERN* is a reserved word variable which is initially set to one. It means only one pattern is involved in the training operation per cycle. However, this value can be changed by assigned a new value to *NPATTERN*.

The weights are updated during training. For example, *weight1* is updated as follows :-

```
oweight1 = weight1;
weight1 += GRBH(alpha,range,aweight1)+beta*cweight1;
cweight1 = weight1-oweight1;
```

where the function *GRBH* (**section 2.2.1**) is a built-in function used to determine the value of *alpha* when *aweigthtl* is in *range*.

Training may be terminated when a global error is less than the limit as defined below :-

```
IF (enormsqr LE limit) BREAK ENDIF;
```

where *enormsqr* is the global error and *limit* can be set to any value, i.e. 0.01 and *enormsqr* is written as :-

```
error = target@ - layer3;
enormsqr += 0.5*(error.error);
```

where @ on *target* means that all entries on a specific row are involved in the above calculation. The value of this row depends on the predefined scalar variable, *ROW*. It is a positive integer used as an index to the matrix row. The operator '.' is known as dot vector product.

The complete program is shown in **Appendix C**.

4.4.2 The Kohonen network simulation program

The Kohonen network is a two layer network that can organise a topological map from a random starting point (**section 2.2.3**). The network combines an input layer with a competitive layer by unsupervised learning.

The network structure declared as global variables is shown below :-

```
REAL layer1[n1], layer2[n2],
     weight[n2,n1], pattern[n4,n1];
```

where *layer1* is a vector for the input layer, *layer2* is a competitive layer to be assigned the distance calculated between the input vector and their connection strengths, *weight* is a matrix for the connection strength between the input layer and the competitive layer, and *pattern* is similar as in **section 4.4.1.** The values of *n1*, *n2* and *n4* will be assigned at run-time.

The training loop statement, *TRAINING ... END* and the pattern loop statement, *EPOCH ... END* are also used in the Kohonen training algorithm.

The winner node at the competitive layer is defined as follows :-

```
layer2< = DISTANCE(layer1,weight);
```

where the symbol '<' means get an index of *layer2* when its value is the minimum calculated as the distance between all nodes in *layer1* and their weights connected to each node in the competitive layer. The word *DISTANCE* is NEUCOMP's built-in function. This index is assigned to *ROW* where *layer2*[*ROW*] is the minimum. *ROW* itself is the winner node in the competitive layer.

Although the competitive layer is declared as one-dimensional, it can also be used for two-dimensions. Since *ROW* is the winner node in one-dimension, then the winner node in two dimensions can be calculated as follows :-

```
REPEAT
  c = ROW - r*grid;
  IF (c GE grid) r = r + 1 ENDIF
UNTIL ( c LT grid);
```

where *r* and *c* is the winner node in two-dimensional layer which represents the map, and *grid* is the square root of the competitive size.

Therefore updating the weights in the neighbourhood can be defined as :-

```
r1 = r-neighb;
r2 = r+neighb;
IF ( r1 LT 0 ) r1 = 0 ENDIF;
IF ( r2 GE grid) r2 = grid - 1 ENDIF;
c1 = c-neighb;
c2 = c+neighb;
IF ( c1 LT 0 ) c1 = 0 ENDIF;
IF ( c2 GE grid) c2 = grid - 1 ENDIF;
FOR (i = r1,r2 + 1)
   FOR (j = c1,c2 + 1)
      ROW = i*grid+j;
      weight@ += lrate*(layer1-weight@ )
   ENDFOR
ENDFOR;
```

where *neighb* is the size of the neighbourhood. The points *r1,c1* and *r2,c2* are in the two-dimensions of map.

The complete program is shown in **Appendix D.**

4.4.3 The ART1 network simulation program

The ART1 network is used to classify the binary pattern (**section 2.2.4**). It is a two layer network with the first layer an input layer and the second layer a competitive layer. There are two network connections called feedforward and feedback weights. The vigilance threshold can be set between 0 and 1.

The network structure declared as global variables is shown below :-

```
REAL layer1[n1], layer2[n2], weightf[n1,n2],
     weightb[n1,n2], pattern[n4,n1];
```

where *layer1* is a vector for the input as well as the comparison layer, *layer2* is a vector for the output as well as the recognition layer, *weightf* is a feedforward weight, *weightb* is a feedback weight and *pattern* is similar as in

section 4.4.1. The values of *n1*, *n2* and *n4* will be assigned at run-time.

Only the statement *EPOCH ... END* is used in the ART1 network algorithm since the training loop statement as mentioned in the previous simulations are not used. This is because training in ART1 involves two stages which are the recognition and comparison stages.

Calculating the best exemplar is done at the recognition stage and is as shown below :-

```
layer2> = weightf*layer1;
```

where the symbol '>' means get an index of *layer2* when its value is the maximum then assign it to variable *ROW* to be used by variable with sign @.

The comparison stage is done as follows :-

```
IF (vigilance GE .99)
 weightf@ = weightb@*layer1/(0.5+weightb@.layer1);
 weightb@ = weightb@*layer1
ELSE
  get next best exemplar when vigilance LT 0.99;
```

where *vigilance* is used to distinguish that the new input pattern is different from the existing pattern. For the next best exemplar, we have to set *weightf@ = 0* and then apply '>' again as before so that the new *ROW* is identified.

The complete program is shown in **Appendix E.**

4.4.4 The Counterpropagation network simulation program

The Counterpropagation network consists of a three layer feedforward network (**section 2.2.5**). The first layer is the input layer, the second layer is the competitive layer and the third layer is the output layer.

The network structure as declared global variables is shown below :-

```
REAL    weight1[n2,n1], weight2[n3,n2],
        layer1[n1], layer2[n2], layer3[n3],
        pattern[n4,n1], target[n4,n3];
```

where *layer1* is a vector variable for the input layer, *layer2* is a vector variable for the competitive layer, *layer3* is a vector variable for the output layer, *weight1* is a matrix variable for the connection weight between the input and competitive layers, and *weight2* is a matrix variable for the connection weight between the competitive and output layers, *pattern* and *target* are similar as in **section 4.4.1**. The values of *n1*, *n2*, *n3* and *n4* will be assigned at run-time.

Training in the Counterpropagation network involves two steps. The first step is to train the competitive layer which is based on the Kohonen method. The second step is to train the output layer which is based on the Grossberg method. To find the winning node and update the first weight are similar to the Kohonen network algorithm (**section 4.4.2**) provided that the grid is a one-dimensional array.

After the winner node is identified, the next step is to update the connection weight between the winner node in the competitive layer and the output layer. Since it is a supervised learning, the output vector is then compared to the target vector.

The *weight2* is updated as follows :-

```
layer3 = weight2#;
error = tlayer-layer3;
weight2# += brate*error;
```

where *brate* is the learning rate at the output layer and *tlayer* is the target vector.

The complete program is shown in **Appendix F**. Examples to solve problems are explained in **chapter 6**.

4.5 IMPLEMENTING GRAPHICAL FEATURES

The NEUCOMP program combines with the graphical package, i.e. Mathematica software package [Wolfram (1991)] to portray some graphical features. Through graphs, the user can view items such as the structure of the NN being considered and analyse simulation results during training.

The use of an existing graphical package is recommended as our own design will make NEUCOMP more complicated. Choosing Mathematica does not mean that Mathematica provides the type of graph that is needed. A program had to be written in the Mathematica language in order to create a graph that is required. However, programming using Mathematica is not difficult. Mathematica provides many graphical functions, i.e. drawing circles, lines, etc. as well as numerical and symbolic computation. These can be combined to provide the appropriate graph. It is more convenient this way as it allows user to write any graph to suit his application if the available graph library is insufficient.

The NEUCOMP program and Mathematica program for graphical features are two separate programs. The NEUCOMP program cannot communicate directly with Mathematica. This is the limitation that NEUCOMP had to face. The original plan was for the NEUCOMP program to call the Mathematica program. By doing this, we can analyse items such as the way the weights and activation nodes adapt themselves during training. This is because NEUCOMP was implemented on the SEQUENT Balance machine at PARC. The terminal used is an ASCII terminal. So there is no graphical package for the Balance machine. An alternative approach is to use the Mathematica software (version 2.0) that is available on the PC. It is a text-based interface. The data from the NEUCOMP program can be transferred to Mathematica by a file.

The type of graphical features that have been implemented so far are :-

(1) Displaying the NN structure, i.e. single or multilayer network.
(2) Plotting the XY-graph.
(3) Plotting (x,y) for data clustering and valid cities for the travelling salesman problem.
(4) The three-dimensional plotting for the spiral problem.

The travelling salesman and spiral problems will be explained in **chapter 6.**

4.5.1 Implementing the Neural Network structure

The program to display a NN structure was implemented through the function called *network* with two arguments, *layer* and *link*. The Mathematica command to define this function is :-

```
network[layer_,link_] := ...
```

where '_' is the required symbol applied to a function argument, *layer* stands for a variable which accepts a set containing the number of nodes in each layer, and *link* is a variable which accepts a set containing connections between the layers and the type of connection. For example :-

```
network[{2,3,1},{{1,2,0},{2,3,0}}]
```

means call the function with the first set {2,3,1} assigned to *layer* and the second set {{1,2,0},{2,3,0}} assigned to *link*. The set {2,3,1} means that *layer* contains a three layer network. The first layer (input node) has 2 nodes, the second layer (hidden layer) has 3 nodes and the third layer (output layer) has a single node. In the set {{1,2,0},{2,3,0}}, the first element, {1,2,0} means the first layer is connected to the second layer with '0'

representing feedforward connection. The feedforward and feedback connection are represented by '1'. Similarly, {2,3,0} means the feedforward connection from the second layer to the third layer. We can add further connections such as a connection from the first layer to the third layer which is written as {1,3,0}.

By using the same function, we can display any fully connected layer. For example, the single layer network can be written as :-

```
network[{10},{{1,1,1}}]
```

where the first set, {10} is one layer network containing 10 nodes and the second set, {{1,1,1}} means the first layer is connected to the same layer with lateral connections. The two layer network with feedforward and feedback connections can be written as :-

```
network[{2,5}, {{1,2,1}}]
```

The network can also contain a layer node arranged in a two-dimensional or topological map, for example :-

```
network[{2,{10,10}},{{1,2,0}}]
```

The set {10,10} within the first set is the second layer which has nodes arranged in 10*10. Its connection is a feedforward connection from the first layer to the second layer.

As explained earlier, the NEUCOMP program cannot call Mathematica directly. An alternative approach is to save the simulation results in a file. This file is then read by a Mathematica program. The file must be declared first before it is used. It is declared under the declaration section as below :-

```
FILES file1;
```

where *FILES* is the reserve word for file type and *file1* is the variable of type file. There are two types of files that can be used which are the file for reading and the file to be written. For the file to be written, it is defined as follows :-

```
OPENWRITE(file1,filename);
```

where the *OPENWRITE* is the reserved word for open file to be written and *filename* can be a string constant or a variable of type string. It is used to give the name of the file to be written.

Let us consider *file1* which is used for displaying the backpropagation network from the simulation program of **section 4.4.1.** Its *filename* is named as "bp.net". The following shows the standard command to display such a graph :-

```
PRINTFILE(file1,"The backpropagation network\n");
PRINTFILE(file1,"{%d,%d,%d}\n",n1,n2,n3);
PRINTFILE(file1,"{{1,2,0},{2,3,0}}\n");
```

The first statement is used to display the title of the graph. The second statement is to display, the nodes *n1*, *n2* and *n3* in the three layer network. They can be any integer constant because their values are assigned at run time. The third statement is to display a feedforward connection between the first layer and the second layer and from the second layer to the third layer. To make a flexible display, such as the number of layers, is quite difficult because it depends on how the NN structure is declared using the vector variable, i.e. *layer1*, *layer2* and *layer3*, and their connections which are the matrix variables, i.e. *weight1* and *weight2*. If the fourth layer wants to be used then the additional declarations are written as :-

```
REAL layer4[n4], weight3[n4,n3];
```

We can then adjust the respective *PRINTFILE* statements to be able to display the four layer network. Similarly for the other networks where we just follow how the network has been written initially.

We now can run the Mathematica program to display the NN structure. The Mathematica program to evaluate the NN structure is called 'displaynet'. When invoked, it prompts for the name of the file to be typed in, i.e. *bp.net*. The function *network* is then called to display the respective graph.

Figure 4.1 shows an example of a three layer network with the first and second layers containing two nodes and the third layer is a single node. **Figure 4.2** shows an example of a single layer network with 5 nodes. **Figure 4.3** shows an example of a two layer network with the feedforward and feedback connections. The figure shows that the input layer contains 2 nodes and the output layer contains 5 nodes. An example of a network containing the layer nodes arranged in a two-dimensional map is shown in **figure 4.4**. The input layer contains 2 nodes and the output layer contains 8 x 8 nodes. The number of nodes will be shown when the size of the node is bigger than 6. Mathematica program cannot cope with the situation when all the nodes wanted to be displayed.

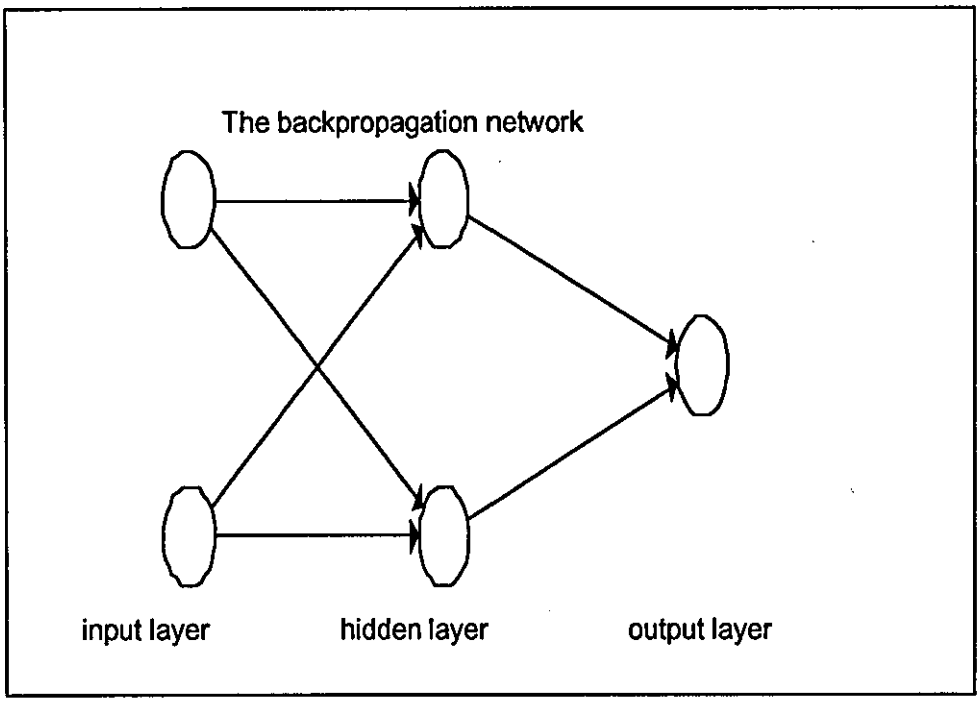


Fig. 4.1: A three layer network

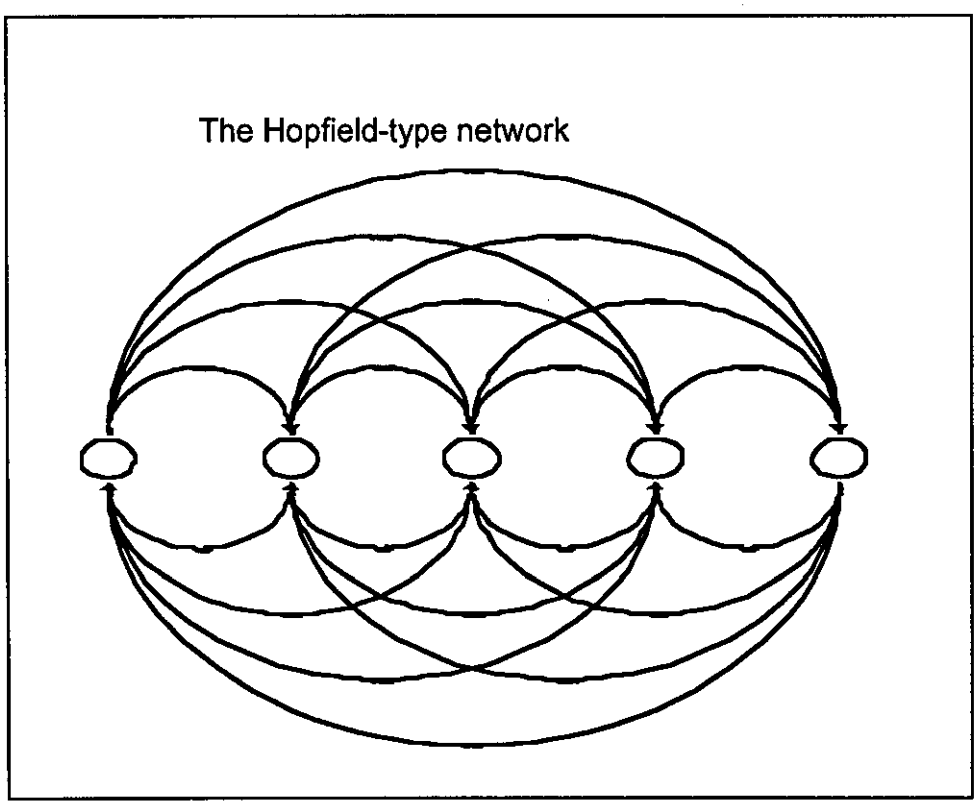


Fig. 4.2: A single layer network

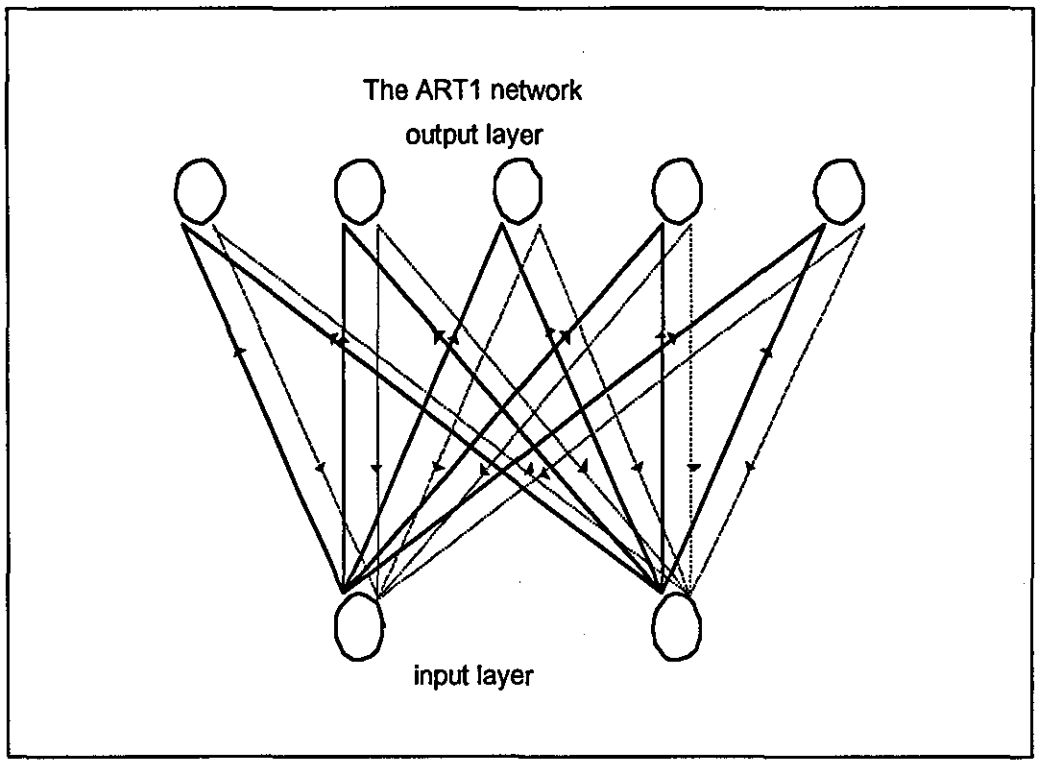


Fig. 4.3: A two layer network with feedforward and feedback connections

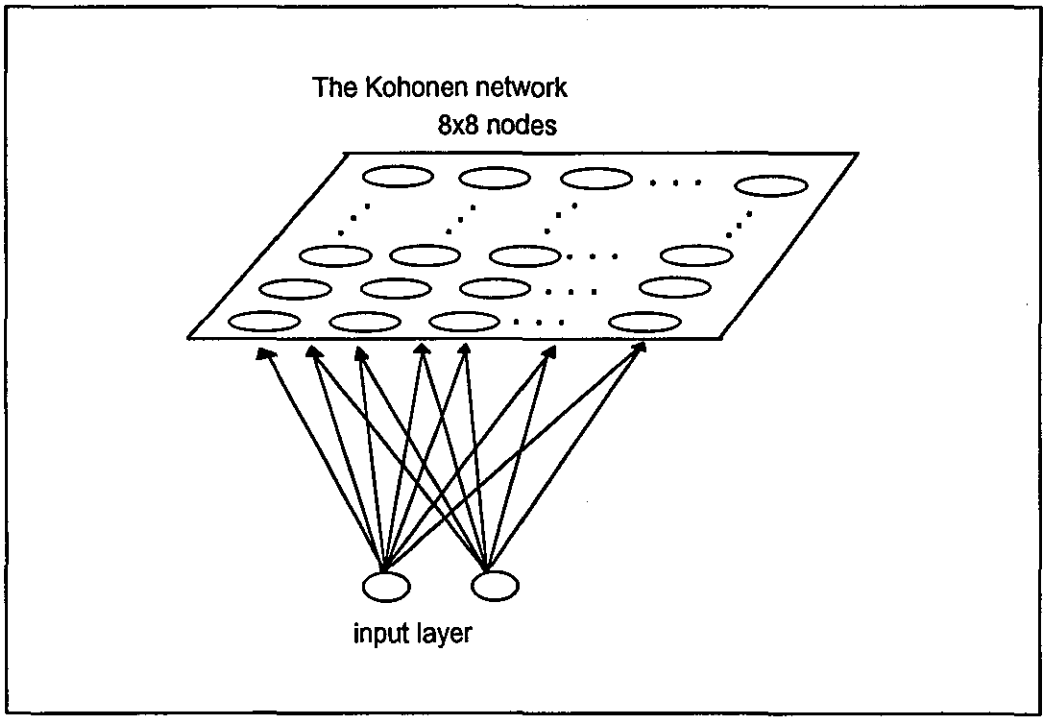


Fig. 4.4: A Topological network

4.5.2 Implementing the XY-graph

The XY-graph is a two-dimensional graph to display a curve of points (x,y). For example, displaying the result of the number of iterations vs. error measures in the backpropagation algorithm. We can plot a single graph or more than one on the same axis.

The algorithm in the Mathematica program for this purpose is written as follows :-

```
read the title of the graph
read the title of the x-axis
read the title of the y-axis
while (more files) do
  read the file name for the graph
  read the title of this graph
  plot the graph using ListPlot
endwhile
Display the whole graph
```

An example of such a graph is shown in **figure 4.5.** The graph shows the different results of solving the XOR problem. To obtain such a graph, the parameters for experiment1 were set as *alpha* = 30 and *beta* = 0.9, and the parameters of experiment2 were set as *alpha* = 8 and *beta* = 0.5. Both experiments were set with the same seeds on weights and biases as 1000, 2000, 3000 and 4000. It is possible to accommodate more than two graphical results on the same axes when further comparison between the results is required.

4.5.3 Implementing other graphs

Another interesting graph to plot are the points for data clustering. It is useful for classification problems using the Kohonen network (**section 2.2.3**).

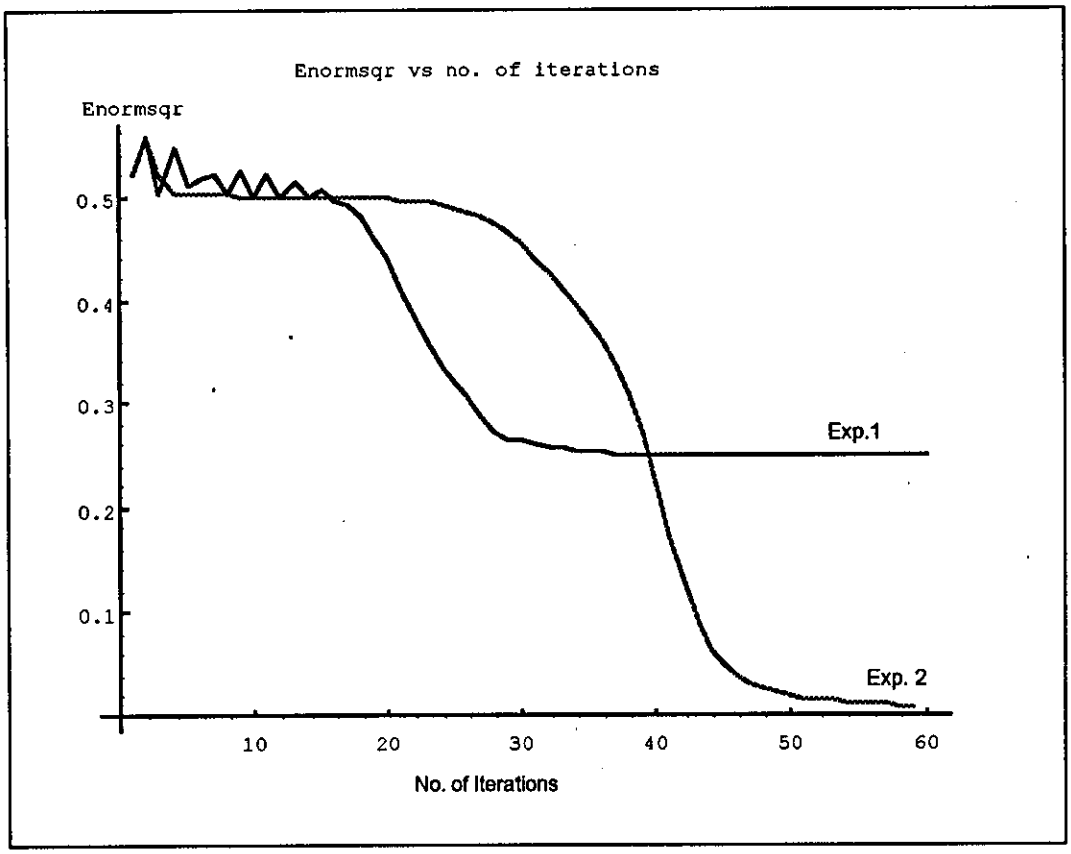


Fig. 4.5: Exp. 1 shows the line has not converged.
Exp. 2 shows the line has converged.

For the data clustering problem, *n1* and *n2* are assigned values of 2 and 64 respectively and the size of the pattern is obtained by the random numbers generated between 0 and 1. The initial weights of the network are set to the value 0.5 plus a small randomised value, i.e. within 10%. **Figure 4.6a** shows the plot of these initial weights. Each unit in the competitive layer is a point on this graph. **Figure 4.6b** shows the network after 1,000 iterations and **figure 4.6c** shows the state of the network after 6,000 iterations. **Figure 4.6d** shows the final state of the network after 20,000 iterations. Each axis of the square in **figure 4.6** goes from 0 to 1 because this is the range of the entries in the input patterns.

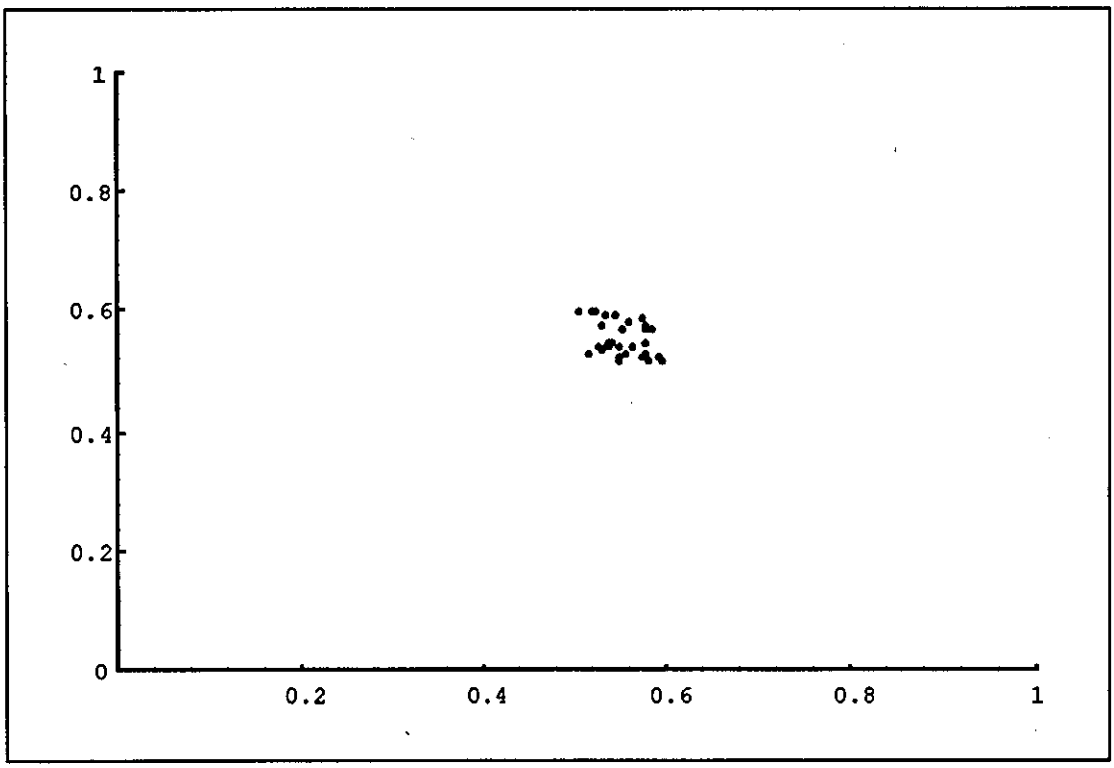


Fig. 4.6a: Initial weights distribution

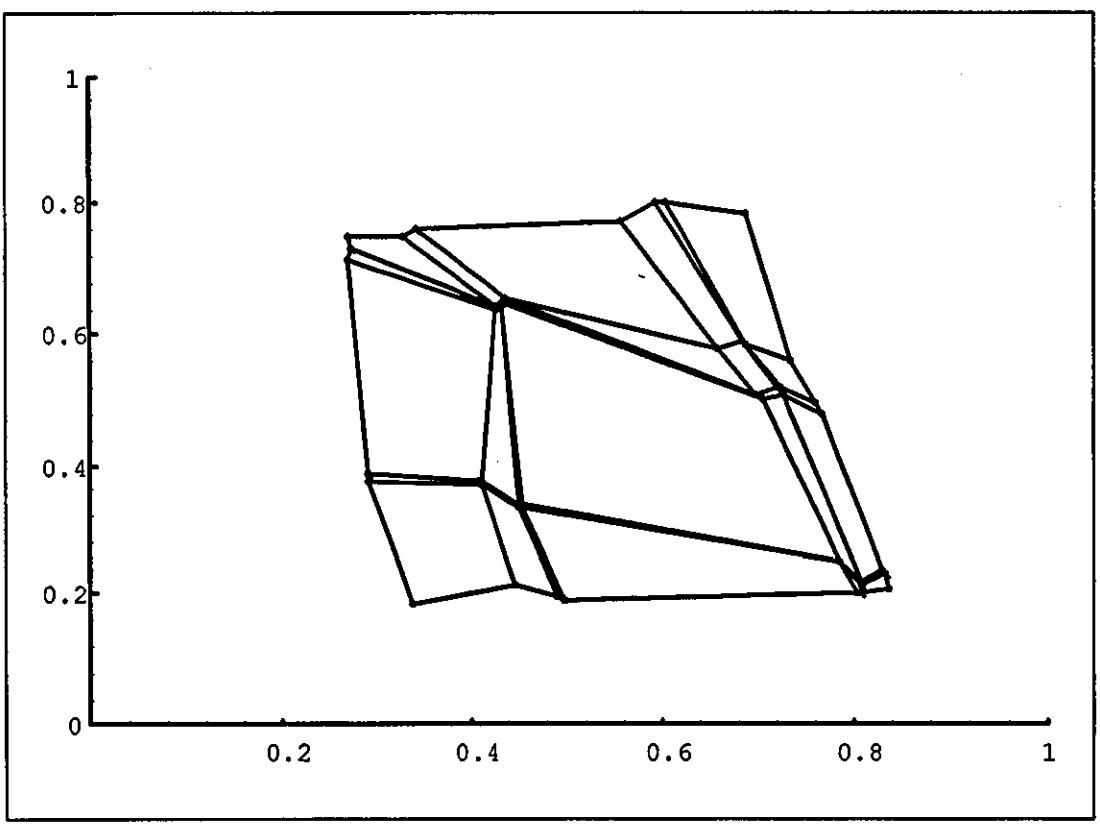


Fig. 4.6b: Weights distribution after 1,000 iterations

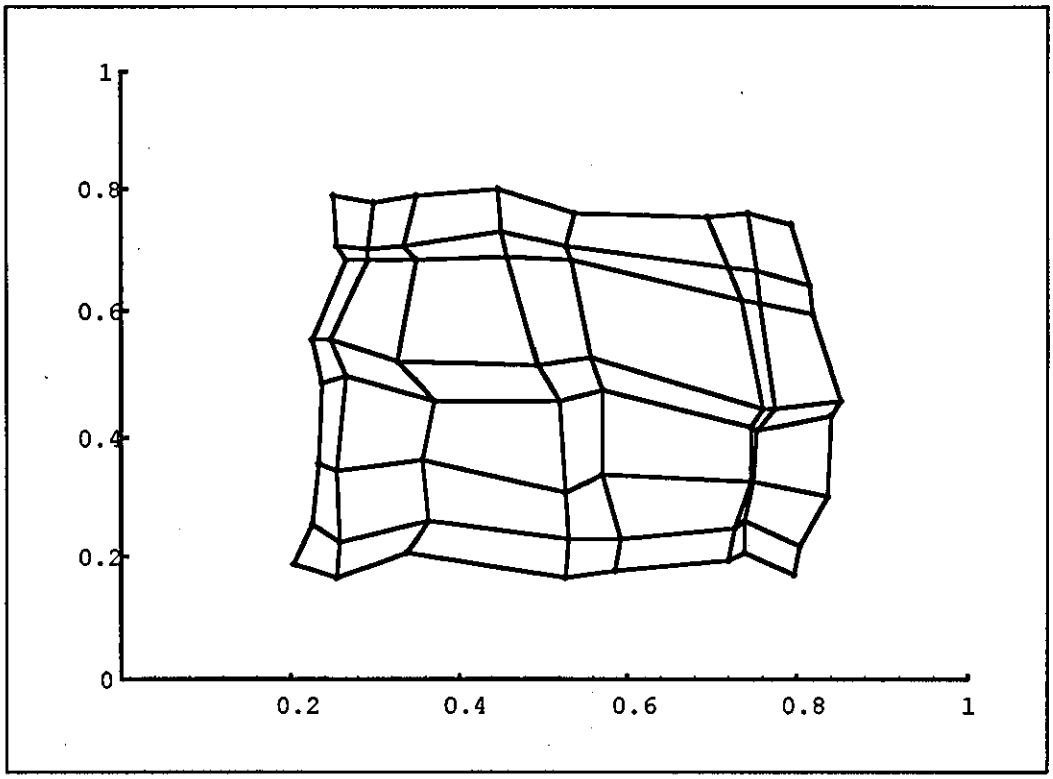


Fig. 4.6c: Weights distribution after 6,000 iterations

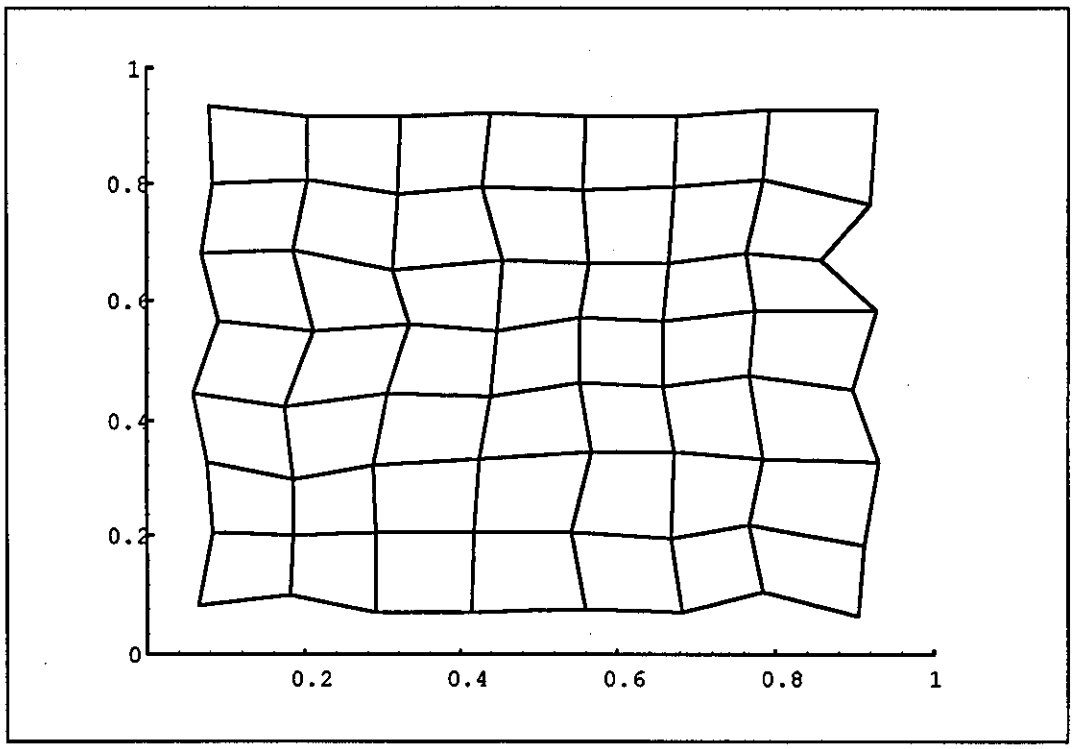


Fig. 4.6d: Weights distribution after 20,000 iterations

CHAPTER 5

A PARALLEL NEURAL NETWORK COMPILER

This chapter discusses the design and implementation of a parallel NN compiler. The design of the compiler follows the strategies of the parallelising compiler [Padua *et al.* (1986), Zima *et al.* (1990)]. However implementation of the compiler is hardware dependent. The technology of computer hardware has been advanced from single processor computers to multi-processor computers. These multi-processor computers are also known as parallel computers or supercomputers. They are capable of producing better and faster performance as more than one processor can work in parallel to solve different parts of a single problem. The parallel NN compiler known as NEUCOMP2 is developed to attain this objective.

