A study of artificial neural networks and their learning algorithms

This item was submitted to Loughborough University’s Institutional Repository by the/an author.

Additional Information:

- A Doctoral Thesis. Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University.

Metadata Record: [https://dspace.lboro.ac.uk/2134/11194](https://dspace.lboro.ac.uk/2134/11194)

Publisher: © H. Y. Y. Sanossian

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/
A Study of Artificial Neural Networks and their Learning Algorithms

By

Hermineh Yusuf Yacoub Sanossian

B.Sc. (ENG.) HONS., M.Sc.

A Doctoral Thesis Submitted in Partial Fulfilment of the Requirements For the Award of Doctor of Philosophy of Loughborough University of Technology 1992.

Declaration

I declare that this thesis is a record of research work carried out by me, and that the thesis is at my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

Herminah Yusuf Yacoub Sanossian.
DEDICATED TO

My Parents
Acknowledgements

I wish to express my sincere gratitude to Professor D. J. Evans for his guidance, suggestions and advice throughout the preparation of this thesis and in the context of research generally.

I would like to thank my parents for giving me the incentive to start and complete this project. Also thanks to my sisters and brother for their moral support.

I would like also to acknowledge Mu'tah University for their Financial support.

Finally, I would like to thank Miss Howard for helping me with the typing of this thesis.
Abstract

The work presented in this thesis is mainly involved in the study of Artificial Neural Networks (ANNs) and their learning strategies. The ANN simulator incorporating the Backpropagation (BP) algorithm is designed and analysed and run on a MIMD parallel computer namely the Balance 8000 multiprocessor machine.

Initially, an overview of the learning algorithms of ANNs are described. Some of the acceleration techniques including Heuristic methods for the BP like algorithms are introduced.

The software design of the simulator for both On-line and Batch BP is described. Two different strategies for parallelism are considered and the results of the speedups of both algorithms are compared.

Later a Heuristic algorithm (GRBH) for accelerating the BP method is introduced and the results are compared with the BP using a variety of expositing examples.

The simulator is used to train networks for invariant character recognition using moments. The trained networks are tested for different examples and the results are analysed.

The thesis concludes with a chapter summarizing the main results and suggestions for further study.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
</tbody>
</table>

**Chapter 1: Introduction**  
1.1 Learning  
1.2 ANN Implementations  

**Chapter 2: Basic Concept of Neural Computing**  
2.1 The Neuron  
2.1.1 The Soma  
2.1.2 The Dendrite  
2.1.3 The Axon  
2.2 ANN Models  
2.2.1 Single Layer Perceptron  
2.2.2 Multi-Layer Perceptron  
   A. Training Procedure for Multi-Layer Perceptron  
   B. Network Configuration  
   C. Training The Network  
2.2.3 Iterative Backpropagation  
2.2.4 Hopfield Networks  
   A. Hopfield Network Architecture  
   B. Hopfield Updating Algorithm and Convergence  
   C. Hopfield Networks as Associative Memory  
   D. Hopfield Networks for Optimization Problems  
2.2.5 The Boltzmann Machine  
   A. Simulated Annealing  
   B. Training Procedure in the Boltzman Machine  
2.2.6 Competitive Learning

iii
A. Network Architecture 37
B. The Learning Algorithm 38

2.2.7 The Hamming Net 39
   A. Hamming Network Configuration 39
   B. The Hamming Net Algorithm 40

2.2.8 The Carpenter and Grossberg Classifier 41

2.2.9 Kohonen’s Self-Organizing Feature Maps 44

Chapter 3: Acceleration Methods of Learning ANNs 46

3.1 Nonlinear Optimization Techniques 47
3.2 Computing The Learning Rate 48
   3.2.1 Newton’s Method 49
   3.2.2 False Position Method 50
   3.2.3 Quadratic Fit Method 51
   3.2.4 Cubic Fit Method 54
3.3 Inexact Line Search 56
3.4 Computing The Direction of Search 58
   3.4.1 Steepest Descent Method 58
   3.4.2 The Backpropagation 62
3.5 The Problems Associated with Backpropagation and
   Steepest Descent Learning Algorithms 62
3.6 Acceleration Methods 66
   3.6.1 Newton’s Method 66
   3.6.2 Conjugate Direction Method 67
   3.6.3 The Conjugate Gradient Method 69
      A. The Conjugate Gradient Algorithm for Quadratic Form 69
      B. Generalization of the Conjugate Gradient Algorithm 70
   3.6.4 Quasi-Newton Method 71
   3.6.5 The Runge Modification Rule 74

iv
3.7 Acceleration Methods Using Heuristic Algorithms

3.7.1 Momentum Rule
3.7.2 Delta-Delta Learning Rule
3.7.3 Delta-Bar-Delta Rule
3.7.4 Heuristic Learning Algorithm (HLA)
3.7.5 Quickprop

Chapter 4: Parallel Simulation of Neural Networks

4.1 Parallel Processing for ANNs
4.1.1 The SIMD Computer Architecture
4.1.2 The MIMD Computer Architecture
4.2 Parallel Programming on MIMD Machines
4.3 Factors that limit Speedup on MIMD Machines
4.4 Some Parallel Pascal Features of the Sequent Balance 8000
4.5 Parallelism in ANNs
4.6 Neural Network Simulator
4.6.1 The On-Line Backpropagation Procedure
4.6.2 The Batch Backpropagation Procedure
4.7 Description of the Experiments
4.7.1 The Results of the on-Line Simulator
4.7.2 The Results of the Batch Simulator
4.8 Conclusion

Chapter 5: Accelerating the Backpropagation Algorithm

5.1 A Gradient Range-Based Heuristic Method for Increasing the Rate of Convergence (GRBH)
5.2 Experimental Description
5.2.1 The Exclusive-OR Problem
5.2.2 Shift Register Problem
5.2.3 Sorting Problem
5.3 The Steepest Descent Algorithm
5.4 Conjugate Gradient Method
5.5 The Batch BP Experimental Results
  5.5.1 The XOR Problem
  5.5.2 The Shift Register Problem
  5.5.3 Sorting Problem
5.6 Two Intertwined Spirals
  5.6.1 The Spiral Task
  5.6.2 The Network Architecture
  5.6.3 BP and Quickprop Results
  5.6.4 The GRBH Results
5.7 Conclusion

Chapter 6: Character Invariance Using ANN
6.1 ANNs Architecture Invariant to Geometrical Changes
  6.1.1 The structure of the Neocognitron
  6.1.2 Calculating the output of a cell
  6.1.3 Training the Neocognitron
6.2 Training Set
6.3 Moment Invariant
  6.3.1 Geometric Moments (GM)
  6.3.2 Complex Moments (CM)
  6.3.3 Zernike Moments (ZM)
6.4 Minimum Number of Hidden Nodes for ANNs
6.5 Experimental Description
  6.5.1 Training and Testing Vector Pairs
  6.5.2 Network Architecture
  6.5.3 Experimental Results using GMs
CHAPTER I

Introduction
Current serial computers are very good at executing well structured problems, but problems that require massive parallelism such as image processing, pattern recognition, speech recognition and combinatorial problems etc. are not suitable for conventional computers. Artificial Neural Networks (ANNs) are a possible solution to such problems. They are designed to mimic the human brain. The brain is composed of more than \(10^{11}\) neurons and a neuron can have from 1000 to 100000 connections with other neurons. These neurons receive signals from the environment through our sensors and react accordingly. ANNs are also called connectionist systems, adaptive networks, neurocomputers, or neural networks. The ANN consists of a large number of very basic processing elements (PEs) which represent the neurons. These neurons are connected with each other so that each neuron can receive signals from other neurons and produce an output which will be received by other neurons. These connections carry different weights. A simple neuron sums all its weighted inputs and passes the result through a nonlinearity function or any other function. Figure 1.1 is a simple example of a neuron with \(I\) inputs.

**Figure 1.1** A simple neuron.
In a network neurons have different functions, some neurons are input nodes and some of them are output nodes. There can also be some hidden nodes. Figure 1.2 shows a simple network with \( I \) input nodes, \( O \) output nodes and \( H \) hidden nodes.

![Diagram of a simple neural network]

**Figure 1.2** A simple neural network.

The input signal is applied to the input nodes, these are passed to the hidden nodes and then to the output nodes. The output result would be correct if the network was trained properly. The required information is stored in the connection weights. A training rule is required to find a suitable set of weights that can perform the required action. Much of ANNs are based on the model formalised by McCulloch and Pitts [1943]. They used threshold logic elements as their neural function. Interest in the field declined after Minsky and Papert's [1969] publication. They showed that neural networks might be limited in their applications to real problems. Later it became apparent that Minsky and Papert's criticisms were true only for very simple types of neural networks. Interest in neural networks has grown in recent years for many reasons, these are
1. Massive parallelism which provides high speed performance and fault tolerance, damage to a few nodes or links will not weaken the overall performance.

2. Adaptivity of ANNs since they can be trained rather than programmed. This allows the network to improve with experience.

3. Generalization from previous examples to new ones.

The ANNs are applied in many fields such as vision, knowledge processing, control, data compression, pattern recognition and so on.
1.1 Learning

Learning is one of the basic features of intelligence. To quote Simon [1983] "Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same tasks drawn from the same population more efficiently the next time". Learning occurs in ANN by modifying the connection weights. One of the first known learning algorithms is due to Hebb [1949]. He suggested that if a node \( a_i \) receives an input from another node \( a_j \) and both are active, then the weight connection between the two nodes should be strengthened. This algorithm was extended so that it can be used more efficiently. Different learning algorithms were also developed to suit various net topologies (Judd [1990]). For example, a network without hidden nodes, where the output nodes are directly connected to the input nodes, an algorithm developed by Widrow and Holf [1960] can be used. In this algorithm the error is found by calculating the difference between the output response and the desired response for each output node. Then the weights are changed to minimise the squares of these errors. This method is also known as Least Mean Square (LMS). If the network has hidden nodes, the LMS algorithm is no longer suitable because the desired response of the hidden nodes are not known. Error-Backpropogation (also called Backpropagation by Rumelhart, Hinton and Williams [1986]) can calculate the error of the hidden nodes by propagating errors backwards through the net so that the hidden nodes connections are adjusted according to how much they participated in the output error. Learning in general can be divided into two catagories.

1. Supervised Learning

In this case the optimal response is known and a learning algorithm is used to perform correctly each time an input is applied. In this method error correction stategies are adopted.
2. Unsupervised Learning

In this case the optimal response is not known and it is required to be identified. In this method extreme search strategies are adopted.

Different learning strategies will be discussed in Chapter 2. Learning and training are closely linked, for both supervised and unsupervised learning. The training data should cover the true reality of the world which is noisy, imprecise and incomplete. Not much is known about the performance of neural nets using different training parameters. The size of the training set, the number of order of presentations and the similarity of items within a set are all important parameters which effects the learning performance. There are many points to be considered when a specific learning model is used. These are

1. Theoretical Capability:- Some examples or patterns are not possible to mapped onto a certain network. Therefore it is important to identify network models that can learn the required patterns.

2. Fault Tolerance:- The ability of a network to withstand some noisy or incomplete input data and also some internal faults caused by a faulty node or disconnection in some of the links.

3. Memory Capacity:- A network is capable of storing a certain amount of information before saturation or cross-talk takes place.

4. Scaling:- Learning becomes faster with increasing sizes of the training sets, but if the number of examples increases for certain networks then the performance slows down.

The main obstacle in the research progress of ANNs in real world applications is the slow speed at which the current algorithms learn. At present
the Backpropagation algorithm (Rumelhart and McClelland [1986]) is one of the fastest learning methods, but it still requires a large number of iterations for a small network to learn. A few real time application problems need small networks which can be trained using the Backpropagation algorithm. However the remainder require large and complex networks which cannot be handled using the current learning algorithms. To test the performance of ANNs for large networks, faster learning algorithms are needed.

1.2 ANN Implementations

There are several approaches in implementing ANNs, these can range from a very complex hardware VLSI design to virtual implementation in general purpose sequential and parallel machines. Hardware implementations are faster than software simulators but they are confined to special purpose ANNs. On the other hand, simulators can be designed for general purpose ANNs but with the disadvantage of being slow. ANNs are computationally expensive mainly for two reasons:-

1. They have a large number of neurons.

2. The learning algorithm is an iterative method that requires a large number of iterations to converge to the 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 then the sequential machines.
CHAPTER II

Basic Concept of Neural Computing
The remarkable performance of the nervous system in many fields is achieved despite the fact that its switching time is in the order of milliseconds compared to nanoseconds for the electronic gate switching time. Even though ANN resembles the human brain in some of its functions, such as learning from examples rather than programming. Unfortunately very little is known about our nervous system.

Since the beginning of 1980's the interest in ANNs has increased and a large range of models have been developed for different purposes (Aleksander [1990], Simpson [1990], Dayhoff [1990]). All these models share some common functions.

These are:

1. Every network contains nodes. These nodes are very simple neurons. The state of each node is a single value and it indicates the strength of the activation of the node.

2. The nodes are connected with each other and each connection has some strength which is called the connection weight. Due to these links each node effects all other nodes that are connected to it.

3. The new state of a node is a nonlinear function of all its input weighted sums.

4. Some of the nodes in the network is used as input nodes and their states are dependent on the input data. Therefore the number of input nodes equals the number of the features of the input data.

5. Some of the nodes in the network is used as output nodes so that an output will result when an input signal is applied to the input nodes.

6. A training algorithm is used to change the connection weights until the desired response is achieved.
In this chapter a brief introduction to the nerve cells or neurons is given and then a review of many important ANN models is introduced.
2.1 The Neuron

The neurons (nerve cells) are complex analog processors. The shape of these neurons, the connections between the neurons and their characteristics determine the parameters of the process (Wasserman [1989]). There are many different types of neurons but they are mainly divided into two categories.

1. **Local processing (Interneuron Cells):-**
   
   These are neurons confined in a small region and they are connected to each other.

2. **Output neurons:-**
   
   These type of neurons allow different parts of the brain to communicate with each other and communication is also possible from the muscles to the brain and vice versa.

A neuron has an inside and an outside separated by a plasma membrane. The outside of the neuron is surrounded by a fluid. A potential difference is created due to different concentrations between the inside and outside of the neuron. Therefore the neuron gets its energy from this potential difference which acts as a battery to the neuron with approximately 100mV potential across the membrane. A neuron consists of three sections the cell body or soma, the dendrites, and the axon. Figure 2.1 shows a neuron with all its three sections.

### 2.1.1 THE SOMA

Each neuron has a large round central body where all the chemical activities of the neuron take place. These should last a lifetime as they do not regenerate.
2.1.2 THE DENDRITE

These are thin, long and irregularly shaped nerve processes, which are branched from the soma. Signals are received by a neuron from other neurons through contacts on the dendrites. These contacts are called synapses. The dendrites are electrically passive so they can be represented as capacitors and resistors in series. The shape of these dendrites play a large role on the type of functions that a neuron can process. The same synaptic input at different dendrites have different effects.

2.1.3 THE AXON

This is another type of nerve process which is branched from the soma. They are located on the output neurons. The length of these axons could be as short as 0.1mm or as long as 1 metre. Unlike the dendrite, they are electrically active. In fact, they function as a nonlinear threshold device,
which causes a sudden change in voltage. A voltage pulse occurs when the potential in the soma exceeds some threshold value. The potential in the soma is due to the synaptic inputs connected to the dendrite of the neuron. If the synaptic inputs are activated, then the potential is passively conducted to the soma. If the sum of all these inputs exceeds the threshold value, then a voltage pulse is generated. These axons terminate at a synapse which can transmit its signal to other neurons.

The synaptic terminal is a small bulbous expansion of the axon and it contains spherical structures called synaptic vesicles. When the potential on the axon side (pre-synaptic side) increases, the synaptic vesicles release some of their content (known as Neurotransmitters) to the synaptic cleft as shown in Figure 2.2. Charged ions will flow either into or out of the neuron and hence produce a change in the dendrite potential.

![Figure 2.2 Simplified sketch of a synapse.](image)

### 2.2 ANN Models

ANNs can be divided into two categories regarding the type of input data. Some networks can have only binary valued inputs others can have continuous
valued inputs or both. The learning procedures for these networks in general can be divided into two categories 1) Supervised learning 2) Unsupervised learning. Different types of models are described in this chapter. Figure 2.3 gives all the models described in this chapter with their categories.

![ANN Models Diagram](image)

**Figure 2.3** ANN models and their categories.

### 2.2.1 SINGLE LAYER PERCEPTRON

The single layer perceptron (Rosenblatt [1962]) is made up of an input layer and an output layer. They are called single layer perceptron because there is only one layer beyond the input layer. The output of each node can be calculated by computing a weighted sum of the input elements, subtracting a threshold $\theta$ and the result is passed through a hard limiter. Figure 2.4(a) shows a diagram of the perceptron with one output node and $I$ inputs, Figure 2.4(b) shows the hard limiter function.

The output of a node can be expressed as

$$a_i = f \left( \sum_{\substack{j=0 \atop i \neq j}}^{I-1} w_{ij} a_j - \theta_i \right),$$  \hspace{1cm} (2.1)

where $f()$ is the hard limiter, $w_{ij}$ is the connection weight from node $j$ to node $i$, $a_i$ is the output state of node $i$, and $\theta_i$ is the threshold value of node $i$. 

12
Figure 2.4(a) A simple perceptron with one node and I inputs

(b) is a graph of the hard limiter.

The output of a node can be either zero or one, if the total weighted sum is larger than $\theta$ then the output is one otherwise it is zero.

Different algorithms can be used to train the perceptron, one of the earliest methods is the perceptron convergence procedure, which was developed by Rosenblatt [1962]. This method can be described as follows:

1. Initialize the connection weights as well as the thresholds to small random values.

2. Apply the input vector to the input layer.

3. Calculate the output values using Equation (2.1).

4. Calculate the new connection weights as follows:

   a) If the output node is in error and it is zero then the new weight is calculated as

   $$w_{ij}(n + 1) = w_{ij}(n) + \alpha a_j,$$  \hspace{1cm} (2.2)

   where $n$ is the iteration number.

   $\alpha$ is the learning step (or rate) and it is a positive value $\leq 1$. 

   

13
b) If the output node is in error and it is one then the new weight is calculated as

\[ w_{ij}(n + 1) = w_{ij}(n) - \alpha a_j. \]  

(2.3)

c) If the output node is correct, then the connection weight retains its value.

\[ w_{ij}(n + 1) = w_{ij}(n) \]  

(2.4)

5. Go back to Step 2

This procedure ignores the magnitude of the error. To understand the perceptron convergence procedure, let us assume that a network has only one output node and \( I \) inputs. A graph of \( I \) dimensions is drawn with one dimension for each weight. For each input vector a hyperplane is drawn in the weight space which indicates that the output is correct if the set of weights lie on one side of the hyperplane and it is incorrect if they lie on the other side of the hyperplane. For the network to behave correctly on the training set, the weights should lie on the correct side of all the hyperplanes. Figure 2.5 shows three hyperplanes in 2-dimensional space.

![Figure 2.5](image)

**Figure 2.5** Three hyperplanes in 2-dimensional space, the output is correct for each hyperplane on the unhashed side.
If the set of weights were on the wrong side of the hyperplane, then using the perceptron convergence procedure the weights are moved perpendicularly towards the hyperplane. Hence the distance between the current set of weights and any ideal set of weights is reduced provided that the set of weights move by less than twice the distance to the plane. The updated weight is guaranteed to reduce the following measure.

\[
\sum_i (w_{i,\text{actual}} - w_{i,\text{ideal}})^2, \tag{2.5}
\]

where \(w_{i,\text{actual}}\) is the actual connection weight of node \(i\), and \(w_{i,\text{ideal}}\) is the ideal connection weight of node \(i\). This perceptron convergence procedure has many problems. If there is no set of weights which can represent a particular problem then it is possible for the decision boundaries to oscillate. Also for multilayer and complex networks, this procedure cannot be generalized to suit such networks.

Widrow and Hoff [1960] developed another learning algorithm for the perceptron which is a generalization of the perceptron convergence procedure. This algorithm is known as the Delta Rule or Least Mean Square (LMS). It is extended to apply to continuous as well as binary input signals, this is achieved by replacing the hard limiter with a threshold logic (Widrow and Stearns [1985]). The algorithm is based on the minimization of the mean square error between the desired output and the actual output. The LMS algorithm is the same as the perceptron convergence procedure except for step 4, where it is replaced by the following:

4. Calculate the error \(\delta\) as

\[\delta_i = t_i - a_i, \tag{2.6a}\]

where \(t_i\) is the desired response of node \(i\).
The new weight is calculated as

\[ w_{ij}(n + 1) = w_{ij}(n) + \alpha \delta_i a_j. \] (2.6b)

This learning algorithm can be interpreted in a geometrical form. Let us assume a network of one output node and \( I \) input lines. The weight space is \( I \)-dimension with one dimension for each weight. This can be represented in a geometrical form by drawing one axis for each weight and an extra axis (the height) is required for the error. For each weight combination, the error is calculated and a point is drawn. These points will form a surface called the error surface. This is depicted in Figure 2.6 with \( I = 2 \).

![Figure 2.6 The error surface of a node with 2 inputs.](image)

For linear output nodes of one layer network, the error surface forms a bowl with one minimum. The error surface has parabolic vertical cross sections and elliptic horizontal cross sections. Any gradient descent method on the error surface will guarantee a correct solution. As the output nodes are nonlinear, the error surface for each training vector is a deformed parabolic trough. The total error surface would be the sum of all troughs of the training vectors. When a number of troughs are added it is possible to create local
minima due to a change in the total gradient sign. If the problem in hand has a set of weights that will give zero error, then no local minima is created.

Rosenblatt [1962] showed that the perceptron could learn anything that it can represent. Representation is different from learning since representation is the ability of a network to simulate a particular function. On the other hand, learning is an iterative procedure in which the weights are changed each time until the function is achieved. The single layer perceptron cannot represent very many simple problems, as shown in the following XOR problem. By using a simple network of one output node and two input nodes. Let us assume that $a_0$ and $a_1$ are the input signals, $w_{i0}$ and $w_{i1}$ are their corresponding weights and let us assume that $\theta_i$ is the threshold value of the output node. Figure 2.7(a) illustrates such a network. The output of this network can either be one or zero depending on the following equations,

$$a_0w_{i0} + a_1w_{i1} > \theta_i \Rightarrow \text{output} = 1$$

$$a_0w_{i0} + a_1w_{i1} \leq \theta_i \Rightarrow \text{output} = 0$$

This shows that the net divides the space spanned by the input into two regions separated by a line (Figure 2.7(b)). Let us now assume that we would like to represent the XOR problem using such a network. Figure 2.7(c) shows the input vectors and their corresponding outputs (truth table) for the XOR problem. The four input vectors can be drawn on the input space as shown in Figure 2.7(d). As can be seen from the diagram there is no straight line which can separate these two classes. Therefore single layered networks cannot represent such functions.
Figure 2.7 (a) A network with two inputs and one output

(b) The network divides the space spanned by the input into two regions A and B

(c) The XOR vector pairs

(d) The XOR classification region.

2.2.2 MULTI-LAYER PERCEPTRON

The capabilities of a multi-layer perceptron is due to the fact that nonlinearities are used within the nodes (Hinton[1989], Fahlman and Hinton [1987], Souček [1989]). If the nodes were linear elements then the multi-layer net can be presented as a single layer net. As was mentioned earlier, a single layer
perceptron is limited to its function representations. Adding another layer allows the network to perform more general classifications. Multi-layer networks are capable of separating points that are contained in an open or closed convex region. A convex region is when two points in a region can be connected by a straight line without leaving the region. A closed region is when all points are contained within a boundary. An open region is when some of the points are outside the boundary. Figure 2.8 depicts some examples of such regions.

![Figure 2.8](image-url)

**Figure 2.8** (a) Is an open region

(b) and (c) are closed convex regions.

A two layer network can represent the XOR problem. This can be described by taking a network with two nodes in the first layer and one node in the second layer. Each node in the first layer has two inputs as shown in Figure 2.9(a)

![Figure 2.9](image-url)

**Figure 2.9** (a) Is a network of two layers

(b) A convex open decision region due to the network.
Each node in the first layer acts as a single layer perceptron where its output is high only for points on one side of the hyperplane. So let us assume that one of the nodes in layer one has the hyperplane \( P_1 \) (in Figure 2.9(b)) and the node is active only when it lies below \( P_1 \). Let us also assume that the second node in layer one has the hyperplane \( P_2 \) and the node is active only when it lies above \( P_2 \). If the sum of the weights from layer one to layer two is larger than the threshold value of the output node, then an AND operation is performed by the output node. Therefore, the output node would be active only when both nodes in the first layer are active, the dashed region of Figure 2.9(b) represents the output node when it is active. This shows that multi-layer networks can represent the XOR problem and that the number of hidden nodes are important to form the required decision region.

Two layer networks cannot represent decision regions shown in Figure 2.10. Therefore, three layer networks are required. In such networks the decision region is partitioned into small hypercubes (or squares for two inputs). To create these hypercubes, \( 2I \) nodes are required in the first layer, where \( I \) is the number of inputs, one node for each side of the hypercube. One node in the second layer is required to perform the AND operation. The output node performs an OR operation.

![Figure 2.10 A decision region which cannot be trained on a two layered network.](image)
A. TRAINING PROCEDURE FOR MULTI-LAYER PERCEPTRON

The perceptron convergence procedure as well as the LMS algorithm cannot be used for training multi-layer networks due to the fact that the required response of the hidden nodes are not known. Rumelhart, Hinton and Williams [1986] described a learning algorithm for multi-layer network known as Backpropagation or the generalized delta rule. Earlier others (Parker [1982], Werbos [1974]) described this algorithm.

The Backpropagation procedure is a generalized form of the LMS procedure. The Backpropagation is used for networks with hidden nodes, where the desired response values are not known. As mentioned earlier, networks with hidden nodes are capable of representing more complex functions other than the single layer networks. However, the learning is much slower because it must search the space of possible ways using the hidden nodes. This algorithm is an iterative gradient method which tries to minimize the mean square error between the actual output and the desired response. The state of each node is calculated as in the perceptron except for the nonlinearity function where the hard limiter is replaced by a sigmoid logistic nonlinearity function. The error surface of such a network has many local minima in contrast to the single layer network where it has only one minima (when a solution exists).

B. NETWORK CONFIGURATION

The network is connected in a feedforward manner, that is, the input signals to a node comes from nodes in a lower layer. The output of each node is connected to a layer above it. The Backpropagation can be used for any number of layers.
Each node consists of a summation and a sigmoid logistic function also called a squashing function. The state of node $i$ is $a_i$ and it is calculated as

$$a_i = \frac{1}{1 + e^{-net_i}},$$  \hspace{1cm} (2.7)

where

$$net_i = \sum_{j=1}^{I} w_{ij}a_j + \theta_i,$$  \hspace{1cm} (2.8)

where $w_{ij}$ is the connection weight from node $j$ to node $i$, $\theta_i$ is the threshold value of node $i$, and $I$ is the number of input signals to node $i$. The output of the sigmoid function ranges from zero to one as shown in Figure 2.11.