There are many different types of parallel computers available today [Babb (1988)]. The parallel computer that is used to develop NEUCOMP2 is the SEQUENT Balance 8000 at PARC. It is a shared-memory parallel machine. It belongs to MIMD (Multiple Instruction Multiple Data) architecture or also known as multi-processor systems.

A parallel computer provides the software tools and a programming system to help a programmer to write a parallel program in order to achieve the high performance of the parallel computer. Parallel programming on the Balance machine can be implemented in two ways depending on the type of application. For executing different tasks or processes (functions or statements) in parallel, function partitioning is used. For executing the same task or process (i.e. matrix/vector operation) in parallel, data partitioning is appropriate. NEUCOMP2 is implemented based on the data partitioning method since its program contains mostly matrix/vector operations.

Experiments are carried out to study the performance of the NN simulations generated by the compiler in terms of the execution time and speedup. The results are then compared with the existing special purpose simulator that used the same parallel machine [Sanossian (1992)]. The only NN model used for this purpose is the backpropagation network since the available results for

comparison was also based on the SEQUENT Balance machine. The backpropagation algorithm written in NEUCOMP2 program is similar to that implemented by Sanossian. However, for other models such as the Kohonen, Counterpropagation, ART1 and Hopfield-type networks, their parallel performance results which are based on the applications will be discussed in the next chapter.

5.1 PARALLEL ARCHITECTURES

The architectures for parallel computer systems are commonly categorised into the SIMD (Single Instruction Multiple Data) and the MIMD (Multiple Instruction Multiple Data) computers [Forrest *et al.* (1987), McBryan (1989), Lafferty *et al.* (1993)]. The Array and Pipelined computers belong to the first type. Most multiprocessor and multicomputer systems belong to the class of MIMD computers. These machines have a set of independent and autonomous processors and every processor is able to execute different instructions concurrently.

The MIMD computers are classified into two further categories, the Shared-Memory Parallel Computers and the Message-Passing Parallel Computers. This classification is based on two different methods of communication amongst the processors. The Shared-Memory Parallel Computers are tightly coupled multiprocessor whilst the Message-Passing Parallel Computers are loosely coupled multiprocessors.

5.1.1 The SIMD Computer Architecture

The SIMD Computer Architecture such as array processors consist of simple processing units (or nodes) that are synchronised to operate in parallel. Each unit consists of an ALU (Arithmetic Logic Unit) and a number of registers. These units are connected to a control unit where the instructions are decoded and broadcast to all

the units in the system. Therefore the units execute the same instruction simultaneously with each unit holding different sets of data. **Figure 5.1** depicts a simple Array computer. As can be seen from the diagram the processing units, i.e. P_1 .. P_n are connected to each other via a data routing network. The shared memory can have multiple modules. Examples of these machines are the Active Memory Technology Distributed Array Processor (AMT DAP) and Connection Machines (CM) of the Thinking Machines Corp. [Zima *et al.* (1990), Lafferty *et al.* (1993)].

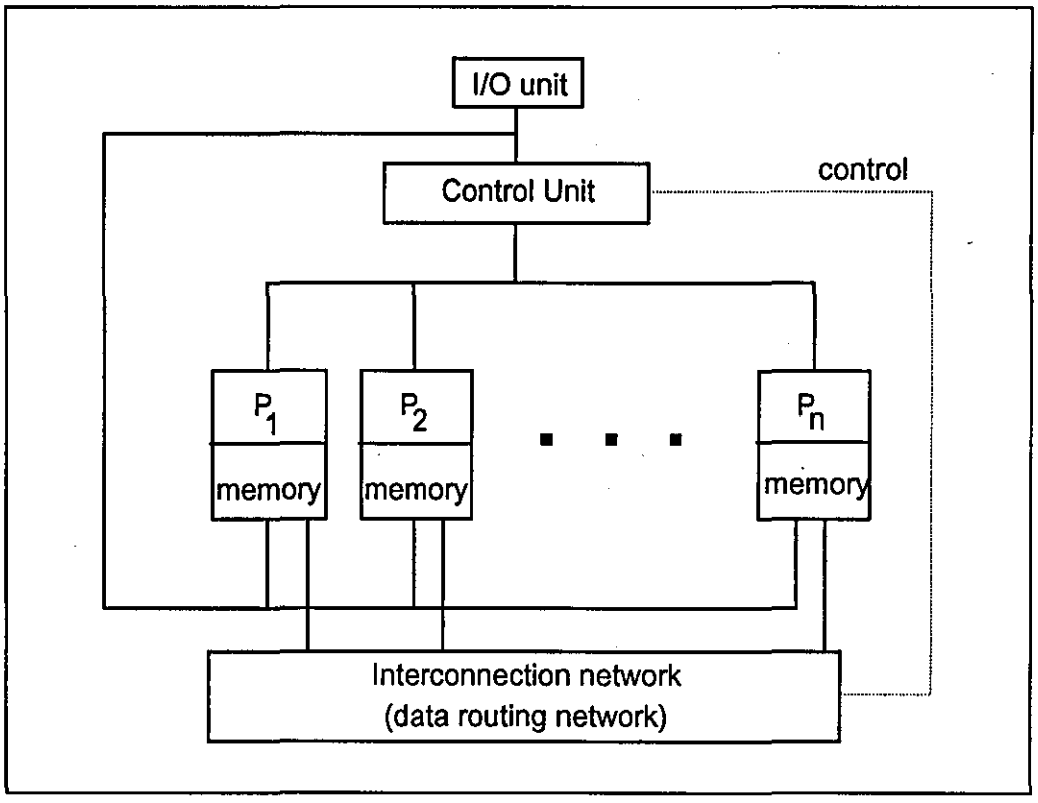


Fig. 5.1: The structure of the SIMD Array processor

5.1.2 The MIMD Computer Architecture

The MIMD computer architecture has a set of independent and autonomous processors. Each processor is able to execute different instructions. The two categories of the MIMD are the Shared-Memory Parallel Computers and Message-Passing Computers.

5.1.2.1 The Message-Passing Parallel System

In a Message-Passing Parallel system, each processor has a number of input/output devices connected to it and a local memory where most of the instructions and data are stored. These systems are also known as Local Memory Systems, Loosely-Coupled Systems or Distributed Memory Systems. Communications among the processors are performed through a message transfer system. Such systems are efficient for tasks that require minimum interactions between the processors. The message transfer system is usually a routing network. **Figure 5.2** depicts a simple loosely coupled system. The transputer system is an example of such an architecture [Almasi *et al.* (1989), Hwang *et al.* (1984)].

5.1.2.2 The Shared-Memory Parallel System

The Shared-Memory Parallel System, sometimes called the Tightly-Coupled System, has a set of processing units and a pool of memory available to all processors through which they communicate via a simple time shared bus or interconnection network. This type of architecture is illustrated in **figure 5.3**. Examples of these systems are the SEQUENT Balance, Encore Multimax and Alliant FX/8 [McBryan (1989), Lafferty *et al.* (1993)]. Due to the problem imposed by the communication through the shared memory, they usually have a relatively small number of PEs. For example, the SEQUENT Balance and Encore Multimax can only have at most 12 and 20 processors respectively for efficient operation.

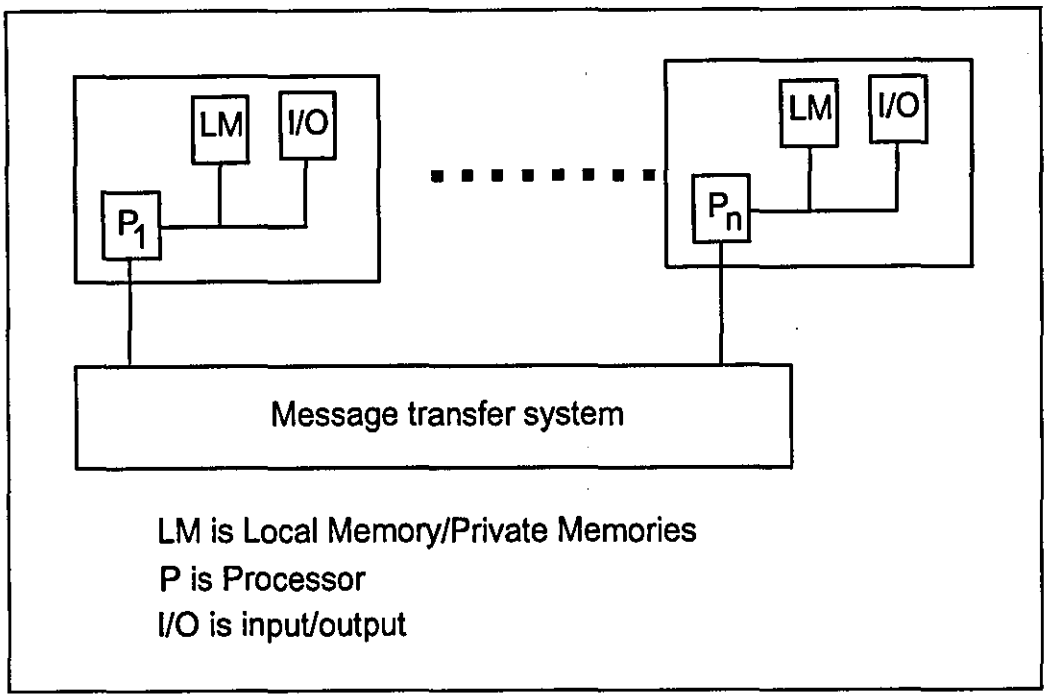


Fig. 5.2: The structure of the Message-Passing systems

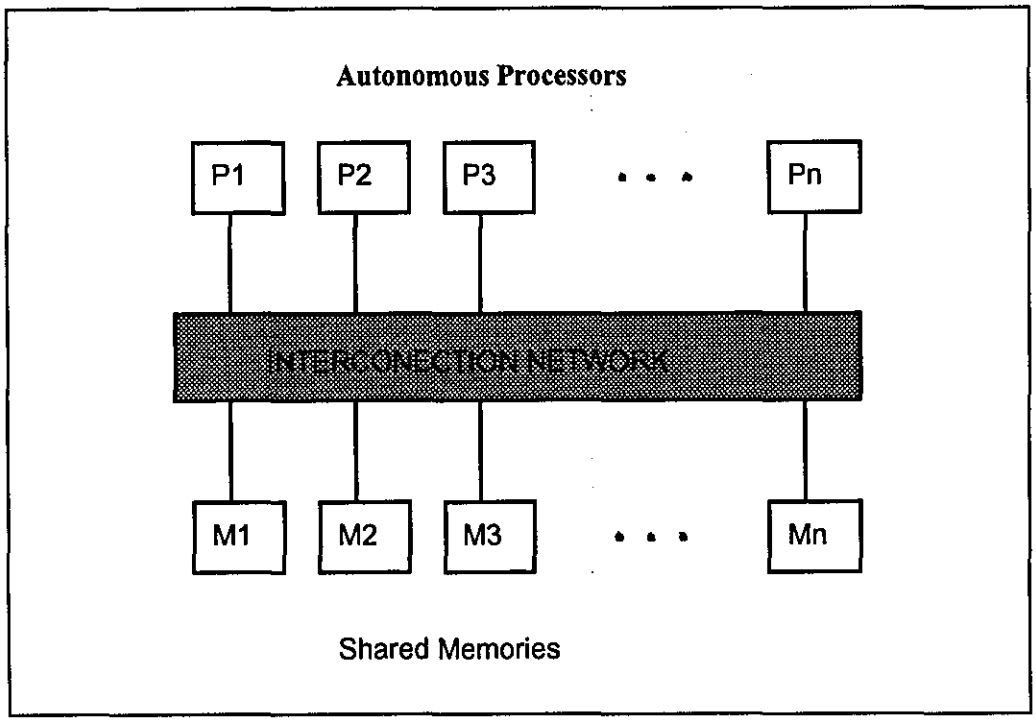


Fig. 5.3: Configuration of the Shared-Memory Parallel systems

5.1.3 The SEQUENT Balance 8000

SEQUENT systems are homogeneous multiprocessors, i.e. computers that incorporate multiple identical processors (CPUs) and a single common memory [Osterhaug (1989)]. The SEQUENT CPUs are general-purpose, 32-bit microprocessors.

The computer used throughout the work presented here is the SEQUENT Balance 8000. It is a tightly coupled (Bus) based MIMD machine with up to 12, 32 bit microprocessors each capable of executing 0.7 MIPS. Each processor consist of a CPU, a hardware floating point accelerator and a paged virtual memory management unit. A two level page table is used to access 16 Mbytes of virtual memory. Each processor contains a cache memory of 8 Kbytes which holds the most recently accessed instructions and data. When a processor updates some data in its cache, the data in the main memory and other caches are updated at the same time. The cache is intended to reduce the traffic burden on the bus.

The operating system is DYNIX, which is derived from UNIX. In particular, the scheduler in DYNIX has the choice of any one of the 12 processors to allocate tasks to, so that even if no parallel program is being run, the total work load is distributed amongst the available processors.

5.2 PARALLEL PROGRAMMING SYSTEMS

In order to utilise the available processors and make use of their parallel capabilities, software must be provided. The development of the parallel software is partly dependent on the hardware available. For example, in the most general model of parallel architecture, i.e. the MIMD system, there must exist language constructs that allow the programmer to program the individual processors and to define the data on which they are to operate.

Parallelism in a computer system can be achieved through two ways, multiprogramming (or timesharing) and multitasking [Osterhaug (1989)]. Multiprogramming allows several jobs (or programs) to be processed at the same time and this will give the maximum throughput of the computer. This is common on most computers nowadays which allow more than one user to log on to the machines, although they may have one processor [Brawer (1989), Silberschatz (1991)]. The operating system in the computer, such as UNIX, is able to handle multiprogramming by allocating jobs in a ready queue to the CPU as soon as it is free.

In the other situation, multitasking is a programming technique that allows a single application to consist of multiple processes executing concurrently [Zima *et al.* (1990)]. Each one of these processes will be handled by the different available processors. Multitasking yields an improved execution speed of an individual program.

5.3 PARALLEL PROGRAMMING ON THE SEQUENT BALANCE

The operating system for the SEQUENT Parallel Machine namely DYNIX supports multiprogramming and multitasking [Osterhaug (1989)]. It has library commands to create processes and to synchronise them such as the 'fork', 'join' and 'lock' instructions. An illustration of the fork and join operations is shown in **figure 5.4**. So, it is left to the programmer to write a parallel program specifying which tasks are to be executed in parallel.

In the multitasking programming methods, there are two methods available for the users to implement the programs. They are data partitioning and function partitioning. Data partitioning involves creating multiple, identical processes and assigning a portion of the data to each process. This method is also called homogeneous multitasking. Data partitioning is appropriate for applications that perform the same operations repeatedly on large collections of data, i.e. vectors.

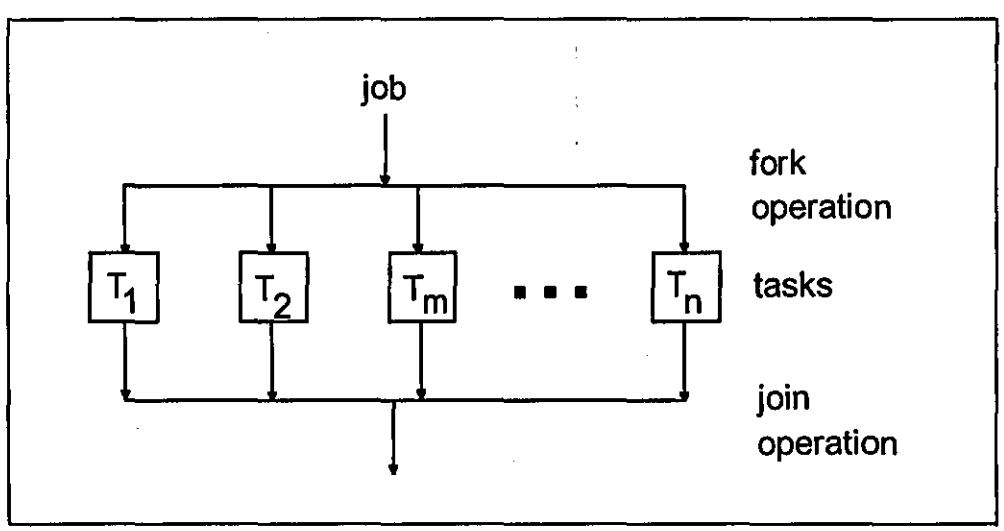


Fig. 5.4: Multitasking environment

The other method, function partitioning, involves creating multiple unique processes and having them simultaneously perform different operations on the shared

memory data set. It is suitable for applications that include many unique subroutines or functions. This method is also called heterogeneous multitasking. Applications such as flight simulation, program compilation and traditional process control adapt well to function partitioning.

While some applications require function partitioning or a combination of data and function partitioning, most problems adapt more easily to data partitioning. This last method offers some advantages over function partitioning, such as less programming effort is required to convert a serial program to a parallel algorithm. Furthermore, with data partitioning, it is easier to achieve an even load balancing among processors and also easier to adapt the programs automatically to the number of available processors.

In this chapter, discussion will only refer to the data partitioning technique. This method is used to implement a parallel NN compiler for the shared-memory parallel computer.

5.3.1 The Data Partitioning method

The data partitioning method is suitable to execute loop iteration in parallel [Osterhaug (1989)]. The loop iteration is chosen as the code section to be parallelised because the parts that offer the best opportunities amenable to parallelism are the loops [Padua *et al.* (1986), Zima *et al.* (1990), Mohd-Saman *et al.* (1993)].

Data partitioning involves creating multiple, identical processes (i.e. loop iteration) and assigning a portion of data to each process. Assigning portions of data means each iteration can be executed simultaneously depending on the number of available processes. Load balancing amongst processes is achieved by a scheduling strategy.

The following describes the process of performing data partitioning :-

(1) A special function is used to fork a subprogram that contains a loop iteration into a set of child processes and assigns an identical copy of the subprogram to each process for parallel execution. This function creates a copy of any private data for each process.

(2) Each copy of the subprogram executes a segment of the loop iteration either using static or dynamic scheduling [Osterhaug (1989)]. The static scheduling divides the loop iterations evenly among the processes. In dynamic scheduling, the loop iterations are treated as a task queue, and each process removes one or more iterations from the queue, executes those iterations, and returns for more work. Dynamic scheduling requires communication between processes.

(3) If the loop being executed in parallel is not completely independent which means there exist data dependencies (**section 5.3.3**), the subprogram may contain calls to a function that synchronises the parallel processes at critical points by using locks or barriers.

(4) When all the loop iterations have been executed, control returns from the subprogram. The parallel execution processes can either be terminated after the calling subprogram, suspends their execution until they are needed to execute another subprogram, or left to spin in a busy wait state until they are needed again.

5.3.2 Parallel Programming tools

Some of important DYNIX Parallel Programming tools used to implement the parallel compiler are discussed here.

The DYNIX Parallel Programming Library includes three sets of routines. These are a microtasking library, a set of routines for general use with data partitioning programs, and a set of routines for memory allocation in data partitioning programs.

The microtasking library routines allows the creation (fork) of a set of child processes, assign the processes to execute loop iterations in parallel, and synchronise the processes as necessary to provide a proper data flow between loop iterations.

For example, the function *m_fork* is used to create new processes. A new process is called a child process and it is a copy of the original process (called parent process). The child process is allowed to access the main memory and any open file. The number of processes created can be set using the function *m_set_procs*. Each child process has an ID number associated with it when it was created. During the execution of the process it might be necessary to require the ID number, this can be done by calling function *m_get_myid*. The parent ID number is 0. The function *m_kill_procs* is used to terminate child processes which is written after the function *m_fork*, the function *m_park_procs* is used to suspend the execution of the child processes while the parent process is involved in some operation. The execution of the child processes can be resumed using the function *m_rele_procs*. When many processes running in parallel try to modify a shared variable (**section 5.3.3**), they have to be synchronised. This can be controlled by shared data structures called 'semaphores'. The simplest of all semaphores is the function *lock* that allows a user to create a critical code region that can be accessed by only one process or using the function *m_sync* to check at a barrier. A barrier is a synchronisation point where a process waits at a barrier until other processes arrive before it can proceed.

The general-purpose data-partitioning routines include a routine to determine the number of available CPUs and several process synchronisation routines that

are more flexible than those available in the microtasking library. For example, *cpus_online* returns the numbers of the CPUs on-line and *s_wait_barrier* is wait at a barrier.

The memory allocation routines allow a data-partitioning program to allocate and de-allocate shared memory and to change the amount of shared and private memory assigned to a process. For example *shmalloc*, allocate shared data memory.

Figure 5.5 gives an example of using some of these functions.

5.3.3 Analysing Data Dependencies

Before implementing data partitioning, data dependencies in the loop have to be analysed in order to guarantee correct results. The analysis involves finding variables that depend on previous operations and variables that may be executed in any order. Data dependence analyses is an important task in parallelising a sequential program. This analysis will give information on the inter-relation of statements based on how the data in the program is computed and used.

Data dependencies occur in two parts. In the first part, data dependencies occur in the programs' statements [Padua *et al*. (1986), Osterhaug (1989), Polychronopoulos (1988)]. For example :-

```
s1 :   a = b + c;
s2 :   d = a - e;
```

where statement *s1* must be executed first since it contains the variable *a* being stored data, then followed by statement *s2* in which *a* is being read. The second part of data dependencies is on the loop iteration which is the concern of this presentation. The outermost loop is chosen as a code section to be executed in parallel.

```
#include <stdio.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>
  ...
main()
{
   void subprogram(), m_fork(),
        m_kill_procs();
   int nprocs;
   printf("Type number of processors: ");
   scanf("%d",&nprocs);
   m_set_procs(nprocs);
   m_fork(subprogram);
   m_kill_procs();
}

void subprogram()
/*This subprogram contain loop iteration
  to be executed in parallel */
{
  int nprocs;
  nprocs = m_get_numprocs();
   ...
}
```

Fig. 5.5: Example of parallel program

Before analysing the data dependencies, identifying which data can be shared amongst parallel processes and which data is local to each process is discussed first. The data that is shared is called shared variables and the data that is local is called private variables. The private variables are initialised in each iteration. They are usually scalar variables. Shared variables need further attention. Data dependencies may occur when a

program attempts to read and write shared variables in more than one loop iteration. These variables can sometimes pass incorrect information between loop iterations if the iterations are executed out of order or two loop iterations try to write the variable simultaneously.

If the shared variable is a read-only variable or an array where each element is referenced by only one loop iteration, then they are considered as independent shared variables. However, the shared variables which are dependent can belong to three categories [Osterhaug (1989)] which are :-

(1) Reduction variables
(2) Locked variables
(3) Ordered variables

which distinguishes the ways the variables are used.

5.3.3.1 Analysing Reduction variables

A reduction variables can be an array or a scalar which has the following properties :-

```
variable op= expression
```

where *op* is either '+','-','*' or '/'. The following example shows the addition of all numbers in an array variable:-

```
for (i = 0; i<n; i++)
    sum += a[i]
```

where *n* and *a* are independent shared variables because they are read only, *i* is a local variable because it is initialised on every iteration and *sum* is a shared reduction variable because it involves the '+='

assignment. If the loop is executed in parallel *sum* may contain incorrect results.

5.3.3.2 Analysing Locked variables

A locked variable can be an array or a scalar involving read and write operations in more than one iteration. The following example shows how the minimum value is searched in an array variable :-

```
least = 999;
for (i = 0; i<n; i++) {
     min = a[i];
     if ( min < least ) {
       least = min;
     }
}
```

where *n* and *a* are independent shared variables because they are read only, *i* and *min* are local variables because they are initialised on every iteration, and *least* is locked shared variable. Since the loop iterations will be executed in parallel, this variable can hold any value in each process which is not necessarily the minimum result that is required. In order to make sure that only one loop iteration is using this variable at a time, it has to be locked.

5.3.3.3 Analysing Ordered variables

An ordered variable is an array variable which yields correct result only if the operations involving the variable are executed one iteration at a time, in sequential order. The following example contains an ordered variable :-

```
for (i = 0; i<n; i++) {
  x[i] = x1[i] + x2[i];
  y[i] = x[i+1] - x[i-1];
}
```

where *n* and *y* are independent shared variables because *n* is a read only variable and *y* is an array where each element is referenced by only one loop iteration, *i* is the local variable because it is initialised on every iteration and *x* is an ordered shared variable because the expressions *x[i+1]* and *x[i-1]* would contain incorrect values if the loop iterations were executed in any order.

5.3.4 Transforming into Parallel code

Figure 5.5 has shown some functions to implement parallel programs. In this section, the transformation of loop iterations into parallel code is discussed.

The first task is distributing loop iterations to processes. This is known as scheduling. The following static scheduling is introduced in the outermost loop of the *subprogram* that is being forked (**figure 5.5**) :-

```
for (i = m_get_myid(); i<n; i+=nprocs) {
    ...
}
```

where *i* is set to process ID produced by function *m_get_myid* and the variable *nprocs* is set to the number of processes produced by function *m_set_numprocs* (**figure 5.5**). The loop iterations are divided evenly among the processes.

The next task is to impose parallel mechanisms to protect dependent variables in order to produce correct results. The following section describes techniques for transforming reduction, locked and ordered type data dependencies.

5.3.4.1 Transforming Reduction variables

The following example shows the handling of reduction variables from an example of **section 5.3.3.1 :-**

```
lvar = 0;
for (i = m_get_myid(); i<n; i+=nprocs)
  lvar += a[i]
m_lock();
  sum += lvar;
m_unlock();
```

A local variable *lvar* is used to hold the sum of array variable within each loop iteration. At the end of each loop iteration the function *m_lock* is called to perform the reduction operation to combine *lvar* with the shared variable, *sum* and call the function *m_unlock*. The functions *m_lock* and *m_unlock* are used to ensure that the code section within it is executed by one processor at a time.

5.3.4.2 Transforming Locked variables

A locked variable cannot be executed simultaneously, so the functions *m_lock* and *m_unlock* are used as before. The function *m_lock* call should appear on the line immediately preceding the first reference to a locked variable, and the function *m_unlock* call should appear after the last reference of a locked variable.

The following example shows the handling of reduction variables from an example of **section 5.3.3.2 :-**

```
least = 999;
for (i = m_get_myid(); i<n; i+=nprocs) {
  min = a[i];
  m_lock();
    if ( min < least ) {
      min = a[i];
    }
  m_unlock();
}
```

The functions *m_lock* and *m_unlock* are used within the parallel loop to ensure that the code section is executed by one loop iteration at a time.

5.3.4.3 Transforming Ordered variables

The code section that contains the ordered variables must be executed in order. The following example shows the transformation of ordered variable from an example of **section 5.3.3.3 :-**

```
for (i = m_get_myid(); i<n; i+=nprocs) {
  while (xguard != i) continue;
  x[i] = x1[i] + x2[i];
  y[i] = x[i+1] - x[i-1];
  xguard = xguard + 1;
}
```

where *xguard* is a new shared integer variable. It is declared in the main and set to the starting value of the loop iteration. The conditional statement used before the first reference to the order variable is to allow the loop execute only when the loop index is equal to *xguard*. This variable is then incremented at the end of the last reference of the ordered variable to allow for the next sequential execution.

5.4 PARALLEL NEURAL NETWORK COMPILER (NEUCOMP2)

A study of the NN compiler called NEUCOMP (**chapter 4**) to generate a general purpose NN simulation programs have been successfully implemented. These simulation programs were executed sequentially.

A further study of designing the NN compiler for a parallel machine has been carried out. This section discusses the development of an upgraded version of NEUCOMP named NEUCOMP2. NEUCOMP2 can generate a parallel NN simulation program running on a shared-memory parallel machine.

NEUCOMP2 contains an additional stage for detecting the existence of parallelism in the sequential program generated by NEUCOMP and transforms it into a parallel version specifically for a shared-memory parallel machine. When a different parallel machine is introduced, this routine can be changed to suit the specification required by that machine.

5.4.1 Design of Parallel Neural Network Compiler

Designing a parallel NN compiler basically follows the design of a parallelising compiler. A parallelising compiler (sometimes referred to as a supercompiler) is a software system that compiles programs targeted for execution on a parallel architecture system [Padua *et al.* (1986), Zima *et al.* (1990)]. This software tool takes as input the sequential program, detects any form of parallelism that exists and carries out the transformation process.

Figure 5.6, shows the process of generating a parallel NN simulation program. The step from the source program (NEUCOMP2 program) to generate a sequential simulation program, follows the step compiled by NEUCOMP.

The next compilation phase is the parallelising stage. It contains routines to detect parallelism and transform into parallel codes. The design of the routine

is dependent on the architecture of the parallel machine. In this section, the design and implementation of the parallelising routine on a shared-memory parallel machine such as the Balance machine (**section 5.3**) is discussed.

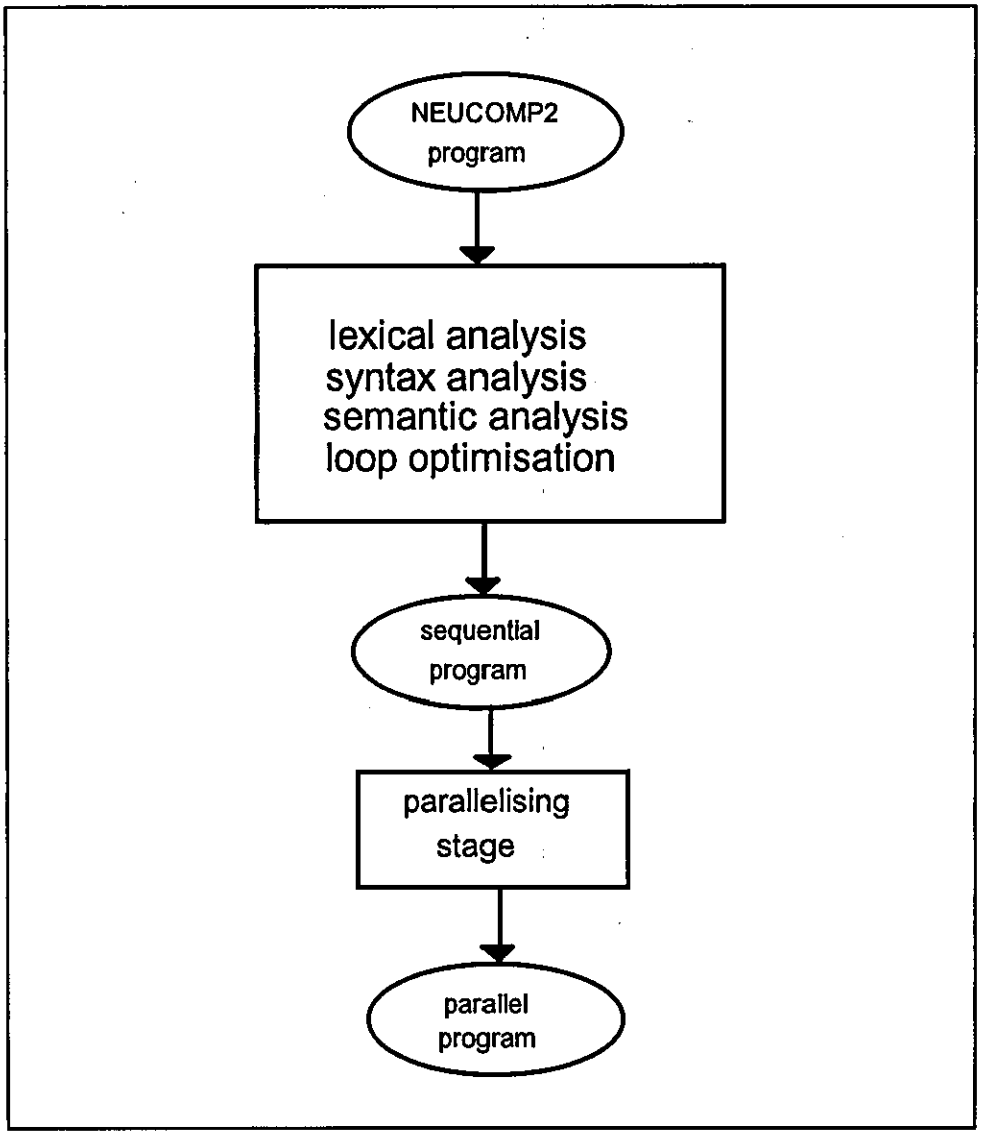


Fig. 5.6: Process of compilation on a NEUCOMP2 program

The language for NEUCOMP2 is called the NEUCOMP2 language. The NEUCOMP2 program has an extra reserved word called PARALLEL which must be included when a certain procedure is to be executed in parallel. In this case the most crucial part in NN simulation is a procedure that involves training the network. For example, the NEUCOMP/NEUCOMP2 program is written as :-

```
MAINPROGRAM
    ...
  CALL training;
    ...
END;
```

To parallelise the procedure training, statement CALL is replaced with PARALLEL, as shown belows :-

```
MAINPROGRAM
    ...
  PARALLEL training;
    ...
END;
```

without the statement PARALLEL, NEUCOMP2 treats the program as a sequential program.

5.4.2 Implementing the Parallelising stage

The code section identified by NEUCOMP2 for parallel execution is the loop. These parts offer the best opportunities amenable to parallelism [Padua *et al.* (1986), Zima *et al.* (1990), Mohd-Saman *et al.* (1993)].

The routine for parallelism will be evoked when the word PARALLEL is included in the respective procedure. The routine then undergoes the following stages :-

(1) Detection of the loop iteration
(2) Creating new procedure for loop iteration
(3) Analyse data dependencies
(4) Transformation process

The following sections discuss the development of the above stages.

5.4.2.1 Detection of the loop iteration

The loop iterations for all matrix-vector statements are chosen as code sections to be executed in parallel. There are two types of loops to be generated: the 'for loop' and 'while loop'. This follows the explanation given in **section 4.2.5.2** under 'Translating an assignment statement'.

The matrix assignment statements are generated into two 'for loops', i.e. *for* (*I* = ...) and *for* (*J* = ...). For example, the NEUCOMP/NEUCOMP2 program code for updating the weights using the backpropagation algorithm (**section 4.4.1**) has the following form :-

```
weight += alpha*dweight + beta*cweight
```

where *weight*, *dweight* and *cweight* are the matrix variables and, *alpha* and *beta* are the scalars. The translated statements are as follows :-

```
for (I = ... )
for (J = ... )
  weight[I][J]+=alpha*dweight[I][J]+beta*cweight[I][J];
```

where *I* and *J* are the system variables (reserved words) which are written as capital letters.

The vector assignment statements are generated into a single 'for loop', i.e. *for* (*I* = ...). For example, to assign the training pattern into the input layer, it is written as follows:-

```
layer1 = pattern@;
```

where *layer1* is an input layer and *pattern* is a matrix variable. The symbol '@' means all its elements at the specific row determined by reserved word *ROW* are assigned to *layer1* (**sections 4.4.1** to **4.4.4**). The translated statements are as follows :-

```
for (I = ... )
  layer1[I] = pattern[ROW][I];
```

The second type of loop statement is the 'while loop'. The following assignment statements provided by NEUCOMP/NEUCOMP2 will be translated into the 'while loop' statement are :-

```
variable< = expression
variable> = expression
```

where the first symbol '<' is used in the Kohonen and Counterpropagation algorithms for finding the winner node based on the minimum calculation of the expression. It is written as :-

```
layer2< = DISTANCE(layer1,weight1);
```

and the second symbol '>' is used in the ART1 algorithm for finding the winner node based on the maximum calculation of the expression. It is written as :-

```
layer2> = weightf*layer1;
```

NEUCOMP translates the Kohonen algorithm into the following code statements which contains the 'while loop' statement.

```
I = 0;
SCALAR0 = DISTANCE(layer1,weight1,I,n1);
ROW = 0;
while ( ++I < n2) {
  layer2[I] = DISTANCE(layer1,weight1,I,n1);
  if (layer2[I] < SCALAR0) {
    SCALAR0 = layer2[I];
    ROW = I;
  }
}
```

Fig. 5.7: The Sequential code for the Kohonen algorithm

where *I*, *SCALAR0* and *ROW* are the system variables, *n1* is the size of *layer1* and *n2* is the size of *layer2*. *DISTANCE* is the built-in function. The final result of the above translation is that *ROW* or the winner node contains the index of which *layer2* is the minimum and *layer2* holds that minimum value.

The translated statement for the second statement, i.e. ART1 algorithm, is similar provided that the sign '<' is replaced by '>' and the final result is that the *ROW* or winner node contains the index of which *layer2* is the maximum and *layer2* holds that maximum value.

5.4.2.2 Creating new procedure for loop iteration

Once the respective loop iteration has been detected, NEUCOMP2 extracts that loop from its position and places it into a newly created procedure called PROCESS followed by an integer number starting with 0 to distinguish it from another newly created procedure, if any. It's original place will then be replaced by this name as a calling procedure. For example, the translated code for the statement

```
layer1 = pattern@;
```

is written as follows :-

```
void training()
...
{
  for (I = ... )
    layer1[I] =     pattern[ROW][I];
  ...
}
```

NEUCOMP2 translates the above 'for loop' into the following code statement :-

```
  ...
void training()
void PROCESS0()
...
{
  PROCESS0();
   ...
}

void PROCESS0()
{
  for (I = ... )
    layer1[I] = pattern[ROW][I];
  ...
}
```

PROCESS0 is a unique name and written in capital letters. If more than one loop is detected, the next new procedure will be named as *PROCESS1* and so on.

If there are more loops being considered previously arranged consecutively, they are then combined into a single procedure. For example, calculating the activation value for all layers in the backpropagation algorithm written in NEUCOMP/NEUCOMP2 codes is as follows :-

```
PROC training
   ...
  layer1 = pattern@;
  layer2 = SIGMOID(weight1*layer1 + bias2);
  layer3 = SIGMOID(weight2*layer2 + bias3);
   ...
END;
```

where *layer1*, *layer2* and *layer3* are the vector variables. The translated codes are generated in the form of sequential codes as shown below :-

```
    ...
void training()
    ...
{
    ...
  for (I = ... )
    layer1[I] = pattern[ROW][I];
  for (I = ... )
    layer2[I] = SIGMOID(Mul_mat_vec(weight1,layer1,n1,I) + bias2);
  for (I = ... )
    layer3[I] = SIGMOID(Mul_mat_vec(weight2,layer2,n2,I) + bias3);
}
```

where *Mul_mat_vec* is the C function used to calculate a matrix-vector multiplication.

NEUCOMP2 then combines the above loops into the translated codes as can be seen in **figure 5.8.**

5.4.2.3 Analysing Data Dependencies

The loops to be executed in parallel are now in the newly defined procedure. All variables usage within the loop iterations have to be analysed in order to identify which variable depends on previous operations. This is to guarantee correct results when these statements are executed simultaneously.

```
   ...
void training()
void PROCESS0()
   ...
{
   ...
  PROCESS0();
   ...
}

void PROCESS0()
{
  int I;
  for (I = ... )
    layer1[I] = pattern[ROW][I];
  for (I = ... )
    layer2[I] = SIGMOID(Mul_mat_vec(weight1,layer1,n1,I) + bias2);
  for (I = ... )
    layer3[I] = SIGMOID(Mul_mat_vec(weight2,layer2,n2,I) + bias3);
}
```

Fig. 5.8: PROCESS0 holds the 'for loop'

During analysis, NEUCOMP2 groups the variable usage into 5 groups namely group0, group1, group2, group3 and group4. The variable usage in each group have the following characteristics :-

(1) The group0 contains a variable written in the form:-

x += ....

where *x* is read and written by a single statement.

(2) The group1 contains a variable written in the form:-

```
... = x
x = ...
```

where *x* is read first and then written in other statement.

(3) The group2 contains a variable written in the form:-

```
... = x
```

where *x* is read only.

(4) The group3 contains a variable written in the form:-

```
x = ...
```

where *x* is written only.

(5) The group4 contains a variable written in the form:-

```
x = ...
... = x
```

where *x* is written first and then read in other statement.

From the group classification, NEUCOMP2 can then classify which type of data dependencies that may occur. NEUCOMP2 assumes scalar variables may cause data dependencies but not a vector or matrix variable. They are operated independently within the loop iteration where each element is referenced by only one loop iteration. The scalars that exist in group0 are of type reduction variables. The scalars that exist in group1 and group3 are of type locked shared variables because they are written many times when running in parallel. The scalars that exist in group2 are independent shared variables because they are read only. The dependent variables can be removed by a transformation process

which contains parallel mechanisms to transform its part to run correctly in parallel (**section 5.4.2.4**). The scalars that are in group4 are local because they are initialised on every iteration.

Other cases that the parallelising routine in NEUCOMP2 does not consider are :-

(1) The statements FOR and WHILE loop provided by the NEUCOMP/NEUCOMP2 language. Since the main purpose of using the NEUCOMP/NEUCOMP2 program, is to make use of matrix/vector assignments, the use of FOR and WHILE statements is not common. If used it is assumed that the number of loops used is very small.
(2) A shared ordered variable.
(3) When the size of the loop is less than 5.

For cases (2) and (3), NEUCOMP2 will then consider the next inner loop.

5.4.2.4 Transformation Processes

Transformation processes involve the translation of the sequential part into its parallel version after information about variable usage is done. It uses the parallel library routines provided by SEQUENT Balance (**section 5.3.2**) for handling data dependencies, etc.

All variables in a loop iteration declared as global are redeclared as shared variables. The calling procedure created by NEUCOMP2 as discussed earlier, i.e. *PROCESS0* is then forked by the routine *m_fork*. The use of parallel routines such as *m_get_numprocs* and *m_get_myid* are also included. The following example shows the transformation of the program from **figure 5.8**.

```
  ...
shared float *layer1; /* global variable */
  ...
void training()
void PROCESS0();
  ...
{   ...
  m_fork(PROCESS0,ROW);
   ...
}

void PROCESS0(ROW)
int ROW;
{  int NPROCS,I;
   NPROCS = m_get_numprocs();
   for (I = m_get_myid(); I<n1; I+=NPROCS)
      layer1[I] = pattern[ROW][I];
   ...
}
```

where *ROW* has a value needed in the loop iteration and therefore it is passed through the argument list of *PROCESS0*.

Transforming a Reduction variable

For a reduction scalar variable that exists in group0, NEUCOMP2 performs two types of translations. The first type of translation is that if the reduction scalar variable is declared by the user as global. The following example shows the transformation of the loop iteration which contains the reduction scalar variable, *sumerror*.

```
  ...
shared float *error, sumerror;
  ...
void training()
{  ...
  m_fork(PROCESS0);
  ...
}

void PROCESS0()
{ float SCALAR0;
  int NPROCS,I;
  NPROCS = m_get_numprocs();
  SCALAR0 = sumerror;
  for (I = m_get_myid(); I<n1; I+=NPROCS)
    SCALAR0 += error[I];
  m_lock();
    sumerror += SCALAR0;
  m_unlock();
}
```

where *sumerror* is originally declared as a global variable. Its type is then declared as shared. In the *PROCESS0*, it is replaced with a local variable, i.e. *SCALAR0*. The variable *SCALAR0* is a system variable (reserved word) which is initially set to *sumerror*. There can also be more unique *SCALARs* such as *SCALAR1* and *SCALAR2*, when more reduction variables are found. The routines *m_lock* and *m_unlock* ensure that the shared lock variable *sumerror* does the addition in each processor one at a time.

The second type of translation is that if the reduction scalar variable is originally declared as local in the procedure where it is used. The following example shows how the reduction scalar variable, *sumerror* declared as local, is transformed.

```
  ...
shared float PSCALAR0;
   ...
void training()
{  ...
   void PROCESS0();
   float sumerror;
      ...
   m_fork(PROCESS0,sumerror);
   sumerror = PSCALAR0;
      ...
}

void PROCESS0(sumerror)
float sumerror;
{ int NPROCS,I;
   PSCALAR0 = 0.;
   NPROCS = m_get_numprocs();
   for (I = m_get_myid(); I<n1; I+=NPROCS)
     sumerror += error[I];
   m_lock();
     PSCALAR0 += sumerror;
   m_unlock();
}
```

In this case, the reduction variable, *sumerror* is an argument to the function *m_fork* which passes its initial value to *PROCESS0*. The system variable, i.e. *PSCALAR0*, declared as shared, is used in handling the data dependencies. There can be more unique *PSCALARs*, i.e. *PSCALAR1* and *PSCALAR2*, when more reduction variables are found in the loop iteration.

Transforming a Locked variable

If a scalar exists in *group1* or *group3*, then this variable is a locked variable. As an example, **figure 5.7** contains two locked scalar variables, *ROW* and *SCALAR0*. Variable *SCALAR0* is in *group1* since it is read in the 'if condition' then written within it. Variable *ROW* is in *group3* since it is written in the loop iteration. The loop iteration to be executed in parallel in this case is the 'while loop'.

Figure 5.9 shows the transformation code of the 'while loop' of **figure 5.7**. The shared locked variables, *SCALAR0* and *ROW*, are declared as local by NEUCOMP2 when the program is translated into the sequential version. In order to overcome the data dependencies for both variables, they need to be declared globally as shared. Alternatively NEUCOMP2 replaces the global variables declared with shared variables namely *PSCALAR0* and *PROW*. They then take initial values from these local variables via parameter passing. The final results of these shared variables are then assigned to their respective local variables. The parallel loop from this example is different from the 'for loop' discussed earlier. This parallel loop follows a dynamic scheduling technique [Osterhaug (1989)] specifically generated when NEUCOMP2 locates the 'while loop'. This loop is only applied to an assignment statement that uses the symbol '>' or '<' (**section 5.4.2.1**). The function *m_next* belongs to DYNIX library function, the increment global counter which is automatically set to one when first called. The second call returns to two, and so on.

```
  ...
shared float PSCALAR0;
shared int PROW;
  ...
void training()
{ float SCALAR0;
  int ROW, I;
  I = 0;
  SCALAR0 = Mul_mat_vec(weight,layer1,n1,I);
  ROW = 0;
  m_fork(PROCESS0,ROW,SCALAR0);
  ROW = PROW;
  SCALAR0 = PSCALAR0;
     ...
}

void PROCESS0(ROW,SCALAR0)
float SCALAR0;
int ROW;
{
  int I,J,K;
  PSCALAR0 = SCALAR0;
  PROW = ROW;
  while ( (K = m_next()) < n2) {
    J = K + 1;
    for (I = K; I<J; I++) {
     layer2[I] = Mul_mat_vec(weight,layer1,n1,I);
     m_lock();
     if ( layer2[I] > PSCALAR0) {
       PSCALAR0 = layer2[I];
       PROW = I;
     } m_unlock();
    }
  }
}
```

Fig. 5.9: The transformation code for the 'while loop'

Synchronisation points

Synchronisation needs to be introduced when parallel results from one execution is required by the next operation otherwise an incorrect result will occur. For example, **figure 5.8** requires *m_sync* to be included between the loop iterations as shown below :-

```
    ...
void training()
void PROCESS0()
    ...
{   ...
  m_fork(PROCESS0,ROW);
    ...
}

void PROCESS0(ROW)
int ROW;
{ int I,NPROCS;
  NPROCS = m_get_numprocs();
  for (I= m_get_myid(); I <n1; I += NPROCS )
    layer1[I] = pattern[ROW][I];
  m_sync();
  for (I =  m_get_myid(); I <n2; I += NPROCS )
    layer2[I] = SIGMOID(Mul_mat_vec(weight1,layer1,n1,I) + bias2);
  m_sync();
  for (I =  m_get_myid(); I <n3; I += NPROCS )
    layer3[I] = SIGMOID(Mul_mat_vec(weight2,layer2,n2,I) + bias3);
}
```

where the first *m_sync* is introduced because *layer1* which is being written from the first parallel execution will be read by the next parallel execution. The final *m_sync* is not needed because at the end of the routine, synchronisation is done automatically.

5.5 EXPERIMENTAL RESULTS

Experiments similar to those in Sanossian (1992), were carried out to study the performance of a parallel NN simulation program generated by NEUCOMP2 and those produced by the Neural Network Simulator (NNS). NNS was designed specifically for the backpropagation network. The results of the two programs were then compared.

NNS is an interactive NN simulation developed by Sanossian (1992) using Parallel Pascal running on the Balance machine at PARC. Its data structure is a linked list of a one-dimensional array. A number is assigned to each node in the network. Each node has a linked list that holds all the node numbers connected to it and the connection weights. A one-dimensional array is used for the state of the nodes. Parallelism on the NNS was implemented using two methods i.e., the 'On-line' and 'Batch' methods. In the 'On-line' method, starting from the first layer, the network is partitioned according to the number of nodes onto each processor. The weights were updated for every training pattern. In the 'Batch' method, all input patterns are divided equally among processors. The weights were updated after all training patterns have been processed.

For the simulation program generated by NEUCOMP2, parallelising the loop on every matrix-vector statement is considered as the 'On-line' method. This is because the elements of the matrix/vector variables are partitioned among the processors. The 'Batch' method for NEUCOMP2 does not implement parallelism on the training patterns but in the loops of the matrix/vector statements.

In measuring the performance, the execution time is taken as the difference between the time at the beginning of calling the training procedure and the time at the completion of the procedure. The speedup is measured as:-

$$\text{speedup} = \frac{time_1}{time_p}$$

where $time_1$ is the execution time for one processor and $time_p$ is the execution time for p processors.

There are two sets of experiments. The first set was done using the 'On-line' method and the second one using the 'Batch' method. Both sets of experiments were run for 10 iterations. The effect of increasing the number of nodes in a network or the number of training pattern on the speedup of the parallel simulation program was tested.

5.5.1 The On-line results

The results for the 'On-line' method generated by NEUCOMP2 were compared with NNS. These are shown in **tables 5.1, 5.2** and **5.3**. The execution times were measured for different numbers of processors and different sizes of network (i.e. 5x5x5, 10x10x10, 40x40x40) with fixed training patterns (i.e. 50). The training patterns contain a set of input and target patterns or vector pairs.

Graphs of speedup vs. number of processors for both the NNS and NEUCOMP2 (**figures 5.10, 5.11** and **5.12**) were plotted after each table to show graphically the different speedups. Both programs showed a linearly increase of speedup as the number of processors increases. It also showed that the parallel program generated by NEUCOMP2 is slightly better. This difference is probably due to the way the programs were implemented. The NEUCOMP2 program was implemented using an array while the NNS was implemented using a one-dimensional array of a linked-list. An array data structure has the advantage of getting the value by referring its subscript, but to get the value from an array of lists, a pointer is used to travel along the linked-list until the address is reached.

	NNS		NEUCOMP2	
Number of Processors	Execution time (sec.)	Speedup	Execution time (sec.)	Speedup
1	14.4	1.00	13.3	1.00
2	9.23	1.56	8.47	1.58
3	7.75	1.85	6.17	2.16
4	7.28	1.97	6.13	2.17
5	6.05	2.37	4.03	3.30
6	6.22	2.31	4.05	3.28
7	6.76	2.12	4.10	3.24
8	7.07	2.03	4.20	3.17
9	7.40	1.94	4.32	3.08
10	7.12	2.02	4.44	3.00

Table 5.1: The execution times and speedups of a network of 5x5x5 nodes using the 'On-line' method produced by NNS and NEUCOMP2

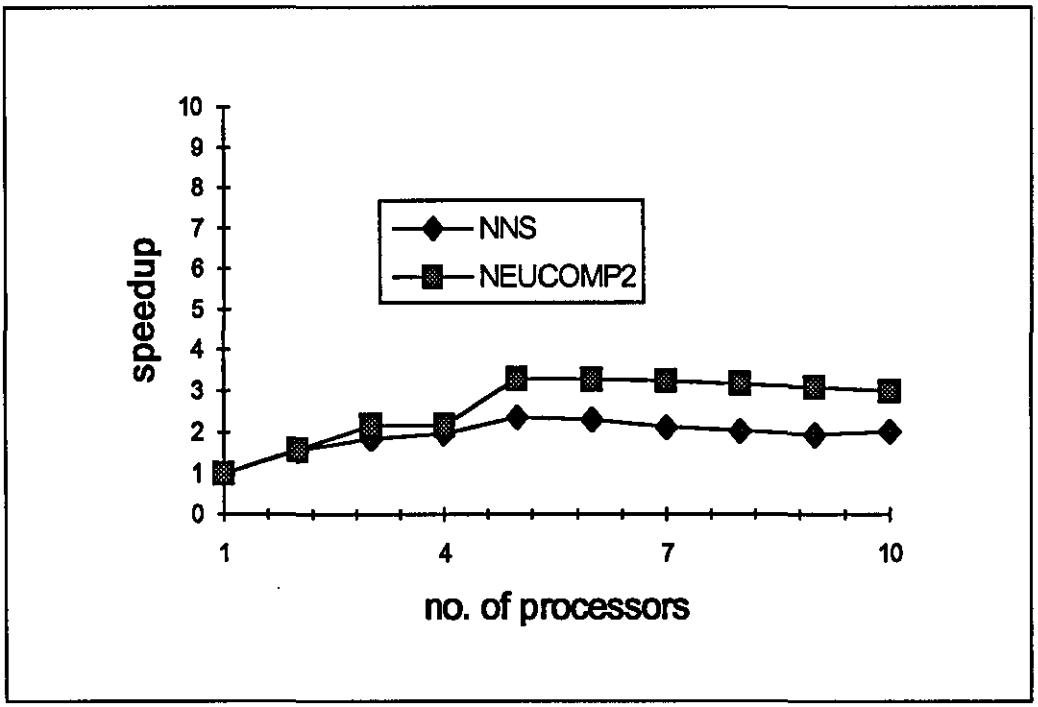


Fig. 5.10: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2

	NNS		NEUCOMP2	
Number of Processors	Execution time (sec.)	Speedup	Execution time (sec.)	Speedup
1	44.5	1.00	43.5	1.00
2	24.3	1.83	22.7	1.92
3	18.7	2.38	18.3	2.38
4	15.1	2.95	14.4	3.03
5	12.9	3.44	10.4	4.18
6	13.0	3.43	10.2	4.27
7	11.9	3.73	10.5	4.14
8	12.2	3.66	10.2	4.27
9	12.4	3.60	10.2	4.27
10	10.5	4.24	6.61	6.58

Table 5.2: The execution times and speedups of a network of 10x10x10 nodes using the 'On-line' method produced by NNS and NEUCOMP2

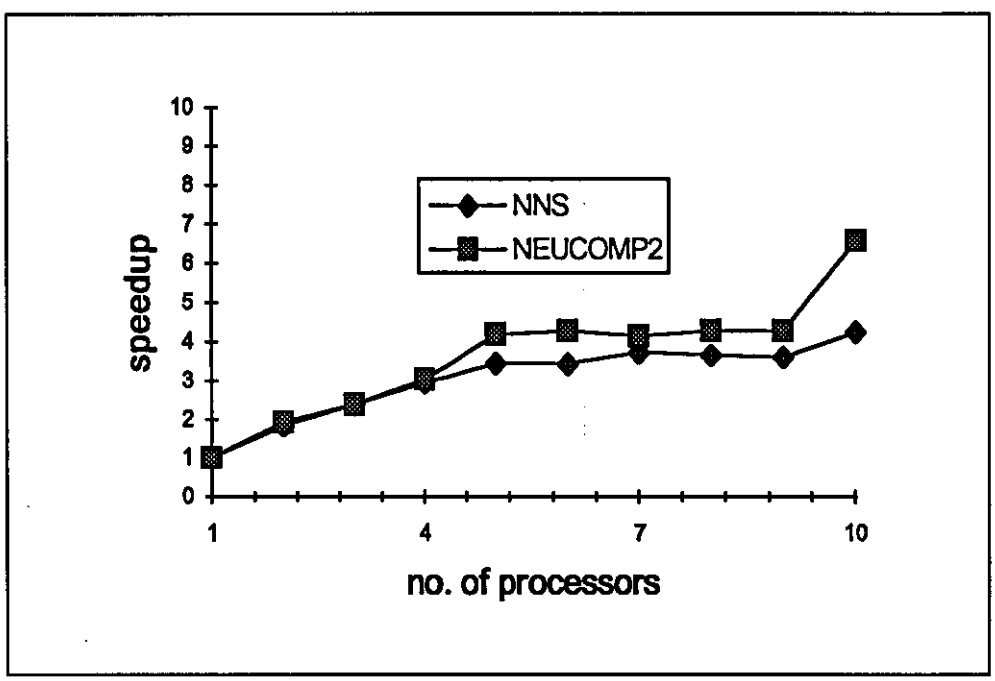


Fig. 5.11: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2

	NNS		NEUCOMP2	
Number of Processors	Execution time x 10^1 s	Speedup	Execution time x 10^1 s	Speedup
1	59.1	1.00	58.7	1.00
2	31.2	1.89	30.0	1.96
3	21.2	2.79	20.8	2.82
4	15.7	3.78	15.0	3.91
5	12.8	4.62	12.2	4.81
6	11.1	5.31	10.6	5.54
7	9.62	6.15	9.22	6.37
8	8.18	7.23	7.73	7.59
9	7.73	7.65	7.58	7.74
10	6.79	8.70	6.39	9.19

Table 5.3: The execution times and speedups of a network of 40x40x40 nodes using the 'On-line' method produced by NNS and NEUCOMP2

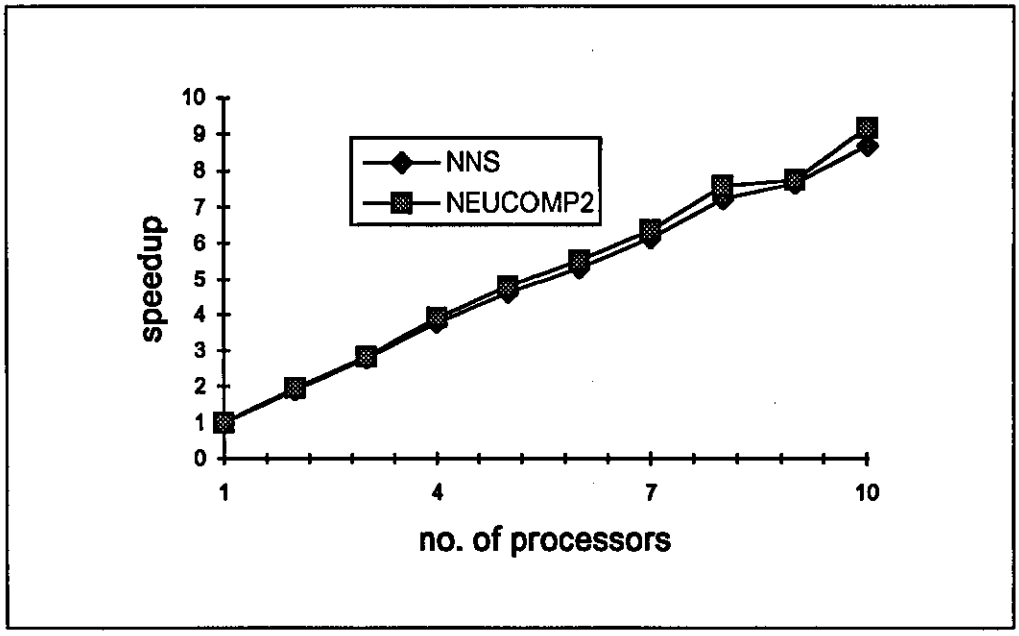


Fig. 5.12: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2

5.5.2 The Batch results

There were two sets of experiments in the 'Batch' method generated by NEUCOMP2. The first experiment (experiment1) was implemented using parallelism amongst the training patterns and the second experiment (experiment2) was implemented using parallelism on all the loop iterations that involve the matrix/vector operations. Results for experiment1 were compared with the NNS also using the 'Batch' method; **tables 5.4** to **5.8** show the comparison. In **tables 5.4** and **5.6**, the execution times were measured for different number of processors and a network of different number of nodes, i.e. 5x5x5, 20x20x20 and 40x40x40 nodes, with fixed vector pairs i.e. 50. In **tables 5.7** and **5.8**, the execution times were measured for different numbers of processors as well as different numbers of vector pairs, i.e. 80 and 100, with a fixed size of network (i.e. 10x10x10 nodes). There are also comparisons made between experiment1 and experiment2 of the execution times and speedups obtained. These are shown in **tables 5.9** to **5.11**. **Table 5.9** contains the differences of execution times and speedups for a network of size 40x40x40 nodes with 50 vector pairs. **Tables 5.10** and **5.11** contain the differences of execution time and speedup for a network of size 10x10x10 nodes and different sizes of vector pairs, i.e. 80 and 100.

Graphs of comparison of speedup vs. number of processors for experiment1 and NNS are shown in **figures 5.13** to **5.17**. There is a great difference in terms of the speedup for the NEUCOMP2 simulation that execute the training patterns in parallel as compared to the NNS that use the 'Batch' method. This is because in the program of experiment1, the only loop that executed in parallel was amongst the training patterns whereas within this loop, there exists many loop iterations that operate on the matrix/vector operations. Such loops are the vector operations for calculating the activation function of the hidden layer and the output layer, calculating the sum of

errors for the output nodes and the hidden nodes, matrix operations on the weight derivatives, etc. This factor affects the execution time of the processors. However, experiment2 gave better results when all the loop iterations within the training patterns were executed in parallel.

Graphs of the comparison between experiment1 and experiment2 are shown in **figures 5.18** to **5.20**. This is done to show graphically, that executing the loop iterations on the matrix/vector operation proved to produce faster execution times. Another advantage is that the execution time for the 'Batch' method that execute the loop iteration in parallel generated by NEUCOMP2 are two times faster than the NNS that were executed in the 'On-line' method. These differences are graphically shown in **figures 5.21** and **5.22**.

The graph of speedup vs. number of processors did not show a linear increase due to load imbalance where some of processors are still busy while others remain idle.

	NNS		NEUCOMP2 (experiment1)	
Number of Processors	Execution time x 10^0s	Speedup	Execution time x 10^0 s	Speedup
1	9.19	1.00	7.38	1.00
2	5.00	1.84	3.91	1.89
3	3.82	2.41	3.25	2.27
4	3.20	2.87	3.06	2.41
5	2.67	3.44	2.79	2.65
6	2.66	3.46	2.76	2.67
7	2.61	3.52	2.77	2.66
8	2.64	3.48	2.86	2.58
9	2.61	3.52	2.87	2.57
10	2.37	3.88	2.91	2.54

Table 5.4: The execution times and speedups of a network of 5x5x5 nodes using the 'Batch' method produced by NNS and NEUCOMP2

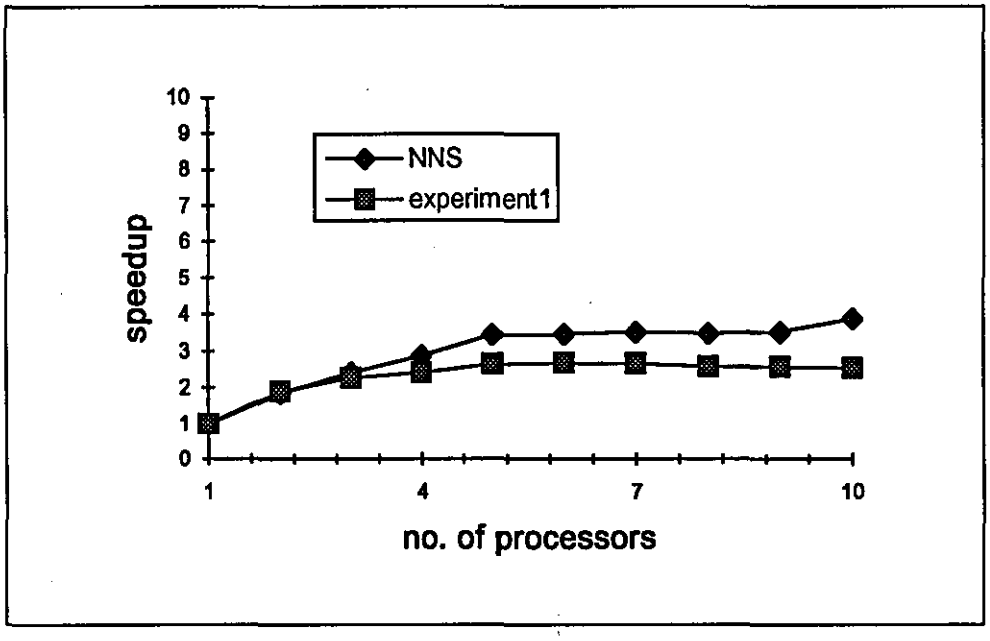


Fig. 5.13: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2 using the 'Batch' method

	NNS		NEUCOMP2 (experiment1)	
Number of Processors	Execution time x 10^1s	Speedup	Execution time x 10^1 s	Speedup
1	9.62	1.00	7.38	1.00
2	4.84	1.99	4.42	1.67
3	3.37	2.86	3.38	2.18
4	2.54	3.79	3.18	2.32
5	2.01	4.78	3.00	2.46
6	1.83	5.26	2.94	2.51
7	1.62	5.95	2.92	2.53
8	1.44	6.68	2.86	2.58
9	1.33	7.23	2.87	2.56
10	1.14	8.45	2.88	2.57

Table 5.5: The execution times and speedups of a network of 20x20x20 nodes using the 'Batch' method produced by NNS and NEUCOMP2

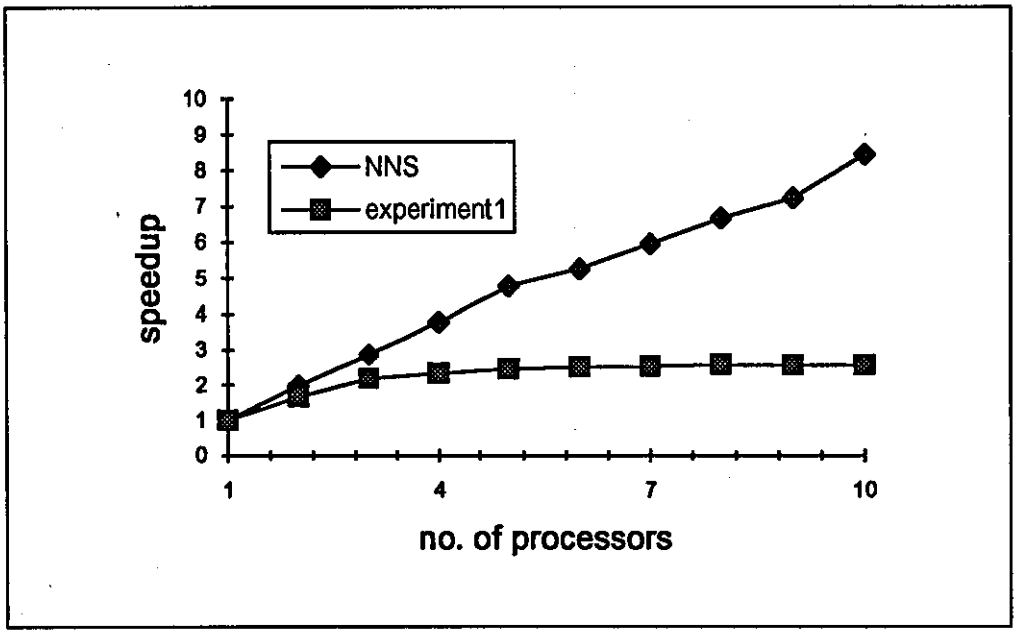


Fig. 5.14: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2 using the 'Batch' method

	NNS		NEUCOMP2 (experiment1)	
Number of Processors	Execution time x 10^1s	Speedup	Execution time x 10^1 s	Speedup
1	35.5	1.00	27.2	1.00
2	18.1	1.97	15.8	1.72
3	12.2	2.90	12.9	2.11
4	9.37	3.79	12.0	2.27
5	7.35	4.83	11.5	2.37
6	6.57	5.40	11.1	2.45
7	5.76	6.16	11.0	2.49
8	5.01	7.08	11.0	2.48
9	4.46	7.96	10.8	2.51
10	3.72	9.53	10.8	2.51

Table 5.6: The execution times and speedups of a network of 40x40x40 nodes using the 'Batch' method produced by NNS and NEUCOMP2

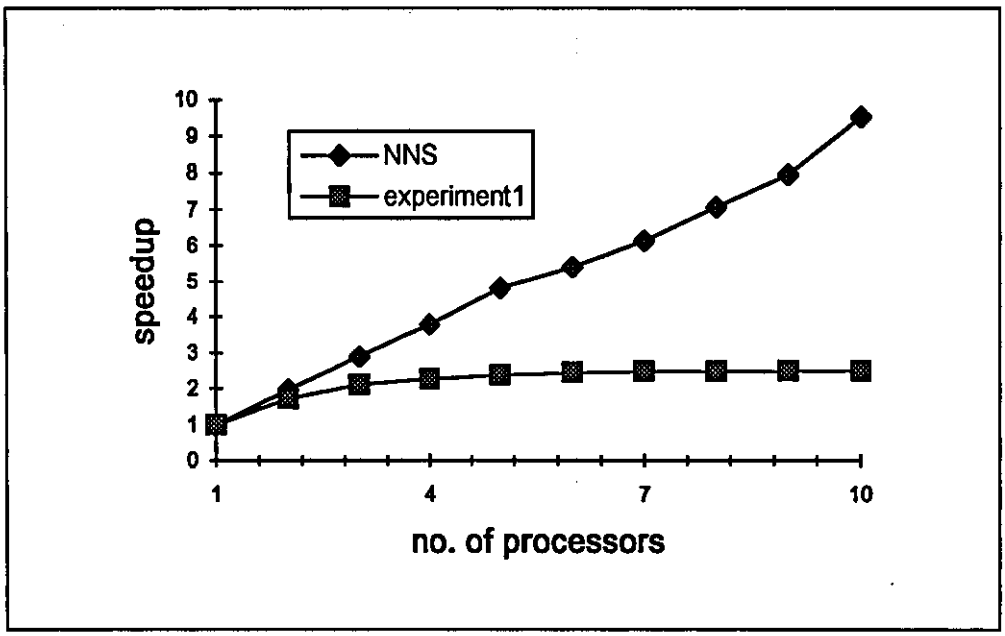


Fig. 5.15: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2 using the 'Batch' method

	NNS		NEUCOMP2 (experiment1)	
Number of Processors	Execution time x 10^0s	Speedup	Execution time x 10^0 s	Speedup
1	44.7	1.00	34.5	1.00
2	22.7	1.97	18.3	1.89
3	16.2	2.77	14.6	2.37
4	11.9	3.75	13.5	2.56
5	9.87	4.53	13.0	2.66
6	9.23	4.85	12.5	2.76
7	7.88	5.68	12.6	2.74
8	7.28	6.14	12.5	2.76
9	7.04	6.35	12.5	2.76
10	6.07	7.37	12.5	2.76

Table 5.7: The execution times and speedups of a network trained on 80 vector pairs using the 'Batch' method produced by NNS and NEUCOMP2

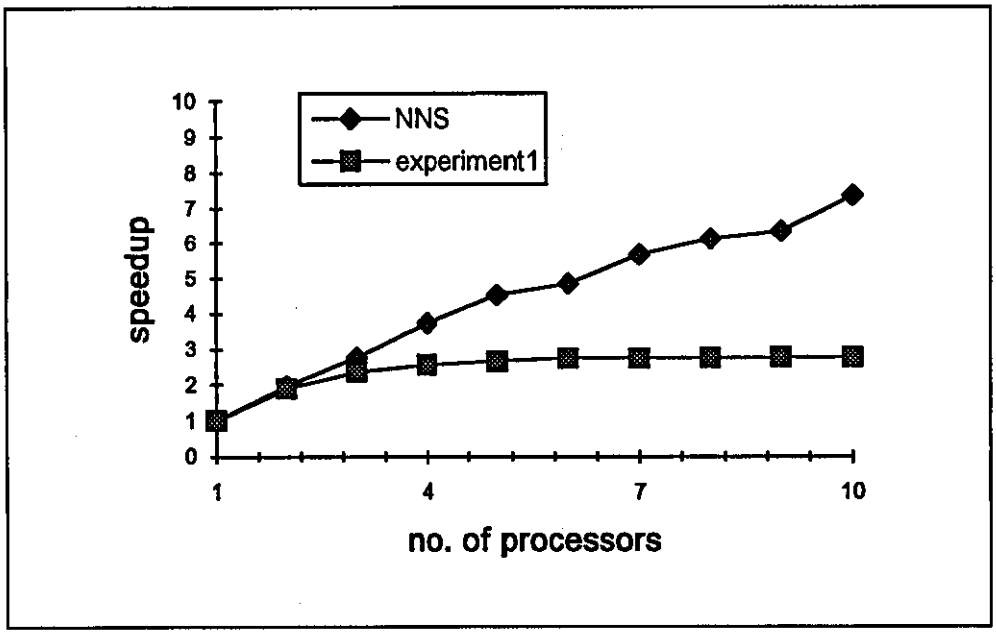


Fig. 5.16: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2 using the 'Batch' method

	NNS		NEUCOMP2 (experiment1)	
Number of Processors	Execution time x 10^0s	Speedup	Execution time x 10^0 s	Speedup
1	55.9	1.00	43.2	1.00
2	28.5	1.96	22.8	1.88
3	19.8	2.82	18.1	2.37
4	14.8	3.79	16.8	2.57
5	12.2	4.60	15.9	2.70
6	11.0	5.07	15.6	2.76
7	9.57	5.80	15.5	2.78
8	8.95	6.24	15.4	2.80
9	8.65	6.46	15.3	2.82
10	7.25	7.71	15.4	2.80

Table 5.8: The execution times and speedups of a network trained on 100 vector pairs using the 'Batch' method produced by NNS and NEUCOMP2

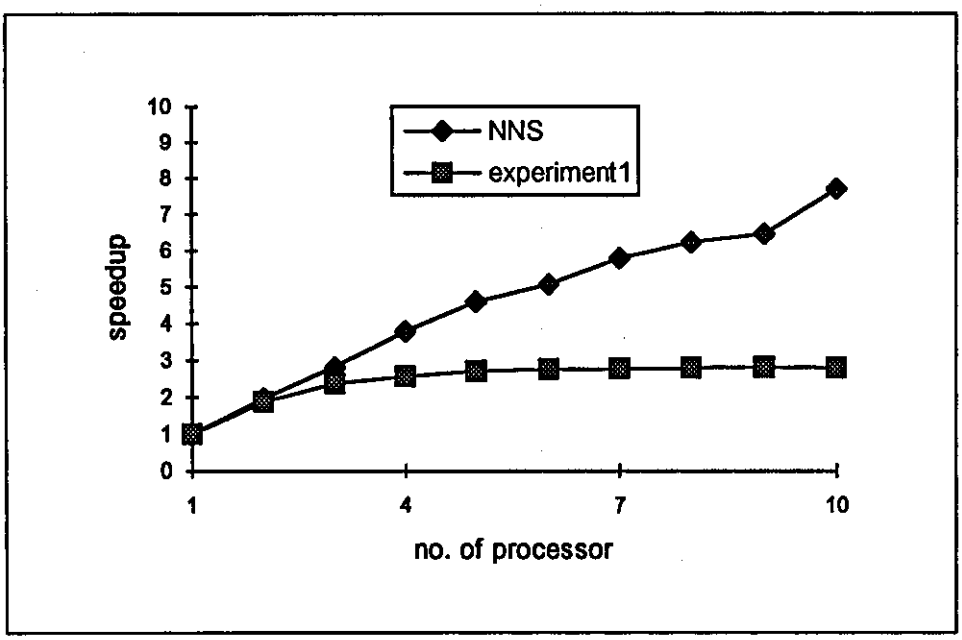


Fig. 5.17: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2 using the 'Batch' method

	NEUCOMP2 (experiment1)		NEUCOMP2 (experiment2)	
Number of Processors	Execution time x 10^1s	Speedup	Execution time x 10^1 s	Speedup
1	27.2	1.00	28.0	1.00
2	15.8	1.72	14.1	1.98
3	12.9	2.11	9.86	2.84
4	12.0	2.27	7.14	3.92
5	11.5	2.37	5.81	4.83
6	11.1	2.45	5.12	5.48
7	11.0	2.49	4.41	6.36
8	11.0	2.48	3.76	7.45
9	10.8	2.51	3.76	7.45
10	10.8	2.51	3.09	9.08

Table 5.9: The execution times and speedups of 40x40x40 nodes using the 'Batch' method on the first and second experiments of the NEUCOMP2 programs

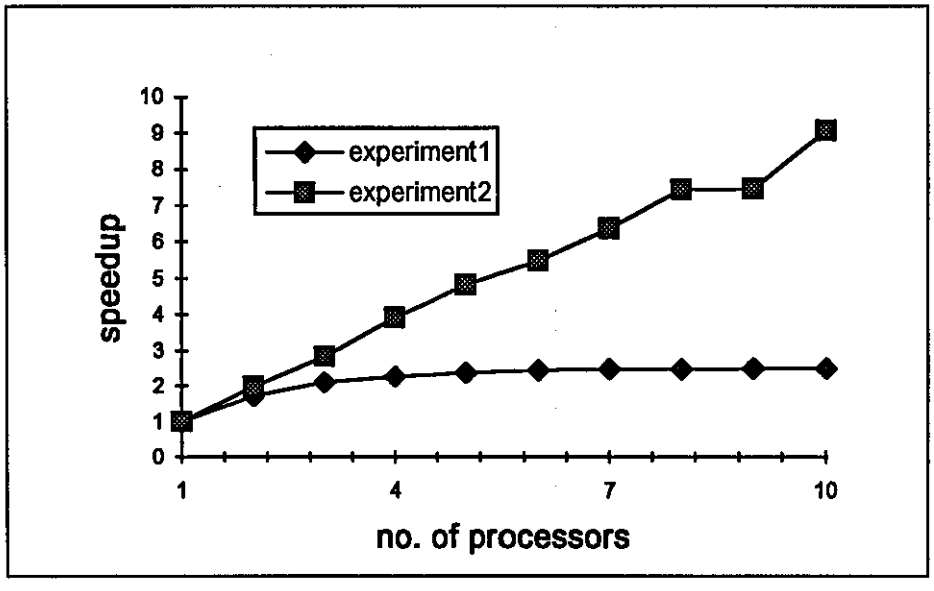


Fig. 5.18: Comparison of speedups vs. no. of processors for both experiment1 and experiment2 using the 'Batch' method