![Figure 2.11 The Sigmoid Logistic Function.](image)

Other functions can be used (Holt [1991]) provided that it has a bounded derivative. This function is used because it has a simple derivative where

$$\frac{\partial a_i}{\partial net_i} = a_i (1 - a_i).$$  \hspace{1cm} (2.9)

C. TRAINING THE NETWORK

For every iteration the mean square error is reduced until the error approaches zero. Let us assume that the state of node $i$ when pattern $p$ is
presented is \( a_{pi} \) and that the required response for the same pattern and node is \( t_{pi} \), then the mean square error can be calculated for pattern \( p \) as

\[
E_p = \frac{1}{2} \sum_i (t_{pi} - a_{pi})^2. \tag{2.10}
\]

The total error due to all patterns (or training set) is \( E \) where

\[
E = \sum_p E_p. \tag{2.11}
\]

To minimize \( E \) using the gradient descent method, the partial derivative of the error function should be calculated with respect to each weight in the network, \( \partial E / \partial w_{ij} \). This value is used to calculate the change in the connection weight \( \Delta w_{ij} \) so

\[
\Delta w_{ij} = -\alpha \frac{\partial E_p}{\partial w_{ij}}, \tag{2.12}
\]

where \( \alpha \) is a positive quantity defined to be smaller or equal to 1 called the learning rate. Using the chain rule we have

\[
\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial \text{net}_{pi}} \frac{\partial \text{net}_{pi}}{\partial w_{ij}}. \tag{2.13}
\]

Also we have

\[
\frac{\partial E_p}{\partial \text{net}_{pi}} = \frac{\partial E_p}{\partial a_{pi}} \frac{\partial a_{pi}}{\partial \text{net}_{pi}}, \tag{2.14}
\]

then from Equation (2.10) we have

\[
\frac{\partial E_p}{\partial a_{pi}} = -(t_{pi} - a_{pi}). \tag{2.15}
\]

and from Equation (2.9) we have

\[
\frac{\partial a_{pi}}{\partial \text{net}_{pi}} = a_{pi}(1 - a_{pi}). \tag{2.16}
\]
Therefore by taking Equations (2.15) and (2.16) we have

$$\frac{\partial E_p}{\partial \text{net}_{pi}} = -(t_{pi} - a_{pi})a_{pi}(1 - a_{pi}). \quad (2.17)$$

Now \(\partial \text{net}_{pi}/\partial w_{ij}\) can be calculated from Equation (2.8),

where

$$\frac{\partial \text{net}_{pi}}{\partial w_{ij}} = a_{j}. \quad (2.18)$$

Hence Equation (2.13) can be written as

$$\frac{\partial E_p}{\partial w_{ij}} = \delta_{pi}a_{j}, \quad (2.19)$$

where

$$\delta_{pi} = -(t_{pi} - a_{pi})a_{pi}(1 - a_{pi}). \quad (2.20)$$

Now \(\delta_{pi}\) can be calculated from Equation (2.20) for the output nodes only

where the desired response \(t_{pi}\) is known. For the hidden nodes the following

chain rule is used to calculate \(\partial E_p/\partial a_{pi}\):

$$\frac{\partial E_p}{\partial a_{pi}} = \sum_k \frac{\partial E_p}{\partial \text{net}_{pk}} \frac{\partial \text{net}_{pk}}{\partial a_{pi}}. \quad (2.21)$$

where \(k\) is applied to all nodes above node \(i\).

Then from Equation (2.8) we have

$$\frac{\partial E_p}{\partial a_{pi}} = \sum_k \frac{\partial E_p}{\partial \text{net}_{pk}} w_{ki} \quad (2.22)$$

$$= \sum_k \delta_{pk} w_{ki}$$

Therefore \(\delta_{pi}\) for the hidden nodes is calculated as

$$\delta_{pi} = a_{pi}(1 - a_{pi}) \sum_k \delta_{pk} w_{ki}. \quad (2.23)$$
and the change in the connection weights is calculated as

\[ \Delta w_{ij} = \alpha \delta_i a_{pj}. \]  

(2.24)

where \( \alpha \) is the learning rate and it is a positive value. To train these networks, the connection weights in the network is initialized by small random values. The iterative procedure consists of two passes.

**Pass 1.**

1. Obtain the next vector pair and apply it to the input of the network.

2. By using Equation (2.7) calculate the output of each node. In this step, the output of the nodes within a layer can be calculated in parallel but the output of each layer is calculated sequentially starting from layer 1 and ending up with the output layer.

**Pass 2.**

The error caused by each node in the output layer is calculated from Equation (2.20). Then the error of the hidden nodes is calculated using Equation (2.23). Again, in this case, the error of the nodes within a layer is calculated in parallel but the error of each layer is calculated sequentially starting from the output layer and ending up with the first layer.

After going through pass one and pass two, the connection weights of the network can be changed according to Equation (2.24). These changes can take place in parallel. This process is repeated for all vector pairs. One cycle is performed after going once through all the vector pairs.

Many problems have been applied to the multi-layer networks, Hinton [1986] used such networks to train family relationships between 24 different
people. Sejnowski and Rosenberg [1986] used such networks to map text to speech.

This method of training has proved to be slow. Different methods were developed to accelerate the Backpropagation procedure. These will be discussed in Chapter 3. Another deficiency of such a system is that it is implausible as a biological model.

2.2.3 Iterative Backpropagation

Rumelhart, Hinton and Williams [1986] showed that the Backpropagation procedure can be extended to apply to iterative networks in which the nodes in the network can be connected in both directions as shown in Figure 2.12. Such networks are used for identifying, generating or completing sequences.

Due to these bidirectional connections, the states of the nodes at time $\tau$ determine the states of nodes at time $\tau + 1$. These networks can be represented in a feedforward net with one layer for each time slice. The weights and nodes are identical for each layer. Let us assume that we have a network with three nodes as shown in Figure 2.12(a). Let us also assume that this network is run synchronously for three iterations, the equivalent feedforward net is shown in Figure 2.12(b).
The Training Procedure

1. The input sequence is applied to their corresponding nodes.
2. The signals are passed on to the next sliced layer. The states of the nodes are calculated according to Equation (2.7).
3. The output of some of the nodes are compared with the target values for that particular time slice and the error is calculated as Equation (2.20).
4. These errors are passed back through the network and the error is calculated according to Equation (2.23).
5. The weight changes are calculated for each time slice and the sum of the changes for each connection is saved.
6. After going back through the network, the weights are changed.

The problem with such networks is the amount of memory required. Each weight connection in the original net requires an extra storage to store the
sum of the weight changes. Also, the state of the nodes for each time slice must be stored.

2.2.4 Hopfield Networks

Hopfield [1982] introduced a new ANN architecture based on the physical models of materials with magnetic properties. Asynchronous parallel processing is used to find the new states of the system. Hopfield used these networks as associative memories with binary input and output vectors. Later these networks (Hopfield [1984]) were improved to accept continuous input values and used to solve combinatorial optimization problems.

A. HOPFIELD NETWORK ARCHITECTURE

These networks are composed of a single layer of nodes (Tank and Hopfield [1987], Hinton [1985]). Each node is connected to every other node as shown in Figure 2.13.

![Figure 2.13 The Hopfield network architecture.](image)

Many or all of these nodes are input nodes and others (or all of them) are output nodes. Also they are symmetrical networks, that is, the connection
weight from node \( i \) to node \( j \) \((w_{ji})\) is equal to the connection weight from node \( j \) to node \( i \) \((w_{ij})\). Hence,

\[
w_{ij} = w_{ji}.
\] (2.25)

The new state of each node is calculated by the following equation

\[
a_i = f\left(\sum_{j=1}^{I} w_{ij} \ a_j + \theta_i\right),
\] (2.26)

where \( f \) is a hard limiter when binary nodes are used and a monotonic sigmoid function when analog nodes are used.

**B. HOPFIELD UPDATING ALGORITHM AND CONVERGENCE**

After assigning the connection weights (this will be discussed later) the following updating algorithm is used.

1. Apply the input vector to the input nodes.
2. Choose a node randomly and calculate its new state according to Equation (2.26). This process is repeated until a stable state is reached. A stable state is achieved when the output nodes remain unchanged with further iterations.

Hopfield showed that such networks converge when symmetrical connections are used. Hopfield associated with the states of the network a measure called the Energy, \( E \), which is expressed as

\[
E = - \sum_{i<j} w_{ij} \ a_i \ a_j + \sum_i \theta_i \ a_i,
\] (2.27)

The second term of Equation (2.27) is zero for binary nodes where the value of \( \theta_i \) is zero. During the updating procedure this energy should be minimized.
The network is said to converge if at each iteration the energy is reduced. The change in energy due to a change in the state of neuron $i$ is given by

$$\Delta E = -\Delta a_i \sum_{j \neq i} w_{ji} a_j + \Delta a_i \theta_i,$$  \hspace{1cm} (2.28)

where the second term is a zero for binary nodes. Equation (2.28) can be written as

$$\Delta E = -\Delta a_i \left( \sum_{j \neq i} w_{ji} a_j - \theta_i \right),$$  \hspace{1cm} (2.29)

where $\Delta a_i = (a_i)_{\text{new}} - (a_i)_{\text{old}}$.

Let us assume that the summation value in Equation (2.29) is greater than $\theta_i$, hence the term in the bracket is positive and the new state of node $i$ is one according to Equation (2.26). Therefore $\Delta a_i$ is either one or zero, and $\Delta E$ is negative or zero. If the summation value in Equation (2.29) is less than $\theta_i$ then the term in the bracket is negative and the new state of node $i$ is zero according to Equation (2.26). Therefore $\Delta a_i$ is either zero or $-1$, and again $\Delta E$ is negative or zero. This shows that the energy of the network is reduced or stays as it is when the state of a neuron is changed. Such networks have many local minima and this updating procedure does not guarantee the network energy to reach the global minima. This is due to the fact that when the energy arrives at a local minima it will get stuck at this point. According to the updating rule, the energy is either reduced or remains unchanged. To arrive at the global minima from the local minima, the energy should be increased before reducing it again. This can be illustrated in Figure 2.14.

![Figure 2.14 The Energy is stuck at local minima.](image)
This problem has been solved by using simulated annealing (which will be described later).

C. HOPFIELD NETWORKS AS ASSOCIATIVE MEMORY

By using associative memories, an item in the memory can be retrieved by presenting sufficient partial information. Hopfield [1982] applied the idea of associative memory on binary networks. The number of nodes used in the network is equal to the number of features of the input pattern where a pattern can be represented as

$$X_p = (x_{p1}, x_{p2}, \ldots, x_{pt}).$$  \hspace{1cm} (2.30)

Let us assume that there are $m$ patterns to be stored, $X_1, \ldots, X_m$. The information to be stored in memory are the connection weights and they are calculated as

$$w_{ij} = \sum_{p=1}^{m} (2x_{pi} - 1) (2x_{pj} - 1).$$ \hspace{1cm} (2.31)

These networks have two major limitations.

1. Sometimes the network evokes spurious states. The spurious state is a pattern which is different from all example patterns. This problem can be improved (Abu–Mostafa and Jacques [1985]) when the number of classes is $0.15 \, N$, where $N$ is the number of nodes.

2. The evoked memory in some cases is not the correct example. This problem is due to the similarities of the examples and can be remedied using orthogonal examples.

D. HOPFIELD NETWORKS FOR OPTIMIZATION PROBLEMS

A larger range of information can be stored using analogue nodes. These nodes use a sigmoid function rather than a hard limiter. Hopfield applied
such networks to the Travelling Salesman problem (TSP). These are difficult optimization problems that belong to the NP-complete class of problems. The requirement is to find the shortest route to arrive from city $A$ to city $Z$ passing through cities $B, C, \ldots$. The order of visiting cities is not important as long as the overall distance is minimized. A tour is a trip that starts and ends at the same place and each city is visited once. The path length is

$$p_l = \sum_i d_{x_i x_{i+1}},$$

(2.32)

where $d_{x_i x_{i+1}}$ is the distance between city $x_i$ and $x_{i+1}$. Hopfield networks give a good solution to these problems rather than the minimum.

The TSP can be represented by a matrix (Hopfield and Tank [1986]), each row of the matrix represents a city and each column of the matrix represents the position in the tour. Hence a network with $m^2$ nodes is required, where $m$ is the number of cities to be visited.

The constraints of the problem are as follows:

1. Every city is allowed to be visited only once, hence only one node in every row is allowed to be active.
2. There can be only one visit to any position, hence each column must have only one node active.
3. There can be only $m$ nodes active for the whole matrix.
4. The sum of the distances must be minimum.

The energy function can be constructed according to these constraints as follows:

$$E = \frac{A}{2} \sum_x \sum_i \sum_{j \neq 1} a_{xi}a_{xj} + \frac{B}{2} \sum_i \sum_x \sum_{y \neq y} a_{xi}a_{yi} +$$

$$\frac{C}{2} \left( \left( \sum_x \sum_i a_{xi} \right) - m \right)^2 + \frac{D}{2} \sum_x \sum_{y \neq x} \sum_i d_{xy}a_{zi} \left( a_{yi+1} + a_{yi-1} \right),$$

(2.33)
where $A$, $B$, $C$, $D$ are constants and the performance of the system depends on these values.

By choosing large values for $A$, $B$ and $C$, the low energy states will represent valid tours. The connection weights are calculated by relating the energy Equation (2.33) with the general energy Equation (2.27). This can be calculated as

\[
\begin{align*}
    w_{xi,yj} &= -A \delta_{xy}(1 - \delta_{ij}) - B \delta_{ij} (1 - \delta_{xy}) - C \\
    &\quad - D d_{xy} (\delta_{j,i+1} + \delta_{j,i-1}),
\end{align*}
\]  

(2.34)

where $\delta_{ij} = 1$ if $i = j$ and is zero otherwise. Figure 2.15 illustrates a network with 25 nodes which can solve a TSP with 5 cities.

Figure 2.15 A network which solves a TSP with 5 cities.
### 2.2.5 The Boltzman Machine

Hopfield networks suffer from getting stuck in local minimum due to the deterministic algorithm where the energy is reduced for every iteration, and once it reaches a local minima it stays there. One way to get out of a local minima is to allow the energy to be increased (simulated annealing). The Boltzman machine uses a stochastic decision rule for updating the state of the nodes. This updating rule, which is called simulated annealing, allows the Boltzman machine to find the low energy states with high probability.

#### A. SIMULATED ANNEALING

This idea is used in condensed matter physics, Kirkpatrick, Gelatt and Vecchi [1983]. A solid in a heat bath is heated by increasing the temperature until its maximum value is reached (All particles of the solid are randomly placed in the liquid state). The next step is to reduce the temperature of the heat bath slowly. At each temperature $T$, the solid is allowed to reach thermal equilibrium. The Monte Carlo method is used to find the thermal equilibrium of the solid for a fixed value of temperature $T$. A randomly chosen particle is perturbed slightly. If the difference in energy ($\Delta E$) between the present and slightly perturbed states is negative then the process is continued with the new state, if $\Delta E \geq 0$, then the probability of the acceptance of the perturbed state is given by

$$P_r = \frac{1}{1 + e^{-\Delta E/T}}. \quad (2.35)$$

This process is repeated for a large number of times, which will cause the probability distribution of the states to approach the Boltzman distribution.
At thermal equilibrium, the relative probability of two global states will follow the Boltzmann distribution:

\[ \frac{P_a}{P_b} = e^{-(E_a - E_b)/T}, \]

where \( P_x \) is the probability of being in state \( x \), and \( E_x \) is the energy of state \( x \).

In a Boltzmann machine the energy is the cost function and the temperature is the control parameter which has the same unit as the cost function.

Kirkpatrick, Gelatt and Vecchi [1983] showed that simulated annealing is very effective for combinatorial problems such as graph partitioning. At high temperatures a coarse search occurs and equilibrium takes place very quickly, on the other hand at low temperatures the search takes place in the low energy states and it requires a long time to reach equilibrium.

**B. TRAINING PROCEDURE IN THE BOLTZMAN MACHINE**

The training procedure (Ackley, Hinton and Sejnowski [1985]) requires two phases:

**A.**

*Phase 1. \( P_{ij} \)*

1. Clamp the input vector on the input nodes and the output vector on the output nodes.
2. Let the network reach thermal equilibrium as explained above.
3. Collect statistics on the active nodes, that is, for a fixed period of time the activation of the pairs of nodes (connected with each other) are recorded.
   \[ < a_i a_j >^+ = P_{ij}^+ \]

*Phase 2. \( P'_{ij} \)*

1. Clamp the input vector on the input nodes and retain the output nodes free.
2. Let the network reach thermal equilibrium.
3. Collect statistics on the active nodes as in phase 1. \( < a_i a_j >^- = P_{ij}^- \)
B.

Repeat the process for different temperatures, starting with a high temperature and gradually reduce it.

C.

Change the connection weights according to the collected statistics. This can be calculated as

$$\Delta w_{ij} = \varepsilon (p^+_{ij} - p^-_{ij}), \quad (2.36)$$

If $\varepsilon$ is chosen to be small, then a gradient descent in the information theoretical measure, $G$, is performed, where

$$G = \sum_{\alpha,\beta} P^+(I_\alpha, O_\beta) \log \frac{P^+(O_\beta|I_\alpha)}{P^-(O_\beta|I_\alpha)}, \quad (2.37)$$

where $I_\alpha$ is the state vector of the input nodes, $O_\beta$ is the state vector of the output nodes, $P^+$ is the probability distribution measured during phase 1. $P^-$ is the probability distribution measured during phase 2.

$G$ is also called asymmetric divergence and it is the measure of the distance from $P^+(O_\beta|I_\alpha)$ to $P^+(O_\beta|I_\alpha)$. $G$ is zero when the distribution is identical for both cases. $P$ depends on the weights and so does $G$. The gradient descent in $G$ (Ackley and Hinton [1985]) is given as

$$\frac{\partial G}{\partial w_{ij}} = -\frac{1}{T}(p^+_{ij} - p^-_{ij}). \quad (2.38)$$

D.

Repeat the whole process using different vector pairs.

This process is repeated until the network converges on the training set.
2.2.6 Competitive Learning

This type of learning is based on unsupervised learning, where the output vectors are not known during the learning process. These were developed by Von der Molsburg [1973], Fukushima [1975] and Grossberg [1976].

A. NETWORK ARCHITECTURE

The network consists of one or more competitive layers (Souček and Souček [1988], Zeidenberg [1990]). Each layer is split into a number of inhibitory clusters. These clusters contain a number of nodes. All these nodes within a cluster try to inhibit each other. Hence only one node at a time can be active in a cluster. Each node in a layer is connected to all the nodes in the previous layer. The connection strength between two layers is always positive (excitatory) and the sum of the connection weights to a node is always one.

$$\sum_{j} w_{ij} = 1 \quad (2.39)$$

Figure 2.16 is a network with two competitive layers.

Figure 2.16 The Competitive learning network.
B. THE LEARNING ALGORITHM

There are many different learning algorithms available and one of the simplest algorithms is due to Rumelhart and Zipser (Rumelhart and McClelland [1986]). This algorithm can be described as follows:

1. The output of each node in a cluster is calculated by taking the weighted sum of all its inputs as in Equation (2.8). The node becomes active if net\(_i > \text{net}_j\) for all \(j\) in a cluster. This can be given as

\[
a_i = \begin{cases} 
1 & \text{if } \text{net}_i > \text{net}_j \quad \forall \ j, \ j \neq i \\
0 & \text{otherwise}
\end{cases}
\]  

(2.40)

If a tie occurs \((a_i = a_j)\) then the node on the left is selected.

2. The connection weights of the winning node are updated as follows:

\[
\Delta w_{ij} = \begin{cases} 
0 & \text{if node } i \text{ loses on pattern } p \\
\alpha \frac{a_p j}{x_p} - \alpha w_{ij}, & \text{if node } i \text{ wins on pattern } p
\end{cases}
\]  

(2.41)

where \(x_p\) is the number of active nodes in the pattern \(p\), and \(\alpha\) is the learning parameter \(0 < \alpha < 1\).

Equation (2.41) shows that when a node wins then each input to this node gives up some of its weights and these weights are equally distributed among the active inputs of the node. This learning algorithm can be presented in a simple geometrical model. Let us assume that we have an input vector with four components of unit length. These can be represented as a point on a unit sphere as shown in Figure 2.17

![Figure 2.17](image)

**Figure 2.17** The process of Competitive learning where \(x\)'s represent the input vectors and \(o\)'s represent weight vectors.
The weights can also be represented as points on the sphere. During the learning algorithm the weights are moved close to the input vector in hand. When each weight vector reaches the centre of gravity of the input vector, the process is said to stabilize.

2.2.7 The Hamming Net

In communication theory, it is required to classify binary fixed length signals which are passed through some communication channel. The classifier calculates the Hamming distance from the examplar for all the classes. The Hamming distance is calculated as the number of the input bits which does not agree with the examplar,

\[ d = \sum_j \text{XOR} (x_j, y_j), \]  

(2.42)

where \( x_j \) is the \( j^{th} \) examplar value and \( y_j \) is the \( j^{th} \) input value. The class with the minimum Hamming distance is selected to be the required result. This idea was used in ANN by Lippmann [1987] and is called the Hamming net.

A. HAMMING NETWORK CONFIGURATION

These networks consist of two layers. The first layer calculates the matching scores and the second layer selects the maximum matching. The input vectors are binary patterns and these inputs are connected to the first layer. The class examples are encoded in the weights of these connections. Each node in the first layer is connected to one of the nodes in the second layer, therefore the number of nodes in the first layer is equal to the number of the
nodes in the second layer. The nodes in the second layer are connected with each other, the weights of these connections can be calculated as

$$w_{ij} = \begin{cases} 1 & \text{for } i = j \\ -\varepsilon & \text{for } i \neq j, \ varepsilon < \frac{1}{O} \ i = j = 1, 2, \ldots, O. \end{cases}$$  \hspace{1cm} (2.43)$$

where $O$ is the number of output nodes.

Figure 2.18 illustrates a Hamming net with $I$ inputs and $O$ outputs. The nodes in the second layer use threshold logic nonlinearity functions to calculate their new states. These states are calculated iteratively as follows:

$$a_i = f_t \left( a_i - \varepsilon \sum_{j \neq i} a_j \right)$$

where $f_t(y) \propto y$, for $y > 0$

$$f_t(y) = 0, \text{ for } y < 0$$ \hspace{1cm} (2.44)$$

Figure 2.18 Hamming network architecture.

B. THE HAMMING NET ALGORITHM

1. Assign the connection weights.
(a) Assign the connection weights for the first layer as

\[ w_{ij} = \frac{x_{pj}}{2}, \]

\[ \theta_i = \frac{I}{2}, \]

where \( 0 \leq j \leq I - 1, \quad 0 \leq i \leq O - 1 \)

and \( x_{pj} \) is element \( j \) of examplar \( p \).

(b) Assign the connection weights for the second layer according to Equation (2.43).

2. Apply the unknown input vector and calculate the initial states of the nodes in the second layer accordingly to the following equation

\[ a_i = f_i \left( \sum_{j=0}^{I-1} w_{ij}x_j - \theta_j \right). \]

3. By using Equation (2.44) iterate until convergence is reached then the output of this step will be the required result.

It can be seen from this algorithm that the output of a node is calculated from the output values of the previous iteration. The output \( a_i \) is laterally inhibited by the other outputs after a number of iterations. The output with minimum Hamming distance will have a positive value and all other nodes will go to zero. Lippmann [1987] reported on the performance of such networks, in which he used a Hamming Net with 1000 inputs and 100 outputs. All the output nodes except one were suppressed to zero. He also showed that such a network requires less than 10 iterations to converge.

2.2.8 The Carpenter and Grossberg Classifier

Carpenter and Grossberg [1986] designed a net which can be trained without supervision to form clusters. The structure of these networks are similar to the Hamming nets where matching scores are calculated and the results are used to find the maximum value of these scores. This network differs from the Hamming net in that it has feedback connections from the top layer to the bottom layer as shown in Figure 2.19.
THE TRAINING ALGORITHM

1. Initialize the connection weights as follows:

   \[ w_{ij}(0) = \frac{1}{1 + I}, \]

   \[ w_{ij}^b(0) = 1, \]

   where \( w_{ij}^b(0) \) is the feedforward connection weights between node \( j \) to node \( i \) at time zero. \( w_{ij}(0) \) is the feedback connection weights from \( j \) to \( i \) at time zero.

2. Apply an input to the network.

3. Calculate the state of the nodes in the first layer as

   \[ a_i = \sum_{j=0}^{I-1} w_{ij}(n)x_j, \]

   \[ 0 \leq i \leq O - 1. \]

4. The output of the first layer is fed to the second layer where the largest value is found as in the Hamming Net.
5. To verify that the input truly belongs to the \( i^{th} \) cluster, the output signals are feedback to the input through the feedback links. This is calculated as

\[
\sum_j w_{ij}^b(n)x_j, \tag{2.48}
\]

then \( x \) belongs to \( i^{th} \) cluster if

\[
\frac{\sum_{j=0}^{I-1} w_{ij}^b(n)x_j}{\sum_{j=0}^{I-1} x_j} > \rho, \tag{2.49}
\]

where \( \rho \) is a vigilance parameter.

If this condition is true then proceed with Step 6 otherwise go to Step 7.

6. Update the connection weights according to the following:

\[
w_{ij}^b(n + 1) = w_{ij}^b(n)x_j, \]

\[
w_{ij}^t(n + 1) = \frac{w_{ij}^b(n)x_j}{.5 + \sum_{j=0}^{I-1} w_{ij}^b(n)x_j}. \tag{2.50}
\]

and go to Step 2.

7. The output with maximum value that was chosen in Step 4 is set to zero temporarily, then go back to Step 3.

The vigilance value ranges from 0 to 1.0 and its value indicates the closeness of the input vector from the examplar. Therefore if a large vigilance value is chosen then the input vector should be very close to an examplar to be considered similar. The number of clusters would be large for large vigilance values. On the other hand, if the vigilance value is small then a few clusters with poor matching will result.
In the training algorithm, the largest matching score is found and is compared with the input by calculating the ratio of the dot product of the input and the best matching examplar to the number of ones of the input vector. If this ratio is greater than the vigilance value then the input is considered to be similar to the best matching examplar. The examplar is updated by performing the AND operation between its bits and the input. If the ratio is smaller, then the input is considered to be different from all the examplars and it is added as a new examplar. Each addition of a new examplar requires one node and $2I$ connections to compute matching scores.

### 2.2.9 Kohonen's Self-Organizing Feature Maps

Kohonen [1984] developed a method which was unsupervised learning. This method forms self-organizing maps which resembles the brain and its function. The nodes in the network are arranged orderly during the learning procedure, where the weights are changed such that the nodes close to each other are sensitive to similar inputs.

**NETWORK ARCHITECTURE**

This network consists of an array of nodes that are interconnected with lateral connections. They also receive signals from the input vector as shown in Figure 2.20. The nodes could also be arranged in a two-dimensional form.

![Lateral interconnection of neurons.](image)

**Figure 2.20** Lateral interconnection of neurons.
The activation of a node at the $n^{th}$ iteration is given by

$$a_i = f(\text{Sum}_i(n)),$$

where $\text{Sum}_i(n)$ is the sum of all inputs into node $i$ at the $n^{th}$ iteration and $f$ is a transfer function applied to $\text{Sum}_i(n)$. The $\text{Sum}_i(n)$ is a function of the signals from the input pattern and is also a function of the output signals from the neighbourhood nodes. Therefore

$$\text{Sum}_i(n) = \phi_i(n) + \sum_{j=-M}^{M} \gamma_j a_{i+j},$$

where $\phi_i(n)$ are the signals from the input pattern, $\gamma_j$ specifies the strength of the lateral connection which is a function of the distance $j$. $M$ is the maximum distance that is allowed for nodes to be interconnected.

The lateral connection function $\gamma$ has the appearance of a Mexican hat as shown in Figure 2.21(a). Figure 2.21(b) is the function used in Equation (2.52).

![Figure 2.21](image)

**Figure 2.21** (a) The Mexican Hat function

(b) Lateral interconnection function.
CHAPTER III

Acceleration Methods of Learning ANNs
ANNs learning algorithms have proved to be very slow. This has confined the researchers to apply them only to small scale problems. Many efforts to speedup and improve learning algorithms have taken place (Fahlman [1988], Sliva and Almeida [1990]). A learning rule should satisfy two criteria:

1. The network should respond correctly most of the time.
2. The learning time should be acceptable and reasonable.

The performance of a learning algorithm depends on these two criteria.

In this chapter the problems associated with the Backpropagation learning algorithm are identified. Some acceleration methods for such learning algorithms are discussed and alternative Heuristic procedures to improve the performance of the Backpropagation algorithm are also considered.
3.1 Nonlinear Optimization Techniques

There are many optimization techniques for finding parameters to minimize a cost function. The parameters in ANNs are the connection weights and the cost function. The cost function is a measure of the error between the target and response value. Optimization algorithms based on gradient procedures have proved to be very effective in many different problems. These algorithms are iterative and at each iteration a search direction $d_n$ and a step $\alpha_n$ (the learning rate) are calculated and the connection weights are updated using $d_n$ and $\alpha_n$ according to the following equation.

$$w_{n+1} = w_n + \alpha_n d_n, \quad (3.1)$$

where $n$ is the iteration number.

Optimization algorithms differ from each other in the way they determine $\alpha_n$ and $d_n$. Using the gradient descent methods the objective function is reduced at every iteration.