	NEUCOMP2 (experiment1)		NEUCOMP2 (experiment2)	
Number of Processors	Execution time x 10^0s	Speedup	Execution time x 10^0 s	Speedup
1	34.5	1.00	35.6	1.00
2	18.3	1.89	18.6	1.92
3	14.6	2.37	15.3	2.33
4	13.5	2.56	12.0	2.98
5	13.0	2.66	8.51	4.18
6	12.5	2.76	8.76	4.07
7	12.6	2.74	8.75	4.07
8	12.5	2.76	8.74	4.07
9	12.5	2.76	8.60	4.14
10	12.5	2.76	6.04	5.90

Table 5.10: The execution times and speedups of a network trained on 80 vector pairs for the first and second experiments of the NEUCOMP2 programs

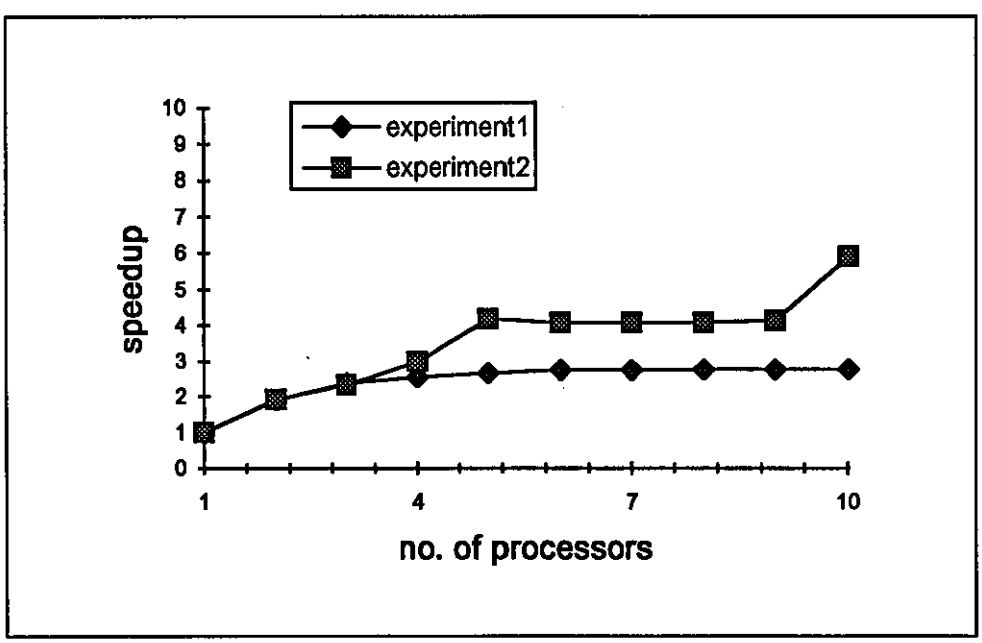


Fig. 5.19: Comparison of speedups vs. no. of processors for both experiment1 and experiment2 using the 'Batch' method

	NEUCOMP2 (experiment1)		NEUCOMP2 (experiment2)	
Number of Processors	Execution time x 10^0s	Speedup	Execution time x 10^0 s	Speedup
1	43.0	1.00	44.4	1.00
2	22.8	1.88	23.1	1.92
3	18.1	2.37	19.2	2.31
4	16.8	2.57	14.7	3.03
5	15.9	2.70	10.7	4.15
6	15.6	2.76	10.7	4.15
7	15.5	2.78	10.6	4.18
8	15.4	2.80	10.7	4.15
9	15.3	2.82	10.6	4.18
10	15.4	2.80	7.10	6.25

Table 5.11: The execution times and speedups of a network trained on 100 vector pairs for the first and second experiment of the NEUCOMP2 programs

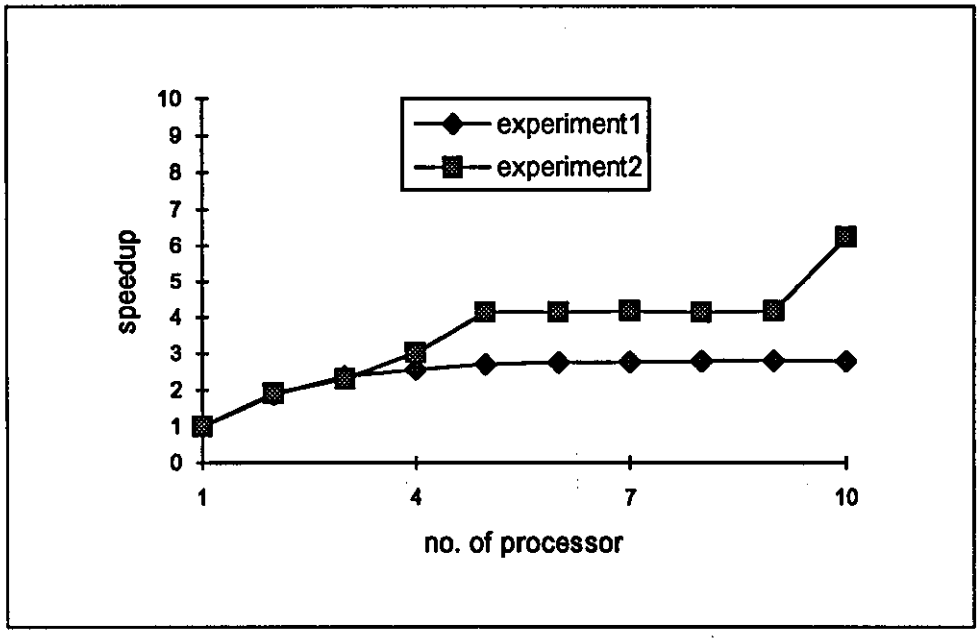


Fig. 5.20: Comparison of speedups vs. no. of processors for both experiment1 and experiment2 using the 'Batch' method

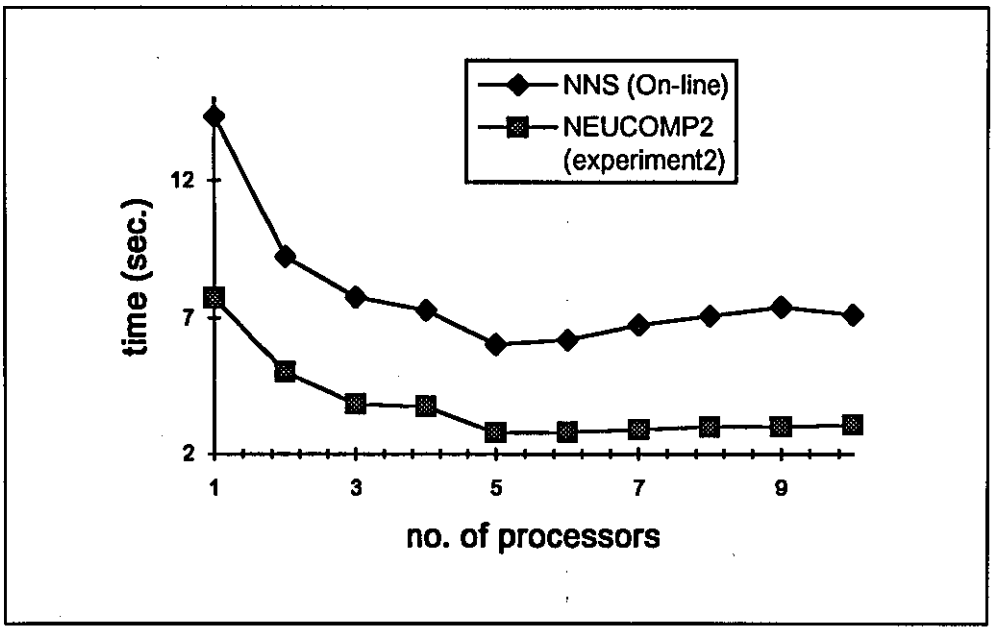


Fig. 5.21: Comparison of execution times vs. no. of processors for 5x5x5 nodes from the NNS using the 'On-line' and the experiment2.

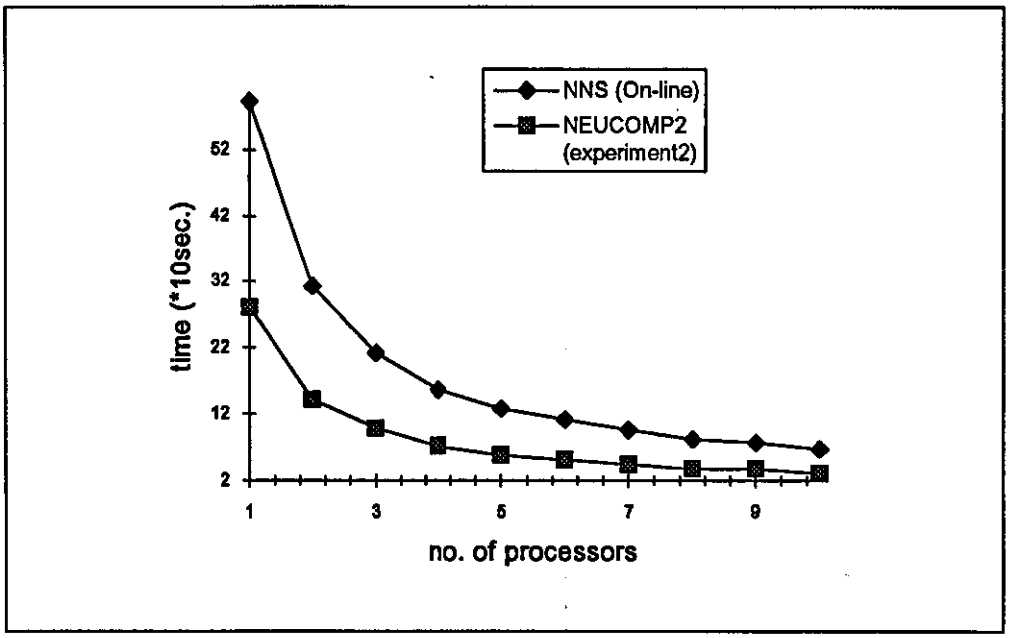


Fig. 5.22: Comparison of execution times vs. no. of processors for 40x40x40 nodes from the NNS using the 'On-line' and the experiment2.

5.6 DISCUSSION

The Parallel Neural Network compiler called NEUCOMP2 was designed for the SEQUENT Balance computer at PARC. The main objective of NEUCOMP2 is to generate a parallel simulation program to be executed on the parallel machine. The work is an extension of the work carried out on NEUCOMP. The only change to the NEUCOMP language from the old version is placing the statement PARALLEL in front of the procedure call in order to run that procedure in parallel. The NEUCOMP2 program is then compiled by NEUCOMP2 to generate the parallel C-code that runs on the parallel machine.

The main characteristics of NEUCOMP2 is the parallelising phase (**figure 5.1**) which can be changed to suit any parallel machine of different architectures, i.e. Transputer network, Intel Hypercube, etc., without changing the whole process of the compilation technique. The parallelising phase that has been implemented so far is for the Shared-Memory parallel machine, i.e. the SEQUENT Balance. The design of the parallelising phase was based on the strategies used in the automatic parallelisation of programs or parallelising compiler.

NEUCOMP2 allows the loop iteration to be executed in parallel on the matrix/vector operations. Therefore the 'On-line' and 'Batch' methods in the backpropagation algorithm are actually parallelising the loops of the program which is suitable for parallelism.

It has been shown that parallelising the loops of the program generated by NEUCOMP2 gives a better performance in terms of execution time and speedup whereas parallelisation by partitioning the training patterns in the NEUCOMP2 did not perform so well. It has also been shown that parallelising the loop iteration using the 'Batch' method generated by NEUCOMP2 is twice faster than the NNS specifically designed for the backpropagation algorithm which uses the one-dimensional array of the linked-list. It has also been shown that the

'On-line' method used by the NEUCOMP2 program gives a slightly better performance than the 'On-line' method of the NNS.

To confirm the correctness of the parallel programs, results were compared and checked satisfactorily with the sequential versions generated by NEUCOMP.

CHAPTER 6

NEURAL NETWORK APPLICATIONS

This chapter discusses the different types of application problems solved using some popular NN models. They are implemented on the parallel computer at PARC using the NEUCOMP/NEUCOMP2 language. These programs are called the NN simulation programs.

There are three types of problems considered to be solvable by NN simulations. These belong to the categories :-

(1) the classification problem
(2) the approximation/classification problem
(3) the optimisation problem

The character recognition problem belongs to the classification category. The spiral problem belongs to the approximation/classification category and the travelling salesman problem (TSP) belongs to the optimisation category.

The chosen network models for the classification problem are the backpropagation (BP), Kohonen, Counterpropagation (CPN) and ART1 networks. In the classification problem, the output nodes for the backpropagation and Counterpropagation networks are arranged in binary form so that when the first node is one and the rest are zeros, it belongs to the first class, and when the second output node is one and the rest are zero it belongs to the second class, and so on. In the Kohonen network the output layer known as map, cluster the patterns into the different classes. The input to the networks are all binary numbers.

The chosen models for the approximation/ classification problem are the backpropagation, Kohonen and Counterpropagation networks. Solving the spiral problem using backpropagation is considered as an approximation, since the output node is one and its content is the Sigmoid function which lies between 0 and 1. When the output value approximates to 1, the spiral belongs to the first class and when the output value

approximates to 0, it is the second type of spiral. The spiral problem using the Kohonen and Counterpropagation networks are said to be a classification problem since both networks cluster the input data into different classes of spirals. The input to and output from the networks are real numbers.

The chosen models for the optimisation problem are the Hopfield-type networks. The Hopfield-type networks considered are the Hopfield-Tank and Potts-Glass models. The input to and output from the networks are again real numbers.

The simulation results for the above problems are shown graphically using the 'Mathematica' programs which are included as NEUCOMP/NEUCOMP2 library functions. The parallel simulation programs are also implemented using NEUCOMP2 for all the problems. Their execution times and speedups are then compared.

6.1 CHARACTER RECOGNITION PROBLEM

A template-matching technique for the identification of the alphabetic characters, A ... Z, has been successful by running the simulation programs on the chosen NN models. The respective models are the backpropagation, Counterpropagation, Kohonen and ART1 networks. The backpropagation and Counterpropagation networks are trained using supervised learning. The Kohonen and ART1 networks are trained using unsupervised learning. The results of the simulation are shown as the time in seconds taken for training and the output of the successfully recognised input patterns.

6.1.1 Simulation Programs for Character Recognition

The simulation programs using the NEUCOMP language for recognising the character set, A ... Z, are implemented on the four respective NN models - the backpropagation, Kohonen, ART1 and Counterpropagation networks.

The input layer consists of 100 nodes and the output layer is organised depending on the type of model used. For the backpropagation, Counterpropagation and ART1 networks, the output layer is defined as 26 nodes. However, the output layer for the backpropagation and Counterpropagation networks corresponds to one of the characters, i.e., for the character A, the first node is one and the rest are zero, and so on. The output node for ART1 represents the winner node when the respected input character is assigned to the network. This winner node combines with the feedback connection for the output result. For the Kohonen network, the output is in the form of a map of a two-dimensional array to display regions controlled by each character.

The simulation programs for the backpropagation, Kohonen, ART1 and Counterpropagation networks were presented in **sections 4.4.1** to **4.4.4**. A three layer network is used for the backpropagation network and its

hidden layer is set to 20 nodes. For the Kohonen network, the size of the map is 30x30 while for the Counterpropagation network, the size of the competitive layer is set to 100.

In the backpropagation network, the weights are updated after presenting all the vector pairs (i.e., set of input and target patterns). This is known as the 'Batch-method'. The weights for the remaining NN models such as the Kohonen, Counterpropagation and ART1 networks, are adjusted for every training pattern.

6.1.2 Experimental Description

Experiments have been carried out to train the 26 characters, A .. Z, using the respective networks. Every character is written as an array of 10x10 binary numbers to represent an image of that character and are kept in an input file. To test the trained networks, two sets of data are presented. The first set contains the original data and the second set contains the noisy image.

An example of the original and noisy characters for A is shown in **figure 6.1** where a black box represents "1" and a white box represents "0".

6.1.2.1 Simulation results for the backpropagation network

The connection weights and biases for all the nodes in the hidden layer and the output layer are initially set randomly in the interval (-1.0,1.0). The weights and the biases are assigned with different seeds used for starting the random number generator, which are 10, 100, 1000, and 5000 respectively. The *beta* value is set to 0.5. When using the GRBH method (**section 4.4.1**), the values of *alpha* are 10, 0.5, 0.1 respectively according to the *range* set to $1.0x10^{-4}$ and $1.0x10^{-2}$. The training cycle is terminated when *enormsqr*<0.1 .

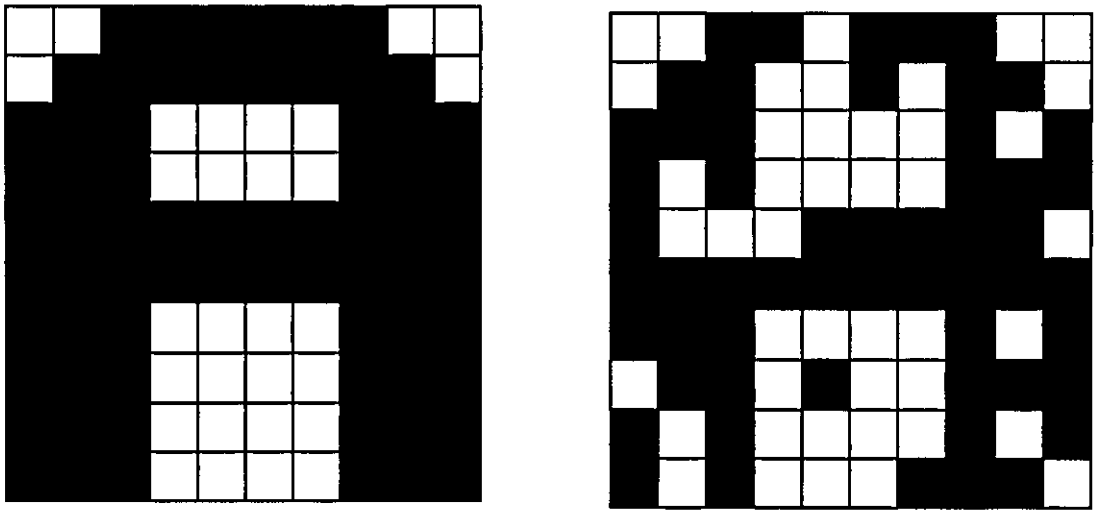


Fig. 6.1: The original and noisy characters for A

The number of iterations required to train the 26 characters is 388 which took 4.05×10^3 seconds. The output nodes hold the following results when all the characters were recalled.

0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.98 for character A

0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.95 0.01 for character B

....

0.99 0.00 0.00 0.00 0.01 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.03 0.00 0.00 0.00 0.00 0.00 for character Z

An example of a noisy character for A when presented to the network, shows the output nodes to have the following values :-

0.00 0.00 0.00 0.00 0.01 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.16 0.00 0.00 0.01
0.00 0.22 0.00 0.03 0.00 0.00 0.00 0.00 0.38

The result shows that the output node on the right holds the maximum value for a noisy character A.

6.1.2.2 Simulation results for the Kohonen network

The initial learning rate, neighbourhood size and random number seed were set to 0.5, 15 and 1000 respectively. The random weights were set between 0 and 1. After 60 cycles, all 26 characters were well distributed in the map with size 30x30. The result is shown in **figure 6.2** where the "o"s represent inactive nodes and each displayed character represents the winner node for that respective character.

The results shows that characters which are not closely related are set well apart on the map, e.g. character S, J, and Z but characters that are closely related are near to each other, e.g. G, O, D. The time taken after 60 cycles is 16.8×10^3 seconds.

When the original characters are presented one at a time to the trained network, the winner node is marked with "*", showing that the character has been successfully recognised and when a noisy character is presented the winner node also marked with "*" appeared very close to its original character. An example of a noisy character for A after being presented to the network is shown in **figure 6.3**.

6.1.2.3 Simulation results for the ART1 network

This network does not require any iterations when it is trained since its feedforward and feedback weights are not updated. The feedforward weight is adjusted using the formula given earlier (**section 4.4.3**) when a new character is presented to the network and the feedback weight is adjusted with binary values equivalent to the input character. These binary values will be produced as output when the winner node is identified for that pattern.

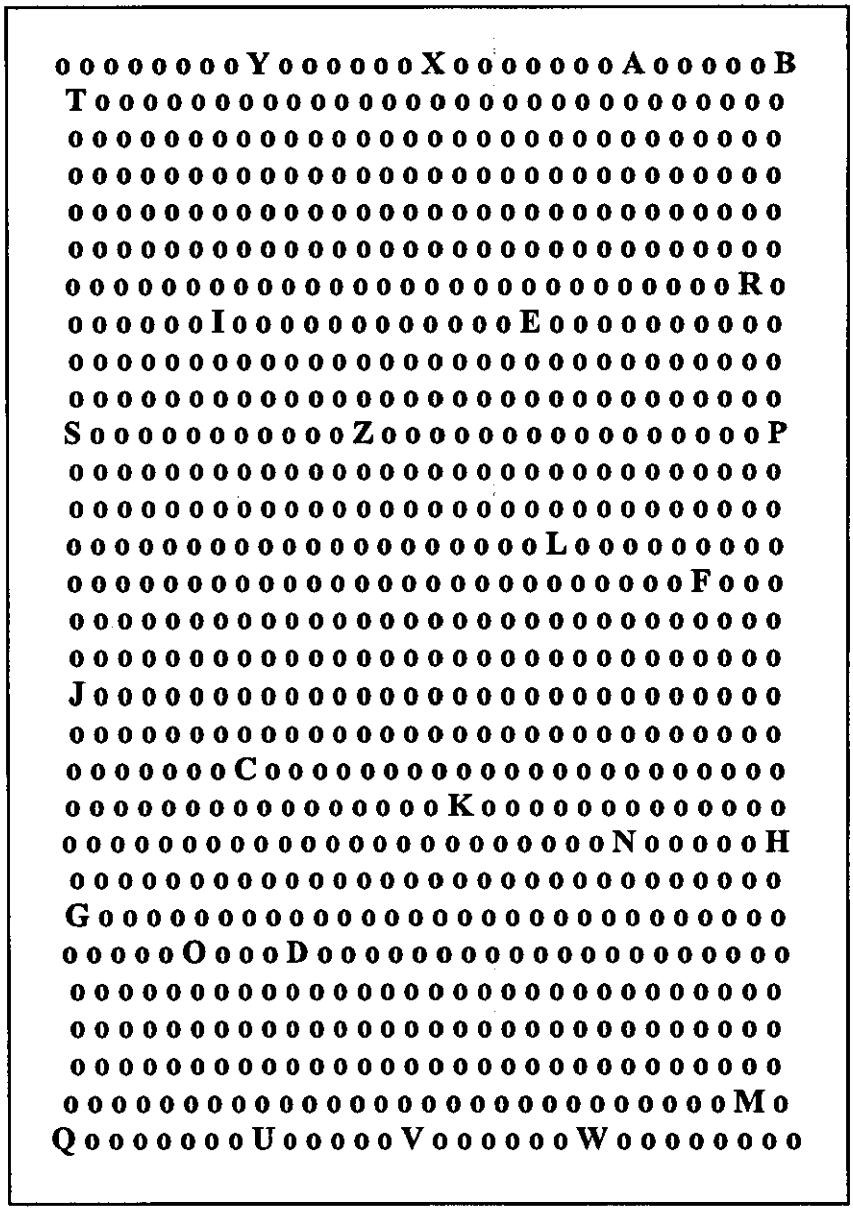


Fig. 6.2: Final map of characters after 60 iterations

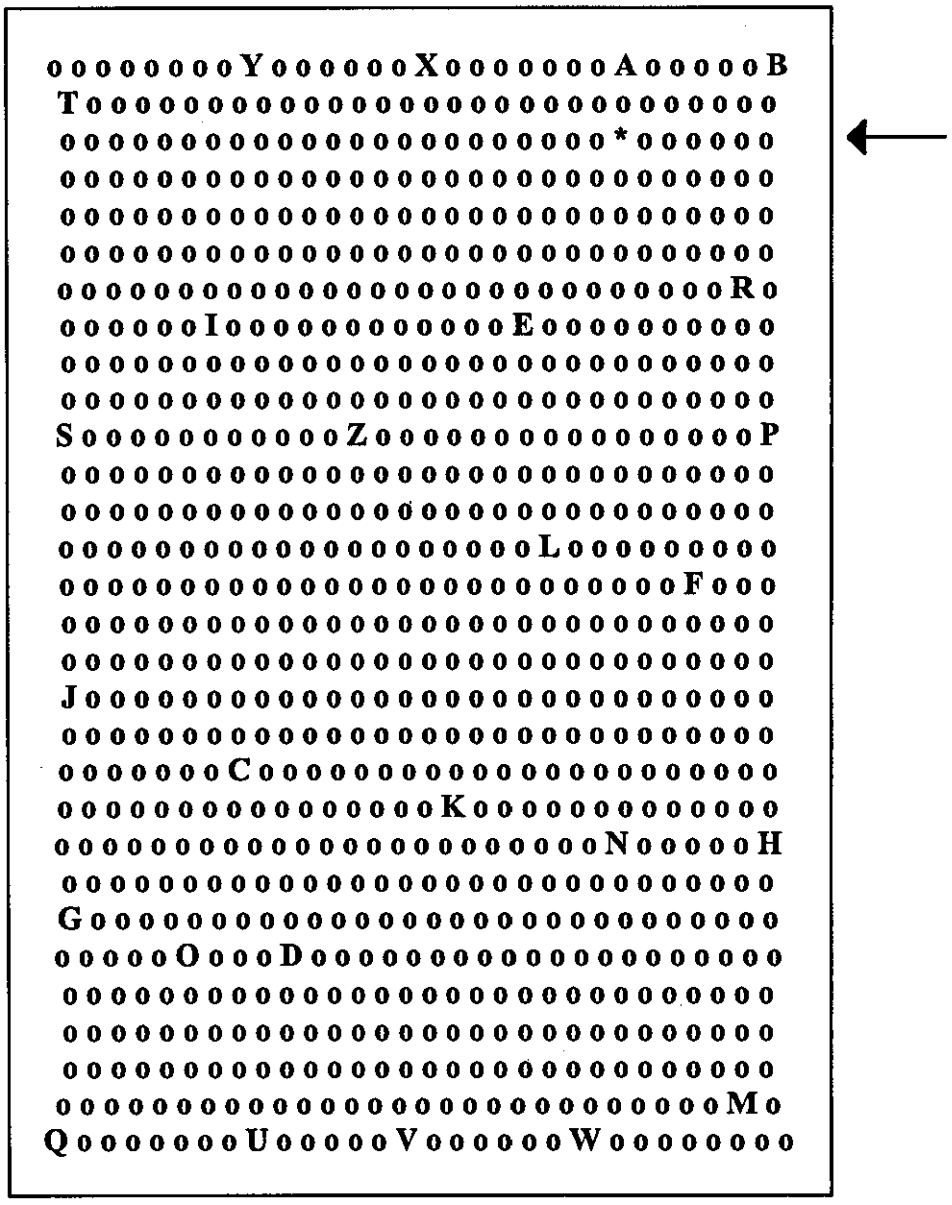


Fig. 6.3: The winning node for noisy A is marked with '*' close to A as indicated

The time taken to train all 26 characters is 22 seconds. All the characters were successfully recalled. The noisy characters were also correctly identified. The output characters are displayed similar to their inputs. As an example, for the original and noisy character for A, the output is shown in **figure 6.4** as pattern A.

6.1.2.4 Simulation results for the Counterpropagation network

Since there are two different layers to be trained on this network, both weights are updated at the same time per cycle. The number of iterations is 60. This number of cycles is required to allow the 26 characters to self-organise in the competitive layer. At the same time, the weights between the winner node and the output layer are then updated according to the target output presented to this layer.

The initial learning rates to update the first and second weights are 0.5 and 1 respectively. The first weights are initially assigned with a random number from the interval (0,1) with the seed equal to 1000. The second weights are initially set to zero. Since the output is a binary value, the final value for the second weight is equal to the target value.

The time taken for 60 training cycles is 2.05×10^3 seconds. All the characters were successfully recalled. The noisy characters, were also correctly identified. The output nodes hold the following binary values when the original and noisy characters were presented.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 represents A
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 represents B
....
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 represents Z

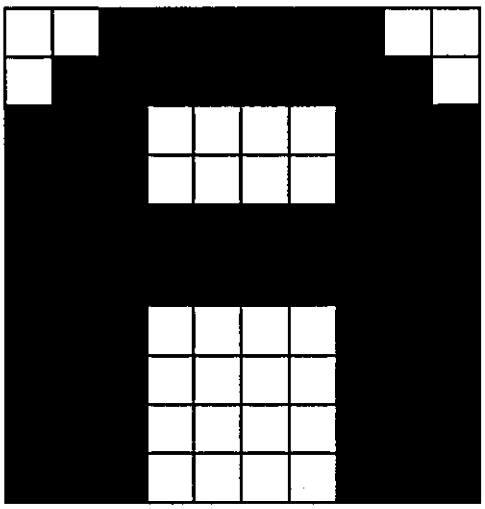


Fig. 6.4: The output of pattern A

6.1.2.5 Parallel Simulation results

To generate a parallel program, the NEUCOMP programs for the four NN models studied previously were added with the parallel statements stated earlier. These were then compiled and executed using NEUCOMP2. The performance of the parallel execution ranging from one processor to many processors were measured. The speedup are calculated based in **section 5.5.**

The performance results for the backpropagation network are shown in **table 6.1.** The performance results for the Kohonen network are shown in **table 6.2.** The performance results for the Counterpropagation network are shown in **table 6.3.** The performance results for the ART1 network are shown in **table 6.4.**

The graphs of speedup versus number of processors for all the above results are shown in **figure 6.5.** The graph of the execution times for the results are shown in **figure 6.6.**

6.1.3 Discussion of the results

For the first simulation in the character recognition problem, the ART1 network performed the fastest learning

time which is approximately 22 seconds. This is because in the ART1, training the network is not through updating the weights for every iteration. Instead its feedforward weights are adjusted by a formula so that when a similar pattern is presented, these weights guarantee to give the maximum value. The node that is connected to those weights is the winner node. The Counterpropagation network took about $2.05x10^3$ seconds for 60 cycles, the backpropagation network took about $4.05x10^3$ for 388 cycles and the Kohonen network took $16.8x10^3$ seconds for 60 cycles. The Kohonen network training took the longest because with the size of its competitive layer, i.e. 30x30, updating the weights of the neighbourhood affect the execution time.

A thorough comparison amongst the networks in response to different levels of noise was not carried out because the purpose of the experiment was to prove that the NEUCOMP language is capable of developing a simulation program for any NN model.

When all the network were executed in parallel, the execution times became less as the number of processors increased. They are shown in **figure 6.6.** The graphs show that the execution time for the Kohonen network is the slowest and the execution time for the ART1 network is the fastest. The speedup for the Kohonen network almost reached the ideal speedup whereas for the ART1, the speedup indicates poor performance. For the backpropagation network, load imbalancing among the processors occurred. The reason that can be deduced is, the Kohonen network is a two layer network of size 100x(30x30). Thus partitioning the nodes among the processors were evenly distributed. This could not happen to the backpropagation network as the nodes are varying in size, i.e. 100x20x26. The ART1 network has a small size, i.e. 100x26 and the size of the Counterpropagation network is 100x100x26.

No. of Processors	Execution time x 10^3sec.	Speedup
1	4.06	1.00
2	2.05	1.98
3	1.44	2.82
4	1.05	3.87
5	0.88	4.61
6	0.84	4.83
7	0.66	6.15
8	0.65	6.25
9	0.62	6.55
10	0.47	8.64

Table 6.1: The execution time and speedup for the BP network

No. of Processors	Execution time x 10^3sec.	Speedup
1	16.9	1.00
2	8.67	1.95
3	5.85	2.89
4	4.42	3.82
5	3.57	4.73
6	3.03	5.58
7	2.54	6.65
8	2.22	7.61
9	1.98	8.54
10	1.79	9.44

Table 6.2: The execution time and speedup for the Kohonen network

No. of Processors	Execution time x 10^3sec.	Speedup
1	2.10	1.00
2	1.07	1.96
3	0.73	2.88
4	0.56	3.75
5	0.46	4.57
6	0.40	5.25
7	0.35	6.00
8	0.31	6.77
9	0.28	7.50
10	0.26	8.08

Table 6.3: The execution time and speedup for the CPN

No. of Processors	Execution time x 10^1sec.	Speedup
1	2.21	1.00
2	1.13	1.96
3	0.80	2.76
4	0.63	3.51
5	0.53	4.17
6	0.48	4.60
7	0.43	5.14
8	0.41	5.39
9	0.38	5.82
10	0.36	6.14

Table 6.4: The execution time and speedup for the ART1 network

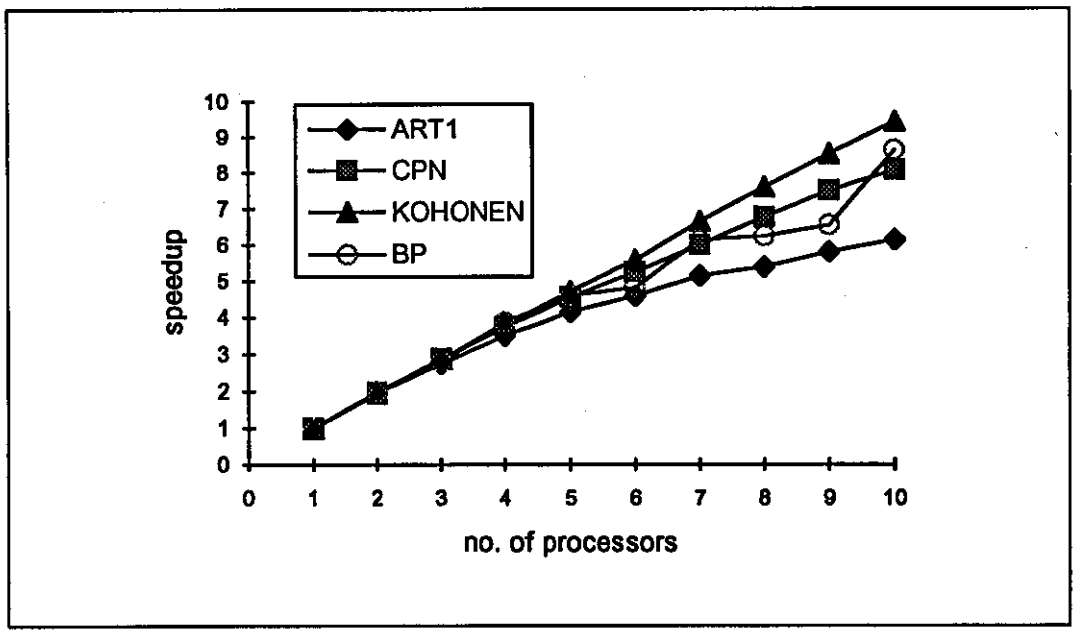


Fig. 6.5: The speedups versus no. of processors

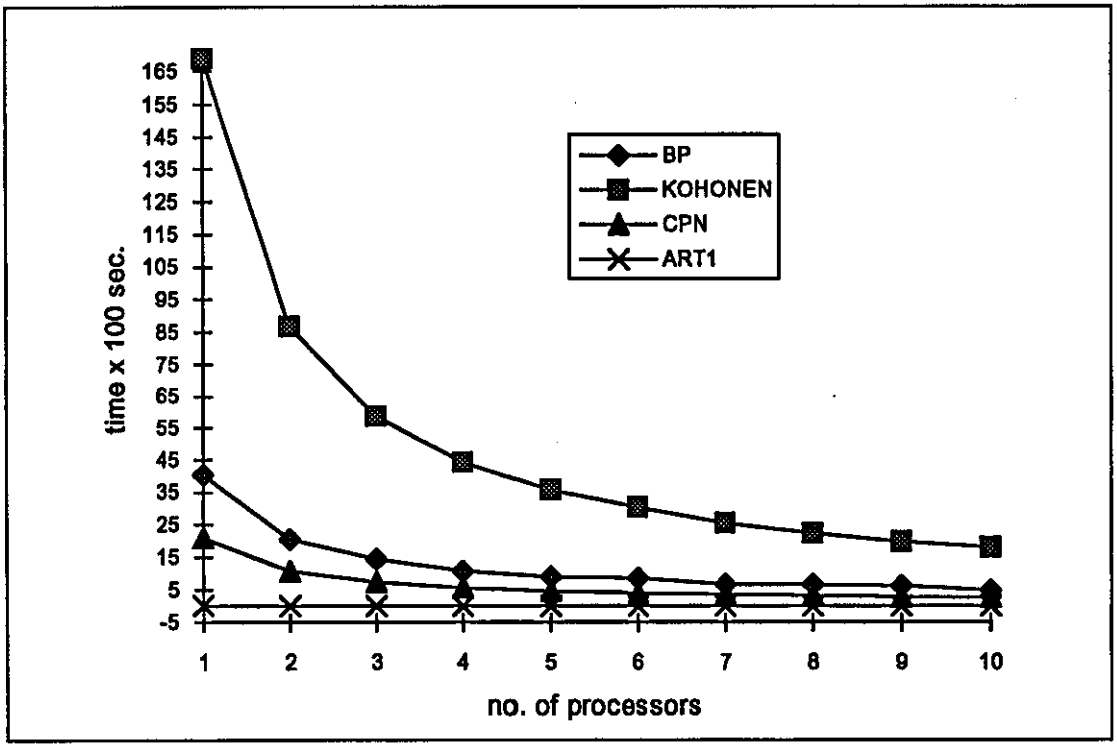


Fig. 6.6: Time in second versus no. of processors

6.2 INTERTWINED SPIRALS PROBLEM

To distinguish between two intertwined spirals is an example of a difficult pattern recognition problem. The data in this example are the x and y co-ordinates of two spirals as shown in **figure 6.7.** A set of points (i.e. input vectors) is trained to distinguish between the two spirals. The goal of this example is to train the network to map x and y co-ordinates into the proper spiral.