Hence

$$E_{n+1} < E_n, \quad (3.2)$$

where $n \geq 0$.

Optimization algorithms (Becker and Le Cun[1988]) can be divided into two major categories i.e., stochastic and deterministic gradient methods. In the deterministic gradient method the gradient of all points are calculated and accordingly the weights are updated. On the other hand, in stochastic gradient methods (Lee [1987], Bottou [1991]) the gradient of a single training vector is calculated and the weights are updated giving an approximation to the gradient. In this case more steps are involved in the search. The learning rates modification can be divided into two major categories i.e.,
deterministic or heuristic methods. In the deterministic method a line search technique is used to calculate a value for the learning rate. On the other hand, in the heuristic method every weight of the network should be given its own learning rate and that these learning rates should be changed with time. Some of these methods are illustrated in Figure 3.1.

![Optimization Methods](image)

**Figure 3.1 Optimization Methods and their Categories**

### 3.2 Computing the Learning Rate

For a gradient descent method to converge, the learning rate $\alpha$ must produce a sufficient decrease in $E$. A curve fitting method can be used to calculate a value for $\alpha$ which gives the maximum decrease in $E$. These are iterative methods and based on drawing a smooth curve which pass through the previously measured points derived from line searches and an estimate for the minimum point can be calculated from this curve. Many different methods can be employed depending on the derivatives of the objective function and the number of the previous points needed to calculate the new position. Some of the fitting methods are described below.
3.2.1 NEWTON'S METHOD

It is required to minimize the objective function (Scales [1985], Ortego and Rheinboldt [1970]) along a particular direction of search.

$$\min_{\alpha_k} \left( E (w_n + \alpha_k d_n) \right), \quad (3.3)$$

where $k$ is the iteration number within the line search. Let us assume that the three values $E (w_n + \alpha_k d_n)$, $E'(w_n + \alpha_k d_n)$ and $E''(w_n + \alpha_k d_n)$ are known, then it is possible to construct a quadratic function $q(\alpha)$ which agrees with these values.

$$q(\alpha) \approx E_k + E'_k(\alpha - \alpha_k) + \frac{1}{2} E''_k (\alpha - \alpha_k)^2, \quad (3.4)$$

where $E_k = E (w_n + \alpha_k d_n)$. By equating the derivatives of Equation (3.4) to zero, the value of $\alpha$ corresponding to a minimum of $E$ can be calculated.

The derivative of $q(\alpha)$ is calculated as

$$q'(\alpha) \approx E'_k + E''_k (\alpha - \alpha_k) = 0,$$

Therefore

$$\alpha_{k+1} = \alpha_k - \frac{E'_k}{E''_k}. \quad (3.5)$$

By using Equation (3.5) the value of $\alpha_{k+1}$ can be calculated from its previous value $\alpha_k$ and this process is repeated by replacing $\alpha_k$ with its new value $\alpha_{k+1}$ until $|\alpha_{k+1} - \alpha_k| < \epsilon$ where $\epsilon$ is a small value. This process is illustrated in Figure (3.2).
Luenberger [1984] showed that the order of convergence of this method is at least two.

3.2.2 FALSE POSITION METHOD

In Newton's method the new value of $\alpha$ is calculated according to the information at a single point (the previous value of $\alpha$) of the quadratic curve. Beside the value of $\alpha_k$, the first and second derivatives of $E$ are needed. The false position method (Luenberger [1984]), on the other hand, requires more points with less information at each point. So if we have two previous values of $\alpha$ and their first derivatives of $E$ then a quadratic curve can be fitted to these points according to the following equation.

$$q(\alpha) \approx E_k + E_k'(\alpha - \alpha_k) + \left( \frac{E_{k-1}' - E_k'}{\alpha_{k-1} - \alpha_k} \right) \frac{(\alpha - \alpha_k)^2}{2}. \quad (3.6)$$

The minimum point can be calculated by equating the first derivative of Equation (3.6) to zero. Therefore we have

$$\alpha_{k+1} = \alpha_k - E_k' \left( \frac{\alpha_{k-1} - \alpha_k}{E_{k-1}' - E_k'} \right). \quad (3.7)$$
In fact this equation is an approximation of Newton's method. The order of convergence of such methods is 1.618 (Luenberger [1984]). Figure 3.3 illustrates this method.

\[
q(a)
\]

\[
\alpha_1 \quad \alpha_3 \quad \alpha_2
\]

**Figure 3.3** The False Position minimization method.

### 3.2.3 QUADRATIC FIT METHOD

If the objective function values are known at three points \( \alpha_1, \alpha_2 \) and \( \alpha_3 \) then it can be approximated by a quadratic equation such as

\[
q(\alpha) = A \alpha^2 + B \alpha + C, \quad (3.8)
\]

where \( A, B \) and \( C \) are unknown coefficients and can be calculated using the three values of \( \alpha \). The minimum point (Scales [1985], Luenberger [1984]) can be found by equating the derivative of Equation (3.8) to zero. The derivative of Equation (3.8) can be written as

\[
q'(\alpha) = 2A\alpha + B = 0,
\]

hence,

\[
\alpha = -\frac{B}{2A}. \quad (3.9)
\]
This equation shows that $\alpha$ depends on the two coefficients ($A$ and $B$) of Equation (3.8). To calculate the coefficients of Equation (3.8), let us assume that $\alpha_1$ and $\alpha_2$ are the two extreme points and that $\alpha_3$ lies between these two points as shown in Figure (3.4). Let us also assume that the objective function values for these three points are $E_1$, $E_2$ and $E_3$. The $\alpha$ points should be chosen such that

$$E_3 < E_1 \text{ and } E_3 < E_2.$$ 

![Figure 3.4 The Quadratic Fit minimization method.](image)

$A$, $B$ and $C$ are determined by the equations

$$A\alpha_1^2 + B\alpha_1 + C = E_1$$

$$A\alpha_2^2 + B\alpha_2 + C = E_2$$

$$A\alpha_3^2 + B\alpha_3 + C = E_3. \quad (3.10)$$

These equations give values of $A$, $B$ and $C$ as follows:

$$A = \frac{((\alpha_3 - \alpha_2)E_1 + (\alpha_1 - \alpha_3)E_2 + (\alpha_2 - \alpha_1)E_3)}{\Delta}$$

$$B = \frac{((\alpha_2^2 - \alpha_3^2)E_1 + (\alpha_3^2 - \alpha_1^2)E_2 + (\alpha_1^2 - \alpha_2^2)E_3)}{\Delta} \quad (3.11)$$

$$C = \frac{(\alpha_2\alpha_3(\alpha_3 - \alpha_2)E_1 + \alpha_3\alpha_1(\alpha_1 - \alpha_3)E_2 + \alpha_2\alpha_1(\alpha_2 - \alpha_1)E_3)}{\Delta}.$$
where

\[ \Delta = (\alpha_1 - \alpha_2)(\alpha_2 - \alpha_3)(\alpha_3 - \alpha_1). \]

By substituting \( A \) and \( B \) in Equation (3.9) we have

\[ \alpha = \frac{1}{2} \left( \alpha_2^2 - \alpha_3^2 \right) E_1 + \left( \alpha_3^2 - \alpha_1^2 \right) E_2 + \left( \alpha_1^2 - \alpha_2^2 \right) E_3. \]  \hspace{1cm} (3.12)

The new value of \( \alpha \) should lie between the two extreme values \( \alpha_1 \) and \( \alpha_2 \). The process of calculating \( \alpha \) is repeated after replacing one of the points by \( \alpha \).

The following is the procedure of the Quadratic Fit method.

1. Choose a value for \( \alpha_1 \) and calculate \( E_1 \). Also choose a value for \( \alpha_2 \) and calculate \( E_2 \), where \( \alpha_2 = \alpha_1 + h \), and \( h \) is the step length.

2. If \( E_1 < E_2 \) then the third value can be taken to be \( \alpha_3 = \alpha_1 - h \) otherwise the third value is \( \alpha_3 = \alpha_1 + 2h \). Now calculate \( E_3 \).

3. Calculate the new value of \( \alpha \) using Equation (3.12) and calculate \( E(w_n + \alpha d_n) \).

4. If \( \alpha - \alpha_3 \) is less than the required accuracy then \( \alpha \) is taken to be the solution otherwise go to step 5.

5. Remove the point with the highest \( E \) and go back to step 3.
3.2.4 CUBIC FIT METHOD

This method is more accurate than the quadratic fit method (Scales [1985], Luenberger [1984]), where the function is approximated by a cubic polynomial rather than a quadratic polynomial. The cubic polynomial can be written as

\[ q(\alpha) = A\alpha^3 + B\alpha^2 + C\alpha + D. \]  

(3.13)

The derivative of Equation (3.13) is given by,

\[ q'(\alpha) = 3A\alpha^2 + 2B\alpha + C. \]  

(3.14)

To find the minimum of Equation (3.13) then Equation (3.14) is equated to zero. Let us assume that the objective function is known for two values of \( \alpha \) (say \( \alpha_1 \) and \( \alpha_2 \)). Also assume that their derivatives are also known \( \alpha'_1 \) (= \( g_1 \)) and \( \alpha'_2 \) (= \( g_2 \)). By substituting these points in Equation (3.13) and (3.14) and taking \( \alpha_1 = 0 \), we have the following results,

\[ E_1 = D \]

\[ E_2 = A\alpha_2^3 + B\alpha_2^2 + C\alpha_2 + D \]  

(3.15)

\[ g_1 = C, \]

and \( g_2 = 3A\alpha_2^2 + 2B\alpha_2 + C \).

From Equation (3.15) the coefficients of Equation (3.13) can be found as

\[ D = E_1 \]

\[ C = g_1 \]

\[ B = \frac{g_1 + z}{\alpha_2} \]

\[ A = \frac{g_1 + g_2 + 2z}{3\alpha_2^2}, \]  

(3.16)

54
where
\[ z = \frac{3(E_1 - E_2)}{\alpha_2} + g_1 + g_2. \]

Substituting these coefficients in Equation (3.14) we have
\[ \frac{g_1 + g_2 + 2z}{\alpha_2^2} \alpha^2 - 2(g_1 + z) \frac{\alpha}{\alpha_2} + g_1 = 0. \] (3.17)

Therefore
\[ \frac{\alpha}{\alpha_2} = 1 - \frac{g_2 + w - z}{g_2 - g_1 + 2w}, \] (3.18)

where
\[ w = (z^2 - g_1 g_2)^{1/2}. \]

The initial value of \( \alpha_2 \) can be chosen according to the Davidon, Fletcher and Powell equation
\[ \alpha_2 = \min \{ \eta, -2(E_2 - m)/g_1 \}, \] (3.19)

where \( \eta \) is a constant usually taken to be 1 or 2 and \( m \) is a guess value.

The procedure of the Cubic Fit method can be described as follows:

1. Choose \( \alpha_1 \) and calculate \( g_1 \) and \( E_1 \).
2. If \( g_1 \geq 0 \) then search along \( d \) and choose \( \alpha_2 \) using Equation (3.19).
3. Calculate \( E_2 \) and \( g_2 \).
4. If \( g_2 > 0 \) or \( E_2 > E_1 \) then the minimum is bracketed otherwise \( \alpha_2 = 2\alpha_2 \)
   and go back to step 3.
5. Calculate the new value of \( \alpha \) using Equation (3.18).
6. Terminate the process if the required accuracy is achieved otherwise go to step 7.
7. Go back to step 5 using the interval \((0, \alpha)\) if \(g > 0\) or interval \((\alpha, \alpha_2)\) if \(g \leq 0\).

### 3.3 Inexact Line Search

It is impossible to get the exact minimum point therefore a crude minimization can be carried out instead. In ANNs for each cycle of the line search it is required to pass through all the vector pairs and calculate \(E\) and \(g\), hence as the number of iterations increase the computation time increases too. By using inexact line searches accuracy can be sacrificed for computational time.

It is important to find a value of \(\alpha\) that causes a large decrease in \(E\), the inexact line search terminates the search for \(\alpha\) before it converges. So it is important to know when to terminate the line search. There are many different terminating criteria that have been devised for approximate line search algorithms. The terminating criteria should ensure that \(\alpha\) is not too large nor too small. One such criteria is the Armijo and Goldstein test ([Luenberger [1984], Ortego and Rheinboldt [1970]]). The following condition is applied to ensure that \(\alpha\) is not too large.

\[
\phi(\alpha) \leq \phi(0) + \varepsilon \phi'(0)\alpha, \quad (\text{Armijo}) \tag{3.20}
\]

where \(\phi(\alpha) = E(w_n + \alpha d_n)\) and \(\phi'(\cdot)\) is the derivative of \(\phi\) with respect to \(\alpha\) and \(0 < \varepsilon < \frac{1}{2}\).

To ensure that \(\alpha\) is not too small the following condition is applied,

\[
\phi(\alpha) > \phi(0) + (1 - \varepsilon)\phi'(0)\alpha. \quad (\text{Goldstein}) \tag{3.21}
\]
Figure 3.5 shows the acceptable range of $\alpha$ using the Armijo and Goldstein test.

![Armijo and Goldstein Test](image)

**Figure 3.5** Armijo and Goldstein Test.

Another alternative terminating criteria is the Wolfe test and by using this method Equation (3.20) is used to ensure that $\alpha$ is not too large. The following condition is also applied to ensure that $\alpha$ is not too small,

$$\phi'(\alpha) \leq -\eta \phi'(\alpha), \quad (3.22)$$

where $0 \leq \eta < 1$.

This method is used whenever the derivatives of the objective function can be calculated easily. Figure 3.6 shows the acceptable range of $\alpha$. 57
3.4 Computing the Direction of Search

There are an infinite number of choices for the direction value $d$. In this section different methods for specifying $d$ are considered.

3.4.1 STEEPEST DESCENT METHOD

Let us take the linear approximation of $E(w_n + \alpha d_n)$ using a Taylor series expansion i.e.,

$$E(w_n + \alpha d_n) \approx E(w_n) + \alpha g^T(w_n)d_n,$$  \hspace{1cm} (3.23)
where $g(w_n) = \nabla E(w_n)^T$. Now according to Equation (3.23) the largest decrease in $E_{n+1}$ (or $E(w_n + \alpha d_n)$) is obtained by choosing a large negative value of $g_n d_n$, ($g_n = g(w_n)$)

$$
\begin{aligned}
g_n^T d_n < 0. 
\end{aligned}
$$

(3.24)

Then Equation (3.23) can be written as

$$
E_{n+1} - E_n \approx \alpha \|g_n\| \cdot \|d_n\| \cos \theta,
$$

(3.25)

where $\theta$ is the angle between $g_n$ and $d_n$. Now $\alpha \|g_n\| \cdot \|d_n\| \cos \theta$ is most negative when $\theta = \pi$, hence the search direction is most negative when

$$
\begin{aligned}
d_n = -g_n. 
\end{aligned}
$$

(3.26)

When the search direction is chosen to be in the opposite direction of the gradient of the objective function then the method is called the Steepest Descent (Gill, Murray and Wright [1981], Fletcher [1980]).

The procedure of the Steepest Descent method is as follows:

1. Initialize the connection weights by small random values.
2. Calculate the gradient of the objective function for all vector pairs and let $d_n = -g_n$.
3. Calculate $\alpha$ using a line search technique.
4. Update the connection weights according to Equation (3.1).
5. If the network converged to a solution then terminate the process otherwise go back to step 2.
The Rate of Convergence.

Many functions behave like a quadratic in a small region. Therefore, the convergence rate of a quadratic function can apply to non-quadratic functions. In this section, the properties of a quadratic function will be analysed. Let us assume that the function \( E(w) \) is a quadratic in \( w \) such as

\[
E(w) = \frac{1}{2} w^T A w - w^T b, \tag{3.27}
\]

where \( A \) is a symmetric and positive definite \( m \times m \) matrix. Because the matrix is positive definite, all its eigenvalues are positive \( \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_m \).

The minimum of Equation (3.27) can be found by equating the derivative of \( E(w) \) w.r.t \( w \) to zero. Therefore

\[
A w^* = b, \tag{3.28}
\]

where \( w^* \) is the solution.

Let us take the function \( F(w) \) where

\[
F(w) = \frac{1}{2} (w - w^*)^T A (w - w^*) \tag{3.29}
\]

\[
= E(w) + \frac{1}{2} w^*^T A w^*.
\]

Equation (3.29) is the same as Equation (3.27) with an extra constant term \( \frac{1}{2} w^*^T A w^* \). Therefore the minimum point of \( E(w) \) is the same as \( F(w) \) so we will use \( F(w) \) instead of \( E(w) \).

The function \( E(w_n - \alpha g_n) \) can be written in a quadratic form as

\[
E(w_n - \alpha g_n) = \frac{1}{2} (w_n - \alpha g_n)^T A (w_n - \alpha g_n) - (w_n - \alpha g_n)^T b. \tag{3.30}
\]
To minimize this equation (by differentiating with respect to $\alpha$) we have

$$-(w_n - \alpha g_n)^T A g_n + g_n^T b = 0,$$

which gives the following value for $\alpha$,

$$\alpha = \frac{g_n^T g_n}{g_n^T A g_n}. \quad (3.31)$$

Let us assume that $y_n = w_n - w^*$, and by using Equation (3.29) we have

$$\frac{F(w_n) - F(w_n - \alpha g_n)}{F(w_n)} = \frac{2\alpha g_n^T A y_n - \alpha^2 g_n^T A y_n}{y_n^T A y_n}. \quad (3.32)$$

By using Equation (3.31) and $g_n = A y_n$ in Equation (3.32) then we have

$$\frac{F(w_n) - F(w_n - \alpha g_n)}{F(w_n)} = \frac{(g_n^T g_n)^2}{(g_n^T A g_n)(g_n^T A^{-1} g_n)}. \quad (3.33)$$

To find a bound for the convergence rate, the Kantorovich inequality can be used. The Kantorovich inequality states that when $A$ is positive definite, symmetric $m \times m$ matrix and $z$ is a vector then

$$\frac{(z^T z)^2}{(z^T A z)(z^T A^{-1} z)} \geq \frac{4\lambda_{\min} \lambda_{\max}}{(\lambda_{\min} + \lambda_{\max})^2}, \quad (3.34)$$

where $\lambda_{\min}$ and $\lambda_{\max}$ are the smallest and largest eigenvalues of the matrix $A$.

By using Equation (3.34) in Equation (3.33) we have

$$F(w_n - \alpha g_n) \leq \left\{ 1 - \frac{4\lambda_{\min} \lambda_{\max}}{(\lambda_{\min} + \lambda_{\max})^2} \right\} F(w_n) \leq \left( \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} \right)^2 F(w_n) \quad (3.35)$$

Equation (3.35) shows that the convergence rate of the Steepest Descent method is linear, and depends on the two eigenvalues $\lambda_{\min}$ and $\lambda_{\max}$, or their ratio $P$, the $P$ condition number.
3.4.2 THE BACKPROPAGATION

The Backpropagation algorithm can be divided into two methods, the Batch method and the On-line method. The connection weights in the On-line method are updated after passing through a single vector pair. Therefore the gradient $g_n$ used in this method is based on a single vector pair. On the other hand, the Batch method calculates the gradient $g_n$ over the entire vector pairs and the weights are updated once for all the vector pairs. The Batch method is very similar to the Steepest Descent method in that $d_n = -g_n$. However the learning rate $\alpha$ is a constant value rather than being determined by a line search. In the On-line method, if $\alpha$ is chosen to be very small then it can be approximated to the Steepest Descent method.

3.5 The Problems Associated with Backpropagation and Steepest Descent Learning Algorithms

Steepest Descent-like methods require a large number of iterations. This is due to a number of problems associated with the algorithm. Using Equation (3.1) it can be seen that the largest change occurs for weights that have the largest gradient $\partial E(w)/\partial w_{ij}$. This method has many disadvantages and they can be described as follows:

1. A small derivative indicates that the error is shallow, wide and smooth, hence a large change in the weights is possible rather than a small change. On the other hand, a large derivative may indicate that the error surface is steep and therefore a small change in the weights is required rather than a large change.

2. The weights which have large derivatives might mean that they have captured some of the features of the problem and therefore it is required
to retain these weights without too much change and allow weights that are not in use to develop the required features.

3. Using this method the direction of the gradient does not necessarily point towards the minimum along different dimensions of the weights space. This is shown in Figure 3.7.

Figure 3.7 The error surface of two dimensional weight space.

The amount of gradient needed to update the weights is very much dependent on the error surface. Figure 3.8a shows the contour map of an error surface, where the region $x$ is steep and therefore small steps are required for updating the weights. On the other hand, region $y$ is shallow and therefore, large steps can be taken for updating the weights.
Figure 3.8 (a) An error surface with shallow and steep slopes at different regions.

Figure 3.8b is another surface error, where the surface has shallow as well as steep slopes at different directions.

Figure 3.8 (b) An error surface with shallow and steep slopes at different directions.
To analyze the Steepest Descent method, we can see from Equation (3.35) that the convergence rate is dependent on $\lambda_{\text{max}}/\lambda_{\text{min}}$. The eigenvectors corresponding to these eigenvalues are in the same direction as the principal axis of the error surface. This shows that when only one eigenvalue is large then the convergence rate is slow. The gradient is small in the direction of the eigenvector corresponding to $\lambda_{\text{min}}$ and large in the direction of the eigenvector corresponding to $\lambda_{\text{max}}$. If the learning rate is chosen large enough for the weights corresponding to the eigenvector of $\lambda_{\text{min}}$ then it will be too large for the weights corresponding to the eigenvector of $\lambda_{\text{max}}$. Therefore, the learning is dependent on the value of $\lambda_{\text{max}}$, the larger the value of $\lambda_{\text{max}}$ the smaller is the learning rate and hence the slower the convergence rate. Therefore, with a small learning rate and ellipsoidal error-surface the algorithm has a zig-zag path as shown in Figure 3.9.

![Figure 3.9 A zig-zag path of the Steepest Descent process.](image-url)
3.6 Acceleration Methods

There are a number of methods which improves the convergence rate. In this section some of these methods are discussed.

3.6.1 NEWTON’S METHOD

Newton’s method (Fletcher [1980], Gill, Murray and Wright [1981]) is based on approximating the error function $E(w)$ by a quadratic function. If the first and second derivatives of the error function are available, then a quadratic function of $E(w)$ can be obtained using the first three terms of the Taylor-series expansion as

$$E(w) \approx E(w_n) + g_n^T (w - w_n) + \frac{1}{2} (w - w_n)^T H_n (w - w_n). \quad (3.36)$$

This equation can be minimized by equating the derivatives of $E(w)$ to zero. Therefore we have

$$w_{n+1} = w_n - H_n^{-1} g_n, \quad (3.37)$$

where $H_n (= H(w_n))$ is the Hessian matrix which is positive definite. For a nonlinear function, Newton’s method converges quadratically to the solution $w^*$ if $w_0$ is near to the solution. This can be proved as follows:

Let us assume that there are $\rho > 0$, $\rho_1 > 0$ and $\rho_2 > 0$ such that for all $w$ with $|w - w^*| < \rho$ and also $|H_n^{-1}| < \rho_1$ and that $|g(w^*) - g(w)| - H(w)(w^* - w)| \leq \rho_2 |w - w^*|^2$.

Let us also assume that $w_n$ is selected with $\rho_1 \rho_2 |w_n - w^*| < 1$ and $|w_n - w^*| < \rho$. Then from Equation (3.36) we have

$$|w_{n+1} - w^*| = |w_n - w^* - H_n^{-1} g_n|$$

$$= |H_n^{-1} [g(w^*) - g_n - H_n (w^* - w_n)]|$$

$$\leq |H_n^{-1}| \rho_2 |w - w^*|^2$$

$$\leq \rho_1 \rho_2 |w - w^*|^2 < |w_n - w^*|.$$
Hence the convergence rate is of order two. As was mentioned earlier Newton's method converges to the solution when initially \( w \) is near the solution, therefore Equation (3.37) should be modified to apply to problems far from the solution. This can be achieved by the following modification:

\[
    w_{n+1} = w_n - \alpha_n H_n^{-1} g_n. \tag{3.39}
\]

Another modification is required when the Hessian matrix is not positive definite. This can be achieved by adding some constant values to \( H(w_n) \) as

\[
    H_{\text{new}}(w_n) = \varepsilon_n I + H(w_n), \tag{3.40}
\]

where \( I \) is the identity matrix and \( \varepsilon_n \) is any number which makes the Hessian matrix positive definite.

### 3.6.2 CONJUGATE DIRECTION METHOD

There is a number of practical problems associated with Newton's method. It requires a large number of calculations to get the Hessian matrix as well as the inverse of \( H(w) \), the method also requires a large memory allocation. Conjugate Direction methods are intermediate between the Steepest Descent method and the Newton method. They accelerate the Steepest Descent method with a reduction in the calculations needed by the Newton method (Luenberger [1984], Gill, Murray and Wright [1981]).

Let us assume that the error function is quadratic in \( w \) (it can be approximated to a quadratic function) as

\[
    E(w) = c - b^T w + \frac{1}{2} w^T A w, \tag{3.41}
\]

where \( A \) is a symmetric positive definite matrix \((m \times m)\). Two vectors \( d_n \) and \( d_j \) are said to be \( A \)-orthogonal or conjugate with respect to \( A \) if \( d_n^T A d_j = 0 \). A
finite set of vectors \( d_0, \ldots, d_{m-1} \) is said to have a conjugate basis if \( d_n^T A d_j = 0 \) for all \( n \neq j \). The conjugate direction method for any quadratic function of the form of Equation (3.41) can be represented as Equation (3.1) with \( d_n \) being \( A \)-orthogonal vectors, and \( \alpha_n \) is calculated using a line search technique. From Equation (3.41) we have

\[
E(w_n + \alpha_n d_n) = c - b^T (w_n + \alpha_n d_n) + \frac{1}{2} (w_n + \alpha_n d_n)^T A (w_n + \alpha_n d_n). \tag{3.42}
\]

To find the minimum of (3.42), we differentiate the equation with respect to \( \alpha \) so that

\[
E'(w_n + \alpha_n d_n) = -b^T d_n + (w_n + \alpha_n d_n)^T A d_n = 0.
\]

Therefore, we have

\[
\alpha_n = \frac{-(Aw_n - b)^T d_n}{(Ad_n)^T d_n}. \tag{3.43}
\]

\( d_0, d_1, \ldots, d_{m-1} \) are linearly independent and therefore the solution \( w^* \) can be expressed as

\[
w^* = \alpha_0 d_0 + \ldots + \alpha_{m-1} d_{m-1}, \tag{3.44}
\]

for some set of \( \alpha_n \)'s. Multiplying Equation (3.44) by \( A \) and taking the scalar product with \( d_n \) we have

\[
\alpha_n = \frac{d_n^T A w^*}{d_n^T A d_n}. \tag{3.45}
\]

Also from Equations (3.43) and (3.44) we have

\[
w^* = \sum_{n=0}^{m-1} \frac{d_n^T (Aw_n - b)}{d_n^T A d_n} d_n. \tag{3.46}
\]

This method converges to the solution in at most \( m \) iterations provided that the function \( E(w) \) is quadratic. This can be shown as follows. Let us assume that \( d_0, \ldots, d_{m-1} \) form a conjugate basis with respect to \( A \). To show
that the algorithm converges to the solution in at most \( m \) iterations we have to show that \( w_n = A^{-1}b \) for \( n \leq m \). From Equation (3.1) and (3.43) we have

\[
(Aw_{n+1} - b)^T d_j = (Aw_n - b)^T d_j - [(Aw_n - b)^T d_n / ((Ad_n)^T d_n)] (Ad_n)^T d_j,
\]

for any \( 0 \leq j \leq m - 1 \), from the \( A \)-orthogonality of the \( d_j \) this results in

\[
(Aw_{n+1} - b)^T d_j = \begin{cases} (Aw_n - b)^T d_j & j \neq n \\ 0 & j = n. \end{cases}
\]

Therefore

\[
(Aw_n - b)^T d_j = (Aw_{n-1} - b)^T d_j = \ldots = (Aw_{j+1} - b)^T d_j = 0.
\]

which implies that \( Aw_n = b \) when \( n < m \).