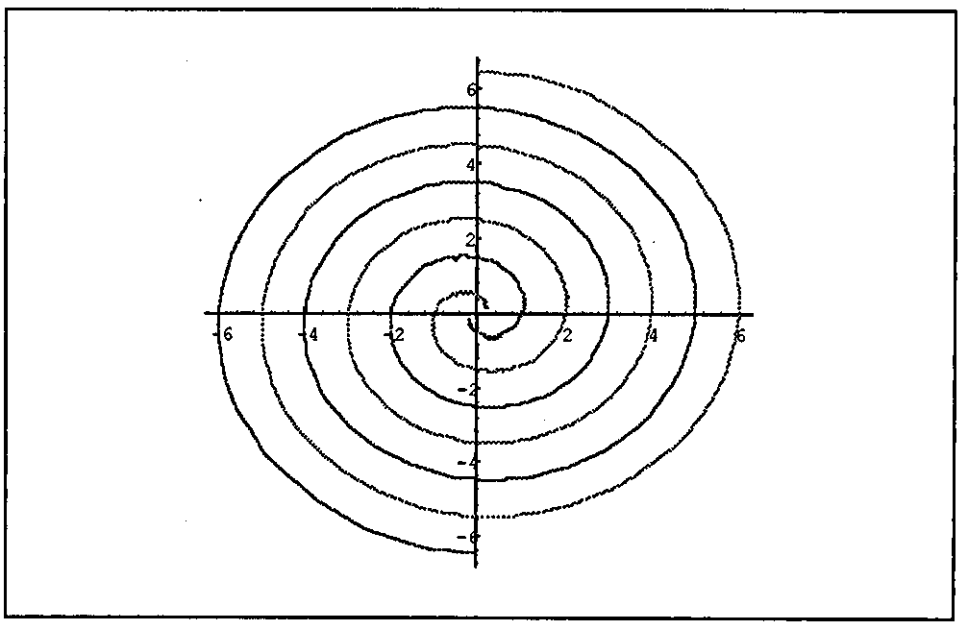


Fig. 6.7: The two intertwined spiral

The training set, i.e. x and y co-ordinates are generated by using the following formula :-

$$x = r \sin\theta$$
$$y = r \cos\theta$$

where

$$r = 6.5\frac{(104 - i)}{104},$$

$$\theta = i\frac{\pi}{16},$$

and $i = 0,1, \ldots, k\text{-}1.$

The x and y co-ordinates for the second spiral are calculated as follow :-

$$x = -r\sin\theta$$
$$y = -r\cos\theta$$

This will generate $2k$ co-ordinates for the training set. The value of k can be changed to give different training set sizes.

The spiral problem was originally conceived by Lang *et al.* (1988) using the backpropagation network. Later it became a benchmark for many researchers in the area of the backpropagation learning algorithms [Leighton *et al.* (1992), Sannosian (1992)].

A study to solve this problem has been carried out using the NEUCOMP language which also includes other network models. They are the Counterpropagation and Kohonen networks. In the backpropagation, the learning algorithm is based on the GRBH method and the weights are updated using the 'Batch' method. The parallel simulation programs for these three networks are also carried out using the NEUCOMP2 language.

6.2.1 Simulation programs for the backpropagation network

The NN architecture for the backpropagation network follows Lang *et al.* (1988). It has two input nodes, three hidden layers and one output node. The number of nodes in each hidden layer is set to seven. Each layer is connected to all the layers, which gives a short cut connection between the layers. The output can have only two states where each indicates one of the spirals. So if the point x,y lies on the first spiral then the output is 0. Otherwise the output is 1 and the point x,y lies on the second spiral. The number of training patterns are 200 (i.e. k = 100).

The simulation program written in the NEUCOMP language is similar to **section 4.4.1.** However, the

variables which represent the additional hidden layers and weights need to be defined. The 'Batch' method with the GRBH algorithms was implemented with α shown in **table 6.5** and β=0.95.

$R = \left\lvert \frac{\partial E}{\partial W} \right\rvert$	α
$R \leq 10^{-6}$	10
$R \leq 10^{-5}$	5
$R \leq 10^{-4}$	0.1
$R \leq 10^{-3}$	0.05
$R > 10^{-3}$	0.01

Table 6.5: The chosen value of α with respect to R

The print format to display the results contains the input and output data. These will give three-dimensional co-ordinates. They can then be plotted graphically using the Mathematica program called 'xyzplot'. To display a graph of the error versus number of iterations, a program called 'xygraph' is used.

6.2.2 Simulation programs for the Kohonen network

The simulation program for the Kohonen network is similar to **section 4.4.2**. The size of the input node is two and the size of the grid is set to 30x30. The number of training patterns are 200 (i.e. k = 100), the learning rate is set to 0.5 and the neighbourhood size is set to 15. The initial weight is set to a random number between 0 and 1 with the seed as 1000.

The print format to display how the Kohonen network self-organises the input pattern contains the two values of the connection weights. These will give two-dimensional co-ordinates. They can then be plotted graphically using the Mathematica program called

'xyspiral'. The information contained in the weights determines the input data that is a close approximate.

6.2.3 Simulation programs for the Counterpropagation network

The simulation program for the Counterpropagation network is similar to **section 4.4.4**. The size of the input node is two, the size of the second layer is set to 1000 and the size of the output node is one. The output node may contain 0 when the co-ordinate lies on the first spiral and 1 for the second. The number of training patterns are 200 (i.e. k = 100) and the learning rate is set to 0.5. The initial weight is set to a random number between 0 and 1 with the seed starting at 1000.

To display the simulation result, a similar print format of the backpropagation is used.

6.2.4 Simulation results

Figures 6.8 to **6.10** are the simulation results for the backpropagation network. **Figures 6.11** to **6.14** are the simulation results for the Kohonen network using NEUCOMP. **Figures 6.15** to **6.16** are the simulation results for the Counterpropagation network.

Figure 6.8 shows the spiral points are scattered around the axes before training. **Figure 6.9** shows the points that belong to each spiral correctly displayed to their respective spiral after 29184 iterations and *enormsqr* was found to be 0.08. The training process was terminated when all the activation values of the output node were less than 0.4 of the target values. **Figure 6.10** displays the progression of the GRBH learning algorithm based on the 'Batch' method. The x-axis of the graph represents the number of iterations required to obtain the solution and the y-axis represents the number of vector pairs in error.

Figure 6.11 shows the distribution of the patterns for the Kohonen network before training. The pattern vectors were randomly distributed within the spirals. **Figure 6.12** shows the patterns distribution after 200 iterations. Although all the patterns have been correctly separated, some patterns stayed close to their original patterns. **Figure 6.13** shows the pattern distribution after 500 iterations and there were still some points that were not on their original patterns. **Figure 6.14** shows the patterns distribution after 1000 iterations. All the patterns from both spirals were in their correct position.

For the Counterpropagation network, **figure 6.15** shows the input and output vectors are randomly distributed before training. **Figure 6.16** shows the pattern vectors were correctly distributed after 200 iterations.

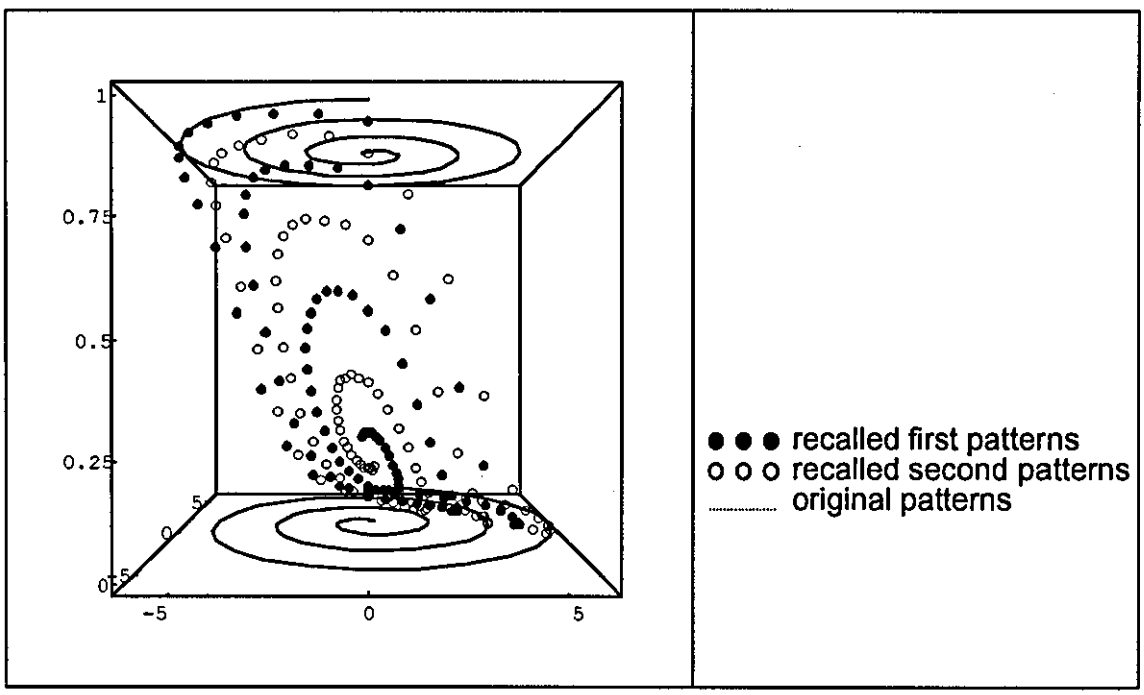


Fig. 6.8: The input and output vectors from the BP simulation before training

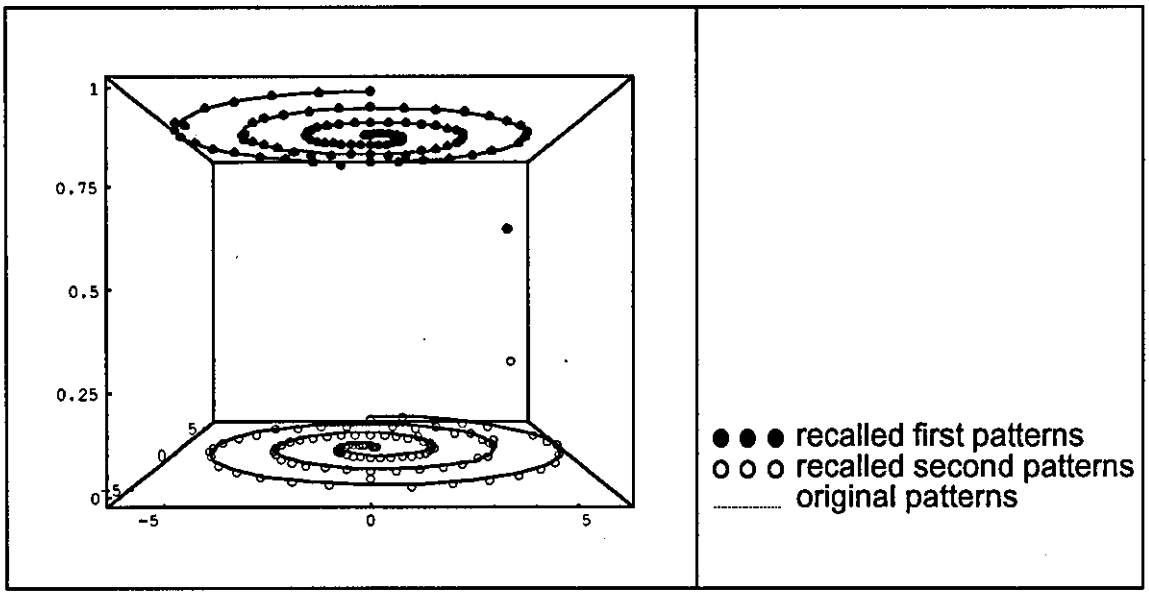


Fig. 6.9: The input and output vectors from the BP simulation after training

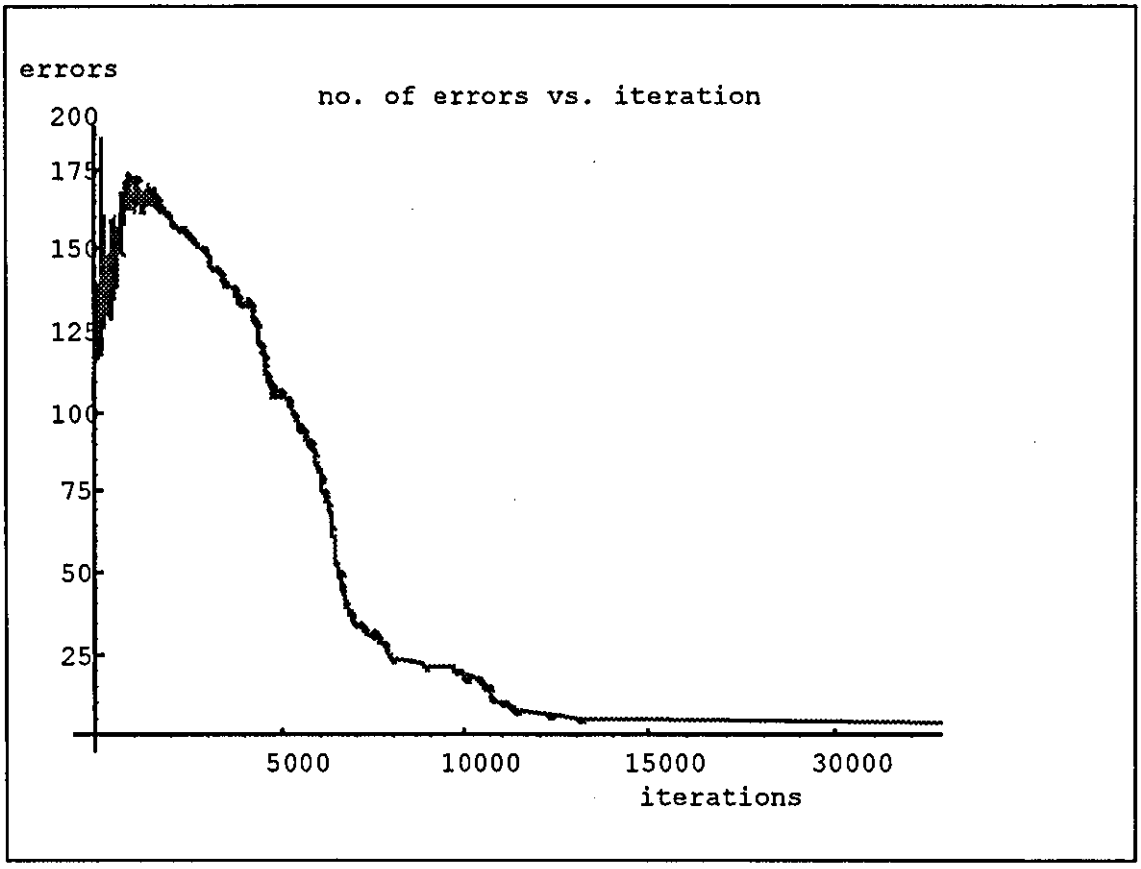


Fig. 6.10: Graph of the number of errors versus number of iterations

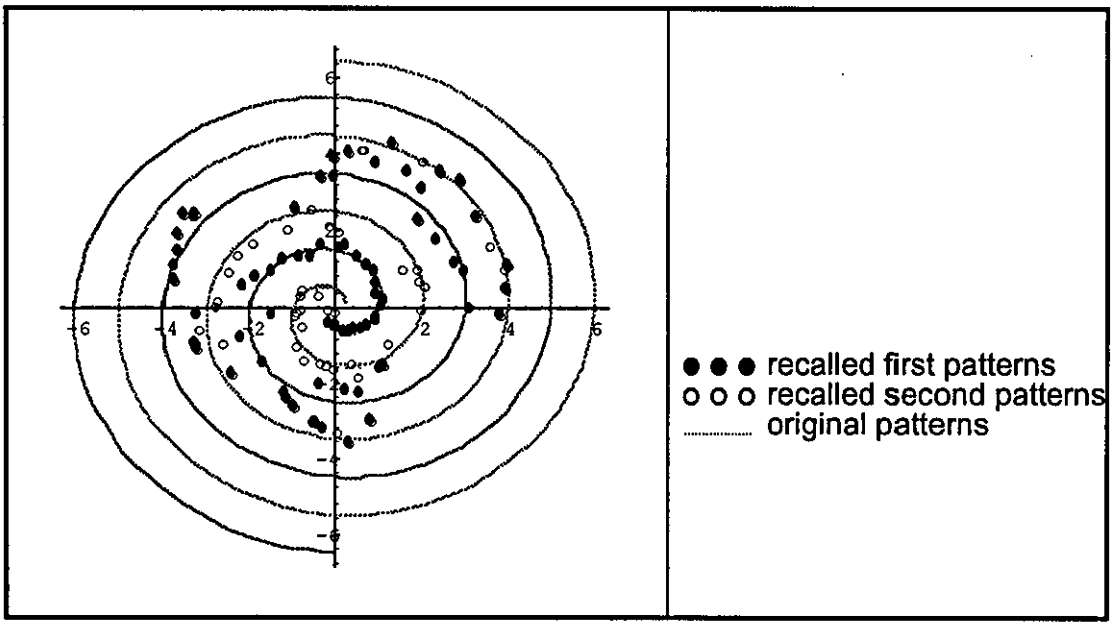


Fig. 6.11: The patterns distribution from the Kohonen network before training

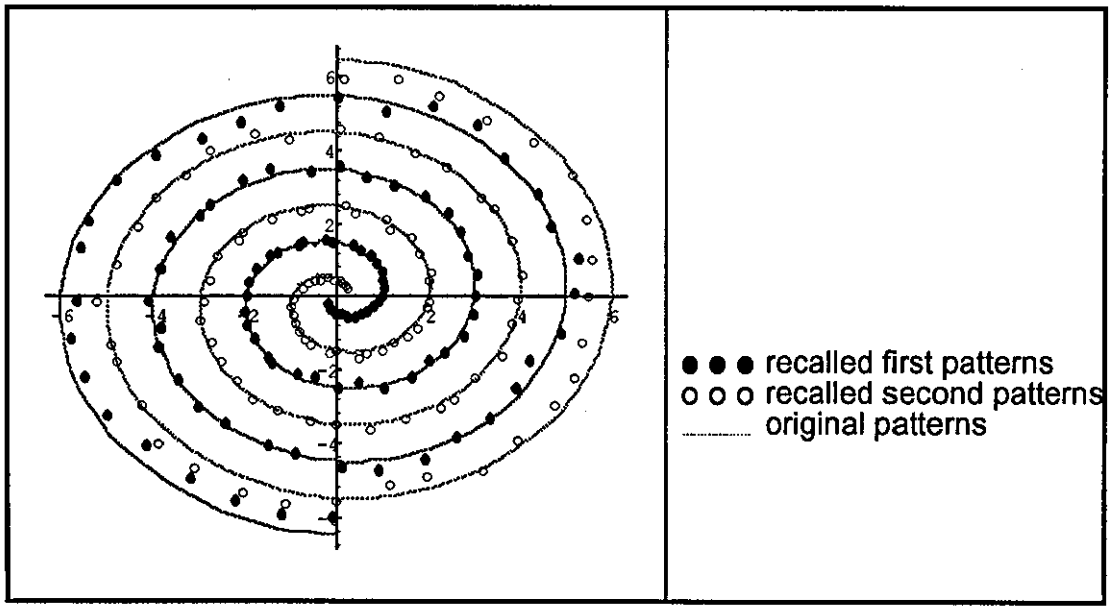


Fig. 6.12: The patterns distribution for two spirals from the Kohonen network after 200 iterations

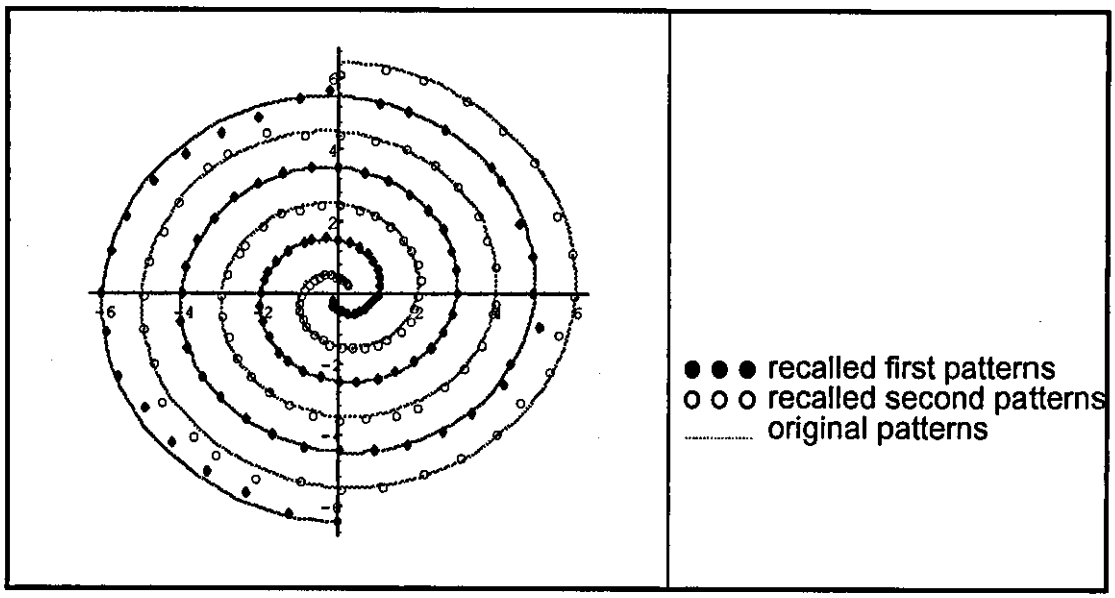


Fig. 6.13: The patterns distribution for two spirals from the Kohonen network after 500 iterations

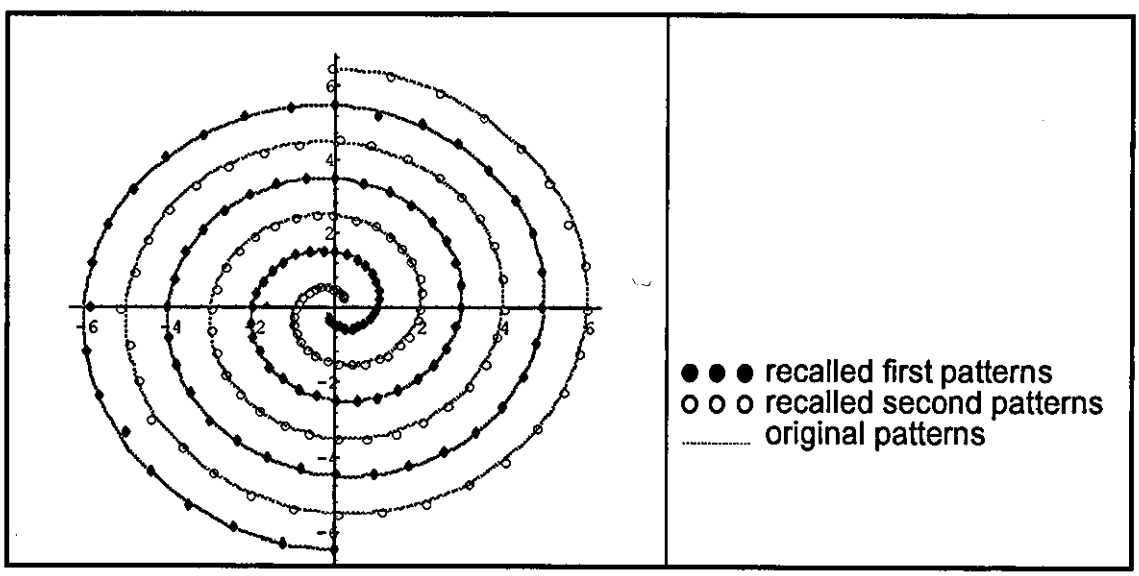


Fig. 6.14: The patterns distribution for two spirals from the Kohonen network after 1000 iterations

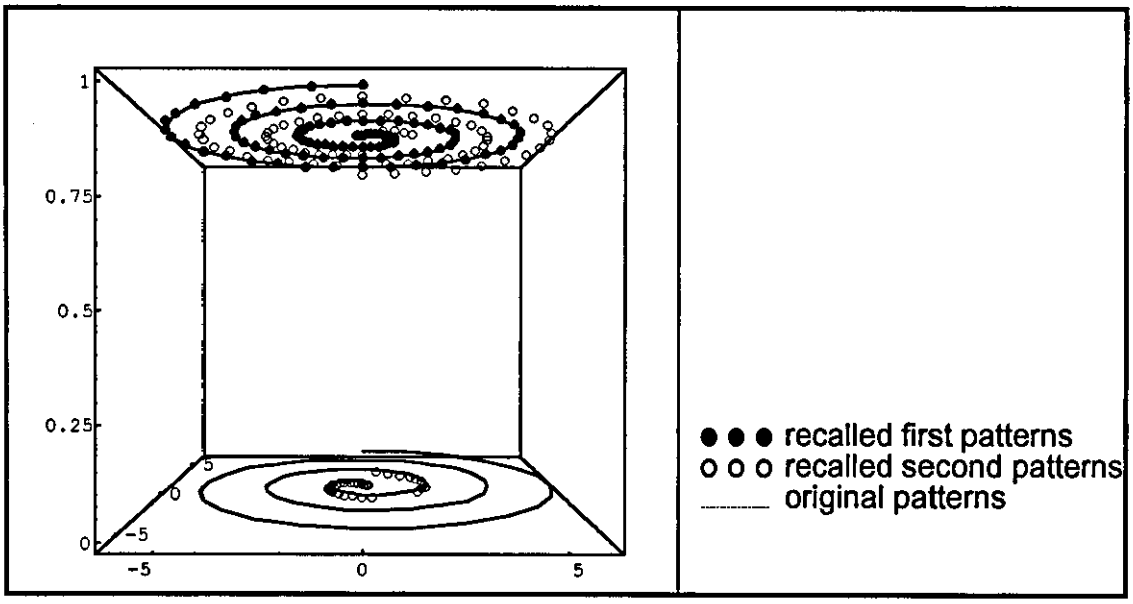


Fig. 6.15: The input and output vectors from the CPN before training

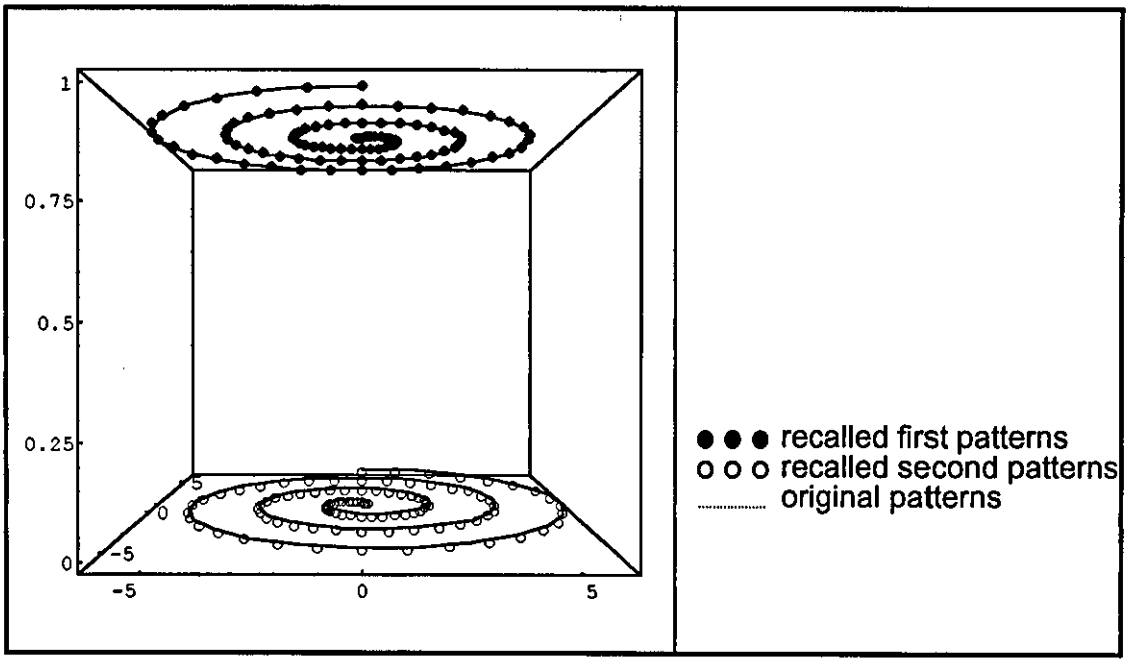


Fig. 6.16: The input and output vectors from the CPN after 200 iterations

6.2.5 Parallel Simulation results

The NEUCOMP2 programs for the backpropagation, Kohonen and Counterpropagation networks are similar to those discussed in **section 6.1.2.5**. However, the generated parallel programs for these networks are different from the previous simulations. There are some loops in the NEUCOMP2 programs for classifying two spirals that are not executed in parallel. Those loops are the input layers for all networks which have only two nodes. The output layers for the backpropagation and Counterpropagation have only one node. NEUCOMP2 does not consider loop iterations which are less than 5 (**section 5.4.2.3**).

To study the parallel performance of the backpropagation, Kohonen and Counterpropagation networks, similar experiments to **section 6.1.2.5** were carried out. The number of iterations was set to 10. The performance results for the backpropagation, Kohonen and Counterpropagation networks are shown in **table 6.6**, **6.7** and **table 6.8** respectively. The graphs of speedup versus number of processors for all the above results are shown in **figure 6.17**. It shows that for the Kohonen and Counterpropagation networks, the speedup increases steadily whilst for the backpropagation network, the speedup increases until 7 processors. Load imbalance occurred when 5 or 6 processors were used. These is because the size of the backpropagation network for spiral problem is small, i.e. 2x7x7x7x1. The speedup for the Counterpropagation network performed better because its competitive layer (i.e. 1000) is greater than the Kohonen (i.e. 30x30). However, the speedups of similar simulations (**section 6.1.2.5**) have better performance since all loops were executed in parallel.

No. of Processors	Execution time x 10^1sec.	Speedup
1	11.4	1.00
2	7.09	1.61
3	5.59	2.04
4	4.32	2.64
5	4.30	2.65
6	4.17	2.73
7	3.10	3.68
8	3.11	3.67
9	3.12	3.65
10	3.16	3.61

Table 6.6: The execution time and speedup for the BP network

No. of Processors	Execution time x 10^2sec.	Speedup
1	11.48	1.00
2	6.25	1.84
3	4.47	2.57
4	3.57	3.21
5	3.05	3.76
6	2.71	4.24
7	2.46	4.67
8	2.28	5.04
9	2.14	5.37
10	2.02	5.68

Table 6.7: The execution time and speedup for the Kohonen network

No. of Processors	Execution time x 10^2sec.	Speedup
1	11.96	1.00
2	6.08	1.97
3	4.11	2.91
4	3.14	3.81
5	2.55	4.69
6	2.17	5.51
7	1.89	6.33
8	1.69	7.08
9	1.53	7.82
10	1.42	8.42

Table 6.8: The execution time and speedup for the CPN

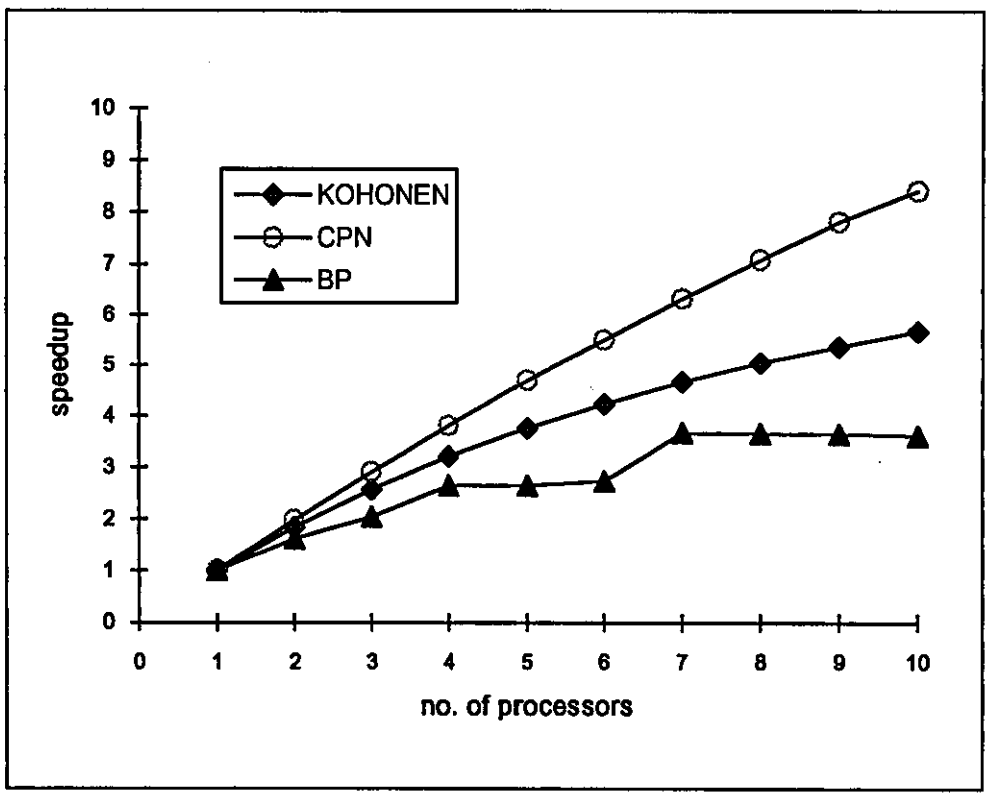


Fig. 6.17: The speedups versus no. of processors

6.3 TRAVELLING SALESMAN PROBLEM

The Travelling Salesman problem (TSP) is a well known combinatorial optimisation problem [Aarts (1989), Freisleben *et al.* (1991), Karp (1977), Platt (1988), Wacholder *et al.* (1989), Wilson *et al.* (1988)]. This is a difficult optimisation problem that belongs to the *NP*-complete class of problems. A set of N cities, *a, b, c, d, ...* have distances of separation d_{ab}, d_{ac}, ... , d_{bc}, d_{bd} ..... The aim of the TSP is to find a valid tour which visits each city once, returns to the starting city, and has the shortest total path length. As N increases, the computational work of the problems increases exponentially.

Two Hopfield-type models are considered for the TSP. They are the Continuous Hopfield model and the Potts-Glass model. The Continuous Hopfield model or Hopfield-Tank model [Hopfield *et al.* (1985)] is chosen because of its original contribution to the TSP. The Potts-Glass model is chosen because it is an alternative to find a better solution. The objective is to study the implementation of the models using the NEUCOMP language to solve the TSP.

The simulation starts with a small value of N, then the size of the problems is doubled until restricted to computational resources. The TSP is easy to solve for small N but as the number of possible solutions increases exponentially with N, it becomes impossible to find the best solution and a good approximation is an acceptable solution.

6.3.1 The Hopfield-Tank model

The details of the Hopfield-Tank model for TSP has been explained in **section 2.2.2.2.** This section discusses the Hopfield-Tank simulation program written in the NEUCOMP language for the TSP, the experimental results for the NN simulation and the parallel simulation results.

6.3.1.1 Simulation Program for the Hopfield-Tank model

The algorithm for the TSP simulation program is based on Müller *et al.* (1990). The data structure is as follows :-

```
INT  seed;
REAL
  xcity[ncity], ycity[ncity], dist[ncity,ncity],
  node[ncity,ncity], u[ncity,ncity],
  deltat, lambda1, lambda2, lambda3, tau, temp,
  energy, energy0, energy1, energy2, energy3;
```

where *seed* is used to generate a different starting number generator, *ncity* is the size of the cities, *xcity* and *ycity* the co-ordinate of the city on a two-dimensional axes, *dist* is the Euclidean distance between the cities, *node* is the activation node, u is the local field, *deltat* is the time increment Δt, *lambda1* , *lambda2* and *lambda3* are the λ's, *tau* is the time constant τ , *temp* is the temperature of the sigmoid function, and *energy* to *energy3* represent $E, E_0, ..., E_3$ of the energy function. The symbols u, Δt, λ, τ and $E, E_0, ..., E_3$ can be referred in **section 2.2.2.2.**

The algorithm for the TSP simulation program and the NEUCOMP program codes are as follows :-

(1) calculate the initial local field, u and activation node, *node* which are written as :-

```
u = -0.5*temp*LOG(ncity)*(1 + 0.1*RAND1(seed));
node = SIGMOID(2*u/temp)
```

The initial value u, is set to some noise level, i.e. with random value generated by function *RAND1* between -1 .. 1. This is to avoid starting at a spurious fully symmetric state and allows for a nondeterministic operation of the program.

(2) calculate the value of $(\sum_i \sum_\alpha n_{i\alpha} - N)$ which is written as :-

```
e3 = SUMALL(node) - ncity;
```

The built-in function *SUMALL* calculates all the elements in the matrix *node* and returns a scalar constant.

(3) calculate the u which is written as :-

```
u += deltat*(-1/tau*u - sum0(e0&,I,J)
                      - lambda1*(sum1(e1&,I) - node)
                      - lambda2*sum2(e2&,I,J)
                      - lambda3*e3);
```

where *sum0*, *sum1* and *sum2* are the user-defined functions. The first function is to calculate the E_0, the rest are for the E_1 and E_2. Their argument lists, *e0*, *e1*, *e2* followed by '&' means that a value is returned from those functions. The arguments I and J are the reserved words which represent the current row and column of the matrix. They are used when a scalar operation requires an element of the matrix. This scalar operation is defined by the user as a function. For example, *sum0* is written as shown in **figure 6.18.**

The argument list of the function *sum0* need not be declared because their types are based on its function call. The two 'if-statements' are used to impose a closed tour cyclic boundary, i.e. when $j+1$ is equal to *ncity* then set j to zero and when $j-1$ is less than zero then set j to *ncity*-1.

```
FUNC sum0(e0,i,j)
INT k,add,minus;
  e0 = 0.;
  FOR (k =0,ncity)
    IF (j + 1 EQ ncity ) add = 0
    ELSE add = j + 1
    ENDIF;
    IF (j - 1 LT 0 ) minus = ncity - 1
    ELSE minus = j - 1
    ENDIF;
    e0 += dist[i,k]*(node[k,add]+node[k,minus])
  ENDFOR RETURN e0;
```

Fig. 6.18: A function defined by user for E_0

3) calculate *energy0*, *energy1*, *energy2*, *energy3* and *energy* which are written as :-

```
energy0 += e0*SUMALL(node);
energy1 += (e1 - node)*SUMALL(node);
energy2 += e2*SUMALL(node);
energy0 *= 0.5;
energy1 *= 0.5;
energy2 *= 0.5;
energy3 = 0.5*e3*e3;
energy = energy0 + lambda1*energy1
         + lambda2*energy2
         + lambda3*energy3;
```

(4) calculate the activation node, *node* which is written as :-

```
node = SIGMOID(2*u/temp);
```

(5) repeat (2) until the following condition is met :-

```
saturation = SUMALL(node*node)/ncity;
IF ( saturation GT .95 ) BREAK ENDIF
```

The variable *saturation* approaches one if a valid solution is reached. However, the final result does not guarantee that a valid tour is found. The following test is then used to find a valid tour.

```
FOR (a = 0,n - 1)
FOR (k = a + 1,n)
  IF (city[a] EQ city[k])
    PRINT("invalid tour on city %d\n",city[a]);
    BREAK
   ENDIF
ENDFOR
ENDFOR;
```

where *city*[*a*] and *city*[*k*] contain an integer value which represent a city being visited at position a and k respectively.