### 3.6.3 THE CONJUGATE GRADIENT METHOD

The Conjugate Gradient method is the Conjugate Direction method where the directions are not specified beforehand but calculated sequentially at each step of the iteration. So at the \( n^{th} \) iteration the negative gradient is calculated and added to the linear combination of the previous direction vectors to obtain a new Conjugate Direction vector.

#### A. The Conjugate Gradient Algorithm for Quadratic Form.

Let us assume that the initial value of \( w \) is \( w_0 \) and the search direction (gradient) is \( d_0 \) as

\[
d_0 = Aw_0 - b.
\]

The weights are updated as

\[
w_{n+1} = w_n + \alpha_n d_n,
\]
where
\[ \alpha_n = \frac{(Aw_n - b)^T d_n}{(Ad_n)^T d_n} \]

\[ d_{n+1} = Aw_{n+1} - b - \beta_n d_n \]

and
\[ \beta_n = \frac{(Aw_{n+1} - b)^T A d_n}{(Ad_n)^T d_n} . \]

The first iteration of this algorithm is identical to the Steepest Descent method, each succeeding iteration moves in a direction that is a linear combination of the current gradient and the preceding direction vector. This method is slightly more complicated than the Steepest Descent method but it converges in a fewer number of iterations. The convergence of an iterative method depends on the condition number \( P(\lambda_{\text{max}}/\lambda_{\text{min}}) \) of the matrix \( A \).

As was shown previously the convergence rate of the Steepest Descent \( \sim O(P) \), whereas the convergence rate of Conjugate Gradient method \( \sim 0 \left( P^\frac{1}{2} \right) \).

The final result for the Conjugate Gradient method is that it satisfies the equation \( w_n = A^{-1} b \) for some \( n \leq m \) and also
\[ (Ad_n)^T d_j = 0, \quad n \neq j \quad 0 \leq n, \quad j \leq m. \]

This shows that \( d_i \) is a Conjugate Direction Vector.

**B. Generalization of the Conjugate Gradient Algorithm**

We would like to generalize the Conjugate Gradient method to non-quadratic functions. One such method is the Daniel Algorithm. The search direction is calculated as
\[ d_{n+1} = g_{n+1} - \beta_n d_n, \quad (3.47) \]
where
\[
\beta_n = \frac{g'_{n+1}g_n^T d_n}{g'_{n+1}d_n d_n}. 
\]  
(3.48)

\(g'(w)\) should be positive definite to ensure that the denominator of \(\beta_n\) does not vanish. The algorithm can be described as follows:

1. Initialize the connection weights by small random values.

2. Calculate the gradient of the objective function for all vector pairs and calculate \(d_{n+1}\) using Equation (3.47). If \(n = 0\) then \(\beta_0 = 0\) else calculate \(\beta_n\) using Equation (3.48).

3. Calculate \(\alpha\) using a line search technique.

4. Update the connection weights according to Equation (3.1).

5. If the network converges to the solution then terminate the process otherwise go back to step 2.

Another method is the Fletcher-Reeves algorithm, the only difference from the previous algorithm is the calculation of the search direction value. Hence \(d_{n+1}\) is calculated as

\[
d_{n+1} = g_{n+1} - \beta_n d_n,
\]

where
\[
\beta_n = \frac{g_{n+1}^T g_{n+1}}{g_n^T g_n}.
\]

This method has the advantage over the Daniel algorithm as only the first derivative of \(E(w)\) is required to be calculated rather than \(g'(w)\).

### 3.6.4 QUASI-NEWTON METHOD

The idea of the Quasi-Newton method (Scales [1985], Gill, Murray and Wright [1981], Fletcher [1980]) is to approximate the inverse Hessian matrix...
in place of the true inverse as required by Newton's method. This method is similar to the Conjugate Gradient method in that the information gathered to minimise a quadratic form requires at most m iterations for an m-dimensional problem.

According to Newton's method the connection weights are updated as:

$$w_{n+1} = w_n - \alpha_n H_n^{-1} g_n.$$ (3.49)

One classical modification of Newton's method is to calculate the Hessian matrix at the initial point $w_0$ and then use it for all other iterations, so

$$w_{n+1} = w_n - \alpha_n H_0^{-1} g_n.$$ (3.50)

The performance of such an algorithm depends very much on how fast the Hessian matrix changes.

For a quadratic form the following is the Quasi-Newton condition,

$$\dot{w}_{n+1} = w_n + H^{-1} (g_{n+1} - g_n).$$ (3.51)

Usually it is impractical to calculate $H^{-1}$ from equation (3.51). Therefore in the Quasi-Newton method a sequence of matrices are built-up and in at most m iterations they approach the true inverse Hessian matrix. So the Quasi-Newton update rule is

$$w_{n+1} = w_n - \alpha_n B_n g_n,$$ (3.52)

where $\lim_{n \to \infty} B_n = H^{-1}$.

**How to calculate $B_n?$**

There are a number of ways to calculate $B_n$. In this section some of these methods are described.
One method is the Rank one update procedure, the Hessian matrix $H$ as well as $H^{-1}$ are symmetric, therefore $B_n$ should be symmetric too. We can define the new value of $B$ as

$$B_{n+1} = B_n + a_n z_n z_n^T.$$  

(3.53)

The term $a_n z_n z_n^T$ is a matrix of rank one.

Let us assume that $\Delta w_n = w_{n+1} - w_n$ and $\Delta g_n = g_{n+1} - g_n$ hence

$$B_{n+1} \Delta g_n = \Delta w_n.$$  

(3.54)

From Equation (3.54) and (3.53) we have

$$\Delta w_n = B_{n+1} \Delta g_n = B_n \Delta g_n + a_n z_n z_n^T \Delta g_n.$$  

(3.55)

By multiplying Equation (3.55) with $\Delta g_n$ we have

$$\Delta g_n^T \Delta w_n = \Delta g_n^T B_n \Delta g_n + a_n (z_n^T \Delta g_n)^2.$$  

(3.56)

Equation (3.53) can be rewritten using Equation (3.56) as

$$B_{n+1} = B_n + \frac{(\Delta w_n - B_n \Delta g_n)(\Delta w_n - B_n \Delta g_n)^T}{a_n (z_n^T \Delta g_n)^2}.$$  

(3.57)

Using equation (3.56) we have

$$B_{n+1} = B_n + \frac{(\Delta w_n - B_n \Delta g_n)(\Delta w_n - B_n \Delta g_n)^T}{\Delta g_n^T (\Delta w_n - B_n \Delta g_n)}.$$  

(3.58)

The connection weights are updated using Equation (3.52) where the search direction $d_n$ is

$$d_n = -B_n g_n.$$  

(3.59)

The parameter $\alpha$ is calculated using a line search technique. There are a number of problems associated with such methods. The new Hessian matrix $B_{n+1}$ calculated by Equation (3.58) can be positive definite only if
\[ \Delta g_n^T(\Delta w_n - B_n \Delta g_n) > 0. \] This is not always possible. Also if it is positive definite it might cause problems if it is small.

Another method was suggested by Davidon and developed by Fletcher and Powell. This method is called a rank two correction procedure where the inverse of the Hessian matrix \( B_n \) is updated by two symmetric rank one matrices. The new value of \( B_{n+1} \) is calculated as

\[
B_{n+1} = B_n + \frac{\Delta w_n \Delta w_n^T}{\Delta w_n^T \Delta g_n} - \frac{B_n \Delta g_n \Delta g_n^T B_n}{\Delta g_n^T B_n \Delta g_n}. \tag{3.60}
\]

The algorithm can be described as follows

1. Choose any symmetric positive definite matrix \( B_0 \) and initialize the weights.
2. Set \( d_n = -B_n g_n \).
3. By using a line search technique calculate a value for \( \alpha \) which minimizes the error function.
4. Calculate \( \Delta g_n \) and \( B_{n+1} \).
5. Update the connection weights according to Equation (3.1).
6. If the network converges to the solution then terminate the process otherwise go back to step 2.

### 3.6.5 The RUNGE MODIFICATION RULE

It was shown by Stornetta and Huberman [1987] that the Backpropagation algorithm can be accelerated by changing the dynamic range of the input, hidden and output nodes. This is done by taking the range \(-\frac{1}{2}\) to \( \frac{1}{2} \) rather than 0 to 1. Then the sigmoid function has to be rewritten as

\[
-\frac{1}{2} + \left[ \exp\left( -\sum_j w_{ij} a_j + \theta_i \right) + 1 \right]^{-1}. \tag{3.61}
\]

The improvement is due to the fact that when the input signal is zero the weights will not change. On the other hand, if the zero was replaced by \(-\frac{1}{2}\) then the weights will be modified by some amount. It was reported that an improvement of 30-50% is possible.
3.7 Acceleration Methods using Heuristic Algorithms

In Section 3.5 the problems associated with the Backpropagation and Steepest Descent learning algorithms were analysed. A number of changes can be implemented to improve the convergence rate by taking the following points into account.

1. During the update of the connection weights, different learning rates are required for different directions. This is due to the fact that the error surface has different gradients in different directions.

2. The learning rates should be changed with time, due to the fact that the error surface changes with time.

3. If the gradient of a point does not change sign for a number of iterations then the learning rate can be increased for that point. Also if the gradient changes sign for a number of iterations then the learning rate should be decreased.

When the learning rate is different for different dimensions the method no longer performs a gradient descent. Therefore the Heuristic method (Sutton [1986], Saridis [1970], Almeida [1987]) does not necessarily perform gradient descent, and there are some error surfaces where the Heuristic methods do not improve the convergence rate.

In this section a number of existing Heuristic algorithms are described.

3.7.1 MOMENTUM RULE

Rumelhart and McClelland [1986] introduced an acceleration method for the Backpropagation by adding a momentum term to the weight update equation and it is defined as

$$\Delta w_{n+1} = -\alpha g_n + \beta \Delta w_n,$$  \hspace{1cm} (3.62)
where $\beta$ is the momentum constant.

It can be seen from Equation (3.62) that the weight update rule depends on the current gradient $g_n$ as well as the gradients of the previous iterations. Equation (3.62) can be rewritten as

$$\Delta w_{n+1} = -\alpha \sum_{i=0}^{n} \beta^i \nabla E(w_{n-i}).$$  \hspace{1cm} (3.63)

The addition of a momentum term is considered to perform a Heuristic algorithm (Watrous [1987], Parker [1987]). This is due to the fact that when the gradients of two consecutive iterations have the same sign then $\Delta w_{n+1}$ increases causing a large change in the connection weight. On the other hand, if the gradients for two consecutive iterations have different signs then $\Delta w_{n+1}$ is reduced hence a smaller change is caused in the connection weight. The solution of a first-order difference equation in the general form $x_{n+1} = a_n x_n + b_n$ is given by

$$x_{n+1} = \prod_{i=1}^{n} a_i x_1 + \sum_{j=1}^{n} \prod_{k=j+1}^{n} a_k b_j.$$  \hspace{1cm} (3.64)

The momentum update rule of Equation (3.62) can be rewritten using Equation (3.64) as

$$\Delta w_{n+1} = \beta^n \Delta w_1 + \alpha \sum_{j=1}^{n} \beta^{n-j} (-g_j).$$  \hspace{1cm} (3.65)

In addition, we have $\Delta w_{n+1} = w_{n+1} - w_n$ and $\Delta w_1 = w_1 - w_0 = -\alpha g_0$ in Equation (3.65) hence

$$w_{n+1} = w_n + \alpha \sum_{j=0}^{n} \beta^{n-j} (-g_j).$$  \hspace{1cm} (3.66)

This equation is also a first-order difference equation with $a_n = 1$ and $b_n = \alpha \sum_{j=0}^{n} \beta^{n-j} (-g_n)$ so

$$w_{n+1} = w_0 + \alpha \sum_{k=0}^{n} \beta^k \sum_{j=0}^{n-k} (-g_j).$$  \hspace{1cm} (3.67)
Let us assume that the gradient is constant that is
\[ g_{n+1} = g_n = g. \]

Therefore, Equation (3.67) can be rewritten as
\[
w_{n+1} = w_0 - \alpha \left( \sum_{k=0}^{n} \beta^k (n-k+1) \right) g. \tag{3.68}
\]

Now Equation (3.68) can be rewritten in terms of a difference of infinite sums as
\[
w_{n+1} = w_0 - \alpha \left[ (n+1) \sum_{k=0}^{\infty} \beta^k - (1 - \beta^{n+1}) \sum_{k=0}^{\infty} k \beta^k \right] g. \tag{3.69}
\]
and by using binomial series expansion we have
\[
w_{n+1} = w_0 - \alpha \left( \frac{n+1}{1-\beta} \right) \left[ 1 - \frac{(1 - \beta^{n+1})}{n+1} \frac{\beta}{1-\beta} \right] g \tag{3.70}
\]
for \( \beta^2 < 1 \).

As \( n \) gets large, \( \frac{1 - \beta^{n+1}}{n+1} \) becomes small and the acceleration factor approaches \( \frac{1}{1 - \beta} \).

Therefore it can be seen that the momentum term increases the learning rate by a factor \( \frac{1}{1 - \beta} \) in regions that has a constant gradient.

3.7.2 DELTA-DELTA LEARNING RULE

This method has two update rules (Jacobs [1988]), one is the weight update rule and the other is the learning rate update rule. This algorithm was introduced by Barto and Sutton [1981] and it can be expressed as
\[
w_{ij}(n+1) = w_{ij}(n) - \alpha_{n+1} \frac{\partial E_n}{\partial w_{ij}(n)}, \tag{3.71}
\]
where $w_{ij}(n)$ is the value of the connection weight from node $j$ to node $i$ at the $n^{th}$ iteration, and the learning rate $\alpha_{n+1}$ is updated at every iteration. Using the steepest gradient on the error surface we have

$$\alpha_{n+1} = \alpha_n + \Delta \alpha_n,$$

where $\Delta \alpha_n$ is the amount of change in the learning rate. To calculate $\Delta \alpha_n$, the error function is minimized with respect to the learning rate. So we have

$$E_n = \frac{1}{2}(t_i - a_i)^2$$

$$= \frac{1}{2} \left( t_i - \frac{1}{1 + e^{-\text{net}_i}} \right)^2.$$  

We want to calculate $\frac{\partial E_n}{\partial \alpha_n}$, using the chain rule as

$$\frac{\partial E_n}{\partial \alpha_n} = \frac{\partial E_n}{\partial \text{net}_i} \frac{\partial \text{net}_i}{\partial \alpha_n},$$

where

$$\text{net}_i = \sum_j w_{ij}(n)a_j = \sum_j (w_{ij}(n - 1) - \alpha_n g_{ij}(n - 1))a_i$$

$$\frac{\partial \text{net}_i}{\partial \alpha_n} = -g_{ij}(n - 1)a_i,$$

where $g_{ij}(n - 1) = -\frac{\partial E_{n-1}}{\partial w_{ij}(n - 1)}$.

Now we have

$$\frac{\partial E_n}{\partial \text{net}_i} = (t_i - a_i) \left( \frac{1}{(1 + e^{-\text{net}_i})^2} e^{-\text{net}_i} \right)$$

$$= (t_i - a_i)a(1 - a_i)$$

$$\frac{\partial E_n}{\partial \alpha_n} = -(t_i - a_i)a_i(1 - a_i)a_jg_{ij}(n - 1)$$

$$= \frac{\partial E_n}{\partial w_{ij}(n)} \frac{\partial E_{n-1}}{\partial w_{ij}(n - 1)}$$

78
\[ \Delta \alpha_n = \gamma \frac{\partial E_n}{\partial w_{ij}(n)} \frac{\partial E_{n-1}}{\partial w_{ij}(n-1)} \]  \hspace{1cm} (3.76) 

where \( \gamma \) is the step size.

From Equation (3.76) we can see that when the gradient sign of two consecutive iterations are the same then the learning rate is increased. On the other hand, when the gradient signs are different for two consecutive iterations the learning rate is reduced.

There are a number of problems associated with this algorithm, first when the point is in a shallow region then most probably the gradient for two consecutive iterations is small and therefore \( \gamma \) should be chosen to be a high value to make \( \Delta \alpha_n \) a reasonable value. By choosing a large value of \( \gamma \) the learning rate \( \alpha \) might increase rapidly causing large connection weights which will have zero gradients. Secondly, in steep regions the gradient signs of two consecutive iterations might be different with large magnitudes therefore their product would be a large negative value. With \( \gamma \) having a large value the learning rate reduces rapidly and it might become a negative value which will cause the error to increase rather than diminish. Hence \( \gamma \) should be chosen to be a small value giving a small improvement over the Steepest Descent method.

3.7.3 DELTA-BAR-DELTA RULE

Jacobs [1988] introduced a Heuristic method to improve the convergence rate of Steepest Descent like methods. This method as in the Delta-Delta Rule uses two update rules, one is the weight update rule and the other is the learning update rule. The weight update rule is the same as Equation
The learning rate update rule differs from the previous method and can be described as follows:

\[
\Delta \alpha_n = \begin{cases} 
\kappa & \text{if } \tilde{g}_{ij}(n-1)g_{ij}(n) > 0 \\
-\gamma \alpha_{n-1} & \text{if } \tilde{g}_{ij}(n-1)g_{ij}(n) < 0, \\
0 & \text{otherwise}
\end{cases}
\]

(3.77)

where

\[
\tilde{g}_{ij}(n) = (1 - \vartheta)g_{ij}(n) + \vartheta \tilde{g}_{ij}(n-1),
\]

\(\tilde{g}_{ij}(n)\) is the exponential average of the current and previous derivatives and \(\vartheta\) is the extrapolation factor. This algorithm is a Heuristic method due to the fact that when two consecutive gradient signs are the same then the learning rate is increased. On the other hand, when the two consecutive gradient signs are different then the learning rate is reduced.

This algorithm rectifies the problems associated with the Delta-Delta rule. This can be seen from Equation (3.77) where the learning rate is increased by a constant value \(\kappa\) to ensure that it does not increase rapidly. Also the learning rate is reduced exponentially keeping it positive. The momentum term is also used with this algorithm and it is called the hybrid rule.

### 3.7.4 HEURISTIC LEARNING ALGORITHM (HLA)

This is another Heuristic algorithm which was introduced by Cater [1987]. In this method the learning rate is increased for the input vector that causes the maximum absolute error e.g. \(\max |t_{pi} - a_{pi}|\). The new learning rate for this vector is calculated as

\[
\alpha_p = 2\alpha + |t_{pj} - a_{pj}|.
\]

(3.78)

This learning rate is used for vector \(p\) in the next iteration. This method might cause the error to oscillate or get stuck in a local minima. To avoid
this problem later Cater[1987] suggested an extra condition whereby if the total error of the network increased then the learning rate is reduced by half and it is used for all the weights so we have:

IF

\[ \sum \sum |\delta_{pi}|_{n+1} >= 1.01 \sum \sum |\delta_{pi}|_{n} \]  

(3.79)

then

\[ \alpha = \alpha/2 \]

ELSE

\[ \alpha = \alpha. \]

The momentum term can also be used with this method.

3.7.5 QUICKPROP

Fahlman [1988] developed a new learning algorithm to accelerate the Backpropagation method called it "Quickprop". This method is a Heuristic algorithm based loosely on Newton's method. The algorithm requires the error derivative of the previous iteration \( \partial E_{n-1}/\partial w_{ij}(n-1) \) as well as the error derivative of the current iteration \( \partial E_{n}/\partial w_{ij}(n) \). Two risky assumptions are made

1. The error surface is quadratic so that it has only one minima.

2. The change in the error due to a change in one of the connections does not effect the other changes of the weight connections within an iteration.

Therefore to calculate the new weight change of a connection \( \Delta w_{ij}(n) \) both the current and previous error derivatives and the previous weight change are needed. This can be calculated as

\[ \frac{\partial E_{n}/\partial w_{ij}(n)}{\partial E_{n-1}/\partial w_{ij}(n-1) - \partial E_{n}/\partial w_{ij}(n)} \Delta w_{ij}(n-1). \]  

(3.80)
It can be seen from Equation (3.80) that if \( \frac{\partial E_n}{\partial w_{ij}(n)} < \frac{\partial E_{n-1}}{\partial w_{ij}(n-1)} \)
and is in the same direction, then the weight is changed in the same direction as the previous iteration. The amount of change will be different depending on both gradients. If the current gradient is in the opposite direction to the previous gradient then the change in the weight would be in the opposite direction to the previous iteration. The problem occurs when the current value of \( \frac{\partial E_n}{\partial w_{ij}(n)} \geq \frac{\partial E_{n-1}}{\partial w_{ij}(n-1)} \). This causes either taking a large step or moving the weights backwards and probably getting stuck in a local minima. Fahlman resolved this problem by using a new parameter \( \mu \) called "maximum growth factor". The weights are not allowed to change more than \( \mu \Delta w_{ij}(n-1) \) that is

\[
\mu \Delta w_{ij}(n) \leq \mu \Delta w_{ij}(n-1). \tag{3.81}
\]

So if the change in a weight is larger than this amount then

\[
\mu \Delta w_{ij}(n) = \mu \Delta w_{ij}(n-1). \tag{3.82}
\]

The value of \( \mu \) depends on the problem in hand. Fahlman [1988] claims that \( \mu = 1.75 \) works for a large number of problems. From Equation (3.80) we can see that the weight change depends on the previous weight change and if the previous weight change is zero then some bootstrap process is needed. Fahlman [1988] suggested using a learning rate \( \alpha \) with the current gradient. This algorithm was applied to the binary encoder problem and was shown that the speedup achieved was an order of magnitude. On the other hand, Lang and Witbrok [1988] applied this algorithm to a much more complex problem using continuous input values and showed that the Quickprop did somewhat better than the BP algorithm.
CHAPTER IV

Parallel Simulation of Neural Networks
Artificial Neural Networks (ANNs) can be constructed in hardware or as software by using simulation. One of the main problems of the hardware is that the large number of connections between the nodes is difficult to implement. The technology so far is capable of building very sophisticated logic circuits from electronic switches on a small chip. But the problem is that the number of connections allowed is limited due to the fact that the wires of the connecting links must be separated by some minimum distance. Optical technology might solve this problem as it is possible to link one optical gate to any number of optical gates where two beams of light can cross each other without affecting one another. However, optical computing is not as accurate but this is not critical for the functioning of a neural network. ANNs are also fault tolerant and a perfect match between the input and output is not required.

Abu-Mostafa and Psaltis [1987] reported that, at CALTECH, they have built an optical recognition system and concluded that advances in optical materials and manufacturing technologies are still required to achieve their goal.

To date very little computer hardware has been built for ANNs so instead simulations have been designed to suit the existing computers Korn [1991]. These simulators play a key role in developing, testing and training different learning algorithms. Since the ANNs are computationally intensive therefore parallel or vector computers are better suited to such problems.

Simulators can be general purpose or special purpose. Different types of parallel machines are used for different types of networks (Mühlenbein and Wolf [1989], Smith [1987]). For example, fully interconnected networks with synchronous updating can be implemented as a matrix vector product, where the matrix is dense. These types of networks can be implemented efficiently
on vector computers, array computers or systolic arrays. Arbitrary connected networks are more difficult to implement.

Parallelism can be achieved by vector-vector, vector-matrix and matrix-matrix operations. Forrest et al [1987], Forrest et al [1988] used the DAP as well as the MEIKO computing surface. Others have used the CRAY machine.

There are three approaches in the production of neural network implementations.

1. Some companies are producing accelerator boards which can be connected to an existing sequential machine such as the IBM PC. This will increase the speedup of the calculations required by ANNs.

2. VLSI designs, both digital and analogue are being made.

3. Using existing general purpose parallel machines to speedup simulations. Many of these machines have been used such as CM-2 connection machine, CRAY, DAP, WARP, MEIKO computing surface and the Sequent multi processor machine.
4.1 Parallel Processing for ANNs

Two commonly used parallel architectures are SIMD (Single Instruction Multiple Data) and the MIMD (Multiple Instruction Multiple Data) computers.

4.1.1 THE SIMD COMPUTER ARCHITECTURE

Array processors belong to this class of computers. The array processors consist of simple processing elements (PEs) that are synchronized to operate in parallel. Each PE consists of an ALU (Arithmetic Logic Unit) and a number of registers. These PEs are connected to a control unit (CU) where the instructions are decoded and broadcast to all the PEs in the system. Therefore the PEs execute the same instruction at the same time with each PE holding different sets of data. Figure 4.1 depicts a simple Array computer. As can be seen from the diagram the PEs are connected to each other via a data routing network. The shared memory can have multiple modules.

![Figure 4.1 The structure of an SIMD Array processor.](image-url)
The SIMD computer organization can be divided into two modes (a) Word Slice and (b) Bit Slice.

(a) Word Slice is processing one word of k-bits in a time

(b) Bit Slice is processing a string of bits one from each word at the same vertical bit position.

4.1.2 THE MIMD COMPUTER ARCHITECTURE

Most multiprocessor systems and multicomputer systems (Hwang and Briggs [1984], Hockney and Jesshope [1981]) belong to this class of computers. These machines have a smaller number of processors but which are more complex. Each processor is able to execute different instructions. The interprocessor communication can be achieved by two different methods.

1. Loosely Coupled Multiprocessors.

2. Tightly Coupled Multiprocessors.

1. Loosely Coupled Multiprocessors

In this system each processor has a number of input/output devices connected to it and a large local memory where most of the instructions and data are stored. Different processors communicate with each other by exchanging messages via a message transfer system. Such systems are efficient for tasks that require minimum interactions between processors. The message transfer system could be a simple time shared bus or a shared memory system. Figure 4.2 depicts a simple loosely coupled system. The transputer is an example of such an architecture. Beynon and Dodd [1987] and also Beynon [1988] have used such systems for ANNs simulator.
2. Tightly Coupled Multiprocessors

This system consists of a number of PEs, memory modules and input/output channels. These are connected by three interconnection networks and are:

1. The memory processor interconnection:- This allows communication between the processor and memory modules. Every processor can access all the memory modules but two processors cannot access the same memory module at the same time. To reduce conflict the number of memory modules are made to equal the number of processors. Another way to reduce conflict is to have a local memory with every processor.

2. The input/output and processor interconnection:- This interconnection allows every processor to communicate with every input/output device.

3. The interrupt signal interconnection:- This communication link allows each processor to direct an interrupt to any other processor.

In some tightly coupled multiprocessors a cache memory is added to every processor to reduce instruction cycle time, as the cache memory captures most of the references made by the processor. Figure 4.3a depicts a simple tightly coupled multiprocessor system without cache memory and Figure 4.3b is a system with cache memory.
Figure 4.3 The structure of Tightly Coupled system.
The Balance 8000 is an example of such an architecture. The Balance 8000 is a tightly coupled (Bus) based MIMD machine with up to 12, 32 bit multiprocessors each capable of executing 0.7 MIPS. Each processor consists 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. It 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 will reduce the traffic burden on the bus.

4.2 Parallel Programming on MIMD Machines

There are three main parallel algorithms used on MIMD machines (Hwang and Briggs [1984], Stone [1980]). These are

1. Pipelined Algorithms or Macropipelining

In this method a task is divided into subtasks, the output of one part is the input of another. This is depicted in Figure 4.4 each subtask is a process and is executed by a processor. So when a task is divided into $p$ subtasks, then $p$ processors are required. The data should pass through all the processors sequentially, when a process is done with the data in hand the next data is allowed to be processed. If a large amount of data is presented, the speed of the system is approximately $p$ provided that the subtasks have equal execution time, otherwise the slowest subtask becomes a bottleneck. If one of the tasks is slower than the others the processes must be synchronized at the end of each stage. With the use of buffers the operation of the pipeline can be made to be asynchronous. By using this method the processes are heterogeneous where by each process is different from the other processes.
2. **Partitioning or Synchronized Algorithm**

When the data is divided between the processors, each processor will perform the task on the data in its own partition. The results of the processors are combined at the end of the task to give the final solution. Synchronization is needed after the completion of all the processes. This can be shown by the following example. Let us assume that we have a large matrix $m \times m$, and some operation is required on this matrix whereby the operation is row independent. Therefore, every row can be executed on different processors in parallel. If we have $p$ processors where $p \leq m$, then each processor can execute $m/p$ rows. No further operation is allowed until all these processors have completed their operation therefore synchronization is a vital component. Figure 4.5 shows the operation of this algorithm. The speedup of this algorithm is $p$, but it is never possible to achieve this value due to software synchronization overheads.

---

**Figure 4.4** The Operation of Macropipelining

**Figure 4.5** The operation of Partitioning.
By using a tightly coupled system with local and cache memory, each processor will have the process code and data segment. Hence the processors do not require to access the main memory. If there is some variables that are needed to be modified by all the processors then these variables will be held in the main memory and accessed through the bus. This causes memory conflict and the efficiency of this algorithm will depend on the amount of variables in the main memory. It is possible that the tightly coupled system does not have a local or cache memory. Therefore the process code and the data will reside in the main memory. This will cause high contention on the bus due to the fact that each processor requires to access the main memory for process code and data. It can be seen from the above discussion that this algorithm has homogeneous processes. We can also have heterogeneous processes where by the partitioning occurs in the tasks rather than the data. For example, let us assume that we have the matrix $Y$ which is calculated as

$$Y = A.B + (C - D)(E + F).$$

In this case we will require three processors each performing different arithmetic operation. This operation is depicted in Figure 4.6.

![Figure 4.6 Partitioning in the tasks.](image-url)
It can be seen that synchronization is important in this problem too. Stage 2 of $P_2$ cannot be executed without completing stage 1 of processor 2 and 3. Also stage 3 of $P_1$ cannot take place without completing stage 1 of $P_1$ and stage 2 of $P_2$.

It can be seen from this problem that a processor can be blocked, waiting for other processors to finish their operation. It is possible that the waiting time could be large which degrades the performance of this algorithm.

3. Relaxation or Asynchronous Parallel Algorithm

Using this method, the processors may be given different tasks as in pipelined algorithms or they may be given the same task to perform as in partitioning. However in this method the processors are independent from other processors. They work on the most recently available data and therefore no synchronization is needed.

4.3 Factors that limit speedup on MIMD machines

There are a number of factors that can effect the performance of an algorithm. Some of these are:-

1. The size of data: - if the data is small then the speedup of the algorithm might not be apparent due to the overhead caused by the creation of the processes.

2. The number of created processes and synchronization: - the overhead increases with the number of created processes as well as synchronization. Therefore they should be kept small if possible. Synchronization is less expensive then process creation, hence synchronization should be used when possible to avoid recreation of processes.
3. The number of processors: if the algorithm requires to access certain variables frequently by all the processors then the contention for a variable increases with the number of processors, and in turn, degrades the performance of the system.

4. Sequential code: if the problem requires to execute a sequential code, then this will limit the speedup of the parallel algorithm.

5. The work load: this should be balanced among the processors otherwise the performance of the algorithm will be degraded.

4.4 Some Parallel Pascal Features of the Sequential Balance 8000

There are a number of parallel programming languages devised for different machines, and all of them have common features. Lester [1987] described the Multi Pascal simulator which runs on the VAX11/780. In this section some of the Parallel Pascal features of the Balance machine are described (Osterhaug [1989]). The Parallel Programming Library is a set of (routines which can be used by Parallel Pascal) programs.

1. **Process Creation and Termination**

The `m_fork` routine is used to create new processes. A new process is called the 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 default number of processes created are the actual number of processors in the machine divided by three. This number can also 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
the function m_get_myid. The parent ID number is 0. The forking operation on the Balance machine takes about 55ms compared to the division operation which takes only 13.7μs. This indicates that forking too many times should be avoided if possible.

To terminate the processes the routine m_kill_procs is used. No operation is allowed to take place until all the processes terminate their work. In fact, this function acts like synchronization. The following is an example of how to use some of these functions.

```
procedure PROC;
begin
  ...
end;
begin {Main Program}
  writeln ('Input the number of processors');
  readln (p);
  ret_val:=m_get_procs (p); { Set p processes }
  m_fork (PROC);
  m_kill_procs;
end;
```

From the previous program segment, if \( p = 5 \) then 5 processes will be set with each process executing the procedure PROC as shown in Figure 4.7.

![Diagram](image-url)

**Figure 4.7** \( p \) processes executing the procedure PROC in parallel.
It can be seen from Figure 4.7 that \texttt{m\_fork} caused a duplicate copy of the procedure \texttt{PROC} in all the \textit{p} processors. It is possible to create child processes with different code, this can be done by rewriting the \texttt{m\_fork} routine as

\[
\texttt{m\_fork (Proc}_1, \texttt{Proc}_2, \ldots, \texttt{Proc}_p);
\]

where \texttt{Proc}_1, \texttt{Proc}_2, \ldots, \texttt{Proc}_p are \textit{p} different procedures.

The \texttt{m\_park\_procs} routine 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 \texttt{m\_rele\_procs}.

2. Synchronization

The function \texttt{m\_lock} ensures that only one process at a time can access a shared data from the shared memory. This is done by using the \texttt{m\_lock} function before accessing the data, and after accessing the data the function \texttt{m\_unlock} is used allowing the other processes to access the data. During the \texttt{m\_lock} and \texttt{m\_unlock} other processes have to wait by spinning in a tight loop. Another synchronization function is the \texttt{s\_wait\_barrier}. In this case, a barrier is initialized using \texttt{s\_init\_barrier} and the process waits until all the processes have reached this point. This is done by testing and setting a shared variable. \texttt{m\_sync} is another synchronization function, when a process reaches \texttt{m\_sync} it spins until all processes reach \texttt{m\_sync}.

4.5 Parallelism in ANNs

To apply parallelism to ANNs, the principle of partitioning has been used. In the On-line Backpropagation procedure the partitioning strategy is applied to the network. The network is then sliced vertically and each slice is assigned to a processor. Each processor will have a number of nodes from
each layer as depicted in Figure 4.8 where a network with two layers has been partitioned into three slices.

![Figure 4.8 A network with 3 vector slices.]

Synchronization between the layers is a vital point. All processors should be processing on nodes from the same layer. No processor is allowed to start with the next layer until all other processors have completed their calculations. The speedup of the parallel procedure using $p$ processors will depend on:

1. **The number of nodes in a layer.** In fact, the number of nodes in each layer and also the total number of nodes in the network are factors in the speedup operation. Let us assume that a network with two layers i.e. 10 nodes in the hidden layer and 2 nodes in the output layer. Then if the training procedure is run on 5 processors, each processor will have two nodes from the hidden layer and only two of the 5 processors will have one node from the output layer and the remainder will be idle. Therefore the speed up increases with the number of nodes.

2. **The number of connections between the nodes.** The speedup increases with the number of connections. This is due to the fact that more calculations are required for each node, this will reduce the relative time spent distributing the jobs to the processors.
For the Batch Backpropagation procedure the above method of partitioning can be used. Another method is to apply partitioning on the vector set. In this method, the vector set is divided amongst the processors, each processor executing the training procedure for different vector pairs. The connection weights are changed by taking the average change in weights for all vector pairs. Therefore, the connection weights are changed only when all the processors have finished executing one cycle of the training procedure.

The speedup of the parallel procedure using this method depends on the number of the total nodes in the network, the number of connections between the nodes and also the size of the vector set.

4.6 Neural Network Simulator

In neural networks, each node is connected to all or some of the other nodes. The states of the nodes, the available connections between nodes and the connection strengths between the nodes are required during learning. Therefore, a 2-dimensional array is needed to store the connection strengths ($w[i,j] \Rightarrow$ the strength from node $j$ to node $i$). A second 2-dimensional array is required for the connections. The states of the nodes can be stored in a 1-dimensional array $a[j]$.

During learning, the new state of a node say node $i$, can be found by going through all the other nodes ($j$). If there is a connection between node $i$ and node $j$ and also if node $j$ is active then the product of the connection strength $\times$ activation level of node $j$ is added to the summation of node $i$ according to the following equation,

$$\text{net}[i] = \sum_j w[i,j]a[j]. \quad (4.1)$$
With $N$ nodes in the network, $2 \times N$ comparisons are required to calculate a new state of one node. To calculate the new states of all the nodes in the network $2N^2$ comparisons are needed. When the number of connections in the network are less than $N^2$, there will be a considerable waste of time using a 2-dimensional array.

Linked lists using 1-dimensional arrays were used in the Backpropagation simulator. 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. 1-dimensional arrays are used for the state of the nodes. Figure 4.9(a) is an example which depicts the structure of a neural network.

\[ \text{Figure 4.9 (a) Linked list structure} \]

\[ \text{(b) The network associated with linked list.} \]
During the feedforward process only the nodes in the linked list are tested for their states. All the nodes in the list \( a[i] \) are connected to the node \( i \) hence checking on the connection is not required, this will result in faster execution times. In the worst case, the nodes are all fully connected and \( N^2 \) comparisons are required. In the Backpropagation procedure, however, the number of comparisons will never approach \( N^2 \), due to the fact that it is only a feedforward network and there are no feedback connections.

The routine \( m\_fork \) is used to create \( p \) processes. The Backpropagation learning procedure is called “Learning”, the processes are created as follows,

\[
m := m\_fork\ (\text{learning});
\]

\[
m\_kill\_procs;
\]

The \( m\_kill\_procs \) are used to terminate all child processes created by \( m\_fork \). The learning procedure for the On-line method, is slightly different from the Batch method. For the On-line method the partitioning is done on the network whereas, for the Batch method the partitioning is done on the vector set. In the following two sections these learning procedures are described.

**4.6.1 THE ON-LINE BACKPROPAGATION PROCEDURE**

The following is a segment of the On-line learning procedure,

```plaintext
procedure learning;
var
    \( l, u : \text{integer}; \)
begin
    ...
    for \( l := 1 \) to cycle_num do
        for \( u := 1 \) to vect_size do
            begin
                Feedforward;
                Backpropagation;
                Change_weights;
                m_sync
```

```
The variables $e$ and $v$ are local variables, each process will change these values locally. None of the processes require the values of $e$ and $v$ of the other processes, hence it is declared in the procedure itself. On the other hand, cycle_num (number of iterations) and vect_size (total number of vector pairs used in training) are global variables shared by all processes.

As can be seen from the learning procedure, each processor will execute the two nested loops $e$ and $v$ simultaneously. The m_sync causes the processor to wait for the slowest processor to finish with its process before all of them proceed with the next instruction. This synchronization is very important because we do not want to start with the next vector pair and some of the processors are still changing the connection weights due to the previous vector pair.

The following is a segment of the Feedforward procedure,

```pascal
procedure Feedforward;
var
  id, e, i, j : integer;
  nproc : longint;
begin
  nproc := m_get_numprocs;
  id := m_get_myid;

  { Clamp the input vector on the input nodes }
  for j := 2 to n_layers do
    begin
      i := id + layer[j];
      while (i < layer[j + 1]) do
        begin
          { Calculate the state of node i }
```
The variables id, \( \ell \), \( i \) and nprocs are local variables, on the other hand n_layers (number of layers of the network) and layer \([j]\) (the first node number in layer \(j\)) are global variables. The m_get_numprocs routine gives the number of processes created by the m_fork routine. The m_get_myid routine gives the id number of the process, so this value will be different for different processors.

The Feedforward procedure calculates the new states of the nodes, the process id calculates the nodes id, id+nprocs, id+2nprocs \ldots\). So different processors will calculate the state of different nodes. For example, let us assume that we have a network of 50 nodes, 10 input, 20 hidden and 20 output nodes. Let us also assume that we want to run the learning procedure on 5 processors.

The state of the input nodes are clamped with the input vector so we require to find only the states of the hidden and output nodes. According to the Feedforward procedure the parent process (id= 0) finds the state of nodes 11, 16, 21, 26, 31, 36, 41, 46. The child process with id = 1 finds the state of nodes 12, 17, 22, 27, 32, 37, 42, 47 and so on. This example is depicted in Figure 4.10.
Figure 4.10 The operation of the parallel learning procedure of the On-line method.

The following is a segment of the Backpropagation procedure

```plaintext
procedure Backpropagation;
var
  i, ii, j : integer;
  nprocs: longint;
  ...
begin
  ii := m.get_myid;
  nprocs := m.get_numprocs;
  i := ii + layer[n.Layer];
  ...
  while (i <= tot.nodes) do
    begin
```
\{ calculate the error of node \( i \),

where \( i \) is an output node \}\n
\( i := i + \text{nprocs} \)
\end;
\text{\texttt{m.sync}};
\text{\texttt{for}} \ j := \text{n.layers} \text{\texttt{downto}} \ 2 \ \text{\texttt{do}}
\begin{align*}
\text{\texttt{i := ii + layer} \ [j] ;} \\
\text{\texttt{while} \ (i < \text{layer} \ [j + 1]) \ \text{\texttt{do}}}
\begin{align*}
\text{\texttt{\{ find the nodes connected to} \ i \}\}} \\
\text{\texttt{m.lock;}} \\
\text{\texttt{\{ Sum the weight errors of node} \ i \text{ due to the above layers}} \\
\text{\texttt{m.unlock;}} \\
\end{align*}
\end{align*}
\text{\texttt{end;}}
\text{\texttt{m.sync;}}
\begin{align*}
\text{\texttt{\{ calculate the errors of the hidden nodes using Eqn (2.33)\}}}
\end{align*}
\text{\texttt{end;}}

The Backpropagation procedure calculates the errors of the network. As can be seen from the procedure \( i, \ ii, \ j \) and \text{nprocs} are local variables. The global variables are \text{tot.nodes} (the total number of nodes in the network), \text{n.layers} and \text{layer} \ [j]. The output nodes are divided amongst the processors to calculate the error. The process \text{id} will calculate the error due to nodes \text{id} + \text{layer} \ [\text{n.layers}], \text{id} + \text{layer} \ [\text{n.layers}] + \text{nprocs}, \ldots. From the previous example, the parent process (\text{id} = 0) calculates the error due to nodes 31 and 36, the child process with \text{id} = 1 calculates the error due to nodes 32 and 37 and so on. After finishing with the output nodes, the hidden layer nodes are divided amongst the processors and their error is calculated. The linked list \text{a}[i] contains the nodes that are connected to node \( i \) as shown in Figure 4.9. To calculate the error for the hidden node \( i \), we require to know to which nodes \( i \) is connected to. From the network of Figure 4.10 it can
be seen that to find the error of node 11, according to Equation (2.22), the
errors of node 31, 32, \ldots, 50 are required with their connection weights to
node 11. The linked list of nodes 31, 32, \ldots, 50 each has the node 11 with
their corresponding connection weights. Therefore, a global 1-dimensional
array is used with one element for each node. The errors of the hidden nodes
are due to the errors caused by the nodes in the layers above it. So we take
all the nodes in the above layers and using their linked list we can sum the
errors. Let us assume that node \( i \) has a number of elements in its linked list,
therefore to find the sum of the weight errors of node \( j \) we have

\[
\text{sum} [j] := \text{sum} [j] + w[ij] \times \text{error} [i], \quad \forall j
\]  

(4.2)

where error \([i]\) is the error of node \( i \).

As this procedure is divided amongst the processors, it is very possible that two or more processes require to access the same variable at the
same time. Therefore the routines m_lock and m_unlock is used to prevent
more than one process access the same location. Again in this procedure
synchronization is important between the layers.

The following is a segment of the Change_weights procedure.

\begin{verbatim}
procedure Change_weights;
var
  ii, i: integer;
begin
  nprocr := m_get_numprocs;
  i := ii + layer [2];
  while (i <= tot_nodes) do
    begin
      \{ calculations of the new values of the connection weights }\n      i := i+nprocr;
    end;
end;
\end{verbatim}
The Change_weights procedure calculates the new values of the connections weights. The nodes are divided amongst the processors and each process is responsible for the connection weights that are connected to the allocated nodes. From the previous example, the id process calculates the connection weights that are connected to nodes id + layer [2], id + layer [2] + nprocs, id + layer [2] + 2x nprocs ... etc. So the nodes that are allocated to the parent process are 11, 16, 21, 26, 31, 37, 41, 46. Synchronization between the layers is not needed in this case.

4.6.2 THE BATCH BACKPROPAGATION PROCEDURE

The following is a segment of the Batch procedure,

```plaintext
procedure learning;
  var
    l, ii : integer;
    