The complete program is shown in **Appendix G.**

6.3.1.2 Simulation results

The initial parameters for the experiments are chosen as follow :-

$$\tau = 1.,\ \lambda_1 = .1,\ \lambda_2 = .1,\ \lambda_3 = .1 \text{ and } \Delta t = 0.0005$$

The values of the λ's are allowed to vary during the iterations as based on Wilson *et al.* (1988) in order to improve the chances of the original Hopfield network on finding the valid tour and shortest path otherwise it is difficult to get a valid tour. The number of cities tested was 20, 30 and 40 cities. A further increase on the size of the cities cannot be carried out due to the memory limitation. The number of iterations allowed for the network to reach a stable state is 1000. A further increase of this number will not change the final result.

A valid tour is easy to get but the shortest path cannot be guaranteed. To do this, a number of experiments was carried out with different initial random numbers (seeds) and temperatures. The temperature is used to determine the slope of the Sigmoid function.

Figure 6.19 shows a tour for 20 cities with a seed set to 200 and temperature of 0.03. The result shows that it is a valid tour but not the shortest. The distance calculated is 8.86. **Figure 6.20** illustrates a tour for 20 cities with a seed set to 400 and temperature of 0.03. The result shows that it is the shortest recorded so far. The distance calculated is 7.48.

Figure 6.21 shows a tour for 30 cities with a seed set to 1100 and temperature of 0.05. The result shows that it is a valid tour but not the shortest. The distance calculated is 11.04. **Figure 6.22** shows a tour for 30 cities with a seed set to 600 and temperature of 0.05. The result shows that it is the shortest recorded so far. The distance calculated is 10.02.

Figure 6.23 illustrates a tour for 40 cities with a seed set to 50 and temperature of 0.05. The result shows that it is a valid tour but not the shortest. The distance calculated is 13.45. **Figure 6.24** illustrates a tour for 40 cities with a seed set to 2600 and temperature of 0.03. The result shows that it is the shortest recorded so far. The distance calculated is 13.31.

The summary of the above results are shown in **table 6.9.**

N	seed	temperature	Distance
20	200	0.03	8.86
	400	0.03	7.48
30	1100	0.05	11.04
	600	0.05	10.02
40	50	0.05	13.45
	2600	0.03	13.31

Table 6.9: The total path for 20, 30 and 40 cities

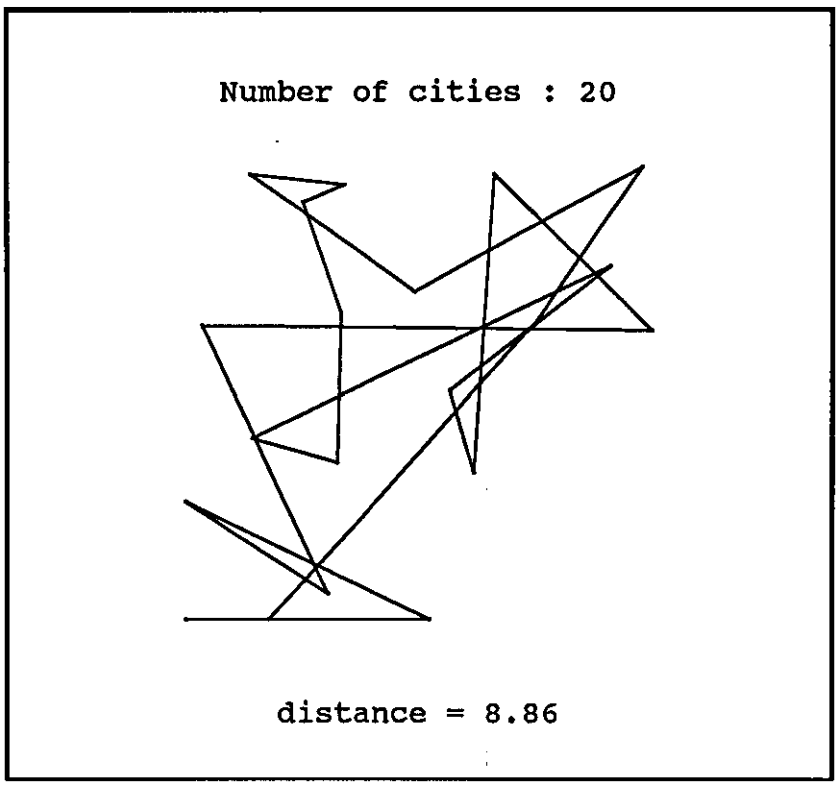


Fig. 6.19 : Valid cities but not the shortest

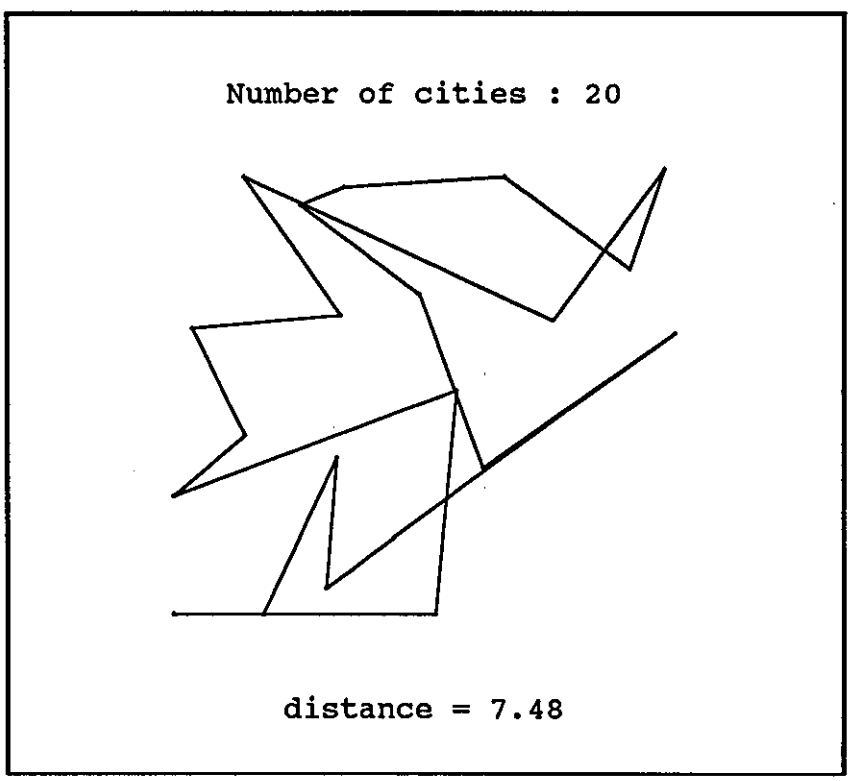


Fig. 6.20 : Shortest path so far for 20 cities

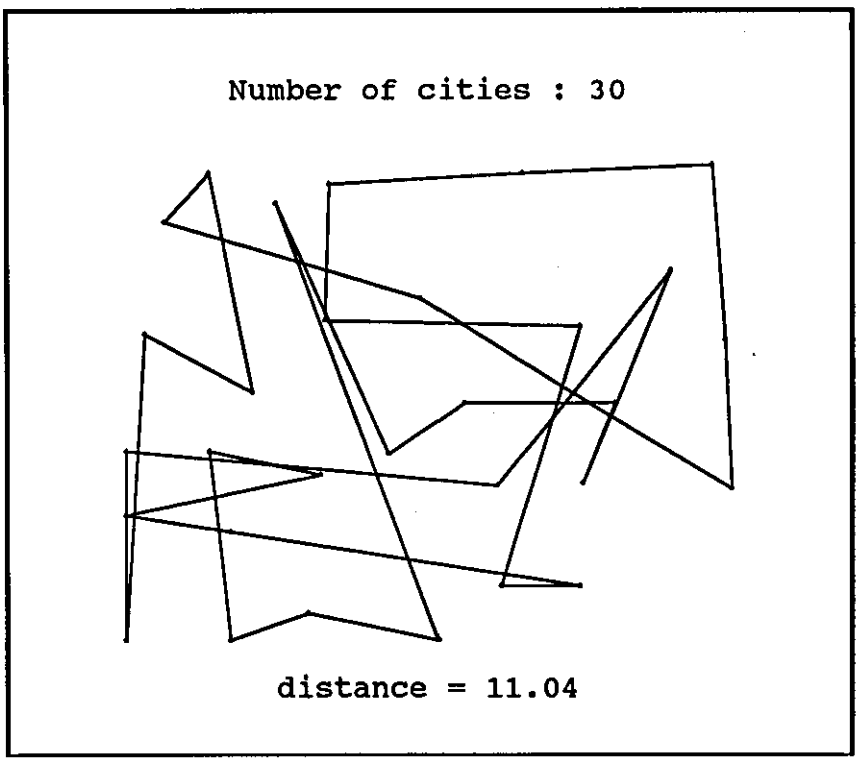


Fig. 6.21 : Valid cities but not the shortest

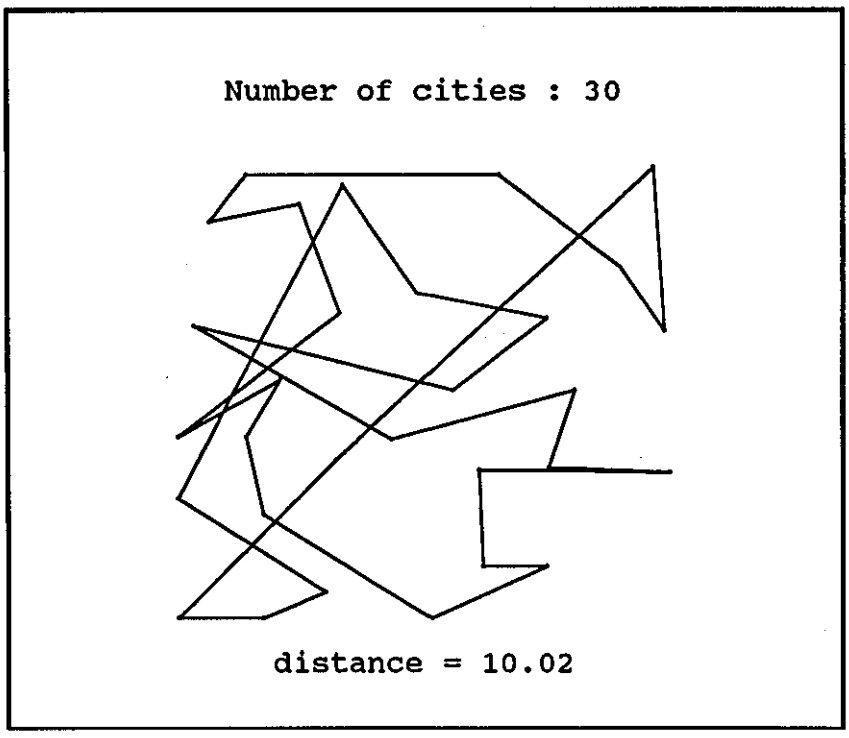


Fig. 6.22 : Shortest path so far for 30 cities

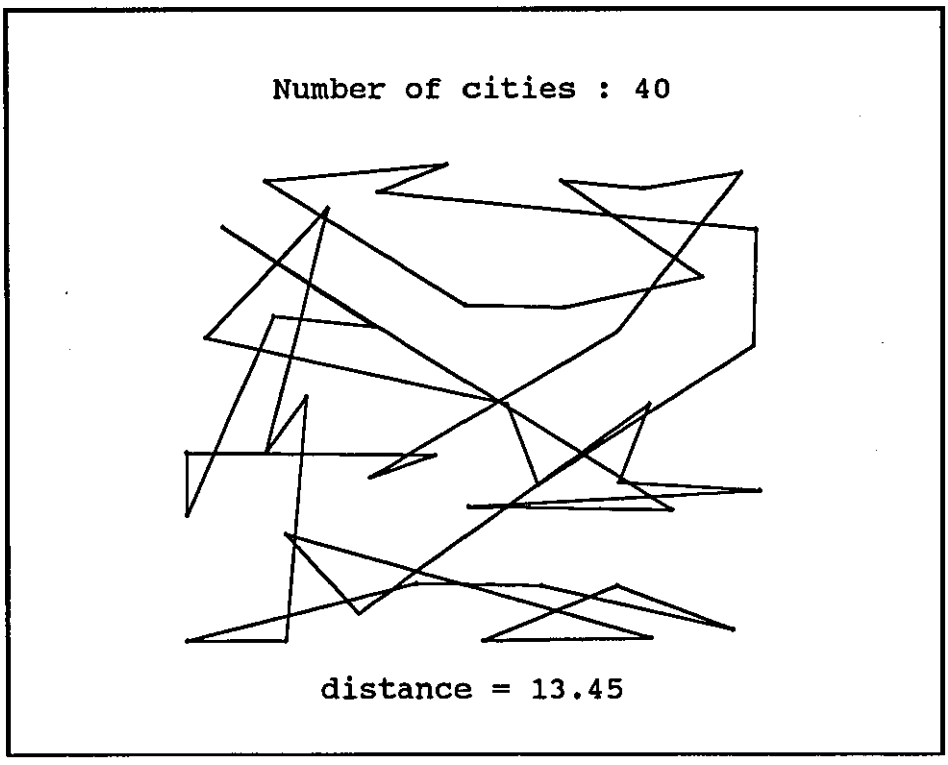


Fig. 6.23 : Valid cities but not the shortest

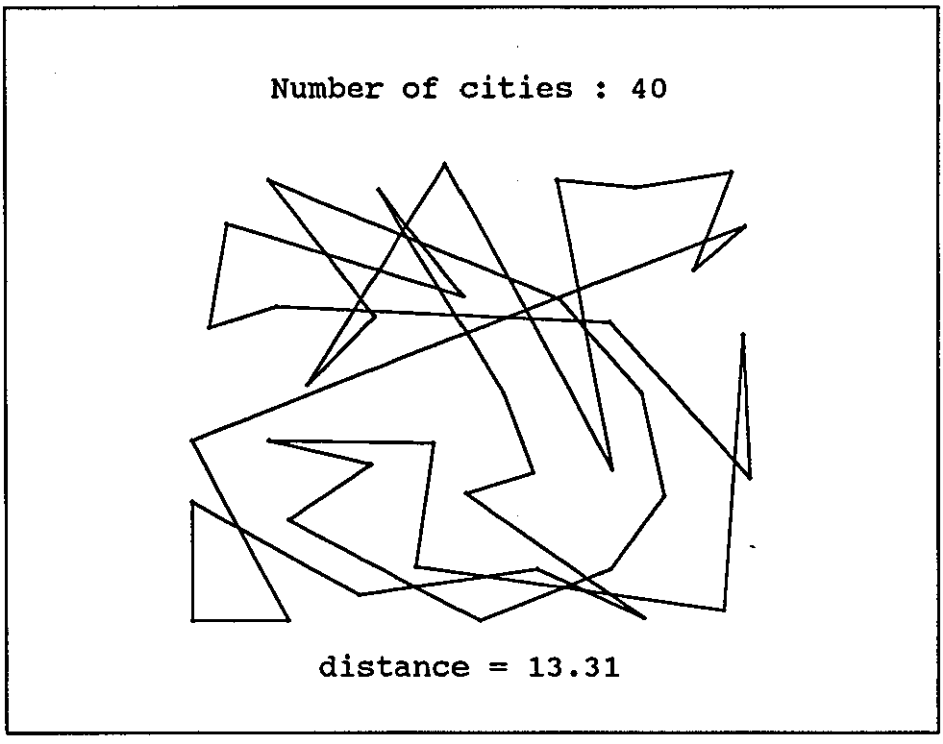


Fig. 6.24 : Shortest path so far for 40 cities

6.3.1.3 Parallel Simulation results

Experiments similar to **section 6.1.2.5** have been carried out to study the performance of a parallel NN simulation program generated by NEUCOMP2 for solving the TSP. **Table 6.10** shows the execution times and speedups for three numbers of cities, i.e. 20, 30 and 40 cities with 300 iterations. **Figure 6.25** gives the graph of speedup versus the number of processors for the above experiments. The graph shows that for a small size problem, load imbalance among the processors occurred resulting in a lost of efficiency. As the city size increases, the load imbalance characteristic almost disappears. Hence, for a network of size 40, the speedup almost reaches to an ideal state. However, due to memory limitation further results cannot be completed.

	20 cities		30 cities		40 cities	
no. of processors.	Execution time $x10^3$sec.	speedup	Execution time $x10^3$sec.	speedup	Execution time $x10^3$sec.	speedup
1	2.29	1.00	4.24	1.00	7.07	1.00
2	1.16	1.97	2.14	1.98	3.57	1.98
3	0.83	2.76	1.49	2.85	2.48	2.85
4	0.63	3.64	1.16	3.66	1.79	3.95
5	0.49	4.67	0.96	4.42	1.46	4.84
6	0.49	4.67	0.79	5.37	1.27	5.57
7	0.37	6.19	0.75	5.65	1.08	6.55
8	0.38	6.03	0.60	6.95	0.95	7.44
9	0.38	6.03	0.61	6.98	0.86	8.22
10	0.27	8.48	0.47	9.01	0.76	9.30

Table 6.10: The execution times and speedups for 20, 30 and 40 cities.

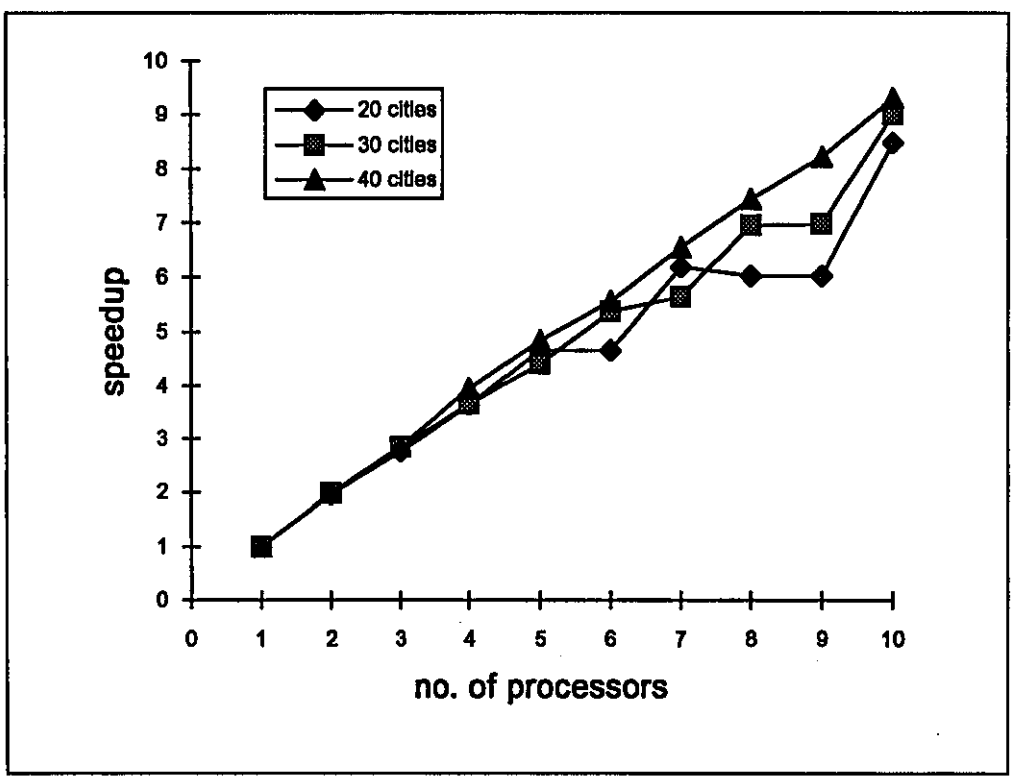


Fig. 6.25: The speedups versus no. of processors

6.3.2 *The Potts-Glass model*

Previous results for solving the TSP using the Hopfield-Tank model have not always located the shortest path. A valid tour is sometime difficult to get for the size of 30 and more cities. Discussions on the reasons for this can be obtained from Wilson *et al.* (1988). The next model to be considered in this section is the Potts-Glass model suggested by Peterson *et al.* (1989).

The problem of not achieving the solutions in the Hopfield model is because the tour is chosen by a set of N^2 independent node variables $n_{i\alpha} = 0$ or 1 where $i, \alpha = 1, \ldots, N^2$. This lead to a situation whereby $n_{i\alpha}$ can be active (i.e. one) on more than one α (i.e. cities). Thus instead of allowing the node to be active and inactive independently, the nodes are set to satisfy the following constraint

$$\sum_{\alpha} s_{i\alpha} = 1$$

where s is known as a Potts spin variable. This guarantees that a city is visited exactly once. In what follows, the encoding scheme is denoted as 'graded neurons'.

Müller *et al.* (1990) outlines the Potts-Glass model for the TSP as follows :-

(1) The energy function is written as

$$E = \sum_{i,j} \sum_{\alpha} d_{ij} s_{i\alpha} s_{j\alpha+1} + \frac{A}{2} \sum_{i} \sum_{\alpha, \beta}^{\alpha \neq \beta} s_{i\alpha} s_{i\beta} + \frac{B}{2} \sum_{\alpha} \left(\sum_{i} s_{i\alpha} - 1 \right)^2$$

The objective is to find a global minimum when the spin variables take on values representing a valid tour with minimum length and then to search for the ground state of the spin system.

(2) In the search for the ground state of the spin variable or at least a state with energy as low as

possible, the 'mean-field approximation' of the Potts-Glass model is used. Thus the mean-field equations are expressed as follows :-

$$v_{i\alpha} = F_N(u_{i\alpha}) \tag{6.1}$$

$$u_{i\alpha} = \frac{1}{T}[-\sum_j d_{ij}(v_{i,\alpha+1} + v_{j,\alpha-1}) + Av_{i\alpha} - B\sum_j v_{j\alpha}] \tag{6.2}$$

where

$$F_N(u_{i\alpha}) = \frac{e^{u_{i\alpha}}}{\sum_\alpha e^{u_{i\alpha}}}$$

Now **equations 6.1** and **6.2** will be iterated at a constant temperature until a stable solution is reached. The next iteration will be performed at a reduced temperature. This strategy follows the simulated annealing method to avoid getting stuck at the local minimum [Kirkpatrick *et al.* (1983)].

(3) The initial spin variable is set as

$$v_{i\alpha} = \frac{1}{N}(1 + 0.1RAND\,1)$$

as was done in the Hopfield-Tank model.

(4) The stopping criteria depends on one of the following conditions :-

(a) The accumulated change of the spin values in the updating procedure,

$$\frac{1}{N}\sum\left|v_{i\alpha}^{new} - v_{i\alpha}^{old}\right| < \delta$$

is smaller than a predetermined constant, i.e. δ=0.1.

(b) the saturation of the solution

$$\eta = \frac{1}{N}\sum_{i\alpha} v_{i\alpha}^2$$

approaches one as in the case of the Hopfield-Tank.

The following sections discuss the NEUCOMP program codes for the Potts-Glass simulation, the results when the size of the problems is increased and the performance of the parallel simulation.

6.3.2.1 Simulation Program for the Potts-Glass model

The algorithm for the Potts-Glass simulation is based on Müller *et al.* (1990). The data structure is as follows :-

```
INT  seed;
REAL
  xcity[ncity],ycity[ncity], dist[ncity,ncity],
  v[ncity,ncity], u[ncity,ncity],
  temp, anneal, delta, aconst, bconst;
```

where *seed*, *ncity*, *xcity*, *ycity* and *dist* serve similar purposes as defined in **section 6.3.1.1**, v represents spin variable (**equation 6.1**), u is the variable of **equation 6.2**, *delta* represents δ, *temp* is the temperature, i.e. T, *aconst* and *bconst* are the constraint parameters for the A and B of **equation 6.2**.

The algorithm for the TSP simulation program and the NEUCOMP program codes is as follows :-

(1) calculate the initial variable for spin which is written as :-

```
v = (1. + 0.1*RAND1(seed) )/ncity;
```

(2) calculate the *u* which is written as :-

```
u=EXP((-sum0(I,J) + aconst*v
                  - bconst*sum1(v,J))/temp );
```

where *sum0* and *sum1* are user-defined functions. The first function has a similar program code to **figure 6.18**. The second function is used to calculate all the elements of the variable *v* with respect to the current column of the matrix, i.e. *J*.

(3) calculate the change of *v* values and the saturation level as written below:-

```
ov = v;
v = u/sum2(u,I);
change=1/ncity*SUMALL(ABS(ov - v));
saturation = 1/ncity*SUMALL(v*v);
```

The variable *ov* is used to hold the old value of *v*. The function *sum2* is defined by the user. It is used to calculate all the elements of the variable *u* with respect to the current row of the matrix where *I* is the reserved word. The built-in function *SUMALL* calculates all the elements in the matrix *v* and returns a scalar constant.

(4) repeat (2) for an iteration at constant temperature until the following condition is met :-

```
IF ( change LT delta) BREAK ENDIF;
```

(5) reduce the temperature by the factor *anneal* written as follows :-

```
temp = temp*anneal;
```

(6) repeat (2) for different temperatures until the following condition is fulfilled

```
IF ( saturation GT .9 ) BREAK ENDIF;
```

A valid tour follows the same method as the Hopfield-Tank simulation program.

The complete program is shown in **Appendix H.**

6.3.2.2 Simulation results

The same size of problems are tested using the simulation program for the Potts-Glass model. The initial parameters for the experiments are as follows :-

(1) δ = 0.01.
(2) Cycle for annealing = 20
(3) Cycle for iteration at constant T = 40
(4) T = 0.4

A number of experiments to find the shortest path for increasing sizes was carried out by changing the seed. For 20 and 30 cities, the shortest paths were easily found. **Figures 6.26** and **6.27** show the graphs of the routes with no intersection. This means an optimal solution. The shortest distance recorded for 20 cities is 4.54. The shortest distance recorded for 30 cities is 5.89. However, for 40 cities, different results were obtained. **Figure 6.28** gives a path which is not the shortest because there are still line crossing occurring. The seed setting was 1000 and the distance found was 6.44. **Figures 6.29** shows the path obtained is the optimum since there is no line intersection. The seed is set to 2000 and the distance recorded is 6.27.

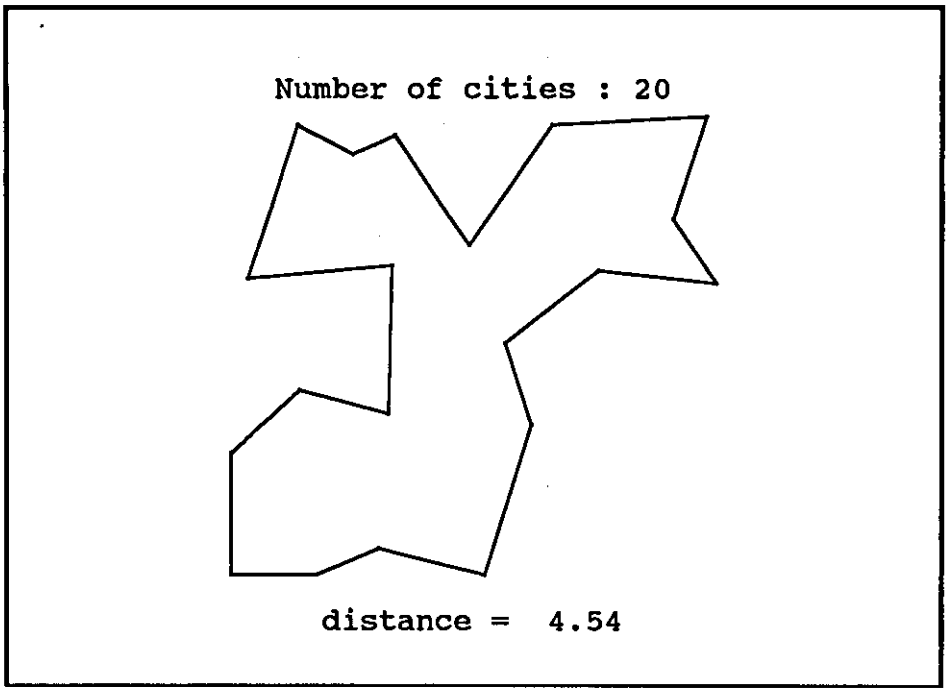


Fig. 6.26 : Shortest path so far for 20 cities

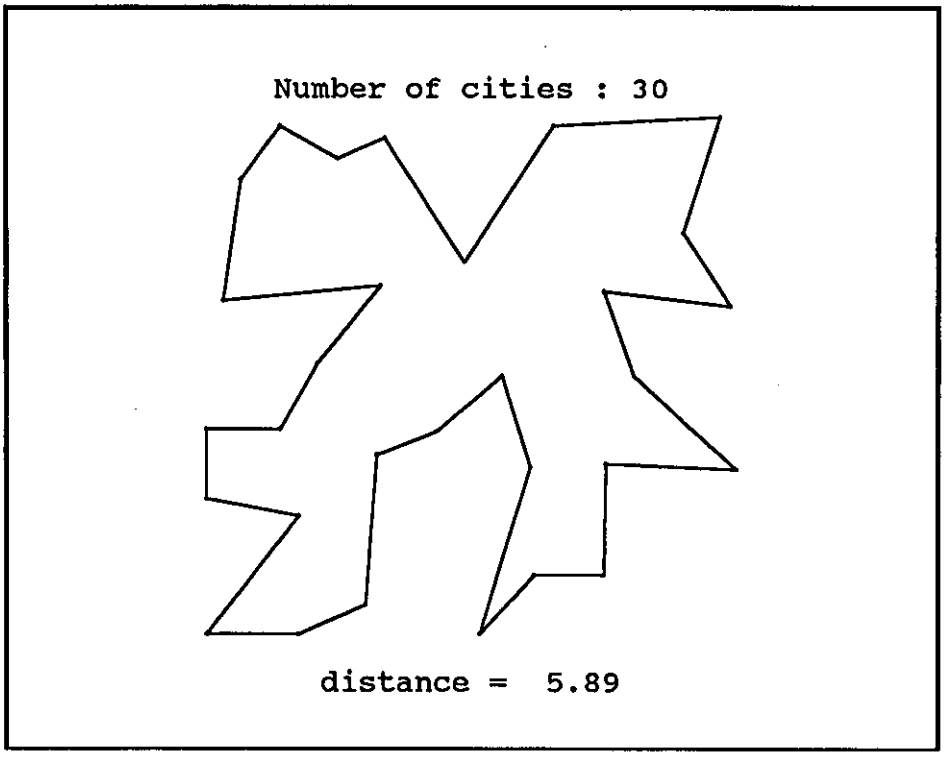


Fig. 6.27 : Shortest path so far for 30 cities

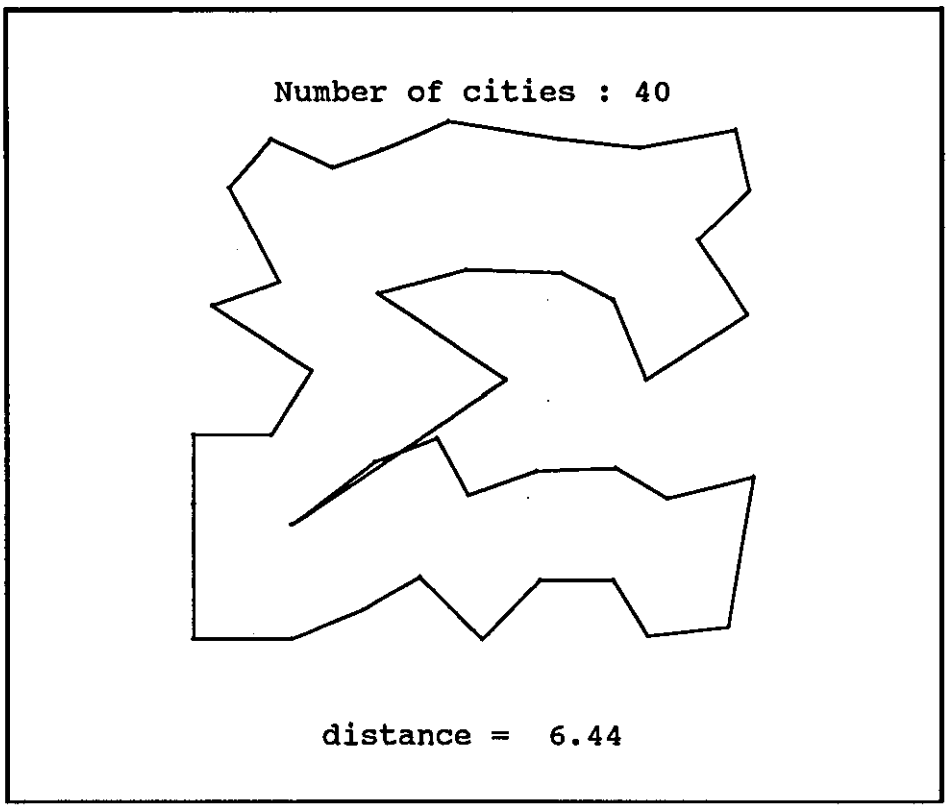


Fig. 6.28 : Non-optimum solution for 40 cities

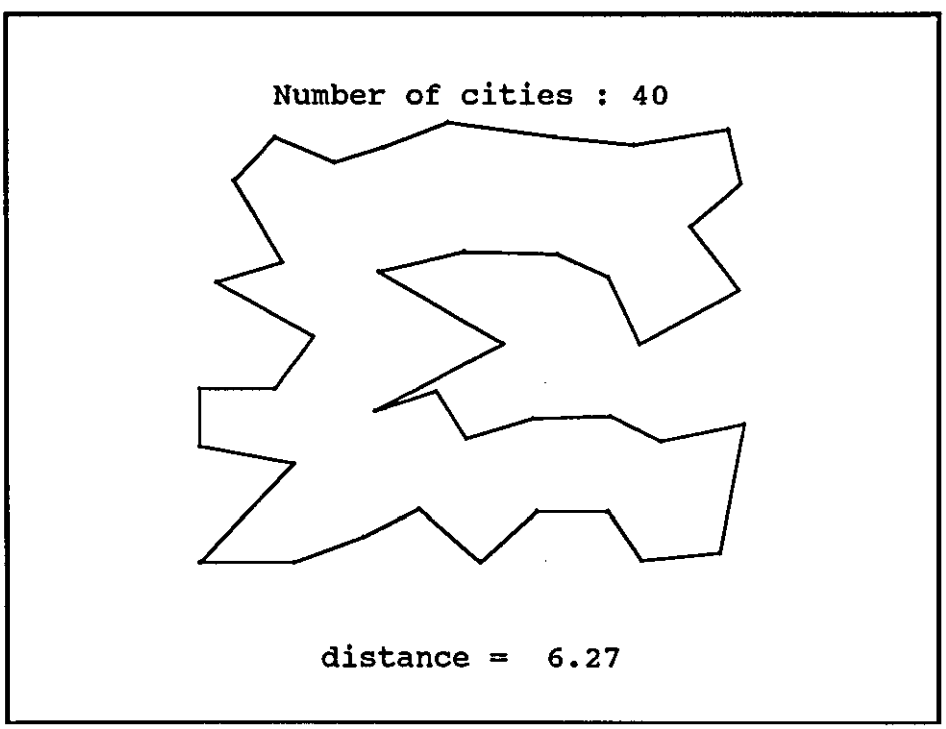


Fig. 6.29 : Shortest path so far for 40 cities

6.3.2.3 Parallel Simulation results

Experiments similar to **section 6.1.2.5** have been carried out to study the performance of a parallel NN simulation program generated by NEUCOMP2 for solving the TSP using the Potts-Glass model. The same parameters of **section 6.3.2.2** was used. At first the same number of cities of the parallel simulation using the Hopfield-Tank was carried out, unfortunately the performance result was very poor. The size of the problem was further increased with the hope that the simulation would perform better. The maximum size permitted is 80 but there was no corresponding improvement in results. The execution times and speedups of 20, 40 and 80 cities are tabulated in **table 6.11. Figure 6.30** shows the graph of speedup versus the number of processors for the above experiments. The graph shows that as the size of the problem increases, the speedup increases only slightly.

The reasons for this are now given. The matrix/vector operations in step 2 and 3 of **section 6.3.2.1** are involved within the training loop. They use the same outer loop (their row size are the same). This loop is chosen by NEUCOMP2 for parallel loop execution. The mathematical operation on variable u in step 2 requires the old value of v to be calculated in function *sum1*. If the loop is executed in parallel, there are new values of v calculated in step 3 which are also involved in the loop iteration. Thus incorrect operation of the simulator has occurred. In order to maintain correctness, NEUCOMP2 omits this loop and considers the next inner loop. However, the next inner loop, has a similar structure as in **figure 6.18**. The column of the matrix contains ordered-shared dependencies which are also omitted by NEUCOMP2. Thus, not many inner loops can be considered for parallelism within the training iteration.

	20 cities		40 cities		80 cities	
no. of processors	Execution time $x10^3$sec.	speedup	Execution time $x10^3$sec.	speedup	Execution time $x10^3$sec.	speedup
1	0.37	1.00	2.16	1.00	14.1	1.00
2	0.37	1.00	2.06	1.05	13.0	1.09
3	0.35	1.06	2.00	1.08	12.8	1.10
4	0.35	1.06	1.98	1.09	12.7	1.11
5	0.35	1.06	1.97	1.10	12.6	1.12
6	0.35	1.06	1.96	1.11	12.6	1.12
7	0.35	1.06	1.95	1.11	12.6	1.12
8	0.35	1.06	1.95	1.11	12.6	1.12
9	0.35	1.06	1.94	1.11	12.6	1.12
10	0.34	1.09	1.93	1.12	12.5	1.13

Table 6.11: The execution times and speedups for 20, 40 and 80 cities using the Potts-Glass model.

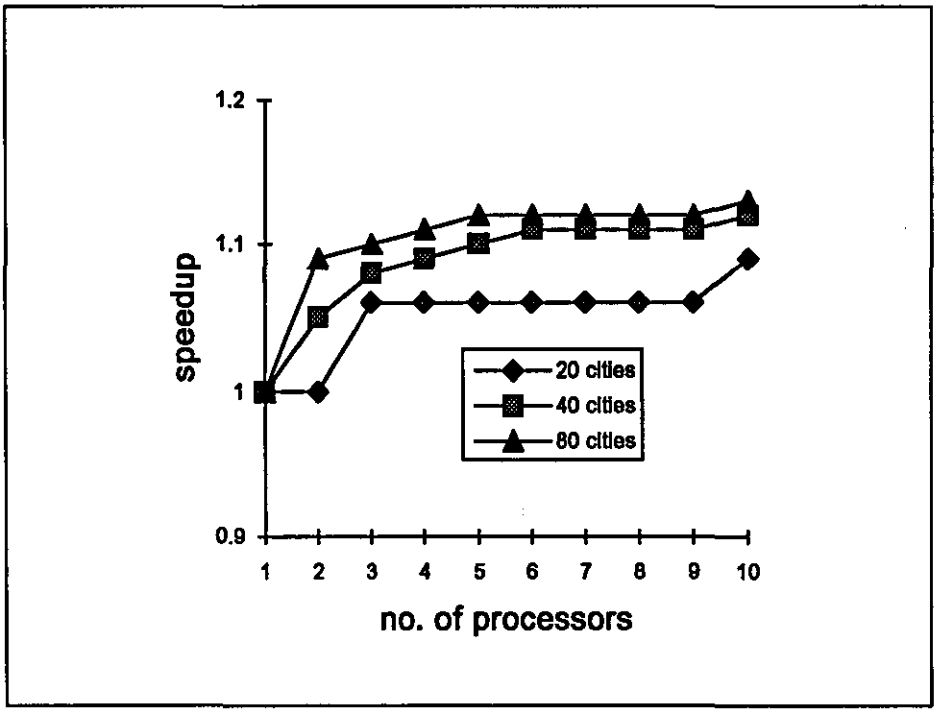


Fig. 6.30: The speedups versus no. of processors

6.3.3 Discussion of the results

The Hopfield-Tank simulation results obtained from the NEUCOMP simulation program have not given the optimum solutions for the size of problem of 20 cities and more. Many experiments failed to find a valid tour. However, its parallel simulation program generated by the NEUCOMP2 performed well where the slope of the graph of speedup versus number of processors nearly reaches the ideal state when the size of the problem increased.