  begin
    ii := m.get_myid + 1;
    for l := 1 to cycle_num do
      begin
        while (ii <= vect_size) do
          begin
            {clamp the input vector on the input nodes }
            Feedforward (ii);
            Backpropagation (ii);
            ii := ii + nprocs;
          end
        m.sync;
        Change_weights;
        m.sync;
      end
  end
```

It can be seen from the above procedure that it is similar to the On-line learning procedure except that the feedforward and backpropagation procedures are repeated for all the vector pairs before executing the Change_Weights
procedure. Also each process will have in its possession the entire network and one part of the vector set. Let us assume that we have a network with 30 nodes, 10 input, 10 hidden and 10 output nodes. Let us also assume that the network is required to be trained on 40 vector pairs using 5 processors. During one cycle the weights are changed once. The learning procedure divides the vector pairs amongst the 5 processors, each processor will have 8 vector pairs in its possession. The id process deals with vector pairs id +1, id +1 + nprocs, id +1 + 2 nprocs, . . . . So the parent process deals with vector pairs 1, 6, 11, 16, 21, 26, 31, 36. Figure 4.11 depicts this process.

This method requires more memory due to the fact that the errors from all the vector pairs should be stored. On the other hand, the On–line method requires the error due to one vector pair. However, this method requires less synchronization than the previous method. Synchronization is needed before changing the weights where all the processes are executed the Feedforward and Backpropagation procedures for all vector pairs.

This simulator was made flexible so that more change of weights per cycle was possible, so if we had 40 vector pairs, 5 processors and 4 weight changes per cycle, then each processor will have two vector pairs to calculate the errors, the weight is changed and repeated again with different vector pairs and so on.

**Figure 4.11** The Operation of the Batch parallel simulator.
4.7 Description of the Experiments

A number of experiments were carried out to test the performance of the parallel simulators (the On-line and Batch simulators). Each experiment was repeated 10 times, each time the simulator was run with a different number of processors. The speedup due to \( p \) processors was calculated for each case, where \( p \) is the number of processors used to run the simulator. Let us assume that the time required for the simulator to execute \( n \) iterations using \( p \) processors is \( \text{time}_p \). The speedup due to \( p \) processors is calculated as

\[
SP = \frac{\text{time}_1}{\text{time}_p}.
\]  

(4.3)

The effect of increasing the number of nodes in a network on the speedup of the parallel simulators was tested. This was done by using 5 different networks where each network consisted of 2 layers and each layer had an equal number of input, hidden and output nodes. For the 5 networks, the number of nodes in each layer was varied to have 5, 10, 20, 30, and 40 nodes. The training set consisted of 50 patterns for all the cases. Another set of experiments were carried out to test the effect of changing the size of the training set. By fixing the size of the network, the training set was increased for each experiment as 50, 80, 100, 200, 300, and 400 vector pairs. The size of the network was chosen to have 10 nodes in each layer. The performance of both On-line and batch parallel simulators were compared with each other.

4.7.1 THE RESULTS OF THE ON-LINE SIMULATOR

In Tables (4.1),(4.2),(4.3),(4.4) and (4.5) are the execution time needed using different number of processors as well as different number of nodes. Each network was allowed to run for 10 iterations.
<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time ×10^0/s</th>
<th>SpeedUp SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14.360</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9.230</td>
<td>1.56</td>
</tr>
<tr>
<td>3</td>
<td>7.750</td>
<td>1.85</td>
</tr>
<tr>
<td>4</td>
<td>7.280</td>
<td>1.97</td>
</tr>
<tr>
<td>5</td>
<td>6.050</td>
<td>2.37</td>
</tr>
<tr>
<td>6</td>
<td>6.220</td>
<td>2.31</td>
</tr>
<tr>
<td>7</td>
<td>6.760</td>
<td>2.12</td>
</tr>
<tr>
<td>8</td>
<td>7.070</td>
<td>2.03</td>
</tr>
<tr>
<td>9</td>
<td>7.400</td>
<td>1.94</td>
</tr>
<tr>
<td>10</td>
<td>7.120</td>
<td>2.02</td>
</tr>
</tbody>
</table>

**Table (4.1):** The execution time and speedup of a network of $5 \times 5 \times 5$ nodes using On-line simulator

<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time ×10^1/s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.450</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.430</td>
<td>1.83</td>
</tr>
<tr>
<td>3</td>
<td>1.874</td>
<td>2.38</td>
</tr>
<tr>
<td>4</td>
<td>1.510</td>
<td>2.95</td>
</tr>
<tr>
<td>5</td>
<td>1.292</td>
<td>3.44</td>
</tr>
<tr>
<td>6</td>
<td>1.296</td>
<td>3.43</td>
</tr>
<tr>
<td>7</td>
<td>1.194</td>
<td>3.73</td>
</tr>
<tr>
<td>8</td>
<td>1.216</td>
<td>3.66</td>
</tr>
<tr>
<td>9</td>
<td>1.237</td>
<td>3.60</td>
</tr>
<tr>
<td>10</td>
<td>1.050</td>
<td>4.24</td>
</tr>
</tbody>
</table>

**Table (4.2):** The execution time and speedup of a network of $10 \times 10 \times 10$ nodes using On-line simulator
<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time × 10¹/s</th>
<th>SpeedUp SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.663</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8.124</td>
<td>1.93</td>
</tr>
<tr>
<td>3</td>
<td>5.831</td>
<td>2.69</td>
</tr>
<tr>
<td>4</td>
<td>4.336</td>
<td>3.61</td>
</tr>
<tr>
<td>5</td>
<td>3.623</td>
<td>4.32</td>
</tr>
<tr>
<td>6</td>
<td>3.405</td>
<td>4.60</td>
</tr>
<tr>
<td>7</td>
<td>2.962</td>
<td>5.29</td>
</tr>
<tr>
<td>8</td>
<td>2.735</td>
<td>5.73</td>
</tr>
<tr>
<td>9</td>
<td>2.724</td>
<td>5.75</td>
</tr>
<tr>
<td>10</td>
<td>2.292</td>
<td>6.83</td>
</tr>
</tbody>
</table>

Table (4.3): The execution time and speedup of a network of 20 × 20 × 20 nodes using On-line simulator

<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time × 10¹/s</th>
<th>SpeedUp SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.766</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>17.503</td>
<td>1.93</td>
</tr>
<tr>
<td>3</td>
<td>11.895</td>
<td>2.84</td>
</tr>
<tr>
<td>4</td>
<td>9.347</td>
<td>3.61</td>
</tr>
<tr>
<td>5</td>
<td>7.626</td>
<td>4.43</td>
</tr>
<tr>
<td>6</td>
<td>6.269</td>
<td>5.39</td>
</tr>
<tr>
<td>7</td>
<td>5.904</td>
<td>5.72</td>
</tr>
<tr>
<td>8</td>
<td>5.227</td>
<td>6.46</td>
</tr>
<tr>
<td>9</td>
<td>4.883</td>
<td>6.92</td>
</tr>
<tr>
<td>10</td>
<td>4.195</td>
<td>8.05</td>
</tr>
</tbody>
</table>

Table (4.4): The execution time and speedup of a network of 30 × 30 × 30 nodes using On-line simulator
<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time ×10^1/s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>59.116</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>31.228</td>
<td>1.89</td>
</tr>
<tr>
<td>3</td>
<td>21.228</td>
<td>2.79</td>
</tr>
<tr>
<td>4</td>
<td>15.658</td>
<td>3.78</td>
</tr>
<tr>
<td>5</td>
<td>12.810</td>
<td>4.62</td>
</tr>
<tr>
<td>6</td>
<td>11.136</td>
<td>5.31</td>
</tr>
<tr>
<td>7</td>
<td>9.616</td>
<td>6.15</td>
</tr>
<tr>
<td>8</td>
<td>8.180</td>
<td>7.23</td>
</tr>
<tr>
<td>9</td>
<td>7.729</td>
<td>7.65</td>
</tr>
<tr>
<td>10</td>
<td>6.791</td>
<td>8.70</td>
</tr>
</tbody>
</table>

Table (4.5): The execution time and speedup of a network of 40 × 40 × 40 nodes using On-line simulator

Figure 4.12 shows diagrammatically the execution time required using a different number of processors.

![Figure 4.12](image)

**Figure 4.12** The effect of the number of nodes on the speedup of the On-line simulator
In Tables (4.6), (4.7), (4.8) and (4.9) are the execution time needed using different numbers of processors as well as different sizes of the training set. Each network was allowed to run for 10 iterations.

<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time $\times 10^1 / s$</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.936</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4.804</td>
<td>1.86</td>
</tr>
<tr>
<td>3</td>
<td>3.677</td>
<td>2.43</td>
</tr>
<tr>
<td>4</td>
<td>2.964</td>
<td>3.02</td>
</tr>
<tr>
<td>5</td>
<td>2.484</td>
<td>3.6</td>
</tr>
<tr>
<td>6</td>
<td>2.505</td>
<td>3.57</td>
</tr>
<tr>
<td>7</td>
<td>2.278</td>
<td>3.92</td>
</tr>
<tr>
<td>8</td>
<td>2.310</td>
<td>3.87</td>
</tr>
<tr>
<td>9</td>
<td>2.320</td>
<td>3.85</td>
</tr>
<tr>
<td>10</td>
<td>1.940</td>
<td>4.61</td>
</tr>
</tbody>
</table>

**Table (4.6):** The execution time and speedup of a network trained on 100 vector pairs using the On-line simulator
<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time $\times 10^1$/s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.883</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9.638</td>
<td>1.86</td>
</tr>
<tr>
<td>3</td>
<td>7.483</td>
<td>2.39</td>
</tr>
<tr>
<td>4</td>
<td>5.991</td>
<td>2.99</td>
</tr>
<tr>
<td>5</td>
<td>4.870</td>
<td>3.67</td>
</tr>
<tr>
<td>6</td>
<td>4.899</td>
<td>3.65</td>
</tr>
<tr>
<td>7</td>
<td>4.449</td>
<td>4.02</td>
</tr>
<tr>
<td>8</td>
<td>4.495</td>
<td>3.99</td>
</tr>
<tr>
<td>9</td>
<td>4.517</td>
<td>3.96</td>
</tr>
<tr>
<td>10</td>
<td>3.658</td>
<td>4.89</td>
</tr>
</tbody>
</table>

**Table (4.7):** The execution time and speedup of a network trained on 200 vector pairs using the On-line simulator

<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time $\times 10^1$/s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26.969</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14.735</td>
<td>1.83</td>
</tr>
<tr>
<td>3</td>
<td>11.365</td>
<td>2.37</td>
</tr>
<tr>
<td>4</td>
<td>8.920</td>
<td>3.02</td>
</tr>
<tr>
<td>5</td>
<td>7.386</td>
<td>3.65</td>
</tr>
<tr>
<td>6</td>
<td>7.309</td>
<td>3.69</td>
</tr>
<tr>
<td>7</td>
<td>6.627</td>
<td>4.07</td>
</tr>
<tr>
<td>8</td>
<td>6.651</td>
<td>4.06</td>
</tr>
<tr>
<td>9</td>
<td>6.633</td>
<td>4.07</td>
</tr>
<tr>
<td>10</td>
<td>5.445</td>
<td>4.95</td>
</tr>
</tbody>
</table>

**Table (4.8):** The execution time and speedup of a network trained on 300 vector pairs using the On-line simulator
<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time x10^1 /s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36.099</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>19.452</td>
<td>1.86</td>
</tr>
<tr>
<td>3</td>
<td>15.017</td>
<td>2.40</td>
</tr>
<tr>
<td>4</td>
<td>11.759</td>
<td>3.07</td>
</tr>
<tr>
<td>5</td>
<td>9.571</td>
<td>3.77</td>
</tr>
<tr>
<td>6</td>
<td>9.932</td>
<td>3.64</td>
</tr>
<tr>
<td>7</td>
<td>8.983</td>
<td>4.02</td>
</tr>
<tr>
<td>8</td>
<td>8.818</td>
<td>4.09</td>
</tr>
<tr>
<td>9</td>
<td>8.869</td>
<td>4.07</td>
</tr>
<tr>
<td>10</td>
<td>7.187</td>
<td>5.02</td>
</tr>
</tbody>
</table>

**Table (4.9):** The execution time and speedup of a network trained on 400 vector pairs using the On-line simulator.

Figure 4.13 shows diagrammatically the execution time required using different number of processors.

**Figure 4.13** The effect different training set sizes on the speedup of the On-line simulator.
It can be seen from the results that the speedup of the On-line simulator depends on a number of factors. These are:-

1. The number of nodes in the network as well as the number of connections. From the experimental results it can be seen that the speedup increases with the size of the network. This is due to the m_sync instructions within the learning procedure. The processors wait for the slowest processor between the layers and also between the Feedforward and Backpropagation procedures. When there are a large number of nodes in each layer, the waiting time due to m_sync will have less effect on the speedup values. The ideal speedup using 10 processors is 10. In general, this value cannot be achieved due to the fact that time is spent in creating processors. From the results it was shown that the speedup achieved using 10 processors is 8.70 when a network of 40 x 40 node was used.

2. The number of nodes per layer. The performance of the parallel simulator depends on the number of the nodes per layer. If the number of nodes within a layer is divisible by the number of processors then the speedup should approach the ideal value. If the number of nodes within a layer is not divisible by the number of the processors then some of the processors will have a smaller number of nodes to calculate and hence the speedup cannot approach the ideal value.

3. The vector size. The size of the training set has a slight effect on the speedup. As the training set increases the speedup increases too. This is due to the fact that more calculations are involved within a cycle.

4.7.2 THE RESULTS OF THE BATCH SIMULATOR

In Tables (4.10),(4.11),(4.12),(4.13) and (4.14) are the execution time needed using different numbers of processors as well as different numbers of nodes. Each network was allowed to run for 10 iterations.
<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time $\times 10^0 / s$</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.190</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5.000</td>
<td>1.84</td>
</tr>
<tr>
<td>3</td>
<td>3.820</td>
<td>2.41</td>
</tr>
<tr>
<td>4</td>
<td>3.200</td>
<td>2.87</td>
</tr>
<tr>
<td>5</td>
<td>2.670</td>
<td>3.44</td>
</tr>
<tr>
<td>6</td>
<td>2.660</td>
<td>3.46</td>
</tr>
<tr>
<td>7</td>
<td>2.610</td>
<td>3.52</td>
</tr>
<tr>
<td>8</td>
<td>2.640</td>
<td>3.48</td>
</tr>
<tr>
<td>9</td>
<td>2.610</td>
<td>3.52</td>
</tr>
<tr>
<td>10</td>
<td>2.370</td>
<td>3.88</td>
</tr>
</tbody>
</table>

Table (4.10): The execution time and speedup of a network of $5 \times 5 \times 5$ nodes using Batch simulator.

<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time $\times 10^0 / s$</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28.080</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14.470</td>
<td>1.94</td>
</tr>
<tr>
<td>3</td>
<td>10.420</td>
<td>2.70</td>
</tr>
<tr>
<td>4</td>
<td>7.950</td>
<td>3.53</td>
</tr>
<tr>
<td>5</td>
<td>6.540</td>
<td>4.29</td>
</tr>
<tr>
<td>6</td>
<td>6.260</td>
<td>4.49</td>
</tr>
<tr>
<td>7</td>
<td>5.620</td>
<td>5.00</td>
</tr>
<tr>
<td>8</td>
<td>5.340</td>
<td>5.26</td>
</tr>
<tr>
<td>9</td>
<td>5.050</td>
<td>5.56</td>
</tr>
<tr>
<td>10</td>
<td>4.320</td>
<td>6.50</td>
</tr>
</tbody>
</table>

Table (4.11): The execution time and speedup of a network of $10 \times 10 \times 10$ nodes using Batch simulator.
### Table (4.12): The execution time and speedup of a network of $20 \times 20 \times 20$ nodes using Batch simulator.

<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time ×10^1/s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.618</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4.840</td>
<td>1.99</td>
</tr>
<tr>
<td>3</td>
<td>3.367</td>
<td>2.86</td>
</tr>
<tr>
<td>4</td>
<td>2.537</td>
<td>3.79</td>
</tr>
<tr>
<td>5</td>
<td>2.014</td>
<td>4.78</td>
</tr>
<tr>
<td>6</td>
<td>1.827</td>
<td>5.26</td>
</tr>
<tr>
<td>7</td>
<td>1.617</td>
<td>5.95</td>
</tr>
<tr>
<td>8</td>
<td>1.439</td>
<td>6.68</td>
</tr>
<tr>
<td>9</td>
<td>1.331</td>
<td>7.23</td>
</tr>
<tr>
<td>10</td>
<td>1.138</td>
<td>8.45</td>
</tr>
</tbody>
</table>

### Table (4.13): The execution time and speedup of a network of $30 \times 30 \times 30$ nodes using Batch simulator.

<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time ×10^1/s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.722</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10.390</td>
<td>1.99</td>
</tr>
<tr>
<td>3</td>
<td>7.134</td>
<td>2.90</td>
</tr>
<tr>
<td>4</td>
<td>5.446</td>
<td>3.81</td>
</tr>
<tr>
<td>5</td>
<td>4.322</td>
<td>4.80</td>
</tr>
<tr>
<td>6</td>
<td>3.789</td>
<td>5.47</td>
</tr>
<tr>
<td>7</td>
<td>3.382</td>
<td>6.13</td>
</tr>
<tr>
<td>8</td>
<td>3.030</td>
<td>6.84</td>
</tr>
<tr>
<td>9</td>
<td>2.655</td>
<td>7.81</td>
</tr>
<tr>
<td>10</td>
<td>2.219</td>
<td>9.34</td>
</tr>
<tr>
<td>Number Of Processors</td>
<td>Execution Time $\times 10^1$/s</td>
<td>Speedup SP</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>1</td>
<td>35.476</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>18.047</td>
<td>1.97</td>
</tr>
<tr>
<td>3</td>
<td>12.243</td>
<td>2.90</td>
</tr>
<tr>
<td>4</td>
<td>9.366</td>
<td>3.79</td>
</tr>
<tr>
<td>5</td>
<td>7.351</td>
<td>4.83</td>
</tr>
<tr>
<td>6</td>
<td>6.566</td>
<td>5.40</td>
</tr>
<tr>
<td>7</td>
<td>5.758</td>
<td>6.16</td>
</tr>
<tr>
<td>8</td>
<td>5.013</td>
<td>7.08</td>
</tr>
<tr>
<td>9</td>
<td>4.458</td>
<td>7.96</td>
</tr>
<tr>
<td>10</td>
<td>3.723</td>
<td>9.53</td>
</tr>
</tbody>
</table>

**Table (4.14):** The execution time and speedup of a network of $40 \times 40 \times 40$ nodes using Batch simulator.

Figure 4.14 shows diagrammatically the execution time required using a different number of processors.

**Figure 4.14** The effect of the number of nodes on the speedup of the Batch simulator.
In Tables (4.15), (4.16), (4.17), (4.18), and (4.19) are the execution times needed using different numbers of processors as well as different sizes of the training set. Each network was allowed to run for 10 iterations.

<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time $\times 10^0/s$</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44.720</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>22.710</td>
<td>1.97</td>
</tr>
<tr>
<td>3</td>
<td>16.160</td>
<td>2.77</td>
</tr>
<tr>
<td>4</td>
<td>11.930</td>
<td>3.75</td>
</tr>
<tr>
<td>5</td>
<td>9.870</td>
<td>4.53</td>
</tr>
<tr>
<td>6</td>
<td>9.230</td>
<td>4.85</td>
</tr>
<tr>
<td>7</td>
<td>7.880</td>
<td>5.68</td>
</tr>
<tr>
<td>8</td>
<td>7.280</td>
<td>6.14</td>
</tr>
<tr>
<td>9</td>
<td>7.040</td>
<td>6.35</td>
</tr>
<tr>
<td>10</td>
<td>6.070</td>
<td>7.37</td>
</tr>
</tbody>
</table>

Table (4.15): The execution time and speedup of a network trained on 80 vector pairs using the Batch simulator.
<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time ×10⁰/s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>55.880</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>28.450</td>
<td>1.96</td>
</tr>
<tr>
<td>3</td>
<td>19.790</td>
<td>2.82</td>
</tr>
<tr>
<td>4</td>
<td>14.760</td>
<td>3.79</td>
</tr>
<tr>
<td>5</td>
<td>12.150</td>
<td>4.60</td>
</tr>
<tr>
<td>6</td>
<td>11.030</td>
<td>5.07</td>
</tr>
<tr>
<td>7</td>
<td>9.570</td>
<td>5.84</td>
</tr>
<tr>
<td>8</td>
<td>8.950</td>
<td>6.24</td>
</tr>
<tr>
<td>9</td>
<td>8.650</td>
<td>6.46</td>
</tr>
<tr>
<td>10</td>
<td>7.250</td>
<td>7.71</td>
</tr>
</tbody>
</table>

**Table (4.16):** The execution time and speedup of a network trained on 100 vector pairs using the Batch simulator.

<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time ×10¹/s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11.097</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5.622</td>
<td>1.97</td>
</tr>
<tr>
<td>3</td>
<td>3.899</td>
<td>2.85</td>
</tr>
<tr>
<td>4</td>
<td>2.920</td>
<td>3.80</td>
</tr>
<tr>
<td>5</td>
<td>2.383</td>
<td>4.66</td>
</tr>
<tr>
<td>6</td>
<td>2.174</td>
<td>5.10</td>
</tr>
<tr>
<td>7</td>
<td>1.842</td>
<td>6.02</td>
</tr>
<tr>
<td>8</td>
<td>1.761</td>
<td>6.30</td>
</tr>
<tr>
<td>9</td>
<td>1.667</td>
<td>6.66</td>
</tr>
<tr>
<td>10</td>
<td>1.410</td>
<td>7.87</td>
</tr>
</tbody>
</table>

**Table (4.17):** The execution time and speedup of a network trained on 200 vector pairs using the Batch simulator.
<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time ×10^1/s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16.668</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8.496</td>
<td>1.96</td>
</tr>
<tr>
<td>3</td>
<td>5.778</td>
<td>2.89</td>
</tr>
<tr>
<td>4</td>
<td>4.313</td>
<td>3.85</td>
</tr>
<tr>
<td>5</td>
<td>3.518</td>
<td>4.74</td>
</tr>
<tr>
<td>6</td>
<td>3.182</td>
<td>5.24</td>
</tr>
<tr>
<td>7</td>
<td>2.665</td>
<td>6.25</td>
</tr>
<tr>
<td>8</td>
<td>2.517</td>
<td>6.62</td>
</tr>
<tr>
<td>9</td>
<td>2.378</td>
<td>7.01</td>
</tr>
<tr>
<td>10</td>
<td>2.011</td>
<td>8.29</td>
</tr>
</tbody>
</table>

Table (4.18): The execution time and speedup of a network trained on 300 vector pairs using the Batch simulator.

<table>
<thead>
<tr>
<th>Number Of Processors</th>
<th>Execution Time ×10^1/s</th>
<th>Speedup SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22.373</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11.360</td>
<td>1.97</td>
</tr>
<tr>
<td>3</td>
<td>7.691</td>
<td>2.91</td>
</tr>
<tr>
<td>4</td>
<td>5.721</td>
<td>3.91</td>
</tr>
<tr>
<td>5</td>
<td>4.656</td>
<td>4.81</td>
</tr>
<tr>
<td>6</td>
<td>4.195</td>
<td>5.33</td>
</tr>
<tr>
<td>7</td>
<td>3.512</td>
<td>6.37</td>
</tr>
<tr>
<td>8</td>
<td>3.274</td>
<td>6.83</td>
</tr>
<tr>
<td>9</td>
<td>3.099</td>
<td>7.22</td>
</tr>
<tr>
<td>10</td>
<td>2.597</td>
<td>8.62</td>
</tr>
</tbody>
</table>

Table (4.19): The execution time and speedup of a network trained on 400 vector pairs using the Batch simulator.
Figure 4.15 shows diagrammatically the execution time required using a different number of processors.

![Graph showing speedup versus number of processors](image)

**Figure 4.15** The effect of the size of the training set on the speedup of the Batch simulator

It can be seen from the experimental results that the speedup of the Batch simulator depends on a number of factors, these are:-

1. The size of the network and hence the number of connections. The performance of the Batch parallel simulator improves by increasing the size of the network. A speedup of 9.53 was achieved when a network of $40 \times 40 \times 40$ was used. This value is near to the ideal value.

2. The vector size. The speedup approaches the ideal value as the size of the training set is increased. This is due to the fact that the processors have a large amount of work to perform within a cycle.
4.8 Conclusion

It was shown that the two parallel simulators perform well when the size of the network increases. The Batch parallel simulator performed slightly better than the On-line parallel simulator when the same size of network was used. This is due to the fact that there are more m_sync instructions in the On-line simulator, hence more processors will be waiting for each other between different procedures or even functions. To get the best performance out of the On-line parallel simulator would depend on the number of nodes per layer. On the other hand, to get the best performance of the Batch parallel simulator will depend on the size of the training set.
CHAPTER V

Accelerating the Backpropagation Algorithm
The gradient used in the BP learning algorithms can take two forms, stochastic and deterministic. In this chapter, the two methods are tested and some improvements of these two algorithms are introduced. The On-line BP algorithm uses stochastic gradient where the gradient is based on a single training vector. Given small learning rates, the individual gradient vectors will be summed to approximate the true gradient descent direction. In the first section of this chapter, a new algorithm is introduced to improve the convergence rate of such a method and some examples are given. Later in this chapter the same algorithm is applied to the Batch BP algorithm. This algorithm uses the deterministic gradient where the gradient now is based on the entire training set. The Steepest Descent and the Conjugate Gradient algorithms were implemented and compared with the BP as well as the new algorithm.
5.1 A Gradient Range-Based Heuristic Method For Increasing The Rate Of Convergence (GRBH)

There have been a number of techniques to accelerate the convergence rate of the BP algorithm. Most of these methods involve extra calculations to determine the value of the learning rate \( \alpha \). Some of these methods were described in Chapter 3. From Section (3.5) it can be seen that the rate of convergence can be improved if the learning rate is allowed to be changed with time and also for different directions depending on the gradient values. By taking these two points into consideration the gradient values were divided into a number of groups according to their values. The modulus values of the gradient usually has the value in the range \( 10^{-1} - 10^{-8} \). Therefore, the gradient values of each group was taken to have equal step sizes. For each group a learning rate is assigned. Large learning rates were assigned for groups that have small modulus values of the gradient, small learning rates were assigned to those with large modulus values of the gradient and for the groups with an intermediate modulus values of the gradient are given an intermediate learning values. The values of \( \alpha \) are dependent on the problem. This is shown graphically in Figure 5.1.
Figure 5.1 The values of $\alpha$ corresponding to the gradient values.

The values of $\alpha$ for each group are chosen at the beginning of the training procedure and they are kept constant. During the weight update procedure the learning rate for each connection weight is found by determining which group the gradient belongs to. The gradient of each weight changes with every iteration, hence $\alpha$ changes with every iteration. Also different weights have different gradients and therefore each weight will have a different value of $\alpha$. The update procedure can be described as follows:

1. Assign $G$ values of $\alpha \to \alpha_1, \alpha_2 \ldots \alpha_G$ where $G$ is the number of groups.
2. Assign a range for each group $R_1, R_2, \ldots, R_G$.
3. Calculate the errors by executing the feedforward and backpropagation procedures.
4. Find the range of the modulus of the gradient for every connection $i.e. \left| \frac{\partial E}{\partial w_{ij}} \right|$ and assume it is $R_k$. 

125
5. Update the connection weight as

\[ w_{ij}(n + 1) = w_{ij}(n) + \alpha_k \frac{\partial E}{\partial w_{ij}} \]

6. Repeat step 4 and 5 for all connection weights.

7. Go back to step 3 until convergence is achieved.

By using this method we can also use the momentum term as

\[ w_{ij}(n + 1) = w_{ij}(n) + \alpha_k \frac{\partial E}{\partial w_{ij}} + \beta \Delta w_{ij}(n), \]

where

\[ \Delta w_{ij}(n) = w_{ij}(n) - w_{ij}(n - 1). \]

This method does not involve additional computation except for \( G \) comparisons to find the range of the gradient.

### 5.2 Experimental Description

In order to evaluate the performance of the Heuristic algorithm, a number of experiments were performed using the On-line GRBH algorithm and they are compared with the usual On-line BP algorithm. The number of groups were varied, and it was found that the best results occurred for \( G = 3 \) and \( G = 6 \). The connection weights of the network were initialized to random values between \([-0.1, +0.1]\). The error measure used was the sum of the squares of the errors.

\[ E = \frac{1}{2} \sum_{p} \sum_{i} (t_{pi} - a_{pi})^2. \]

(5.1)
5.2.1 THE EXCLUSIVE-OR PROBLEM

The network consisted of 2 input, 1 hidden and 1 output nodes. All the nodes in a layer are connected to the nodes below as shown in Figure 5.2. The training set consists of 4 vector pairs.

![Figure 5.2 XOR network.](image)

For the On-line BP algorithm the values chosen were $\alpha = 0.5$ and $\beta = 0.9$. For the GRBH algorithm two sets of experiments were carried out, one with $G = 3$ and the other with $G = 6$. Table (5.1) shows the chosen values of $\alpha$ corresponding to the ranges of the modulus values of the gradient. The value of $\beta = 0.9$ for both cases.

<table>
<thead>
<tr>
<th>Number of groups $G = 3$</th>
<th>Number of groups $G = 6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient $</td>
<td>\partial E/\partial w</td>
</tr>
<tr>
<td>$R_1 &lt; 10^{-3}$</td>
<td>2.5</td>
</tr>
<tr>
<td>$R_2 &lt; R_3$</td>
<td>0.5</td>
</tr>
<tr>
<td>$R_2 &lt; R_3$</td>
<td>2.0</td>
</tr>
<tr>
<td>$R_4 &lt; R_5 &lt; 10^{-2}$</td>
<td>1.0</td>
</tr>
<tr>
<td>$R_5 &lt; R_6$</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table (5.1) The chosen values of $\alpha$ for the XOR problem using the On-line GRBH algorithm.

Figure 5.3 shows diagrammatically the chosen values of $\alpha$ for different ranges.
Figure 5.3 XOR problem: The chosen values of $\alpha$ corresponding to two different groupings.

The learning process was terminated when the sum of the squares of the error reached $2.5 \times 10^{-5}$. Table (5.2) shows the results of these experiments, which were the average of 10 trials.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average number of iterations</th>
<th>% Improvement over BP</th>
<th>Number of cases not converged</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-line BP</td>
<td>67119</td>
<td>-</td>
<td>None</td>
</tr>
<tr>
<td>GRBH with $G = 6$</td>
<td>10279</td>
<td>84.7%</td>
<td>None</td>
</tr>
<tr>
<td>GRBH with $G = 3$</td>
<td>14694</td>
<td>78.1%</td>
<td>None</td>
</tr>
</tbody>
</table>

Table (5.2): The simulation results for the XOR problem using the On-line BP algorithm.
From the results of Table (5.2) it can be seen that the GRBH algorithm improves the convergence considerably. From Figure 5.3, the chosen values of $\alpha$ for $G = 6$ are slightly higher than for $G = 3$ hence the convergence of the GRBH algorithm with $G = 6$ is better than with $G = 3$. It was found that the best combination of values of $\alpha$ with $G = 6$ is to choose these values such that two adjacent groups have values that are a factor of 2, and for $G = 3$ a factor of 5. For this problem it was possible to use larger values of $\alpha$ when $G = 3$ (Sanossian and Evans [1991]).

All the trials converged to the required solution, which is due to the fact that only very small random weights $[-0.1, +0.1]$ were used. This does not guarantee convergence to the required solution but reduces the possibility of getting stuck in a local minima. Figure 5.4 displays the progression of the learning algorithms. The x-axis of the graph represents the number of iterations to the solution and the y-axis represents the sum of the squares of the errors.

![Figure 5.4 The learning progression of the XOR problem.](image-url)
5.2.2 THE SHIFT REGISTER PROBLEM

It is required to train a network to perform an 8 bit shift register where the output of an 8 bit input is shifted one bit to the left. The network consists of 8 input, 4 hidden, and 8 output nodes. Each layer is connected to the nodes of its adjacent layer as shown in Figure 5.5. The training set consists of 25 vector pairs.

![Shift Register Network](image)

**Figure 5.5** The shift register network.

For the On-line BP algorithm the values $\alpha = 0.2$ and $\beta = 0.9$ were chosen. For the GRBH algorithm two sets of experiments were carried out, one with $G = 3$ and the other with $G = 6$. Table (5.3) shows the chosen values of $\alpha$ corresponding to the ranges of the modulus of the gradient values. The value of $\beta = 0.9$ for both cases.
Table (5.3) The chosen values of $\alpha$ for the shift register problem using the On-line GRBH algorithm.

Figure 5.6 shows diagrammatically the chosen values of $\alpha$ for the different ranges.

![Diagram showing chosen values of $\alpha$ for different ranges.](image)

Figure 5.6 The shift register problem with the chosen values of $\alpha$ for two different groupings $G = 3$ and $G = 6$. 

131
The learning process was terminated when the sum of the squares of the error reached $0.5 \times 10^{-3}$. Table (5.4) shows the results of these experiments, which are the average of 10 trials.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average number of iterations</th>
<th>% Improvement over BP</th>
<th>Number of cases not converged</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-line BP</td>
<td>90455</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>GRBH with $G = 6$</td>
<td>14841</td>
<td>83.6%</td>
<td>2</td>
</tr>
<tr>
<td>GRBH with $G = 3$</td>
<td>16793</td>
<td>81.4%</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table (5.4):** The simulation results for the shift register problem using the On-line BP algorithm.

From Table (5.4) it can be seen that the convergence of the GRBH algorithm is faster than the BP algorithm. A number of tests were carried out with a different set of values of $\alpha$ but the best combination was with $G = 6$ and $G = 3$ and these are shown in Table (5.4). For $G = 3$ only one trial got stuck at a local minima, on the other hand two trials got stuck when the BP algorithm as well as the GRBH algorithm with $G = 6$ was used.

Figure 5.7 displays the progression of the learning algorithms. The x-axis of the graph represents the number of iterations to the solution and the y-axis represents the sum of the squares of the errors.
5.2.3 SORTING PROBLEM

It is required to sort three integer numbers in ascending order. Therefore three input nodes are used, each node corresponds to one of the numbers. 9 output nodes are used, where each number corresponds to three nodes. Only one of the three nodes can be active, which then indicates the position of the number in the sorted list as shown in Figure 5.8.

Here a maximum of 2 digit numbers are used, where each number is divided by 100 to bound the input vectors to the range 0-0.99 and the output vectors consist of all 0's and 1's. Two hidden layers are used, each layer consisting of 9 nodes and only adjacent layers are connected with each other.

For the On-line BP algorithm the values of $\alpha = 0.2$ and $\beta = 0.9$ was used while for the GRBH algorithm two sets of experiments were carried out, one with $G = 3$ and the other with $G = 6$. Table (5.5) shows the chosen values of $\alpha$ corresponding to the ranges of the modulus of the gradient values. The value of $\beta = 0.9$ for both cases.
Figure 5.8 The sort network.

Table (5.5) The chosen values of $\alpha$ for the sorting problem using the On-line GRBH algorithm.

Figure 5.9 shows diagrammatically the chosen values of $\alpha$ for different ranges.
Figure 5.9 Sorting problem: The chosen values of $\alpha$ corresponding to two different groups.

The learning process was terminated when the sum of the squares of the error reached $0.5 \times 10^{-3}$. Table (5.6) shows the results of these experiments, which are the average of 10 trials.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average number of iterations</th>
<th>% Improvement over BP</th>
<th>Number of cases not converged</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-line BP</td>
<td>18250</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>GRBH with $G = 6$</td>
<td>4461</td>
<td>75.6%</td>
<td>1</td>
</tr>
<tr>
<td>GRBH with $G = 3$</td>
<td>3548</td>
<td>80.6%</td>
<td>1</td>
</tr>
</tbody>
</table>

Table (5.6): The simulation results for the sorting problem using the On-line BP algorithm.

From the results, it can be seen that the convergence is improved by using the GRBH algorithm. Only one trial got stuck at local minima for the
three chosen algorithms. Figure 5.10 displays the progression of the learning algorithms. The x-axis of the graph represents the number of iterations to the solution and the y-axis represents the sum of the squares of the errors.

Figure 5.10 The learning progression of the sort problem.

5.3 The Steepest Descent Algorithm

As previously mentioned in Chapter 3, the Steepest Descent method uses a line search to estimate the value of $\alpha$. The quadratic interpolation method is chosen to calculate $\alpha$ where the error function is approximated by a quadratic polynomial. This method requires the values of the function at three points. The two extreme points are chosen such that they bracket the minimum. Initially one of the points is taken to be $\alpha_1 = 0$. The second extreme point is found by using a function comparison method where an initial value is supplied say, $\alpha_2$ and the error function is calculated. The process is repeated using $\alpha_2 = 2\alpha_2$ until an increase in the error function value is detected. As $E(\alpha_1) < E(\alpha_2)$, then $[\alpha_1, \alpha_2]$ must bracket a local
minimum. It was found that for function comparison $\alpha_2 = 4\alpha_2$ can be used when training ANNs using the SD method. The third point is taken to be half the value of $\alpha_2$ that is $\alpha_3 = \alpha_2/2$. The process of function comparison is shown in Figure 5.11.

![Diagram of function comparison](image)

**Figure 5.11** The process of function comparison.

Once the minimum is bracketed the quadratic interpolation method is used to calculate $\alpha$. For each iteration in the line search, the error is calculated by averaging the error due to all the vector pairs. When the exact solution is required this method is expensive but fortunately the exact solution is not required. The Wolfe test is used to terminate the search method.

The outline of the Steepest Descent algorithm for the feedforward net-

work can be described as follows:

```plaintext
procedure Quadratic-linesearch
  var
  : 
```

137
begin
initialize $\alpha_1, \alpha_2, \text{test};$
repeat \{ Bracketing the minimum \}
\begin{align*}
\text{change-weights (point2, point1, $\alpha_2$);}
\text{feedforward (point2);} \\
\text{backpropagation (point2);} \\
\text{gradient (point2, $g_2$);} \\
\alpha_2 := 4\alpha_2;
\end{align*}
until ($e_{r2} \geq e_{r1}$) or ($g_2 \geq 0$);
\begin{align*}
\alpha_3 := \alpha_2/2;
\end{align*}
repeat \{ calculate $\alpha$ using quadratic interpolation \}
\begin{align*}
\alpha_1 := e_{r0} + 0.0001g_0\alpha_2; \\
\alpha_2 := 0.9g_0;
\end{align*}
if ($e_{r3} \leq \alpha_1$) and abs (($g_3$) $\leq -\alpha_2$) then 
\begin{align*}
test := \text{true}
\end{align*}
else 
begin
\{ calculate $\alpha_4$ using Equation (3.12) where $\alpha_4 = \alpha$ \}
\begin{align*}
\text{if ($\alpha_4 \geq (\alpha_2 - 0.1$) then \{ Check if $\alpha_4$ is outside the bracket \}} \\
\alpha_4 := \alpha_2 - 0.1(\alpha_2 - \alpha_1); \\
\text{if ($\alpha_4 \leq (\alpha_1 + 0.1$) then} \\
\alpha_4 := \alpha_1 + 0.1(\alpha_2 - \alpha_1); \\
\text{change-weights (point4, point1, $\alpha_4$);} \\
\text{feedforward (point4);} \\
\text{backpropagation (point4);} \\
\text{gradient (point4, $g_4$);} \\
\text{if ($\alpha_4 \geq \alpha_3$) and ($\alpha_4 < \alpha_2$) then} \\
\text{if ($e_{r4} \leq e_{r3}$) then} \\
\begin{align*}
\alpha_1 := \alpha_3; e_{r1} := e_{r3}; g_1 := g_3; \\
\alpha_3 := \alpha_4; e_{r3} := e_{r4}; g_3 := g_4;
\end{align*}
\end{align*}
end
else 
begin
\begin{align*}
\alpha_2 := \alpha_4; e_{r2} := e_{r4}; g_2 := g_4;
\end{align*}
end;
\begin{align*}
\text{if ($\alpha_4 \geq \alpha_1$) and ($\alpha_4 < \alpha_3$) then} \\
\text{if ($e_{r4} \leq e_{r3}$) then} \\
\begin{align*}
\alpha_2 := \alpha_3; e_{r2} := e_{r3}; g_2 := g_3; \\
\alpha_3 := \alpha_4; e_{r3} := e_{r4}; g_3 := g_4;
\end{align*}
\end{align*}
end
else
    begin
      \alpha_1 := \alpha_4; \epsilon r_1 := \epsilon r_4; \epsilon g_1 := \epsilon g_4;
    end;

    until test;
    change-weights (point_0, point_0, \alpha_3);
    end;

where \epsilon r_p is the sum of the squares of the errors for the point \( p \). Also \( g \) is the gradient of the error which is calculated using the procedure Gradient.

For the line search the gradient of the error function is the derivative of the error function with respect to \( \alpha \), so if

\[ \phi(\alpha) = E(w_n + \alpha d_n). \]

Then the derivative of \( \phi(\alpha) \) with respect to \( \alpha \) is

\[ \phi'(\alpha) = E'(w_n + \alpha d_n)^T d_n, \]

where

\[ E'(w_n + \alpha d_n) = \delta_j a_i. \]

The procedure gradient calculates \( g(w_n + \alpha d_n)^T d_n \) for all output elements.

The line search is terminated when the Wolfe test is satisfied, if the Wolfe test is not satisfied the process is repeated using the new value of \( \alpha \) which replaces the poorest point.

The learning procedure for the SD method is described as follows:-
procedure SD-learning;
    var
    .
    begin
    for i:=1 to cycle-num do
    begin
    v:=0;
    while (v < vect-size) do
    begin
    v:=v+1;
    for j:=1 to layer[1] do
        active[v,j]:=input[v,j];
    feedforward(v);
    feedback(v);
    end;
    Quadratic-linesearch;
    end;
    end;

5.4 Conjugate Gradient Method

The line search guarantees that $E(w_{n+1}) < E(w_n)$. Even though the Steepest Descent algorithm can be proved to converge for a large class of problems, unfortunately its convergence rate is linear. Hence a large number of iterations is required to converge.

The Conjugate Gradient method (discussed in Chapter 3) takes advantage of second order information. The search direction is a combination of the current gradient and a previous search direction such that

$$d_{n+1} = -g_{n+1} + \beta_n d_n,$$  \hspace{1cm} (5.2)

where $\beta_n$ can be calculated using various rules. The Fletcher-Reeves rule is used to determine the value of $\beta_n$ as

$$\beta_n = \frac{g_{n+1}^T g_{n+1}}{g_n^T g_n}.$$  \hspace{1cm} (5.3)
Again as in the Steepest Descent algorithm, $\alpha_n$ is calculated using a line search. The quadratic interpolation method is used with the Wolfe test for terminating the search.

The outline of the Conjugate Gradient algorithm for feedforward network can be described as follows.