The optimum solution for the Potts-Glass simulation was easy to find for the small size problem. As the size of the problem increased, finding the optimum solution is not easy. Many trials had to be made by changing the initial random numbers in order to find the shortest path. The optimum solution is found when the graph of the route has no line crossing (**figures 6.26, 6.27** and **6.29**). As another example, **figures 6.31** and **6.32** show two different distances for 80 cities. The first figure is not the optimum solution because there are many line crossings. The second figure is near to the optimum value. Both have seed settings for 2000 and 3200 respectively. Hence, an accurate result cannot be guaranteed because no network of polynomial size in N can exist that will solve the TSP for N cities to a desired accuracy [Müller *et al.* (1990)].

The summary of the comparison for the shortest distance found using the Hopfield-Tank and Potts-Glass models are shown in **table 6.12.**

Although the Potts-Glass simulation program written in the NEUCOMP has succeeded in solving the problem, unfortunately its parallel version did not perform well owing to data dependencies on both the outer and inner loops.

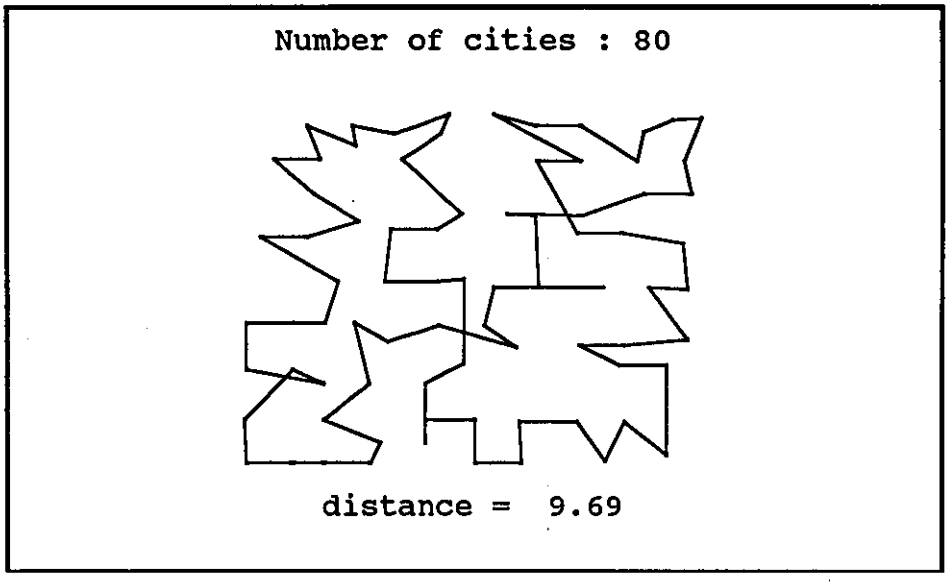


Fig. 6.31 : Non-optimum solution for 80 cities

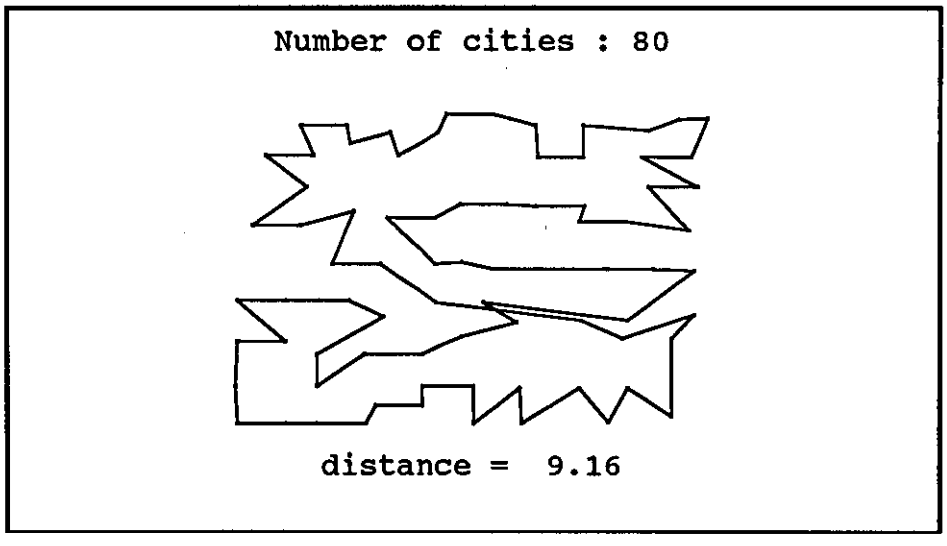


Fig. 6.32 : The shortest so far for 80 cities

Number of cities	Hopfield-Tank simulation	Potts-Glass simulation
20	7.46	4.54
30	10.02	5.89
40	13.31	6.27

Table 6.12: The total path for 20, 30 and 40 cities

CHAPTER 7

SUMMARY AND CONCLUSIONS

Neural network (NN) models have grown rapidly in solving applications involving massively parallelism such as pattern recognition where the traditional programming methods are not viable. This is because NNs are designed to mimic the human brain which is able to operate easily in parallel to solve problems such as pattern recognition. Although the computer is a high-speed serial machine, it is unable to solve quite simple recognition problems. NNs and their computational properties have attracted the interest of researchers in the area of machine perception by presenting an exciting, complementary alternative to symbolic processing paradigms. They hold with them the promise of exceedingly fast implementations coupled with flexibility through self-organisation or learning.

NN models can be implemented in various ways. These can range from a very complex hardware VLSI design to software simulators on a digital computer. Hardware implementations are faster than software simulators but they are confined to special purpose NNs. Computer simulation is more flexible and economical for rapid prototyping and problem solving

A general-purpose NN simulation tool has become a current trend because it is more flexible. The user can easily simulate any NN model or combine these models to suit their applications. To implement this simulator, a proper programming language specifically for NN model is preferred. The existing high-level languages such as C or FORTRAN are not suitable because most of the NN designers do not originate from a computer programming background.

The program translations for NN languages come from either compiler method [Almassy *et al.* (1990), Leighton *et al.* (1992), Panetsos *et al.* (1993)] or a combination of both interpreter and compiler methods [Korn (1989, 1991a&b)]. There exists many programming languages specifically for the NN models [Almassy *et al.* (1990), DasGupta *et al.* (1990), Hu (1991), Korn (1989, 1991a&b), Zell *et al.* (1991), Vellacott (1991), Leighton *et al.*

(1992), Panetsos *et al.* (1993)]. These NN languages cover many methods of programming style such as descriptive (declarative), procedural and object-oriented. The purpose is to provide a free style of writing a simulation program for any of the NN models.

NEUCOMP is a NN compiler used to compile the procedural style of programs known as the NEUCOMP language. It is a high-level language specially designed to cater for any NN model with the complexity of the existing high-level languages such as C being simplified. It also contains graphical facilities such as portraying the NN architecture and displaying a graph of the result, and finally it can run on a parallel shared memory multi-processor system. A NEUCOMP program is written as a list of mathematical specifications of the particular NN model. The mathematical statements can be written as scalar, vector or matrix assignments as required by the NN models. This idea is based on Korn's work [Korn (1989, 1991a&b)]. The DESIRE/NEUNET program is translated by a combination of both an interpreter and a compiler whereas NEUCOMP is based only on compiler.

It is well known that the compilation of high-level programs has been proved to produce a high performance result [Bennett (1990),Ford (1990)]. However, to develop a true compiler is a difficult task. NEUCOMP takes a simpler approach as the objective here is to study the suitability of the NEUCOMP language to perform general implementations of NN models. The reason is to provide an ad hoc and workable compiler at an early stage so that when it is successful a true compiler can be later developed. The C language is chosen as the target language because it is portable to any machine under the UNIX platform.

The procedural approach is chosen because traditionally this approach has been established since the evolvement of the FORTRAN language. Furthermore, a procedural language allows list of mathematical specifications to be easily organised or written

algorithmically. Other approaches as mentioned earlier which are declarative (descriptive), functional and object-oriented are not suitable for writing a series of mathematical specifications.

NEUCOMP has been implemented on the SEQUENT Balance machine at PARC. It is used to generate a sequential program. It can be used on any UNIX based machine. NEUCOMP2 has been implemented on the same machine and used to generate a parallel program for a shared memory parallel computer system. NEUCOMP2 is different from NEUCOMP in that its compiler phases contains a parallel routine called the parallelising stage. It analyses the existence of parallelism in the target program which is written in sequential form and transforms it into an equivalent parallel program. The target machine is the shared-memory parallel processor. Program correctness is based on both sequential and parallel results being compared. Experiments were carried out in parallel because of better execution times and speedups that can be attained.

The NEUCOMP/NEUCOMP2 language has proved to be capable of designing a simulation program for any NN model. So far, 5 models of different structure and training algorithms, and solving 3 NN applications of different problems have been successfully compiled and executed. The results of the simulation programs were very encouraging.

The chosen NN models which represent different classes of the networks were the backpropagation, Kohonen, Counterpropagation, ART1 and Hopfield-type networks. The backpropagation network is a multilayer feedforward network. The Kohonen network is a self-organising topological network. The Counterpropagation network is a three layer network in which the hidden layer is the competitive layer network. The ART1 network is a two layer network with feedforward and feedback connections. The Hopfield-type network is a single layer with feedback connection. The learning algorithms for the

backpropagation and Counterpropagation networks are based on supervised learning whilst the Kohonen, ART1 and Hopfield-type networks use unsupervised learning.

The NN simulators generated by NEUCOMP/NEUCOMP2 for the above mentioned models were used to solve three categories of problems, i.e. the classification, approximation and optimisation. Character recognition belongs to the classification category. It was solved by the backpropagation, Kohonen, ART1 and Counterpropagation simulators. The intertwined spirals problem belongs to the approximation/classification category. The backpropagation simulator was used to approximate the spiral type from two sets of input co-ordinates. The Kohonen simulator classified the intertwined spirals into two clusters which were shown on the map. The Counterpropagation simulator did the clustering of the intertwined spiral on its competitive layer and then classified the clusters to belong to which spiral. The Hopfield-type networks considered were the Hopfield-Tank and Potts-Glass models. They were used in the optimisation category to solve the travelling salesman problem.

The simulation results for all the categories mentioned above have been successfully recorded. They were shown graphically using the 'Mathematica' programs which were included as NEUCOMP/NEUCOMP2 library functions. The parallel simulation for character recognition and spiral problems have shown increasing speedup as the number of processors increases except that the backpropagation simulator for solving the spiral problem has a speedup that reached a maximum for 7 processors. This is because its network size is 2x7x7x7x1. For the travelling salesman problem, the parallel simulation for the Hopfield-Tank has shown good performance but the Potts-Glass model was disappointing because its loop segment contained data dependencies which obstructed the parallelism.

NEUCOMP2

Although NEUCOMP/λ has explored different NN applications, these problems are not new. However, this research is to study the feasibility of implementing a general-purpose simulator based on the compiler method. It is challenging to tackle certain problem and NN model in depth within a limited period of time whereas the characteristic of the general-purpose model has to be achieved. Another difficulty encountered was to find problems with realistic data to be implemented owing to industrial secrecy. The model examples given in the books or papers are often too simple and straight forward. For the more advanced examples, insufficient details are given.

However, NEUCOMP/NEUCOMP2 has the following advantages:-

(1) Flexibility - the user has a free style of developing his own simulation program. A fixed NN or more general NN simulator can be designed.
(2) Efficiency - the program can be run in parallel.
(3) Readability - the statements are English-like commands. An algorithm is easy to follow which is based on structured programming technique.
(4) Dynamic-like structure - the use of dynamic memory allocation allows a simulation program on a model that can be used for any size of the network. The size can be assigned at run time without recompilation.
(5) Simple and straightforward language - the mathematical form written in matrix/vector notation are easily included. This allows the designer to avoid the use of the loop on matrix/vector operations. However, If the matrix/vector operation cannot be used then the loop written as in other high-level language can be used.
(6) Portability - the target program written in C can be used on any UNIX machine. However, the parallel

target program can only be run on a shared-memory parallel machine.

(7) The target program can be used as a source code when more facilities of the C language are required to enhance the complexity of the software.

The development of NEUCOMP/NEUCOMP2 is just the initial stage. Due to lack of man power, equipment and time, many other NN models and application could not be explored. The related topics for the development of NEUCOMP/NEUCOMP2 are the compiler design, NN models, NN simulators, parallel compiler and NN applications. There are many topics that are not covered such as :-

(1) Error handling and recovery routines when a syntax error is found. This is because the compiler generator, Yacc stops execution when an error is located.

(2) Enhancement of the NN applications such as invariant character recognition or pattern completion, using various learning rate strategies to improve learning on the backpropagation network and using other NN models to improve and increase the problem size of the TSP.

(3) Graphical display on network characteristics during iteration such as node activation and the three-dimensional display of the change of weights. However, there is a limitation of displaying the NN architecture. Only a small size network can be displayed because of memory limitation and the Mathematica program is slow to display the network architecture.

(4) Combining NN simulation with other disciplines such as control engineering and information processing. In practice, NNs cannot provide the solution working by themselves alone.

(5) Other NN models such as stochastic NN models, dynamic NN models and cascading NN models or combining subnetworks into a substantial NN.

(6) A comparative study between NEUCOMP/NEUCOMP2 and other NN simulation languages. The performance study should give the real indication of its usefulness.

(7) Implementing the parallel compiler on other parallel machines such as distributed parallel processors, i.e. Transputer, Intel Hypercube, etc.

REFERENCES

[Aarts (1989)] Aarts, E. H. L., Boltzmann machines for travelling salesman problems, *European Journal of Operational Research* 39, 1989, pp: 79-95

[Aho *et al.* (1986)] Aho, A. V., Sethi R. and Ullman, J. D., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986

[Aleksander *et al.* (1990)] Aleksander, I. and Morton, H., *An Introduction to Neural Computing*, Chapman and Hall, 1990

[Almasi *et al.* (1989)] Almasi, S. A. and Gottlieb, A., *Highly Parallel Computing*, The Benjamin/Cummings Publ. Co., 1989

[Almassy *et al.* (1990)], Almassy, N., Köhle, K. and Schönbauer, Concepts in Implementation of the Neural Network Language Condela-3, *InfoJapan '90 : Information Technology Harmonizing with Society,* North-Holland, Amsterdam, Netherlands, Vol. 1, 1990, pp: 241-6

[Baase (1988)] Baase, S., *Computer Algorithms: Introduction to Design and Analysis*, 2nd. Ed., Addison-Wesley, July 1988

[Babb (1988)] Babb, R. G. (editor), *Programming Parallel Processors*, Addison-Wesley, 1988

[Beale *et al.* (1990)] Beale, R. and Jackson, T., *Neural Computing, An Introduction*, Adam Hilger, 1990

[Bennett (1990)] Bennett, J. P., *Introduction to Compiling Techniques: A First Course using ANSI C, LEX and YACC,* McGRAW-HILL, 1990

[Bornat (1979)] Bornat R., *Understanding and Writing Compilers,* MacMillan, 1979

[Brause (1989)] Brause, R., Neural Network Simulation using INES, *Tools for AI Architectures, Languages and Algorithms*, IEEE Workshop, 1989, pp: 556-61

[Brawer (1989)] Brawer, S., *Introduction to Parallel Programming*, Academic Press, Inc., 1989

[Carpenter *et al.* (1988)] Carpenter, G. A. and Grossberg, S., The ART of adaptive Pattern Recognition, *IEEE Computer*, Vol. 21, No. 3, Mar. 1988

[DasGupta *et al.* (1990)] DasGupta, S. and Roy, K., Description Language for a Neural Network Architecture, *Intelligent Autonomous Systems, 2nd. International Conference*, Vol. 1, 1990, pp: 294-304

[Dayhoff (1990)] Dayhoff, J. E., *Neural Network Architecture: An Introduction*, Van Nostrand Reinhold, N.Y, 1990

[Feldman *et al.* (1988)] Feldman, J. A., Fanty, M. A. and Goddard, N. H, Computing with Structured Neural Networks, *IEEE Computers*, 21, Mar 1988

[Ford (1990)] Ford, N. J., *Computer Programming Languages : A Comparative Introduction*, ELLIS HORWOOD, 1990.

[Forrest *et al.* (1987)] Forrest, B. M., Roweth, D., Stroud, N., Wallance, D. J. and Wilson, G. V., Implementing Neural Network Models on Parallel Computers, *The Computer Journal*, Vol. 30, No. 5, 1987, pp: 413-419

[Forrest *et al.* (1988)] Forrest, B. M., Roweth, D., Stroud, N., Wallace, D. J. and Wilson, G. V., Neural Network models, *Parallel Computing* 8, North-Holland, 1988, pp: 71-83

[Freisleben *et al.* (1991)] Freisleben, B. and Schilte, M., A Combined Clustering and Parallel Optimization Approach to the Travelling Salesman Problem, *International Conference on Parallel Processing*, Vol.3, 1991, pp: 310 - 311

[Fujimoto (1992)] Fujimoto, Y., Massively Parallel Architectures for Large Scale Neural Network Simulations, *IEEE Transactions on Neural Networks*, Vol.3, No.6, Nov.1992

[Haykin (1994)] Haykin, S., *Neural Networks: A Comprehensive Foundation*, Macmillan Publ. Co., 1994

[Hecht-Nielsen (1987)] Hecht-Nielsen, R., Counterpropagation networks, *Applied Optics*, Vol. 26, Dec. 1987, pp: 4979-4984

[Hecht-Nielsen (1988)] Hecht-Nielsen, R., Applications of Counterpropagation Networks, *Neural Networks*, Vol. 1, 1988, pp: 131-139

[Hecht-Nielsen (1989)] Hecht-Nielsen, R., *Neurocomputing*, Addison-Wesley, 1989

[Hopfield (1982)] Hopfield, J. J., Neural networks and physical systems with emergent collective computational abilities, *Proc. Natl. Acad. Sci.*, USA, Biophysics, 1982, pp: 2554 - 2558

[Hopfield *et al.* (1985)] Hopfield, J. J. and Tank, D. W., "Neural" Computation of Decisions in Optimization problems, *Biological Cybernetics*, Vol. 52, Spring-Verlag, 1985, pp: 141-152

[Horowitz (1985)] Horowitz, E., *Programming Languages: A Grand Tour*, 2nd. Ed., Computer Science Press, 1985

[Hu (1991)] Hu, D. S., An Object-Oriented Neural Network Language, *IEEE International Joint Conference on Neural Networks*, Vol. 2, IEEE, NY, USA, 1991, pp: 1606-11

[Hush *et al.* (1993)] Hush, D. R. and Horne, B. G., Progress in Supervised Neural Networks: What's New Since Lip, *IEEE Signal Processing Magazine*, Jan. 1993

[Hwang *et al.* (1984)] Hwang, K. and Briggs, F. A., *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984

[Johnson (1978)] Johnson S. C., *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, New Jersey, July 1978

[Jones *et al.* (1992)] Jones, W. T., Vachha, R. K. and Kulshrestha, A. P., DENDRITE: A System for Visual Interpretation of Neural Network Data, *IEEE SOUTHEASTCON*, Vol. 2, 1992, pp: 638-641

[Karp (1977)] Karp, R. M., Probabilistic analysis of partitioning algorithms for the travelling-salesman problem in the plane, *Mathematics of operations research*, Vol.2, No. 3, Aug. 1977

[Kernighan *et al.* (1980)] Kernighan B. W. and Ritchie D. M., *The C programming language*, 2nd. Ed., Bell Laboratories, Murray Hill, New Jersey, Prentice-Hall, 1980

[Kirkpatrick *et al.* (1983)] Kirkpatrick, S., Gelatt, J. R. and Vecchi, M. P., Optimization by Simulated Annealing, *Science*, Vol. 220, No. 4598, May 1983, pp: 671 - 679

[Kohonen (1982)] Kohonen, T., Self-organized formation of topologically correct feature maps, *Biological Cybernetics*, 43, 1982, pp: 59-69

[Koopman *et al.* (1990)] Koopman, P. W. M., Rutten, L. M. W. J., van Eekelen, M. C. J. D. and Plasmeijer, M. J., Functional Descriptions of Neural Networks, *International Neural Networks Conference*, Vol. 2, 1990, pp: 701- 704

[Korb *et al.* (1989)] Korb, T. and Zell, A., A Declarative Neural Network Description Language, *Microprocessing and Microprogramming*, Vol. 27, North-Holland, 1989, pp: 181-188

[Korn (1989)] Korn , G. A., A New Environment for Interactive Neural Network Experiments, *Neural Networks*, Vol. 2, Pergamon Press plc, 1989, pp: 229-237

[Korn (1991a)] Korn , G. A., Design of function-generating mapping networks by iteractive neural-network simulation, *Mathematics and Computers in Simulation*, Vol. 33, North-Holland, 1991, pp: 23-31

[Korn (1991b)] Korn , G. A., *Neural Network Experiments on Personal Computers and Workstations*, A Bradford Book, The MIT Press, 1991

[Kung (1993)] Kung, S. Y., *Digital Neural Networks*, PTR Prentice Hall, Eaglewood Cliffs, New Jersey, 1993

[Lafferty *et al.* (1993)] Lafferty, E. L., Prelle, M. J., Michaud, M. C. and Goethert, J. B., *Parallel Computing: An Introduction*, NDC, New Jersey, USA, 1993

[Lang *et al.* (1988)] Lang, K. J. and Witbrock, M. J., Learning to Tell Two Spirals Apart, *Proceedings of the*

Connectionist Models, Summer School, Morgan-Kaufman, 1988, pp: 52 - 59

[Leighton *et al.* (1992)] Leighton, R. and Wieland A., *The Aspirin/MIGRANES software Tools User's Manual*, MITRE Corporation, Washington, 1992

[Lemone (1992)] Lemone, K. A., *Design of Compilers : Technique of Programming Language Translation*, CRC Press, 1992

[Lesk *et al.* (1975)] Lesk, M. E. and Schmidt, E., *Lex - A Lexical Analyser Generator*, Bell Laboratories, Murray Hill, New Jersey, July 1975

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

[Maeder (1991)] Maeder, R. E., *Programming in Mathematica*, 2nd. Ed., Addison-Wesley, 1991

[Männer *et al.* (1989)] Männer R., Horner, H., Hauser, R. and Genthner, A., Multiprocessor Simulation of Neural Networks with NERV, *SUPERCOMPUTING* 89, ACM CONF, 1989, pp: 457 - 465

[Maren *et al.* (1990)] Maren, A. J., Harston, C. T. and Pap, R.M., *Handbook of Neural Computing Applications*, Academic Press, San Diego, California, 1990

[McBryan (1989)] McBryan, O. A., Overview of Current Developments in Parallel Architectures, In *Parallel Supercomputing: Methods, Algorithms and Applications*, Edited by Corey, G. F., John Wiley, 1989

[McClelland *et al.* (1988)] McClelland, J. L. and Rumelhart, D. E., *Exploration in Parallel Distributed Processing*

A Handbook of Models, Programs and Exercise, The MIT Press, 1988

[Mohd-Saman *et al.* (1993)] Mohd-Saman, M. Y. and Evans, D. J., Investigation of a Set of Bernstein Test for the Detection of Loop Parallelization, *Parallel Computing* 19, 1993, pp: 197 - 207

[Müller *et al.* (1990)] Müller, B. and Reinhardt, J., *Neural Network An Introduction*, Physics of Neural Network, Springer-Verlag, 1990

[Myler *et al.* (1992)] Myler, H. R., Weeks, A. R., Gillis, R. K. and Hall, G. W., Object-oriented neural simulation tools for a hypercube parallel machine, *Neurocomputing*, Vol. 4, Part 5, 1992, pp: 235-248

[Nelson *et al.* (1991)] Nelson, M. M. and Illingworth, W. T., *A Practical Guide to Neural Nets*, Addison-Wesley, 1991

[Nijhuis *et al.* (1989)] Nijhuis J., Spaanenburg, L. and Warkowski, F., Structure and Application of NNSIM: a general-purpose Neural Network SIMulator, *Microprocessing and Microprogramming*, Vol.27, North-Holland, 1989, pp:189-194

[Osterhaug (1989)] Osterhaug, A., *Guide to Parallel programming*, 2nd. Edition, Sequent Computer Systems, Inc., 1989

[Padua *et al.* (1986)] Padua, D. A. and Wolfe, M. J., Advanced Compiler Optimizations for Supercomputers, *Communications of the ACM*, Vol. 29, No. 12, 1986, pp: 1184 - 1202

[Paik *et al.* (1987)] Paik, E., Gungner, D. and Skrzypek, J., UCLA SFINX: A Neural Network Simulation

Environment, *IEEE First International Conference on Neural Networks*, Vol. 3, 1987, pp: 367 - 375

[Panetsos *et al.* (1993)] Panetsos, F., Alonso, J., Barja, E., Isasi, P. and Olmedo, V., NSL: A language for neural network simulation, *Microprocessing and Microprogramming*, Vol. 36, 1993, pp: 127-139

[Perkel (1976)] Perkel, D. H., A Computer Program for Simulating a Network of Interacting Neurons, *Computers and Biomedical Research*, Vol. 9, 1976, pp: 31-43

[Peterson *et al.* (1989)] Peterson, C. and Söderberg, B., A New Method for Mapping Optimization problems onto Neural Networks, *International Journal of Neural Systems*, Vol. 1, No. 1, World Scientific Publishing Co., 1989, pp: 3 - 22

[Polychronopoulos (1988)] Polychronopoulos, C. D., *Parallel programming and Compilers*, Kluwer Academics Publ., 1988

[Platt (1988)] Platt, J. C., Constrained Differential Optimization, *American Institute of Physics*, 1988

[Recce *et al.* (1992)] Recce, M. L., Rocha, P. V. and Treleaven, P. C., Neural Network Programming Environments, In *Artificial Neural Networks*, Vol. 2, Edited by Aleksander, I. and Taylor, J., Elsevier Science Publishers B. V., 1992

[Rumelhart *et al.* (1986)] Rumelhart, D. E., Hinton, G. E. and Williams, R.J., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. I, MIT Press, 1986

[Sadja *et al.* (1992)] Sadja, P., Sakai, K. and Finkel, L. H., NEXUS: A Tool for Simulating large-scale Hybrid

Neural Networks, *Summer Computer Simulation Conference*, 1992, pp: 72-76

[Sanossian *et al.* (1991)] Sanossian, H. Y. Y. and Evans, D. J., An Acceleration method for the backpropagation Learning Algorithmn, *Proceeding of the Neuro-Nimes, Forth International Conference on Neural Networks and their Applications*, Nimes-France, 1991, pp: 377-385

[Sanossian (1992)] Sanossian, H. Y. Y., *The study of Artificial Neural Networks and their learning strategies*, Ph.D. thesis, Loughborough University of Technology, 1992

[Shumsheruddin (1992)] Shumsheruddin, D., The Neural Network Paradigm, In *Advanced Topics in Computer Series, Advances in Parallel Algorithms*, Edited by Kronsjo, L. and Shumsheruddin, D., Blackwell Scientific Publication, 1992, pp: 66 - 84

[Siberschatz (1991)] Siberschatz, A., *Operating System Concepts*, Addison-Wesley, 1991

[Simpson (1990)] Simpson, P. K., Artificial Neural Systems, Foundation, Paradigms, Applications and Implementations, *Neural Networks: Research and Applications*, Pergamon Press, 1990

[Springer *et al.* (1989)] Springer, G. and Friedman, D. P., *Scheme and The Art of Programming*, The MIT Press, 1989

[Tarr *et al.* (1992)] Tarr, G., Priddy, K. and Rogers, S., NeuralGraphics: A general purpose environment for neural network simulation, *Proceeding of the SPIE, Applications of Artifical Neural Networks III*, Vol. 1709, Part 2, 1992, pp: 1047-1056

[Tucker (1986)] Tucker, A. B., *Programming Languages*, 2nd. Ed., McGraw-Hill, 1986

[Turban (1993)] Turban, E., *Decision Support and Expert Systems: Management Support Systems*, MacMillan Publ. Co., 1993

[Vellacott (1991)] Vellacott O. R., ANNECS: A Neural Network Compiler and Simulation, *IEEE Neural Networks - International Joint Conference*, Vol. 2, 1991

[Wacholder *et al.* (1989)] Wacholder, E., Han, J. and Mann, R. C., A Neural Network Algorithm for the Multiple Travelling Salesmen Problem, *Biological Cybernetics*, Vol. 61, Springer-Verlag, 1989, pp: 11-19

[Wasserman (1989)] Wasserman, P. D., *Neural Computing: Theory and Practice*, Van Nostrand Reinhold, New York, 1989

[Wilson *et al.* (1988)] Wilson, G. V. and Pawley, G. S., On the Stability of the Travelling Salesman Problem Algorithm of Hopfield and Tank, *Biological Cybernetics*, Vol. 58, 1988, pp: 63-70

[Wilson *et al.* (1993)] Wilson, L. B. and Clark, R. G., *Comparative Programming Languages*, 2nd. Ed., Addison-Wesley, 1993

[Wirth (1976)] Wirth, N., *Algorithms + Data Structures = Programs*, Prentice-Hall, Inc., 1976

[Wolfram (1991)] Wolfram, S., *Mathematica: A System for Doing Mathematics by Computer*, 2nd. Ed., Addison-Wesley, 1991

[Zell *et al.* (1991)] Zell, A., Mache, N., Sommer, T. and Korb, T., Recent Developments of the SNNS Neural Network Simulator, *SPIE PROC, Applications of Artificial Neural Networks II*, Vol. 1469, Part 1, 1991, pp: 708-718

[Zima *et al.* (1990)] Zima, H. and Chapman, B., *Supercompilers for Parallel and Vectors Computers*, ACM Press, Addison-Wesley, 1990

APPENDIX A

BNF SPECIFICATIONS
OF
THE NEUCOMP/NEUCOMP2 LANGUAGE

```
program :   program_heading
            identifier_declarations
            main_declaration
            subprogram_declarations ;
program_heading : NEURALNET IDENTIFIER ;
identifier_declarations : /* none */ | declaration_list ;
main_declaration :  MAINPROGRAM block_statement ;
subprogram_declarations : /* no subprogram */
          |  subprogram_declarations
             subprogram_declaration ;
subprogram_declaration:
     subprogram_head IDENTIFIER formal_parameter
     block_statement ;
formal_parameter : /* none */ | '(' identifier_list ')';
block_statement : identifier_declarations
                  statement_list
                  return_statement ';';
return_statement :    END /* only procedure */
                    | RETURN variable;
subprogram_head : PROC | FUNC ;
declaration_list : declaration
                 | declaration_list declaration;
declaration : type identifier_list ';' ;
type : INT | REAL | STRING | FILES ;
identifier_list:   identifier
                  |identifier_list ',' identifier ;
identifier : IDENTIFIER
       | IDENTIFIER '=' NUMBER
       | IDENTIFIER '[' numORident ']'
       | IDENTIFIER '[' numORident ',' numORident ']';
numORident : NUMBER | IDENTIFIER ;
statement_list :   statement
                 | statement_list ';' statement ;
statement :  assignment_statement
          | while_statement
          | repeat_statement
          | open_statement
          | case_statement
```

```
          | close_statement
          | read_statement
          | print_statement
          | call_statement
          | pattern_statement
          | train_statement
          | for_statement
          | if_statement
          | break_statement ;
break_statement : BREAK ;
open_statement :  OPENREAD read_parameter
               | OPENWRITE write_parameter ;
read_parameter : '(' IDENTIFIER ',' text_ident ')' ;
text_ident  : TEXT | IDENTIFIER ;
write_parameter : '(' IDENTIFIER ',' text_ident ')';
close_statement : CLOSEFILE '(' IDENTIFIER ')';
pattern_statement : EPOCH statement_list END ;
call_statement : CALL IDENTIFIER use_parameter ;
          | PARALLEL* IDENTIFIER use_parameter ;
use_parameter : /* none */| '(' expression_list ')' ;
case_statement : CASE variable OF case_list END ;
case_list : case_condition
          | case_list case_condition ;
case_condition : NUMBER ':' statement ';' ;
for_statement :
        FOR '(' variable '=' expression ',' expression ')'
        statement_list
        ENDFOR ;
if_statement : IF '(' logical_expression  ')'
               statement_list else_statement;
else_statement :  ENDIF |  ELSE statement_list ENDIF ;
logical_expression : logical_AND_expression
     | logical_expression OR logical_AND_expression;
logical_AND_expression : equility_expression
     | logical_AND_expression AND  equility_expression;
```

* applicable to NEUCOMP2

```
equility_expression : relational_expression
     | equility_expression EQ  relational_expression ;
relational_expression : expression
     | relational_expression LT expression
     | relational_expression GT expression
     | relational_expression LE expression
     | relational_expression GE expression
     | relational_expression NE expression;
assignment_statement : variable assigntype expression ;
assigntype : '+=' | '*=' | '-=' | '/=' | '=';
variable : ROW
     | I
     | J
     | CLOCK
     | CYCLE
     | NPATTERN
     | NPROCS*
     | IDENTIFIER status
     | IDENTIFIER '[' expression ']'
     | IDENTIFIER '[' expression ',' expression ']'
     | ROW ;
expression : expression '+' term
     | expression '-' term | term ;
term : term '*' factor
     | term '/' factor
     | term '.' factor
     | factor ;
factor : function_reference
     | MINUS expression
     | '(' expression ')'
     | NUMBER
     | variable;
status : /* none */ | '@' | '#' | '>' | '<';
function_reference :  SIGMOID '(' expression ')'
     | ABS '(' expression ')'
     | DISTANCE '(' expression ',' expression ')'
     | EXP '(' expression ')'
```

* applicable to NEUCOMP2

```
      | GRBH '(' expression_list ')'
      | IDENTIFIER '(' actual_parameter ')' ;
      | LOG '(' expression ')'
      | RAND '(' expression ')'
      | RAND1 '(' expression ')'
      | SIGMOID '(' expression ')' ;
      | SUMALL '(' expression ')'
      | SQR '(' expression ')'
      | SQRT '(' expression ')'
actual_parameter :  /* none */ | expression_list ;
expression_list : expression
      | expression_list ',' expression;
print_statement : print  | printfile ;
print : PRINT '(' print_items ')' ;
print_items : TEXT | print_list;
printfile : PRINTFILE '(' IDENTIFIER ',' print_items ')';
print_list : print_id | print_list ',' print_id;
print_id : TEXT ',' expression | expression;
read_statement : read | readfile ;
read : READ '(' read_list ')' ;
readfile : READFILE '(' IDENTIFIER ',' read_list ')' ;
read_list : read_id | read_list ',' read_id;
read_id : TEXT ',' variable| variable
while_statement : WHILE '(' logical_expression ')' DO
     statement_list  ENDWHILE ;
repeat_statement :  REPEAT statement_list
     UNTIL '(' logical_expression ')' ;
train_statement :  TRAINING  statement_list END;
```

APPENDIX B

USER GUIDE

A brief description of how to use the NEUCOMP/NEUCOMP2 language is now presented. NEUCOMP is a sequential compiler running on a UNIX operating system and NEUCOMP2 is a parallel compiler running on a shared memory parallel machine, i.e. SEQUENT Balance. Explanations given below are applicable to both compilers unless stated.

The structure of the language is as follows :-

```
NEURALNET program_name
identifier_declarations (global use)

MAINPROGRAM
identifier_declarations (local use)
statement_list
END;

subprogram_declarations
```

The first line is called program heading which has to be included. *NEURALNET* is the reserved word and *program_name* is a variable name that must be given. Names are made up of alphabets, digits or underscore ('_') but the first character must be alphabetic. A single alphabet is a valid name but a combination of characters improve readability. The above name must be unique which cannot be used in the simulation program. It must be in a lower-case letter. Capital letters are the reserved word.

Program heading is used to give a name to the simulation program. *Identifier_declarations* (global and local uses) are declaration sections. MAINPROGRAM ... END is the body of the program. *Statement-list* can be a single statement or more than one. If more than one statement is used, they are separated by a semicolon, i.e. ';' and the last statement has no semicolon. *Subprogram_declarations* are declarations of one or more subprograms of types procedure or function.

DATA TYPES

There are four data types :-

INT	integer
REAL	single-precision floating point
STRING	holding up to 10 characters useful as a file name
FILES	file type

CONSTANTS

A real constant contains a decimal point, i.e. 3.14. An integer constant has no decimal point. A string constant is written as any combination of characters within " ", i.e. "inputfile".

DECLARATIONS

Variables can be reserved words or defined by the user. All reserve word variables are written in capital letters. They are of type integer. These variables are I, J, ROW, CYCLE, NPATTERN and NPROCS*. Variables defined by the user must be declared before use. They are declared in the declaration section (*identifier_declarations*) either as global or local to the body of the main program or a subprogram. A local variable is only applicable to where it is declared.

A variable is specified with a type. A type may contain a list of one or more variables as shown below :-

```
INT seed, nocycle;
REAL dist, lrate;
STRING filename1, filename2;
```

* applicable to NEUCOMP2 which is used to specify number of processors required.

A variable of type integer and real can be declared as a scalar, vector or matrix. They are used in mathematical operations only. From the above example, integer and real variables are scalars. A scalar variable can also be initialised in its declaration, as shown below :-

```
INT seed = 1000;
```

Vector and matrix variables are an array of one and two-dimensional sizes respectively. The size can be an integer constant or a variable. For a variable, its type need not be declared. Its size will be assigned at run time. This makes it more like a dynamic data structure.

Examples of vector and matrix declarations are written as :-

```
REAL
  layer1[n1],layer2[n2],layer3[n3],
  weight1[n2,n1],weight2[n3,n2];
```

A matrix declaration for the connection weights, i.e. the connections between the first and second layers, its row size must be the size of the second layer and its column must be the size of the first layer.

ARITHMETIC OPERATORS

There are five types of arithmetic operators :-

'+' , '-' , '*', '/' and '.'