```plaintext
procedure CG-learning;
 var
 
 begin
 for $i$:=1 to cycle-num do
 begin
 $v$:=0;
 while ($v$<vect-min) do
 begin
 $v$:=v+1;
 for $j$:=1 to layer[1] do
 active[v,j]:=input[v,j];
 feedforward(v);
 feedback(v);
 end;
 { calculate the value of $\beta_{i-1}$ according to Equation (5.3) where $\beta_0 = 0$ }
 if $\beta_{i-1}$ > 1 then
 $\beta_{i-1}$ := $\beta_{i-2}$
 $j$:=layerp[2];
 while ($j$ $\leq$ tot - nodes) do
 begin
 $delta_i[j, k]$ := $-g[j, k] + \beta_{i-1}delta_{i-1}[j, k]$: { where $k$ is all nodes connected to node $j$ and $g[j, k]$ here is the gradient due connection weight from node $k$ to $j$. }
 Quadratic-line-search;
 end;
 end;
 end;
```

5.5 The Batch BP Experimental Results

In this section the experimental results for the Batch BP algorithm is compared with the Batch GRBH algorithm as well as the Steepest Descent
and Conjugate Gradient methods. These algorithms were applied to the same problems as in Section (5.2).

It was observed that much larger learning rates can be used for the Batch GRBH algorithm compared with the Online GRBH algorithm, without causing the network to diverge. For the Batch GRBH algorithm the best results were obtained for $G = 3$ and $G = 5$.

5.5.1 THE XOR PROBLEM

Using the Batch BP method the learning rate used is $\alpha = 35$ with the momentum term being $\beta = 0.9$. The learning rates chosen for the Batch GRBH algorithm are shown in Table (5.7).

<table>
<thead>
<tr>
<th>Number of groups $G = 3$</th>
<th>Number of groups $G = 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient $</td>
<td>\partial E/\partial w</td>
</tr>
<tr>
<td>$R_1 \leq 10^{-5}$</td>
<td>3500</td>
</tr>
<tr>
<td>$R_1 &lt; R_2 \leq 10^{-3}$</td>
<td>350</td>
</tr>
<tr>
<td>$R_2 &lt; R_3$</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>35</td>
</tr>
</tbody>
</table>

Table (5.7) The chosen values of $\alpha$ for the XOR problem using the Batch GRBH algorithm.

Figure 5.12 shows diagrammatically the chosen values of $\alpha$ for the different ranges.
Figure 5.12 XOR problem: The chosen values of $\alpha$ corresponding to two different groups.

The training process was terminated when the sum of the squares of the errors reached $0.5 \times 10^{-5}$. Table (5.8) shows the experimental results where the number of iterations are the average of 10 trials.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average number of iterations</th>
<th>% Improvement over BP</th>
<th>Number of cases not converged</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch BP</td>
<td>16130</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>GRBH with $G = 5$</td>
<td>154</td>
<td>99.1%</td>
<td>1</td>
</tr>
<tr>
<td>GRBH with $G = 3$</td>
<td>79</td>
<td>99.5%</td>
<td>1</td>
</tr>
<tr>
<td>SD</td>
<td>350</td>
<td>97.8%</td>
<td>1</td>
</tr>
<tr>
<td>CG</td>
<td>200</td>
<td>98.8%</td>
<td>2</td>
</tr>
</tbody>
</table>

Table (5.8): The simulation results for the XOR problem using the Batch BP algorithm.
From the Table (5.8) it can be seen that the GRBH algorithm with $G = 3$ and $G = 5$ improves the convergence by 99%. This is due to using large learning rates of the order $10^{+3}$. Even though the SD and CG algorithms have a better convergence rate than the BP algorithm, the execution time per cycle is much greater than the BP algorithm. However, the CG algorithm required less iterations than SD method due to the fact that second order information is used to calculate the new direction.

Figure 5.13 displays the progression of the learning algorithms. The x-axis of the graph represents logarithmically the number of iterations to the solution and the y-axis represents logarithmically the sum of the squares of the errors. It can be seen that the GRBH algorithm shows an improvement after only a few iterations.

![Graph showing learning progression](image)

**Figure 5.13** The learning progression of the XOR problem.

### 5.5.2 THE SHIFT REGISTER PROBLEM

For the Batch BP algorithm a set of experiment using the values of $\alpha = 5$ and $\beta = 0.9$ were chosen. For the Batch GRBH algorithm two sets of experiments were carried out with $G = 3$ and $G = 5$. Table (5.9) shows the chosen values of $\alpha$. 

144
Table (5.9) The chosen values of $\alpha$ for the shift register problem using the Batch GRBH algorithm.

Figure 5.14 illustrates diagrammatically the chosen values of $\alpha$ for different ranges.

<table>
<thead>
<tr>
<th>Number of groups $G = 3$</th>
<th>Number of groups $G = 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient $</td>
<td>\partial E/\partial w</td>
</tr>
<tr>
<td>$R_1 \leq 10^{-5}$</td>
<td>500</td>
</tr>
<tr>
<td>$R_1 &lt; R_2 \leq 10^{-3}$</td>
<td>50</td>
</tr>
<tr>
<td>$R_2 &lt; R_3$</td>
<td>.5</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.14 Shift register problem: The chosen values of $\alpha$ corresponding to two different groups.
The learning process is terminated when the sum of the mean squares of the error reached $0.5 \times 10^{-3}$. Table (5.10) shows the experimental results. The number of iterations are the average of 10 trials.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average number of iterations</th>
<th>% Improvement over BP</th>
<th>Number of cases not converged</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch BP</td>
<td>90375</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>GRBH with $G = 5$</td>
<td>2029</td>
<td>97.8%</td>
<td>2</td>
</tr>
<tr>
<td>GRBH with $G = 3$</td>
<td>1280</td>
<td>98.6%</td>
<td>1</td>
</tr>
<tr>
<td>SD</td>
<td>25023</td>
<td>72.3%</td>
<td>2</td>
</tr>
<tr>
<td>CG</td>
<td>2467</td>
<td>97.3%</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table (5.10):** The simulation results for the shift register problem using the Batch BP algorithm.

From Table (5.10) it can be seen that the GRBH algorithm with $G = 3$ requires only 1280 iterations to obtain the solution whereas the BP algorithm required 90375 iterations. The number of networks that got stuck at a local minima using the CG algorithm is larger than the other algorithms. Figure 5.15 shows the progression of the learning algorithms. The x-axis of the graph represents logarithmically the number of iterations to the solution and the y-axis represents logarithmically the sum of the squares of the errors. It can be seen that the SD method starts very slowly and after a certain point it converges to the required solution in a very few iterations. On the other hand however, the GRBH algorithm starts converging to the solution quite quickly from the start of the algorithm.
**Figure 5.15** The learning progression of the shift register problem.

### 5.5.3 SORTING PROBLEM

For the Batch BP algorithm the values chosen was $\alpha = 4$ and $\beta = 0.9$. For the Batch GRBH algorithm two sets of experiments were carried out with $G = 3$ and $G = 5$. Table (5.11) shows the chosen values of $\alpha$.

<table>
<thead>
<tr>
<th>Number of groups $G = 3$</th>
<th>Number of groups $G = 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient $</td>
<td>\partial E/\partial w</td>
</tr>
<tr>
<td>$R_1 \leq 10^{-5}$</td>
<td>400</td>
</tr>
<tr>
<td>$R_1 &lt; R_2 \leq 10^{-3}$</td>
<td>40</td>
</tr>
<tr>
<td>$R_2 &lt; R_3$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table (5.11) The chosen values of $\alpha$ for the sorting problem using the Batch GRBH algorithm.
Figure 5.16 illustrates diagrammatically the chosen values of $\alpha$ for different ranges.

\[ R_1 \quad R_2 \quad R_3 \quad R_4 \quad G=3 \]

\[ \log(\alpha) \]

\[ R_1 \quad R_2 \quad R_3 \quad R_4 \quad G=5 \quad \text{Range} \]

**Figure 5.16** Sorting problem: The chosen values of $\alpha$ corresponding to two different groups.

The learning process is terminated when the sum of the squares of the errors reached $0.5 \times 10^{-3}$. Table (5.12) shows the experimental results. The number of iterations are the average of 10 trials.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average number of iterations</th>
<th>% Improvement over BP</th>
<th>Number of cases not converged</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch BP</td>
<td>46958</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>GRBH with $G = 5$</td>
<td>475</td>
<td>99.0%</td>
<td>1</td>
</tr>
<tr>
<td>GRBH with $G = 3$</td>
<td>660</td>
<td>98.6%</td>
<td>0</td>
</tr>
<tr>
<td>SD</td>
<td>19638</td>
<td>58.2%</td>
<td>1</td>
</tr>
<tr>
<td>CG</td>
<td>945</td>
<td>98.0%</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table (5.12):** The simulation results for the sorting problem using the Batch algorithm.

From Table (5.12) it can be seen that the GRBH algorithm improves the convergence by 99%. The SD algorithm has a better convergence rate than
the Batch BP algorithm but each iteration requires a larger execution time due to the line search to find the best value of $\alpha$.

Figure 5.17 displays the progression of the learning algorithms. The x-axis of the graph represents logarithmically the number of iterations to the solution and the y-axis represents logarithmically the sum of the squares of the errors. It can be seen that the GRBH algorithm shows an improvement after only a few iterations. On the other hand, the SD algorithm converges very slowly up to a certain point and then it converges quickly to the solution in a few iterations. On the other hand the CG algorithm required much less iterations than SD method.

![Graph showing the learning progression of the sorting problem.](image)

**Figure 5.17** The learning progression of the sorting problem.

### 5.6 Two Intertwined Spirals

In 1988 Lang and Witbrock showed a network architecture which can learn to distinguish between two intertwined spirals. Even though it is easy to distinguish between the two spirals visually, it is not so easy to train a
network due to the fact that the error surface is highly nonlinear. The ANNs architecture uses a number of hidden layers with short cut connections to speedup the learning process. In this section, the GRBH algorithm is used to train the network and the results are compared with the usual BP algorithm as well as the Quickprop strategy (Fahlman [1988]). The progression of the learning by the GRBH algorithm for a number of cases is illustrated and compared with the BP algorithm as well as Quickprop.

5.6.1 THE SPIRAL TASK

Two spirals $A$ and $B$ are intertwined as shown in Figure 5.18 A number of points (i.e training set) are used to train to distinguish between the two spirals.

![Figure 5.18 The two intertwined spirals.](image)

The input data consists of the $x, y$ coordinate of the point and its output can have only two states each indicates one of the spirals. So if the point $x, y$ lies on spiral $A$ then the output is 1 otherwise the the output is zero and the point lies on spiral $B$. 

150
The training set was generated by a program, the $x$ and $y$ values for the spiral $A$ is calculated as

$$x = rsin\theta$$

$$y = rcos\theta,$$

where

$$r = 6.5 \frac{(104 - i)}{104}$$

$$\theta = i \frac{\pi}{16},$$

and

$$i = 0, 1, \ldots, k - 1.$$

The output of the spiral $A$ is made 1.

The $x, y$ values for the spiral $B$ is calculated as in Equation (5.4) except that

$$x = -rsin\theta$$

$$y = -rcos\theta.$$

The output for spiral $B$ is made zero. This will generate $2k$ points for the training set. The value of $k$ is changed to give different training set sizes.

**5.6.2 THE NETWORK ARCHITECTURE**

Lang and Witbrock [1988] suggested a 4 layer network with 3 hidden layers and one output layer. Each layer is connected to all the layers below it, which gives short cut connections between the layers. For a complex problem a large number of layers are needed, but as more layers are added the learning rate slows down. This is due to the fact that the error signal reduces each time it passes backwards one layer. By using a fully connected network it allows more information to each node.
The network used in this application consisted of 2 input nodes, one output node and five nodes per hidden layer. The network is illustrated in Figure 5.19.

![Network Architecture](image)

**Figure 5.19** The network architecture for the two intertwined spiral task.

### 5.6.3 BP AND QUICKPROP RESULTS

Lang and Witbrock [1988] used the Vanilla BP algorithm to train the network. The initial parameters used was $\alpha = 0.001$ and $\beta = 0.5$. The values were gradually increased and the final values of these parameters were $\alpha = 0.002$ and $\beta = 0.95$. Also, the Cross Entropy BP algorithm was used to train the network. The Cross Entropy BP algorithm is the same as the BP algorithm except that the error function is replaced by

$$E_n = \sum_p \sum_i t_{pi} \log_2(a_{pi}) + (1 - t_{pi} \log_2(1 - a_{pi})). \quad (5.5)$$

The training process was terminated when the activation value of the output node was within 0.4 of the target values.
The above three algorithms were tested on three networks with different starting points. The range of the random weights chosen was in the interval $[-0.1, +0.1]$. It was shown that the Quickprop method on average required less iterations than the other algorithms. Table (5.13) shows the results obtained by Lang and Widbrock [1988].

<table>
<thead>
<tr>
<th>Learning Algorithm</th>
<th>Trial BP</th>
<th>Cross Entropy BP</th>
<th>Quickprop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>18900</td>
<td>16200</td>
<td>4500</td>
</tr>
<tr>
<td>B</td>
<td>22300</td>
<td>8600</td>
<td>12300</td>
</tr>
<tr>
<td>C</td>
<td>19000</td>
<td>7600</td>
<td>6800</td>
</tr>
<tr>
<td>Mean</td>
<td>20000</td>
<td>10000</td>
<td>8000</td>
</tr>
</tbody>
</table>

Table (5.13): Number of iterations needed for the intertwined spiral.

Figure 5.20, 5.21, and 5.22 illustrates the progression of the learning by Vanilla BP, Cross-Entropy BP and Quickprop for the two trials A and C that was shown by Lang and Witbrock [1988].

![Graph showing learning progression](image)

**Figure 5.20** The learning progression for trials A and C for the Vanilla BP.
Figure 5.21 The learning progression for trials $A$ and $C$ for the Cross-Entropy BP.

Figure 5.22 The learning progression for trials $A$ and $C$ for the Quickprop.
5.6.4 The GRBH Results

The GRBH algorithm was implemented using the value of $G = 5$ as the number of ranges, Table (5.14) gives the $\alpha$ parameters used for training and $\beta = 0.95$.

| $G = 5$ | $|\partial E/\partial w|$ | $\alpha$ |
|---------|----------------|---------|
|         | $R_1 \leq 10^{-6}$ | 80      |
|         | $R_1 < R_2 \leq 10^{-5}$ | 40      |
|         | $R_2 < R_3 \leq 10^{-4}$ | 8       |
|         | $R_3 < R_4 \leq 10^{-3}$ | 4       |
|         | $R_4 < R_5$ | 0.4     |

Table (5.14): The chosen values of $\alpha$ with $G = 5$.

Figure 5.23 shows diagrammatically the chosen values of $\alpha$.

![Graph showing the values of $\alpha$ corresponding to the gradient values.](image)

Figure 5.23 The values of $\alpha$ corresponding to the gradient values.

The network architecture of Section 5.6.2 is used for training the spiral task. Four networks with different starting points were used to test the
algorithm. Table (5.15) shows the experimental results.

<table>
<thead>
<tr>
<th>Trial</th>
<th>GRBH Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2603</td>
</tr>
<tr>
<td>2</td>
<td>1323</td>
</tr>
<tr>
<td>3</td>
<td>3313</td>
</tr>
<tr>
<td>4</td>
<td>1187</td>
</tr>
<tr>
<td>Mean</td>
<td>2106</td>
</tr>
</tbody>
</table>

**Table (5.15):** Number of iterations needed for the intertwined spiral.

Figure 5.24 displays the progression of the GRBH learning algorithm for the four cases. The x-axis of the graph represents the number of iterations to the solution and the y-axis represents the number of vector pairs in error.

![Graph of iterations vs. errors](image)

**Figure (5.24)** The progression of the GRBH learning algorithm for the intertwined spiral problem.
5.7 Conclusion

It can be observed that the BP algorithm is very slow even when the momentum term is used. In general, the On-line BP algorithm requires less iterations than the Batch algorithm. This is due to the fact that the connection weights are modified more often than the Batch method. The problem with the On-line method is that the execution time required per cycle is larger than the Batch method and is dependent on the size of the training set. The GRBH method improves both these algorithms. For the On-line GRBH algorithm the improvement was around 80% whereas for the Batch GRBH algorithm the corresponding value was around 99%. This is due to the fact that much larger learning rates were used for the Batch GRBH algorithm without causing the network to diverge. In all cases, the convergence of the GRBH algorithm was guaranteed by keeping the learning rates very small for large modulus values of the gradient.

Even though the SD method converges faster than the BP algorithm the execution time per cycle is larger than the BP algorithm. Also the convergence rate of the SD algorithm is slower (up to a certain point) than the other algorithms including BP. The CG method has an improved convergence rate over the SD method, but due to the fact that the execution time per cycle is larger than that of the GRBH method, the GRBH performance is still better than all other algorithms.

From the results it was shown that the GRBH algorithm improved the convergence rate by four fold compared with the Quickprop and five fold compared with the Cross-Entropy BP algorithm. This was due to the fact that the GRBH algorithm converges faster at the beginning of the learning process. It was possible to use large learning rates even though the task is highly nonlinear.
During the experiments it was observed that a network is more likely to converge when the initial sum of the squared errors $e$ is less than half the sum of the modulus of the errors $e_m$ (where $e_m = \sum_p \sum_i |(t_{pi} - a_{pi})|$). Also, a network converges faster to the solution when the rate of reduction of $e$ per cycle is larger than the rate of reduction of $e_m$. This can be explained that it is rather better to have error vectors with about equal values for all nodes and vector pairs than having some error vectors with large values and others with very small values. These objectives can be developed in an Heuristic algorithm in which the group step sizes and corresponding learning rates can be suitably modified to maximize the convergence.
CHAPTER VI

Character Invariance Using ANN
Pattern recognition requires complex computations due to the fact that a pattern has a large number of variables. These variables could have some added noise or geometrical changes. Many techniques have been developed to resolve this problem. (Wechster [1990], Casasent and Psaltis [1977]) ANNs have proved to be good classifiers, where the conventional classifiers are capable of creating boundaries with relatively restricted families (e.g. a set of hyperplanes) whereas the ANNs are capable of more general decision boundaries.

A number of researchers have used ANNs with different techniques to recognize invariant patterns (Lee and Oldham [1990], Burr [1988], Khotanzad and Lu [1990], Barnard and Casasent [1991], Widrow, Winter and Baxter [1987]). These techniques can be broadly classified into three groups i.e.,

1. ANN architecture design that is invariant to geometrical differences.
2. Training set.
3. Preprocessing the input signal to be invariant to geometrical differences.

In the following sections these three groups are described in more detail.
6.1 ANNs Architecture Invariant to Geometrical Changes

A number of researchers such as Fukushima [1980], Widrow and Winter [1988] have designed ANN architectures which are invariant to geometrical changes. In general, the ANN structure has connections between neurons which forces the same input of different transformation to have the same output. For example, let us assume that a network is designed to classify images which are invariant to rotation. Let us also assume that the image is the input signal to the network and that a neuron $a_j$ receives its signals from the input image. To make the network architecture invariant to rotation of its input signals, the sum of the input signals to a neuron should not change with rotation. This can be achieved by allowing the connection weights $w_{ji} = w_{jk}$, where $k$ and $i$ are the pixels that lie at equal distances from the centre of the image. The Neocognitron of Fukushima [1980] (Fukushima, Miyake and Ito [1983]) has an interesting architecture. At first they designed it to be trained using unsupervised learning, later they extended their design to include supervised learning. The Neocognitron architecture is based on the visual cortex structure. This is described in the following section.

6.1.1 THE STRUCTURE OF THE NEOCOGNITRON

The Neocognitron is a multilayered network based on the structure of the visual system. The network consists of a number of modules with an input layer $U_0$. This is shown in Figure 6.1

![Figure 6.1 The structure of Neocognitron.](image-url)
Each module consists of two layers, i.e. the \( U_s \) (simple cells) layer and the \( U_c \) (complex cells) layer. Each \( S \) and \( C \) layer consists of a number of cell planes and each plane may be visualized as a two dimensional array of cells. Within a module, \( S \) and \( C \) cell planes exist as pairs, for each \( S \) cell plane there is a \( C \) cell plane. The \( S \) cells are adaptive so that during training the connections to the \( S \) cells are modified. Each cell in the \( S \) layer responds to certain features of the input image. So each cell receives an input signal from a restricted area of the input image, which is called the receptive range. The receptive ranges of the cells overlap to cover the entire input image. This is shown in Figure 6.2. Each \( C \) cell receives its input from the \( S \) cells in the same module. These cells respond to the same features as the \( S \) cells except that they are less sensitive to a shift in position.

![Neocognitron connection structure](image)

**Figure 6.2** Neocognitron connection structure.

The cells in the higher stages have a higher receptive range, and therefore their sensitivity to the shift in position decreases.
Fukushima, Miyake and Ito [1983] demonstrated the ability of the Neocognitron to recognize Arabic letters. The network consists of nine layers with different cell plane sizes. This is shown in Figure 6.3. From Figure 6.3 it can be seen that the size of the cell planes reduces at every higher stage until the final C layer where a cell covers the entire input image.

![Figure 6.3 The structure of Neocognitron to recognize Arabic letters.](image)

### 6.1.2 CALCULATING THE OUTPUT OF A CELL

The output of a cell is a positive analog variable. The $S$ cells have excitatory inputs as well as inhibitory inputs. Let us assume that the number of cell planes in module $l$ is $K_{cl}$ (for the complex layer) and $K_{sl}$ (for the simple layer). Let us also assume that the area (receptive range) from which an $S$ cell in the $(l+1)$ module receives its signals is $A_l$. The output of an $S$ cell in the $k^{th}$ cell plane of the $l$ module is $a_{sl}(k, p)$ and is calculated as

$$a_{sl}(k, p) = r_1 \phi \left( \frac{1 + \sum_{\kappa=1}^{K_{cl}-1} \sum_{v \in A_l} w_{cl}(\kappa, v, k) a_{cl-1}(\kappa, p + v)}{1 + \frac{r_1}{1+r_1} w_{hl}(k) h_{cl-1}(p)} - 1 \right), \quad (6.1)$$

$k = 1, 2, \ldots, K_{sl}$
where $p$ is the position of the cell, $w_{el}$ and $w_{hl}$ are the connection weights from the excitatory and inhibitory cells respectively, $h$ is the activation value of the inhibitory cell, $r_I$ is a parameter to control the intensity of the inhibitory connection. If $r_I$ is large, the cell becomes more selective to its input features.

and

$$\phi(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0. \end{cases} \quad (6.2)$$

Figure 6.4 shows the input to output characteristics of the $S$ cell.

The output of the inhibitory cell is calculated to be the weighted root mean square of its inputs. This is given as

$$h_{cl-1}(p) = \left( \sum_{\kappa=1}^{K_{cl-1}} \sum_{\nu \in A_1} u_{l-1}(\nu) a_{cl-1}^2(\kappa, p + \nu) \right)^{0.5}, \quad (6.3)$$

where $u_{l-1}(\nu)$ is an unmodifiable connection weight and it should satisfy the following condition

$$K_{cl-1} \sum_{\nu \in A_1} u_{l-1}(\nu) = 1. \quad (6.4)$$

The output of the $C$ cell can be calculated as
\[ h_{cl}(k, p) = \psi \left( \sum_{\kappa=1}^{K_{cl}} j_l(\kappa, k) \sum_{v \in D_l} w_{dl}(v, k)a_{dl}(\kappa, p + v) \right), \] (6.5)

where

\[ \psi(x) = \begin{cases} 
\frac{x}{q_l + x} & x \geq 0 \\
0 & x < 0,
\end{cases} \]

\( q_l \) is a parameter to control the output value, and \( j_l \) can be zero or one. If there is a connection between plane \( \kappa \) and \( k \) then \( j_l = 1 \) otherwise \( j_l = 0 \). The value of \( w_{dl}(v, k) \) is the connection weight of the excitatory from the \( S \) cell to the \( C \) cell.

### 6.1.3 TRAINING THE NEOCOGNITRON

The Neocognitron can be trained using either unsupervised or supervised learning. The connection weights are adjusted layer by layer starting with the first module. For the supervised learning, the \( S \) cell planes are trained one at a time. A cell from the cell plane is chosen, and the connection weights through which non zero signals are coming is modified. The connection weights of all the other cells in the cell plane are modified in an identical manner to the chosen cell which is called the representative cell. The connection weights of the excitatory cells are modified as

\[ \Delta w_{el}(\kappa, v, k) = \alpha_l u_{l-1}(v)a_{cl-1}(\kappa, p + v), \] (6.6)

where \( \alpha_l \) is the learning rate and is a positive constant value. The connection weights of the inhibitory cells are modified as

\[ \Delta w_{hl}(\kappa) = \alpha_l h_{cl-1}(p), \] (6.7)
6.2 Training Set

By using this method a large number of training patterns are used. For each class, a large number of examples with different transformations is required. This is necessary for the network to capture the required features. The input pattern to the network is the image itself without any changes. Rumelhart and McClelland [1986] used such a method to train a network to recognize letters which are invariant to rotation. They used a very small input comprising of $3 \times 3$ pixels. Thus, the method has a number of disadvantages:

1. It requires a large number of training patterns.
2. For realistic problems, the input vector can be very large. Hence, the training time could be very long.
3. The use of such techniques do not explain the way the network generalizes. It is not yet clear why such networks work (if they do work?).

6.3 Moment Invariant

As mentioned previously, the Neocognitron or any ANN architecture for invariant character recognition is too complex for our present technology. Another method is to preprocess the input data and represent them in another form which are invariant to geometrical changes.

Moment invariants were introduced by Hu [1962] as pattern recognition features. These features are invariant to position, size and rotation. The advantage of using such a method is that the number of input features is reduced considerably. One drawback to such methods is that, preprocessing of the input data is needed to extract the new features. To avoid expensive preprocessing calculations, the size of the image should be constrained. There are other invariant feature spaces such as Wedge-ringed samples of magnitude
of the Fourier transform, and Fourier transform in log polar coordinates. These two methods are not invariant to all possible transformations.

6.3.1 GEOMETRIC MOMENTS (GM)

Hu [1962] introduced moment invariants based on algebraic invariant methods. By using nonlinear combinations of the geometric moments, a set of moment functions which have the desired property of being invariant to translation, scale, and rotation are formed (Reddi [1981]).

Let us assume a two dimensional $N \times N$ continuous function $f(x, y)$, the moment of the order $p + q$ can be defined as

$$m_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^p y^q f(x, y) dx dy,$$  \hspace{1cm} (6.8)

where $p, q = 0, 1, 2, \ldots$.

Let us also assume that the image is centred at $(\bar{x}, \bar{y})$ where

$$\bar{x} = \frac{m_{10}}{m_{00}},$$

and

$$\bar{y} = \frac{m_{01}}{m_{00}}.$$  

From this information a set of central moments which are invariant to translation, can be defined as

$$\mu_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (x - \bar{x})^p (y - \bar{y})^q f(x, y) dx dy.$$ \hspace{1cm} (6.9)

The discrete moment is defined as

$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y).$$ \hspace{1cm} (6.10)
and the discrete central moment is defined as

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y). \quad (6.11)$$

To have scaled images, the normalized central moments are used. These can be calculated as

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}}, \quad (6.12)$$

where

$$\gamma = \frac{p + q}{2} + 1.$$

Hu [1962] derived a set of moments that are invariant to translation, scale and rotation by using the normalized central moments. The set consists of seven moments and are given by

$$M_1 = \eta_{20} + \eta_{02}$$

$$M_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$$

$$M_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$$

$$M_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$$

$$M_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12}) [(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2]$$

$$+ (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03}) [3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

$$M_6 = (\eta_{20} - \eta_{02}) [(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$$

$$M_7 = (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12}) [(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2]$$

$$- (\eta_{30} - 3\eta_{12})(\eta_{12} + \eta_{03}) [3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]. \quad (6.13)$$

The functions $M_1$ through $M_6$ are invariant under reflection, on the other hand $M_7$ changes sign.
The values of $M_1$ to $M_6$ are very small and therefore the values of $\log |M_k|$ where $(k = 1, 2 \ldots 6)$ are used. The training patterns are normalized with zero mean and unit variance. This is done to avoid some of the features dominating the training process. The normalized features are calculated as

$$M_k' = \frac{M_k - \bar{M}_k}{\sigma_k},$$

where $\bar{M}_k$ is the mean of the $k^{th}$ feature of all patterns, and $\sigma_k$ is the standard deviation of the $k^{th}$ feature of all patterns. Before calculating the geometrical moments, the image $N \times N$ (where $N$ now denotes image size) is mapped onto a square region such that

$$x \in [-1, +1] \quad \text{and} \quad y \in [-1, +1].$$

This is done to keep the dynamic range of $m_{pq}$ consistent with different image sizes. To calculate the values of $M_1, \ldots, M_6$, it can be seen from Equation (6.13) only eight different moments are needed. These are $m_{00}, m_{11}, m_{20}, m_{02}, m_{12}, m_{21}, m_{03}, m_{30}$. The main features of the Geometrical moment procedure is shown below.

It is important to point out here that the values of the standard deviation and the mean of the features are stored in a file. This is important when a new set of examples is used to test the trained network. The Geometrical moments can be calculated and when normalizing the features, the mean and the standard deviation is loaded from the file rather than being calculated.
procedure GM;

begin

for \( v := 1 \) to \( \text{vector-size} \) do

begin

initialize the variables \( m_{ij} \) and \( \mu_{ij} \)

\( \{ m_{ij} \) is the geometrical moment \}

\( x_m := m[1, 0]/m[0, 0]; \)

\( y_m := m[0, 1]/m[0, 0]; \)

\( p := 0; \)

\( q := 0; \)

for \( \ell := 0 \) to 7 do \{ calculate the eight normalized moments \}

begin

calculate the normalized moment \( \mu_{pq} \);

if \( (p \mod 2) = 0 \) then

\( p := p - 1; \)

if \( p < 0 \) then

\( p := 1 \)

else

\( p \leftrightarrow q; \)

end;

Calculate the six features according to Eqn. (6.13)

end;

save the vector pairs.

end;

6.3.2 Complex Moments (CM)

The notion of Complex moments was introduced by Abu-Mostafa and Psaltis [1984] as a simple and straightforward way to derive moment invariants. The Complex moment of the order \((p, q)\) for a two dimensional image \(f(x, y)\) can be calculated by the following equation:

\[
C_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (x + iy)^p(x - iy)^q f(x, y) dx dy, \tag{6.15}
\]
where \( p \) and \( q \) are positive integers and \( i = \sqrt{-1} \). Now if \( f(x, y) \) is positive then \( C_{pq} \) is a positive number and \( C_{qp} \) is the complex conjugate of \( C_{pq} \). The discrete form of Equation (6.15) is defined as

\[
C_{pq} = \sum_x \sum_y (x + iy)^p (x - iy)^q f(x, y).
\]  

(6.16)

When an image is rotated by an angle \( \theta \), the new \( C_{pq} \) in relation to the unrotated image is given by

\[
C'_{pq} = C_{pq} e^{-i(p-q)\theta}.
\]  

(6.17)

It can be seen from Equation (6.17) that the magnitude of \( C_{pq} \) does not change with rotation, only the phase value is effected. So the absolute value of the Complex moments can be used for invariant rotation features. Taking the absolute values of \( C_{pq} \) causes information loss, this is due to the fact that \( C_{pq} \) is the complex conjugate of \( C_{qp} \) and therefore we can get \( \frac{n}{2} + 1 \) invariants out of \( n + 1 \) CMs of degree \( n \). It is possible to get more features by using the phaser information of CMs. This can be done by choosing a combination of CMs, from Equation (6.17) it can be seen that \( C_{21} C_{24} \) is a complex number with invariant magnitude and phase under rotation.

The Complex moments are a linear combination with the complex coefficients of the Geometric moments \( m_{rs} \), where \( p + q = r + s \). The Complex moments and Geometric moments are related by

\[
C_{pq} = \sum_{r=0}^{p} \sum_{s=0}^{q} \binom{p}{r} \binom{q}{s} i^{p+q-(r+s)} \cdot (-1)^{q-s} m_{r+s,p+q-(r+s)}.
\]  

(6.18)

The absolute value of the CMs are only invariant to rotation. To achieve scale and translation invariancy, the images should be normalized against
scale and translation (Abu-Mostafa and Psaltis [1985]). Let us assume that the image $f_2(x, y)$ is a scaled and translated version of the image $f_1(x, y)$. These two images are related by

$$f_2(x, y) = f_1 \left( \frac{x}{D} + \bar{x}, \frac{y}{D} + \bar{y} \right),$$

(6.19)

where $\bar{x}, \bar{y}$ are the coordinates of the centroid, and $D$ is the dimensional scale. Scale invariance is achieved by enlarging or reducing the image such that the new zeroth order moment $m'_{00}$ has a constant value of $B$ so

$$m'_{00} = B.$$  

(6.20)

From Equation (6.8), the geometrical moment of the new image $f_2(x, y)$ is

$$m'_{pq} = \int_x \int_y x^p y^q f_1 \left( \frac{x}{D}, \frac{y}{D} \right) dx dy$$

$$= \int_x \int_y x^p y^q D^{p+q+2} f_1(x, y) dx dy$$

(6.21)

$$= D^{p+q+2} \int_x \int_y x^p y^q f_1(x, y) dx dy.$$

Therefore the geometrical moments of $f_1(x, y)$ and $f_2(x, y)$ are related such that

$$m'_{pq} = D^{p+q+2} m_{pq}. $$

(6.22)

By choosing $m'_{00} = B$, from Equation (6.22) $D$ can be calculated as

$$D = \left( \frac{B}{m_{00}} \right)^{0.5}.$$ 

(6.23)

Translation invariancy is achieved by making the centroid to be the origin of the new image. This can be done by using the following condition,

$$C_{10} = 0 \quad \text{(for } f_2(x, y)).$$
$C_{10}$ can be written in terms of the GMs as

$$C_{10} = m_{10} + im_{01}.$$ 

Therefore, the coordinates of the centroid of $f_2(x, y)$ is calculated in terms of the GMs of $f_1(x, y)$ as

$$\bar{x} = \frac{m_{10}}{m_{00}} \quad \bar{y} = \frac{m_{01}}{m_{00}}. \quad (6.24)$$

To calculate CMs that are invariant to scale, translation, and rotation, an image should be scaled and then the absolute values of the CMs of the new image is calculated.

It should be noted here that the relative sizes of the CM values can be large. It can be shown that these values reduces sharply as $p + q$ increases.

*What features should be selected?*

The features that we choose to use should have enough information to be able to discriminate between the images. The following are a number of points to be considered before choosing the features:

1. **Information loss.** When CMs are used as features for pattern recognition some information is lost. This is due to the fact that the entire image is replaced by a limited number of features. This can be well explained from the CMs and the circular harmonic expansion of the image. Let us assume that $F(r, \theta)$ is the image function in the polar coordinate, that is

$$F(r, \theta) = f(r \cos \theta, r \sin \theta). \quad (6.25)$$

Therefore Equation (6.15) can be rewritten in the polar coordinates as

$$C_{pq} = \int_0^{2\pi} \int_0^\infty r^{p+q} e^{i(p-q)\theta} F(r, \theta) rdrd\theta. \quad (6.26)$$
The circular harmonic expansion of $F(r, \theta)$ can be written as

$$F(r, \theta) = \sum_{n=-\infty}^{\infty} c_n(r)e^{in\theta}, \quad (6.27)$$

where $c_n(r)$ is a measure of the angular variation of frequency $n$ cycles/cycle at radius $r$. By substituting Equation (6.27) into Equation (6.26) we can have

$$C_{pq} = 2\pi \int_{0}^{\infty} r^{p+q+1} c_{q-p}(r)dr. \quad (6.28)$$

From Equation (6.28) we can see that when all CMs up to order $n$ are used then these features contain only the circular harmonic expansion from $c_{-n}(r)$ to $c_{n}(r)$ of the image $f(x, y)$. This indicates that any information with angular variation $\geq n + 1$ is lost. Therefore such features are not suitable for images with high frequency information. On the other hand, if the image used does not have important high frequency information and the noise presented has certain frequency distribution then the features that contain the noise frequencies can be ignored. In such cases CMs can be useful.

2. Information Suppression. From Equation (6.28) let us restrict the image to a unit disk to obtain the following equation

$$C_{pq} = 2\pi \int_{0}^{1} r^{p+q+1} c_{-q-p}(r)dr. \quad (6.29)$$

From this equation it can be seen that for a constant value of $p$ and $q$ the term $r^{p+q+1}$ varies with $r$ (the radius). This term is a weight to $c_{p-q}(r)$ and this weighting decreases as $r$ is reduced. So for small $r$ the term $r^{p+q+1}$ can be very small and it can be lost in rounding error. Therefore, $C_{pq}$ contains information mainly from the boundary of the image and most of the information in the central part of the image is suppressed.
All the features will have the same information suppressed, which can be critical if the information of the image is mainly in the centre.

3. Information Redundancy. The CMs are related and this is due to the fact that the information presented in $C_{pq}$ is also presented and dominated in $C_{p+1,q+1}$. As the order of the moment increases, the ratio of the new information to the redundant information decreases. Again, the higher order of CMs are information redundant.

The performance of the CMs is tested for pattern recognition using different orders of moments. The training patterns are normalized with zero mean and unit variance. This normalization procedure is the same as the one used for GM. In the CM procedure these moments are calculated from GM as in Equation (6.18). The main features of the CM procedure is described below.