The first four operators are similar to any high-level language. The fifth operator stands for 'dot product'. It is used for two vector multiplications which yields a scalar result. In terms of precedences, it is in line with the operators '*' and '/'. All mathematical

operations involve real and integer types. For example, the calculation for the error measure in the backpropagation simulation is written as :-

```
error = layer3 - target;
enormsqr = 0.5 * (error . error);
```

where *error*, *layer3* and *target* are vectors and *enormsqr* is a scalar.

RELATIONAL OPERATORS AND LOGICAL OPERATORS

The relational operators are :-

GT, LT, GE and LE

In the C language, they stand for symbols '>', '<', '≥' and '≤'. NEUCOMP/NEUCOMP2 prefers to use a word instead of a symbol in order to maintain readability of the program.

Logical operators are written as AND and OR. They are represented in C as '&&' and '||' respectively. The precedences of the operators follow the C language.

STATEMENTS

The statements available for writing any simulation program are :-

assignment statements
conditional statements
loop statements
break statement
subprogram statements
input-output statements

ASSIGNMENT STATEMENT

An assignment statement is presented as follows :-

variable assigntype expression

where *variable* can be a scalar, vector or matrix variable, *assigntype* is a mathematical operator of types '=','+=', '-=', '*=' or '/=' and *expression* can be a variable or variables in a mathematical expression. Their data types must be compatible, i.e. if *variable* is an integer, *expression* must be of type integer. However, an integer variable that is in *expression* is automatically converted into real if *variable* is real. The use of arithmetic operators on symbol '=' is to compress an assignment, i.e. a = a + 1 and b = b*2 are compressed to a += 1 and b *= 2 respectively.

An assignment statement is divided into 3 types - a scalar assignment, vector assignment and matrix assignment. In a scalar assignment, *variable* is a scalar and *expression* can be a digit, scalar variable or mathematical expression which yields a scalar result, i.e. dot product between two vectors.

In a vector assignment, *variable* is a vector and *expression* can be one of the following rules :-

(1) Expression of type scalar.
(2) Vector variable.
(3) Matrix-vector multiplication. The column size of the matrix must be equal to the size of the vector. The result is a vector of size equal to the row size of the matrix.
(4) Function - a built-in function or a user-defined function. Its argument can be a scalar expression or a mathematical expression which yields a vector result. The built-in functions are shown in **table B1.**

(5) Matrix variable. The row*column of the matrix size must be equal to *variable*. An example of a valid assignment is :-

```
READ layer1[100], input[10,10];
...
layer1 = input;
...
```

(6) Matrix variable followed by a special character as listed below :-

@	all elements of a matrix on a specific row
#	all elements of a matrix on a specific column

The specified row or column depends on the status of a reserved word ROW. Both variables become a vector. An example of a valid assignment is :-

```
REAL layer1[10], pattern[5,10];
  ...
EPOCH
  layer1 = pattern@;
  ...
END;
```

EPOCH ... END is a loop statement where each iteration is assigned to ROW. ROW represents the current row of *pattern*.

(7) Mathematical expression of the above rules except rule (5).

In a matrix assignment, *variable* is a matrix and *expression* can be one of the following rules :-

(1) Expression of type scalar.
(2) Matrix variable.
(3) Function - a built-in function (**table B1**) or a user-defined function. Its argument can be a scalar expression or a mathematical expression which yields a matrix result.
(4) Vector variable. The size of the vector must be equal to the size of row*column of *variable*. An example of valid assignment is :-

```
REAL layer3[100], output[10,10];
...
output = layer3;
PRINT("%f ",output);
...
```

The print statement, PRINT, prints the output data in two-dimensional form. The format '%f' denotes a data format of type real.

(5) Outer-product of two vectors yields a matrix with its row size equal to the size of the first vector and its column size is equal to the size of the second vector.
(6) Matrix transpose. It is written as matrix&. **Appendix C** has shown the use of matrix transpose.
(7) Mathematical expression of the above rules except rule (4).

Note on *variable*:-

(1) If *variable* is a vector followed by '@' or '#', it becomes a scalar variable and similarly, a matrix becomes a vector where its size depends on which symbol is used. For this case the above rules are applicable.

(2) If *variable* is a vector followed by > then *expression* must be a vector. The index of *expression* where its element is the maximum, is assigned to ROW. **Appendix E** shows its use.

(3) If *variable* is a vector followed by < then *expression* must be a vector. The index of *expression* where its element is the minimum, is assigned to ROW. **Appendices D** and **F** show its use.

(4) If *variable* and *expression* are of type vector, their sizes must be equal.

(5) If *variable* and *expression* are of type matrix, the sizes of rows and columns must be equal or vice-versa if the matrix is transposed.

CONDITIONAL STATEMENTS

There are two types of conditional statements to express decisions. The 'if-statement' is used to test a single decision and 'case-statement' is used to test multiple decisions. Variables involved in these statements are of type scalar. The if-statement is written as :-

```
IF ( logical-expression )
     statement-list
ENDIF;
```

or

```
IF ( logical-expression )
     statement-list
ELSE
     statement-list
ENDIF;
```

where *logical-expression* involves either a logical operator or relational operator or both. When it yields true the first *statement-list* is evaluated. If it uses 'ELSE' then,

when *logical-expression* yields false, the second *statement-list* is applied. For example, termination of the error measure (**Appendix C**) is written as :-

```
IF (enormsqr LT 0.01)
     PRINT("convergence fulfil");
     PRINT("stop iteration")
ENDIF;
```

The case-statement is written as :-

```
CASE variable OF
  integer constant1 : statement1;
    ...
  integer constantn-1 : statementn-1;
  integer constantn : statementn;
END;
```

where *variable* is of type integer scalar. A single statement is allowed for each integer constant. As an example :-

```
CASE type OF
   10 : a += 1;
   15 : a *= 2;
   31 : a /= 3;
END;
```

where *type* contains any integer value. When it matches one of the above integer constants, the statement is executed.

LOOP STATEMENTS

There are five types of loop statements :-

```
for-statement
while-statement
repeat-statement
training-statement
pattern-statement
```

The for-statement is written as

```
FOR (variable = expression1, expression2)
     statement-list
ENDFOR;
```

where *variable* and *expression* are of scalar type integer. They can be reserved words or defined by the user. The iteration begins on *expression1* and increments one until *expression2*-1. As an example, to update the weights in the neighbourhood of the Kohonen network (**Appendix D**), it is written as :-

```
FOR (i = r1,r2 + 1)
  FOR (j = c1,c2 + 1)
    ROW = i*grid+j;
    weight@ += lrate*(layer1-weight@ )
  ENDFOR
ENDFOR;
```

The while-statement is written as :-

```
WHILE (logical-expression) DO
     statement-list
ENDWHILE;
```

If *logical-expression* is evaluated true, *statement-list* is executed and *logical-expression* is re-evaluated. This cycle continues until the expression becomes false. However, in the repeat-statement, *statement-list* is executed first then

logical-expression is tested. If false, the loop continues until it is true. This loop is written as :-

```
REPEAT
     statement-list
UNTIL (logical-expression);
```

The loops mentioned above are common to any high-level language. However, 'train-statement' and 'pattern-statement' are special loop statements.

To train the network, it can be done by using the following training loop :-

```
TRAINING
   ...
END;
```

where the statement *TRAINING* contains a reserved word variable of type integer called *CYCLE* which is initially set to 100. It means the number of iterations is 100. However, this value can be changed. The training algorithm is within the loop.

To assign an input layer with a pattern, the following pattern loop is used :-

```
EPOCH
  layer1 = pattern@;
    ...
END;
```

where the statement *EPOCH ... END* contains the loop starting from zero to the pattern size minus one set by an integer variable called *NPATTERN*. Each iteration is assigned to the reserved word variable called ROW. The *NPATTERN* is a reserved word variable which is initially set to one. It means only one pattern is involved in the training operation per cycle. However, this value can be changed.

BREAK STATEMENT

The 'break-statement' is used to exit from the loop other than through *logical-expression*. The word BREAK is included in the loop if exit from the loop at an early stage is necessary. For example, in the backpropagation simulation (**Appendix C**), training-loop is stopped when the sum of error is less than 0.01, as shown below :-

```
TRAINING
  ...
  IF (enormsqr LT 0.01)
     BREAK
  ENDIF;
  ...
END;
```

INPUT-OUTPUT STATEMENTS

Statements for input are READ and READFILE, and for output are PRINT and PRINTFILE. READ and PRINT are an input-output statement from or to the terminal. A READ statement allows a variable of type scalar, vector or matrix to be assigned a value. For example :-

```
READ(seed);
```

where *seed* is a scalar variable of type integer. The value to be assigned must be an integer constant. A text written within " " can be included before that variable. It is written as :-

```
READ("type in seed = ", seed);
```

This is used to display a message before the value is typed. A PRINT statement allows a text or value of a variable to be printed on the terminal. For example :-

```
PRINT("The backpropagation\n");
PRINT("%d",seed);
```

The first statement is to print a text. The character \n is to allow a newline to be printed. The second statement is to print a scalar variable *seed* of type integer. The argument %d is the data format for an integer constant. It is written within " ". Other data formats can be included. For a real, the data format is written as %f and a string is written as %s. The width of the constant can be included as :-

%4d	print an integer, at least 4 characters wide
%4f	print as real, at least 4 characters wide
%.3f	print as real, at least 3 characters after the decimal point
%6.3f	print as real, at least 6 characters wide and 3 after the decimal point

A text can be included in the data format to improve readability. For example :-

```
PRINT("seed = %d\n",seed);
```

More variables can be printed using a single print statement, such as

```
PRINT("seed = %d\n",seed, "learning rate =%f",alpha);
```

A vector or matrix variable can be printed by following the above examples. However, for a vector the values are printed in the same row and for a matrix, the values are arranged in row to column. When using '\n' as an example given below :-

```
PRINT("%f\n",weight);
```

where *weight* is a matrix, all its values are printed line by line.

READFILE and PRINTFILE statements are used for an input-output from and to the specified file. A variable of type file must be declared using data type FILES. For example :-

```
FILES file1, file2;
```

The variables *file1* and *file2* must be connected to a file name using OPENREAD or OPENWRITE statements. OPENREAD is used to connect a variable of type file to a file name to be read. OPENWRITE is used to connect a variable of type file to a file name to be printed. For example :-

```
OPENREAD(file1,"inputfile");
OPENWRITE(file2,"outputfile");
```

The texts within " " are the file names. The name of a file can be replaced by a variable of type string so that the file name can be typed using a read-statement. For example :-

```
FILE2 file1,file2;
STRING inputname, outputname;
...
READ("Type input file : ", inputname);
READ("Type output file : ", outputname);
OPENREAD(file1,inputname);
OPENWRITE(file2,outputname);
```

To read data from or write data into a file, the following statements are used.

```
READFILE(file1, ...);
PRINTFILE(file2, ...);
```

The second arguments for the above statements follow READ and PRINT statements as discussed earlier. However, the statement CLOSEFILE has to be written after the file variable has been used. It breaks the connection between the file variable and the file name. It is written as follows :-

```
CLOSEFILE(file1);
CLOSEFILE(file2);
```

SUBPROGRAM STATEMENTS

There are two types of subprograms, function and procedure. They are used to break a large computing task into smaller tasks. They are declared in the 'subprogram_declarations'. The structure of a subprogram is written as

```
PROC procedure_name argument
identifier_declarations (local use)
statement_list
END;

FUNC function_name argument
identifier_declarations (local use)
statement_list
RETURN variable;
```

They follow the same structure as the main program. Procedure is invoked using a CALL statement and function is invoked through an expression. The type of a return value for this expression is based on a type of a variable after statement 'RETURN'. The use of *argument* is optional. Argument may contain one or more variables written within ( ). The argument in the subprogram acts as a passing parameter to the calling subprogram. The argument in the subprogram need not be declared because

the type depends on the type of argument in the calling subprogram.

A function is used to return a single value via a RETURN statement. However, if more values are needed to be returned, the variable in the argument is written followed by &. This can also be applied to a procedure. Examples of using procedures can be seen in **Appendices C** to **F**. Examples of using functions can be seen in **Appendices G** and **H**. A function can allow some part of an expression in a matrix/vector assignment to be evaluated on the current row or column of a matrix/vector variable. Examples of such functions are *sum0*, *sum1* and *sum2*, found in **Appendices G** and **H**.

PARALLEL PROGRAM

To generate a parallel program, the word CALL is replaced by PARALLEL and then compiled by NEUCOMP2. Only one procedure is allowed to be executed in parallel. Other additional statements to be written in the NEUCOMP2 program are shown below :-

```
MAINPROGRAM
INT time1, time2;
REAL time;

READ("No. of processors : ",NPROCS);
time1 = CLOCK;
PARALLEL training;
time2 = CLOCK;
time = (time2 - time1)/100.0;
PRINT("Training time = %.2f\n",time)
END;
```

The use of the CLOCK is to record the execution time on several processors. Its usefulness is that before running an application, the execution time and speedup of that NN

simulation can be tested on a number of processors within a small cycle. When a proper number of processors have been determined, then actual simulation on the application can begin. This is because the use of many processors does not necessarily mean good performance.

COMMENTS

All characters after // are ignored by the compiler. They are used to make documentation on a program. Comments may appear anywhere and are written in one line.

COMPILATION AND EXECUTION

The simulation program is written using 'vi editor' and the file can be given any name, i.e. filename. The name is similar to a variable name.

There are two steps of compilation. The first step is to compile the source program (i.e. the NEUCOMP/NEUCOMP2 program). When there is no error, the second step is to compile the target program using the C compiler. The first compilation is written as

```
NEUCOMP filename     for a sequential program
NEUCOMP2 filename    for a parallel program,
```

The second compilation is written as

```
CC          for a sequential C compiler
PC          for a parallel C compiler.
```

Execution of the object code can be done using NET.

ABS(x)	$\lvert x\rvert$; x is a scalar, $\lvert x_i\rvert$; is a vector; i = 0 .. m-1, or $\lvert x_{ij}\rvert$; x is a matrix; i = 0 .. m-1 and j = 0 .. n-1
DISTANCE(x,w)	x is a vector; j = 0, n-1 and w is a matrix; i = 0 .. m-1 and j =0 .. n-1. $\sqrt{(\sum_{j=0}^{n-1}(x_j - w_{ij})^2}$;
EXP(x)	e^x ; x is a scalar, e^i ; x is a vector; i = 0 .. m-1, or e^{ij} ; x is a matrix; i = 0 .. m-1 and j = 0 .. n-1
GRBH(α ,r,x)	Gradient Ranged Heuristic method. x is weight or bias derivative, α of type vector with size n and r is the range of type vector with size n-1.
LOG(x)	$\log_{10} x$; x is a scalar $\log_{10} x_i$; x is a vector; i = 0 .. m-1, or $\log_{10} x_{ij}$; x is a matrix; i = 0 .. m-1 and j = 0 .. n-1
RAND(seed)	random number between 0 .. 1. Seed is an integer.
RAND1(seed)	random number between -1 .. 1. Seed is an integer.
SIGMOID(x)	$\frac{1}{1+e^{-x}}$; x is a scalar, $\frac{1}{1+e^{-x_i}}$; x is a vector; i = 0 .. m-1, or $\frac{1}{1+e^{-x_{ij}}}$; x is a matrix; i = 0 .. m-1 and j = 0 .. n-1
SQR(x)	x^2 ; x is a scalar, x_i^2 ; x is a vector; i = 0 .. m-1, or x_{ij}^2 ; x is a matrix; i = 0 .. m-1 and j = 0 .. n-1
SQRT(x)	$\sqrt{x}$; x is a scalar, $\sqrt{x_i}$; x is a vector; i = 0 .. m-1, or $\sqrt{x_{ij}}$; x is a matrix; i = 0 .. m-1 and j = 0 .. n-1
SUMALL(x)	$\sum_{i=0}^{m-1} x_i$; if x is a vector or $\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} x_{ij}$; if x is a matrix

Table B1: List of Built-in functions

GRAPHICAL FEATURES

All graphical displays are shown on a PC using the Mathematica software. The result from the NEUCOMP/NEUCOMP2 simulation program are sent via file transfer.

The type of graphical features that have been implemented so far are :-

(1) Displaying the NN structure.
(2) Plotting the XY-graph.
(3) Plotting (x,y) for data clustering.
(4) Plotting (x,y) for the travelling salesman problem.
(5) Plotting a three-dimensional graph.

Displaying the Neural Network structure

The function 'displaynet' is called from the Mathematica text-based interface. It prompts for the name of a file to be displayed on the respective network. The format for this file must contain the title, number of nodes in each layer and the connections. For example, a three layer network which contains 2 input nodes, 3 hidden nodes and 1 output node, is written as

```
The backpropagation network
{2,3,1}
{{1,2,0},{2,3,0}}
```

In the last line, {{1,2,0},{2,3,0}}, the first set, {1,2,0} means the first layer is connected to the second layer with '0' representing feedforward connection. '1' represents feedforward and feedback connections. Similarly, {2,3,0} means the feedforward connection from the second layer to the third layer. We can add further connections such as a connection from the first layer to the third layer which is written as {1,3,0}.

Similarly, a single layer network can be written as:-

```
The Hopfield network
{10}
{{1,1,1}}
```

Set {10} is a one layer network containing 10 nodes and {{1,1,1}} means the first layer is connected to the same layer with feedforward and feedback connections.

A two layer network with feedforward and feedback connections can be written as:-

```
The ART network
{2,5}
{{1,2,1}}
```

For a network that contains a layer node arranged in a two-dimensional or topological map, it can be written as :-

```
The Kohonen network
{2,{10,10}}
{{1,2,0}}
```

Set {10,10} means the second layer has nodes arranged in 10*10.

Plotting the XY-graph.

The XY-graph is a two-dimensional graph to display a curve of points (x,y). We can plot a single graph or more than one on the same axis. The function 'xygraph' is called from the Mathematica text-based interface. It prompts for the title name of the graph, the name of x-axis and y-axis. One or more file names that contain the co-ordinates to be plotted are needed to be typed in. If no more graphs are required then type the space bar and

return key. The data file format are written as values of x and y.

Plotting (x,y) for data clustering

To display data clustering, there are two types of graphs, and they are 'xycluster' and 'xyspiral'. Both are used to display weights characteristics during training in the Kohonen network with 2 input nodes. The weights that are kept in the file are arranged in pairs, i.e. the first and second input nodes that are connected to the winner node. The first graph is used to study the weights distribution on random numbers between 0 and 1, and the second graph is for separating intertwined spirals.

Plotting (x,y) for the travelling salesman problem

The co-ordinates of the tour can be displayed using 'xyplot'. The data format in the file are arranged as a value of x-axis followed by y-axis. The first pair represents city number one, the second is for the next city to be visited and so on, until the last city is connected to the first city.

Plotting three-dimensional graph

A three-dimensional graph has a co-ordinate in the form (x,y,z). The name of a function is 'xyzplot'. It prompts for the name of a file to be typed in. The file format is arranged in ordered values of x followed by y followed by z. This feature has been used for displaying two intertwined spirals using the backpropagation and Counterpropagation simulation programs. The first two values are for the spirals co-ordinates and the third one is the value of the output node.

APPENDIX C

THE BACKPROPAGATION NETWORK SIMULATION

```
NEURALNET backpropagation

REAL
  layer1[n1],layer2[n2],layer3[n3],
  weight1[n2,n1],weight2[n3,n2],
  oweight1[n2,n1],oweight2[n3,n2],
  cweight1[n2,n1], cweight2[n3,n2],
  dweight1[n2,n1], dweight2[n3,n2],
  bias2[n2],bias3[n3], obias2[n2],
  obias3[n3], cbias2[n2], cbias3[n3],
  delta2[n2], delta3[n3], ddelta2[n2],ddelta3[n3],
  pattern[n4,n1], target[n4,n3], error[n3], enormsqr,
  beta, limit, alpha[sizealpha], range[sizerange];

MAINPROGRAM
  CALL parameters;
  CALL training;
  CALL one_recall
END;

PROC parameters
  INT seed1,seed2,seed3,seed4;
  FILES file1,
        file2;
  STRING inputf,outputf;
  READ("No. of cycle =",CYCLE);
  READ("no. training pattern = ",NPATTERN);
  READ("Termination when limit = ",limit);
  READ("beta =",beta);
  READ("Range alpha, = ",alpha);
  READ("Range derivative,  = ",range);
  READ("Input file from = ",inputf);
  OPENREAD(file1,inputf);
  READ("Target file from = ",outputf);
  OPENREAD(file2,outputf);
  READFILE(file1,pattern);
  READFILE(file2,target);
  CLOSEFILE(file1);
```

```
  CLOSEFILE(file2);
  READ("Seed for weight1 =",seed1);
  READ("Seed for weight2 =",seed2);
  READ("Seed for bias2 =",seed3);
  READ("Seed for bias3 =",seed4);
  weight1 = RAND1(seed1);
  weight2 = RAND1(seed2);
  bias2= RAND1(seed3);
  bias3=RAND1(seed4);
  cweight1 = 0;
  cbias2 = 0;
  cweight2 = 0;
  cbias3 = 0
END;

PROC training
INT nocycle = 0;
TRAINING  // use in serial execution only
  nocycle = nocycle + 1;
  PRINT("No. of cycle = %d ",nocycle);
  dweight1=0;
  ddelta2 = 0;
  dweight2 = 0;
  ddelta3 = 0;
  enormsqr = 0;
  EPOCH
    layer1 = pattern@;
    layer2 = SIGMOID(weight1*layer1+bias2);
    layer3 = SIGMOID(weight2*layer2+bias3);
    error = target@-layer3;
    enormsqr += 0.5*(error.error);
    delta3 = error*layer3*(1-layer3);
    dweight2 += delta3*layer2 ;
    ddelta3 += delta3;
    delta2 = weight2&*delta3*(1-layer2)*layer2;
    dweight1 += delta2*layer1;
    ddelta2 += delta2
  END;
```

```
  PRINT(" enormsqr = %f\n ",enormsqr);
  IF (enormsqr LE limit) BREAK ENDIF;
  oweight1 = weight1;
  weight1 +=  GRBH(alpha,range,dweight1) + beta*cweight1;
  cweight1 = weight1-oweight1;
  obias2 = bias2;
  bias2 += GRBH(alpha,range,ddelta2) + beta*cbias2;
  cbias2 = bias2-obias2;
  oweight2 = weight2;
  weight2 +=  GRBH(alpha,range,dweight2) + beta*cweight2;
  cweight2 = weight2-oweight2; // dw(t) = w(t)-w(t-1);
  obias3 = bias3;
  bias3 += GRBH(alpha,range,ddelta3) + beta*cbias3;
  cbias3 = bias3-obias3 // dd(t) = d(t)-d(t-1)
END
END;

PROC one_recall
INT type;
STRING inputf;
FILES file1;
REPEAT
  READ("test data = ",inputf);
  OPENREAD(file1,inputf);
  READFILE(file1,layer1);
  layer2 = SIGMOID(weight1*layer1+bias2);
  layer3 = SIGMOID(weight2*layer2+bias3);
  PRINT ("%.2f ", layer3);
  READ(" continue ? [1=yes/ 0=no] ",type)
UNTIL (type EQ 0 );
CLOSEFILE(file1)
END;
```

APPENDIX D

THE KOHONEN NETWORK SIMULATION

```
NEURALNET kohonet_net
// input as random number between 0 .. 1
// cluster number: kohfile0, kohfile1, kohfile2, kohfile3
REAL layer1[n1], layer2[n2], weight[n2,n1],
     output[n1], initlrate;
INT  grid,initneighb;

MAINPROGRAM  // --- like main routine  ---
  CALL parameter;
  CALL training
END;

PROC parameter
INT  seed;
FILES fp;

OPENWRITE(fp,"kohfile0");
READ("No. of cycle =",CYCLE);
READ("Initial learning rate = ",initlrate);
READ("Initial neighbourhood = ",initneighb);
READ("Size grid on map = ",grid);
weight = 0.5 + 0.1*RAND(0); // 0 and 1
PRINTFILE(fp,"%.3f ",weight);
CLOSEFILE(fp)
END;

PROC training  // --- like main routine  ---
INT  iter, i,j,r,c,r1,c1,r2,c2,neighb;
REAL lrate;
FILES fp1,fp2,fp3;
OPENWRITE(fp1,"kohfile1");
OPENWRITE(fp2,"kohfile2");
OPENWRITE(fp3,"kohfile3");
neighb = initneighb;
lrate = initlrate;
TRAINING // use for serial execution only
         // train weight vector ... kohonen layer
  iter = iter+ 1;
```

```
  PRINT("cycle %d \n",iter);
  layer1 = RAND(0);
  // get ROW = index of layer2 with its value is minimum
  layer2< = DISTANCE(layer1,weight);
  r = 0;
  REPEAT // converting into two-dim subscribt
    c = ROW - r*grid;
    IF (c GE grid) r = r+ 1 ENDIF
  UNTIL ( c LT grid);
  // (r,c) is the winner node in two-dim
  // get neighbourhoods
  r1 = r-neighb;
  r2 = r+neighb;
  c1 = c-neighb;
  c2 = c+neighb;
  IF ( r1 LT 0 ) r1 = 0 ENDIF;
  IF ( r2 GE grid) r2 = grid - 1 ENDIF;
  IF ( c1 LT 0 ) c1 = 0 ENDIF;
  IF ( c2 GE grid) c2 = grid - 1 ENDIF;
  FOR (i = r1,r2 + 1)
    FOR (j = c1,c2 + 1)
      ROW = i*grid+j;
      weight@ += lrate*(layer1-weight@ )
    ENDFOR
  ENDFOR;
  neighb = initneighb*(1- iter/CYCLE);
  lrate = initlrate*(1- iter/CYCLE);
  CASE iter OF
     1000 :  PRINTFILE(fp1,"%.3f ",weight);
     6000 :  PRINTFILE(fp2,"%.3f ",weight);
     20000:  PRINTFILE(fp3,"%.3f ",weight);
  END
END;// end training
CLOSEFILE(fp1);
CLOSEFILE(fp2);
CLOSEFILE(fp3)
END;
```

APPENDIX E

THE ART1 NETWORK SIMULATION

```
NEURALNET art1

REAL layer1[n1], layer2[n2],
     weightf[n2,n1], oweight[n2,n1],
     weightb[n2,n1], pattern[n4,n1],
     data[widthpattern,widthpattern];
STRING inputf, outputf; // string

MAINPROGRAM  // --- like main routine  ---
  FILES file1;

  READ("read pattern from file =",inputf);
  OPENREAD(file1,inputf);
  READFILE(file1,pattern);
  READ("No. of patterns = ",NPATTERN);

  CALL training;
  CALL all_recall
END;

PROC training
REAL xnorm, znorm, compare,
     vigil=0.99;

weightb=1;
weightf=1;
EPOCH
 layer1 = pattern@;
 oweight = weightf;
 REPEAT
   layer2> = oweight*layer1; // get ROW
   xnorm = layer1.layer1;
   znorm = weightb@.layer1;
   compare = znorm/xnorm;
   IF (compare GT vigil) // adapt weights
     weightf@ = weightb@*layer1/(0.5+(weightb@.layer1));
     weightb@ = weightb@*layer1
   ELSE
```

```
      oweight@ = 0
      // initialise oweight with respects to ROW
      // so that its row will not been selected again.
    ENDIF
 UNTIL (compare GT vigil)
END // end pattern
END;

PROC all_recall
FILES file1;

READ("Recall pattern from file : ",inputf);
OPENREAD(file1,inputf);
EPOCH
  READFILE(file1,layer1);
  PRINT("before\n");
  data = layer1;
  PRINT("%.0f ",data);
  layer2> = weightf*layer1;
  PRINT("recall pattern\n");
  data = weightb@;
  PRINT("%.0f ",data)
END
END;
```

APPENDIX F

THE COUNTERPROPAGATION NETWORK SIMULATION

```
NEURALNET counterpropagation
REAL layer1[n1], layer2[n2],
     layer3[n4], tlayer[n4], error[n4],
     weight1[n2,n1], weight2[n4,n2],
     pattern[n4,n1], target[n4,n4],
     data[pattwidth,pattwidth], initlrate, brate;
INT  initneighb;
STRING inputf,outputf;

MAINPROGRAM  // --- like main routine  ---
  FILES fp1,fp2;

  READ("input file = ",inputf);
  READ("target file = ",outputf);
  OPENREAD(fp1,inputf);
  OPENREAD(fp2,outputf);
  READFILE(fp1,pattern);
  READFILE(fp2,target);
  READ("Number of pattern =",NPATTERN);
  READ("Number of cycle =",CYCLE);
  READ("Initial learning rate =",initlrate);
  READ("Initial neighbourhood =",initneighb);
  READ("brate  =",brate);
  READ("Seed for random number =",seed);
  weight1 = RAND(seed);
  weight2 = 0.;
  CALL training;
  CALL all_recall
END;

PROC training
  REAL lrate, brate = 1.;
  INT patt,nocycle, neighb1, neighb2, neighb;

  lrate = initlrate;
  neighb = initneighb;
  TRAINING
    nocycle = nocycle + 1;
```

```
    patt = 0;
    EPOCH
      patt = patt + 1;
      layer1 = pattern@;
      tlayer = target@;
      layer2< = DISTANCE(layer1,weight1);
      neighb1 = ROW-neighb;
      neighb2 = ROW+neighb;
      IF (neighb1 LT 0) neighb1 = 0 ENDIF;
      IF (neighb2 GE n2)
          neighb2 = n2 - 1 ENDIF;
      FOR (ROW=neighb1,neighb2 + 1)
        weight1@ = weight1@ + lrate*(layer1-weight1@);
        //  ... grossberg layer
        layer3 = weight2#;
        error = tlayer -layer3;
        weight2# = weight2# + brate*error
      ENDFOR
    END;
    neighb = initneighb*(1- nocycle/CYCLE);
    lrate = initlrate*(1- nocycle/CYCLE)
  END // training
END;

PROC all_recall
FILES fp1;

  READ("input file as test data = ",inputf);
  OPENREAD(fp1,inputf);
  EPOCH
    READFILE(fp1,layer1);
    layer2< = DISTANCE(layer1,weight1);
    layer3 = weight2#;
    PRINT("%.0f ",layer3)
  END
END;
```

APPENDIX G

THE HOPFIELD NETWORK SIMULATION

```
NEURALNET travelsales1
// hopfield-tank on solving TSP
REAL
  xcity[n],ycity[n], dist[n,n],
  node[n,n], u[n,n],
  deltat, lambda1, lambda2, lambda3,
  tau, temp = 0.03,
  energy, energy0, energy1, energy2, energy3;

MAINPROGRAM

  CALL parameters;
  CALL training;
  CALL valid_city
END;

PROC parameters
INT i,j,seed;
FILES file1; // input file
STRING inputf;
READ("Input file from = ",inputf);
OPENREAD(file1,inputf);
FOR (i=0,n)
  READFILE(file1,xcity[i],ycity[i]);
  dist[i,i] = 0.
ENDFOR;
CLOSEFILE(file1);
FOR (i=0,n - 1)
  FOR (j= i + 1 ,n)
    dist[i,j] = SQRT( SQR(xcity[i]-
                  xcity[j])+SQR(ycity[i]-ycity[j]) );
    dist[j,i] = dist[i,j]
  ENDFOR
ENDFOR;
READ("No. of cycle =",CYCLE);
READ("Time increment =",deltat);
READ("Damping factor, tau =",tau);
READ("lambda1 = ",lambda1);
```

```
READ("lambda2 = ",lambda2);
READ("lambda3 = ",lambda3);
READ("Starting seed for random generator = ",seed);
READ("Temperature = ",temp);
u = -0.5*temp*LOG(n)*(1 + 0.1*RAND1(seed) );
node = SIGMOID(2*u/temp)
END;

PROC training
INT ianneal,nocycle;
REAL e0,e1,e2,e3, saturation;

TRAINING  // use in serial execution only
  nocycle = nocycle + 1 ;
  PRINT("no. cycle = %d \n",nocycle);
  FOR (ianneal =0,10)
    e3 = SUMALL(node);
    e3 = e3 - n;
    u += deltat *
          ( -1./tau*u - sum0(e0&,I,J)
            - lambda1* ( sum1(e1&,I) - node)
            - lambda2*sum2(e2&,I,J)
            - lambda3*e3);
    energy0 += e0*SUMALL(node);
    energy1 += (e1 - node)*SUMALL(node);
    energy2 += e2*SUMALL(node);
    energy0 *= 0.5;
    energy1 *= 0.5;
    energy2 *= 0.5;
    energy3 = 0.5*e3*e3;
    energy = energy0 + lambda1*energy1
               + lambda2*energy2 + lambda3*energy3;
    lambda1 += deltat*energy1;
    lambda2 += deltat*energy2;
    lambda3 += deltat*energy3
    node = SIGMOID(2*u/temp)
  ENDFOR;
  saturation = SUMALL ( node*node );
```

```
  saturation /= n;
  IF ( saturation GT .95 ) BREAK ENDIF
END;
PRINT("%f",node)
END;

FUNC sum0(e0,i,j)
INT k,add,minus;
  e0 = 0.;
  FOR (k =0,n)
    IF (j + 1 EQ n ) add = 0
    ELSE add = j + 1
    ENDIF;
    IF (j - 1 LT 0 ) minus = n - 1
    ELSE minus = j - 1
    ENDIF;
    e0 += dist[i,k]*(node[k,add]+node[k,minus])
  ENDFOR
RETURN e0;

FUNC sum1(e1,i)
  INT k;
  e1 = 0.;
  FOR (k=0,n)
    e1 += node[i,k]
  ENDFOR
RETURN e1;

FUNC sum2(e2,i,j)
  INT k;
  REAL sum;
  FOR ( k=0,n)
    sum += node[k,j]
  ENDFOR;
  e2 = sum - node[i,j]
RETURN e2;
```

```
PROC valid_city
INT a,i,j,k,city[n];
REAL scalar;

// Determine the path
FOR (a=0,n) // every location
 scalar= 0;
 FOR (i=0,n) // find valid city
   IF (node[i,a] GT scalar)
      scalar=node[i,a]; city[a] = i
      // city i at location a
   ENDIF
 ENDFOR
ENDFOR;
scalar=0.;       // Determine pathlength
FOR (a=0, n - 1 )
   i = city[a];
   k = city[a + 1];
   scalar += dist[i,k]
ENDFOR;
i = city[n - 1];
k = city[0];
scalar += dist[i,k]; // Closed path
FOR (a = 0,n - 1)
FOR (k = a + 1,n)
   IF (city[a] EQ city[k])
     PRINT("invalid tour: visit twice %d\n",city[a]);
     BREAK
   ENDIF
ENDFOR
ENDFOR;
FOR (a=0,n)
  i = city[a];
  PRINT("%f ",xcity[i],"%f\n",ycity[i])
ENDFOR;
PRINT("\nPath length: %f ",scalar)
END;
```

APPENDIX H

THE POTTS-GLASS MODEL SIMULATION

```
NEURALNET travelsales1
// using Potss-Glass model

INT seed;
REAL
  xcity[ncity],ycity[ncity], dist[ncity,ncity],
  v[ncity,ncity], ov[ncity,ncity],
  anneal, delta, aconst, bconst, temp = 0.03;

MAINPROGRAM
  CALL parameters;
  CALL training;
  CALL valid_city
END;

PROC parameters
  INT i,j;
  FILES file1; // input file
  STRING inputf;
  READ("Input file from = ",inputf);
  OPENREAD(file1,inputf);
  FOR ( i= 0,ncity)
    READFILE(file1,xcity[i],ycity[i])
  ENDFOR;
  CLOSEFILE(file1);
  FOR (i=0,ncity)
    dist[i,i] = 0.
  ENDFOR;
  FOR (i=0,ncity - 1)
    FOR (j= i + 1 ,ncity)
      dist[i,j] = SQRT( SQR(xcity[i]-
                    xcity[j])+SQR(ycity[i]-ycity[j]) );
      dist[j,i] = dist[i,j]
    ENDFOR
  ENDFOR;
  READ("delta =",delta);
  READ("Constraint A =",aconst);
  READ("Constraint B =",bconst);
```

```
  READ("Annealing factor =",anneal);
  READ("temperature  =",temp);
  READ("No. of cycle =",CYCLE);
  READ("Starting seed for random generator = ",seed)
END;


PROC training
  INT stop, step,iter,nocycle;
  REAL u[ncity,ncity],  saturation, change,
       sumchange, sumsatur;
  v = (1. + 0.1*RAND1(seed) ) / ncity;
  FOR ( step=1,20)
    iter = 0;
    TRAINING  // use in serial execution only
      iter = iter + 1 ;
      PRINT("Iter:%2d\n",iter);
      u = EXP ( ( - sum0(I,J) + aconst*v
                  - bconst*sum1(v,J) )/temp );
      ov = v;
      v = u/sum2(u,I);
      sumchange = 1/ncity*SUMALL ( ABS(v - ov) );
      sumsatur = 1/ncity*SUMALL ( v*v );
      IF ( change LT delta) BREAK ENDIF;
      IF ( saturation GT .9 ) stop = 1; BREAK ENDIF
    END;
    temp = temp*anneal;
    IF ( stop EQ 1) BREAK ENDIF
  ENDFOR
END;

FUNC sum0(i,j)
INT k,add,minus;
REAL sum = 0.;
  FOR (k =0,ncity)
    IF (j + 1 EQ ncity ) add = 0
    ELSE add = j + 1
    ENDIF;
```

```
    IF (j - 1 LT 0 ) minus = ncity - 1
    ELSE minus = j - 1
    ENDIF;
    sum += dist[i,k]*(v[k,add]+v[k,minus])
  ENDFOR
RETURN sum;

FUNC sum1(v,j)
  INT k;
  REAL sum = 0.
  FOR (k=0,n)
    sum += v[k,j]
  ENDFOR
RETURN sum;

FUNC sum2(u,i)
  INT k;
  REAL sum = 0.;
  FOR ( k=0,n)
    sum += u[i,k]
  ENDFOR
RETURN sum;

PROC valid_city
INT a,i,j,k,city[ncity];
REAL scalar;

// Determine the path
FOR (a=0,ncity) // every location
 scalar= 0;
 FOR (i=0,ncity) // find valid city
   IF (v[i,a] GT scalar)
      scalar=v[i,a]; city[a] = i
     // city i at location a
   ENDIF
 ENDFOR
ENDFOR;
scalar=0.;       // Determine pathlength
```

```
FOR (a=0, ncity - 1 )
   i = city[a];
   k = city[a + 1];
   scalar += dist[i,k]
ENDFOR;
j = city[ncity - 1];
k = city[0];
scalar += dist[j,k]; // Closed path
FOR (a = 0,ncity - 1)
FOR (k = a + 1,ncity)
   IF (city[a] EQ city[k])
     PRINT("invalid tour: %d\n",city[a]);
     BREAK
   ENDIF
ENDFOR
ENDFOR;
FOR (a=0,ncity)
  i  = city[a];
  PRINT("city %d ",i,"(%f,",xcity[i],"%f)\n",ycity[i])
ENDFOR;
PRINT("\nPath length: %f ",scalar)
END;
```