```plaintext
procedure scale
    ...

begin
    for $v := 1$ to vector-size do
        begin
            $p := 0$;
            $q := 2$;
            if $l <> 5$ then
                begin
                    for $e := 1$ to 6 do
                        begin
                            calculate moment $m_{pq}$;
                            if $(l \mod 2) <> 0$ then
                                $p \leftrightarrow q$
                            else
                                $p := p - 1; p \leftrightarrow q$
                            end;
                            if $l = 5$ then
                                $p := 1; q := 1$;
                            end;
                            $xm := m[1,0]/m[0,0]$;
                            $ym := m[0,1]/m[0,0]$;
                        end;
                    end;
```
\[ a := \sqrt{B/m[0,0]}; \]

\[ \text{for } i := 1 \text{ to } N \text{ do } \{ N \text{ is the number of columns or rows in an image } \} \]

\[ \text{for } j := 1 \text{ to } N \text{ do } \]

\[ \begin{align*}
& \text{begin} \\
& \quad x := (xm + i/a); \\
& \quad y := (ym + j/a); \\
& \quad f_1[x, y] := f[i, j]; \{ f_1(x, y) \text{ is the new image } \} \\
& \text{end;} \\
& \text{save these values in the vector pair } V; \\
& \end{align*} \]

procedure CM

\[ \begin{align*}
& \begin{align*}
& \text{begin} \\
& \quad \text{for } u := 1 \text{ to vector-size do } \\
& \quad \quad \text{begin} \\
& \quad \quad \quad \text{initialize the variables } rc[i, j] \text{ (real part of CM)} \\
& \quad \quad \quad \text{and } ic[i, j] \text{ (the imaginary part of CM).} \\
& \quad \quad \quad \text{for } p := 0 \text{ to order do } \{ \text{order is the order of CM } \} \\
& \quad \quad \quad \text{for } q := 0 \text{ to } p \text{ do } \\
& \quad \quad \quad \quad \text{begin} \\
& \quad \quad \quad \quad \quad \text{for } r := 0 \text{ to } p \text{ do } \\
& \quad \quad \quad \quad \quad \quad \text{for } s := 0 \text{ to } q \text{ do } \\
& \quad \quad \quad \quad \quad \quad \quad \text{begin} \\
& \quad \quad \quad \quad \quad \quad \quad \quad \text{calculate } ic[i, j] \text{ and } ic[i, j] \text{ according to Eqn. (6.18)} \\
& \quad \quad \quad \quad \quad \quad \quad \quad \text{end;} \\
& \quad \quad \quad \quad \quad \quad \quad \quad c[p, q] := \sqrt{\text{sqr}(ic[p, q]) + \text{sqr}(rc[p, q])} \\
& \quad \quad \quad \quad \quad \quad \quad \quad \text{end;} \\
& \quad \quad \quad \quad \quad \quad \quad \quad \text{save these values for vector pair } V \\
& \quad \quad \quad \quad \quad \text{end;} \\
& \quad \quad \text{end;} \\
& \quad \text{end;} \\
& \text{end;} \\
& \end{align*} \]

6.3.3 ZERNIKE MOMENTS (ZM)

Zernike [1934] introduced the Zernike polynomials which are complex values and orthogonal on the unit disk \(x^2 + y^2 \leq 1\). The Zernike polynomials (Teh and chin [1988], Bhatia and Wolf [1954]) can be expressed as

\[ v_{nl}(x, y) = V_{nl}(r \cos \theta, r \sin \theta) = R_{nl}(r)e^{i\theta}, \quad (6.30) \]

175
where $n = 0,1,2,\ldots,\infty$, $l$ is a positive or negative integer value which satisfies the following conditions

$$n - |l| = \text{even}$$

(6.31)

$$|l| \leq n.$$  

These polynomials satisfy

$$\int_0^{2\pi} \int_0^1 [V_{nl}(r, \theta)]^* V_{mk}(r, \theta) r dr d\theta = \frac{\pi}{n + 1} \gamma_{mn} \gamma_{kl},$$

(6.32)

where $*$ is the complex conjugate and

$$\gamma_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise.} \end{cases}$$

The real valued radial polynomials ($R_{nl}(r)$) of Equation (6.30) satisfies the condition

$$\int_0^1 R_{nl}(r) R_{ml}(r) r dr = \frac{1}{2(n + 1)} \gamma_{mn},$$

(6.33)

and $R_{nl}(r)$ is defined as

$$R_{nl}(r) = \sum_{s=0}^{(n-|l|)/2} (-1)^s \frac{(n-s)!}{s!(n-|l| - s)!} \frac{(n-|l| - 2s)!}{(n-|l| - s)!} r^{n-2s},$$

(6.34)

$$= \sum_{\substack{k=|l| \\ n-k = \text{even}}} ^n B_{n|l|} k r^k.$$

The Zernike moments of order $n$ with repetition $l$ are defined as

$$A_{nl} = \frac{n + 1}{\pi} \int \int_{x^2 + y^2 \leq 1} v_{nl}(x, y) f(x, y) dx dy.$$  

(6.35)

For discrete images Equation (6.35) can be rewritten as

$$A_{nl} = \frac{n + 1}{\pi} \sum_y \sum_x v_{nl}(x, y) f(x, y).$$

(6.36)
where $x^2 + y^2 \leq 1$. Equation (6.35) can also be expressed in polar coordinates as

$$A_{nl} = \frac{n+1}{\pi} \int_0^{2\pi} \int_0^\pi [V_{nl}(r, \theta)]^* f(r \cos \theta, r \sin \theta) r dr d\theta. \quad (6.37)$$

The Zernike moments are related to the Geometric moments according to the following equation

$$A_{nl} = \frac{n+1}{\pi} \sum_{n-k=\text{even}}^n \sum_{j=0}^q \sum_{m=0}^{\mid l \mid} \omega \cdot \left( \begin{array}{c} q \\ j \end{array} \right) \left( \begin{array}{c} \mid l \mid \\ m \end{array} \right) B_{nl ||k-m-2j-m,2j+m},$$

where

$$\omega = \begin{cases} -i & l > 0 \\ +i & l \leq 0 \end{cases}, \quad (6.38)$$

$$q = \frac{1}{2}(k - \mid l \mid),$$

and

$$i = \sqrt{-1}.$$ 

The Zernike moments and the Complex moments are related by

$$A_{nl} = \frac{n+1}{\pi} \sum_{n-k=\text{even}}^n B_{nl ||k C_{1/(2(k-l))},1/(2(k+l))}. \quad (6.39)$$

If the Zernike moments are calculated from previously calculated Complex moments, then these Zernike features will also suffer from radial information suppression which was discussed in the previous section.

The image $f(x, y)$ can be constructed from the Zernike moments and their polynomials using the following equation

$$f(x, y) = \sum_{n=0}^\infty \sum_{\mid l \mid=0}^\infty A_{nl} v_{nl}(x, y). \quad (6.40)$$
When an image is rotated by an angle $\theta$, the new $A_{nl}$ in relation to the unrotated image is given by

$$A'_{nl} = A_{nl}e^{-i\theta}. \quad (6.41)$$

It can be seen from Equation (6.41) that the magnitude of $A_{nl}$ does not change with rotation, only the phase value is effected. Also $A_{nl}$ is the complex conjugate of $A_{n,-l}$, hence $|A_{nl}| = |A_{n,-l}|$. Therefore when Zernike moments are used as invariant features, the magnitude of the ZMs are used and also only $l \geq 0$ are considered. The ZMs are invariant only to rotation, to achieve scalar and translation invariancy, the scale procedure of Section (6.3.2) should be used. So an image is scaled and translated before its ZMs are calculated. It can be seen from Equation (6.38) that due to the scale procedure $|A_{00}|$ and $|A_{11}|$ have a constant value for all images. For $A_{00}$ we have

$$A_{00} = \frac{1}{\pi} m_{00},$$

where $m_{00}$ for the scaled images is a constant value. For $|A_{11}|$ we have

$$A_{11} = \frac{2}{\pi} m_{10} + i\frac{2}{\pi} m_{01},$$

where $m_{10} = m_{01} = 0$ for the scaled images hence $|A_{11}| = 0$

So what features should be selected?

The Zernike polynomials are orthogonal on the unit disk and therefore they do not suffer from information suppression or redundancy as in the CMs. This is true provided that ZMs are not calculated from the CMs. One problem with ZMs is that they do suffer from information loss. This can be explained using Equation (6.40). A limited number of features are used for
classification. Let us assume that all the ZMs up to order $N$ are used. From Equation (6.40) we will have

$$f(x, y) \approx \sum_{n=0}^{N} \sum_{l=0}^{N} A_{nl} v_{nl}(x, y).$$

(6.42)

It can be seen that it is inevitable to lose some information and the image cannot be reconstructed fully from the limited number of ZMs and their polynomials. The performance of ZMs is tested for pattern recognition, using different orders of moments. The training patterns are normalized with zero mean and unit variance. This normalization procedure is same as the one used for GM. In the ZM procedure these moments are calculated from GM as in Equation (6.38). The main features of the ZM procedure is described below.

```
procedure ZM
    var
        :;
    begin
        scale;
        for $v := 1$ to vect-size do
            begin
                { initialize the variables $r z[i,j]$ (the real part of ZM)
                and $i z[i,j]$ (the imaginary part of ZM) }
                for $n := 0$ to order do
                    begin
                        { where order is the order of the ZM }
                        for $l := 0$ to $n$ do
                            begin
                                if ($(n - l) \mod 2) = 0$ then
                                    begin
                                        for $k := l$ to $n$ do
                                            begin
                                                if ($(n - k) \mod 2) = 0$ then
                                                    begin
                                                        { calculate $B_{n,l,k}$ according to Equation (6.34) }
                                                        for $j := 0$ to $q$ do
                                                            for $m := 0$ to $l$ do
                                                                begin
```

179
\{ calculate \( A_{nl} \) according to Equation (6.38) \}

if \((m \mod 4) = 0\) then


else

if \((m \mod 2) = 0\) then


else

if \(((m + 1) \mod 4) = 0\) then


else


end

end

for \( n := 2 \) to \( \text{order} \) do

for \( l := 0 \) to \( n \) do

\( z[n, l] := \sqrt{\sqrt{rz[n, l]}^2 + \sqrt{iz[n, l]}^2}; \)

end;

end

end;

\{ normalize the features according to Equation (6.14) \}

\{ save the vector pairs \};

\section*{6.4 Minimum Number Of Hidden Nodes For ANNs}

Before describing the experimental results, it is important to show how to calculate the minimum number of hidden nodes needed for a certain problem. Recent results (Lippmann [1987]) show that the minimum number of hidden nodes for feedforward nets are dependent on the number of the vector pairs \( T \) used for training. The results indicates that the relation between the minimum number of hidden nodes \( H \) and \( T \) is given as

\[ H = \log_2(T). \] \hspace{1cm} (6.43)
Mirchandani and Cao [1989] extended this relation and showed that the number of separable regions $RG$ in the input space is a function of both $H$ and the dimension of the input vector $D$. They proved that the relation between $RG$, $D$, and $H$ is given as

$$RG(H, D) = \sum_{k=0}^{D} \binom{H}{k}, \quad (6.44)$$

where

$$\binom{H}{K} = 0, \quad H < K.$$

From Equation (6.44) it can be seen that as the number of hidden nodes increases, more separable regions of the input space is created. It is important to mention that the number of the vector pairs should at least equal the number of separable regions. That is

$$T \geq RG. \quad (6.45)$$

### 6.5 Experimental Description

A number of experiments were performed using ANNs as a classifier for the English letters. The three types of moments GM, CM, and ZM described earlier in this chapter were used to train different networks. The performance of the trained networks were compared with each other. These trained networks were also tested for noisy images. The networks were trained using both the GRBH and BP algorithms and the acceleration factor for the GRBH algorithm compared to the BP algorithm was shown.
6.5.1 TRAINING AND TESTING VECTOR PAIRS

Each image consisted of $16 \times 16$ binary pixels. These images were the English capital letters. Two sets of vector pairs were generated, one was used as a training set and the other was used as a testing set. For the training set, three examples of each letter were used with different sizes, orientations and translations. Therefore, 78 letters were used for the training set. The testing set also consisted of three examples of each letter. Figure 6.5a are the three examples of the letter A used for training the networks. Figure 6.5b are the three examples of the letter A used for testing the trained networks.

Figure 6.5 (a) The three training letters of A
(b) The three testing letters of A.
Noisy images were also created for testing the trained networks. This was achieved by changing randomly some of the values of the pixels of the normalized images. Three levels of noise were used with these images, such that their signal to noise ratios (SNR) are 50, 25, and 12 db. The signal to noise ratio of these images can be calculated as

$$SNR = 20 \log \left( \frac{p - l}{l} \right),$$

where $p$ is the total number of pixels in the image, and $l$ is the number of pixels in the noisy image that differs from the original image. Figure 6.6 shows the letter $A$ with three level of noise.

![Figure 6.6 The letter A with three levels of noise 50db, 25db and 12db respectively.](image)
6.5.2 NETWORK ARCHITECTURE

The networks were trained with different numbers of hidden nodes and different numbers of input features. Each network consisted of 26 output nodes each node representing one of the English letters. The number of input nodes were the same as the number of features taken from an image. The number of hidden nodes was made variable from 5 to 40 nodes. For all the networks used, the initial connection weights were in the interval \([-0.1, +0.1]\]. The momentum term \(\beta = 0.5\) and the values of the learning rates \(\alpha\) are given later for each experiment. Only the adjacent layers are connected with each other as shown in Figure 6.7.

![Figure 6.7 The network architecture.](image)

The training process of the GRBH algorithm was terminated when the sum of the squared errors \(e \leq 5 \times 10^{-2}\). For the BP algorithm the training process was terminated when \(e \leq 1 \times 10^{-1}\), this is due to the fact that a large number of iterations was required using the BP algorithm.

6.5.3 EXPERIMENTAL RESULTS USING GMs

By using GMs only 6 features from an image can be obtained. Therefore the networks consisted of 6 input nodes and 26 output nodes. For this problem, the number of separable regions is 26 (=RG) and the input dimension
By having this information, the minimum number of hidden nodes can be calculated using Equation (6.44), this is given as

\[ 26 = \sum_{k=0}^{6} \binom{H}{k}. \]

With \( H = 4 \) only 16 separable regions are possible, and with \( H = 5 \) there can be 32 separable regions. Hence, we need at least 5 hidden nodes. The experiments were repeated with \( H = 5, 10, 20, 30, \) and 40.

For the GRBH algorithm, the modulus of the gradient was divided into three groups. It was observed that larger learning rates can be used for networks with smaller numbers of hidden nodes. The training process for a network was repeated with slightly larger \( \alpha \) values until a value was found that caused the network to oscillate or diverge.

Table (6.1) shows the chosen values of \( \alpha \) against the modules of the gradient for a network with 10 hidden nodes.

| \(|\partial E/\partial w|\) Range | learning rates (\(\alpha\)) |
|-------------------------------|-----------------------------|
| \(R_1 \leq 10^{-5}\)         | 800                         |
| \(R_1 < R_2 \leq 10^{-3}\)   | 80                          |
| \(R_3 < R_2\)                 | 8                           |

Table (6.1): The chosen values of \(\alpha\) for a network with

\[ H = 10 \] and GRBH algorithm.
Figure 6.8 The values of \( \log(\alpha) \) against the modulus of the gradient range. For all the cases, the relative values of \( \alpha \) are given as
\[
\alpha_2 = 10\alpha_3
\]
\[
\alpha_1 = 100\alpha_3
\]  
(6.47)

Figure (6.9) shows the maximum possible values of \( \alpha_3 \) against different number of hidden nodes.

Figure 6.9 The maximum values of \( \alpha_3 \) against the number of hidden nodes \( H \).
Table (6.2) gives the number of iterations required for a network to converge using the GRBH algorithm. The training process was terminated when $e = 5 \times 10^{-2}$.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha_1$</th>
<th>Learning rate $\alpha_2$</th>
<th>Learning rate $\alpha_3$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>s</td>
</tr>
<tr>
<td>10</td>
<td>800</td>
<td>80</td>
<td>8</td>
<td>1685</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>40</td>
<td>4</td>
<td>680</td>
</tr>
<tr>
<td>30</td>
<td>400</td>
<td>40</td>
<td>4</td>
<td>646</td>
</tr>
<tr>
<td>40</td>
<td>300</td>
<td>30</td>
<td>3</td>
<td>1198</td>
</tr>
</tbody>
</table>

**Table (6.2):** The number of iterations required for a network to converge to $e = 5 \times 10^{-2}$ using GRBH algorithm.

Table (6.3) gives the number of iterations required for a network to converge using the BP algorithm. The training process was terminated when $e = 1 \times 10^{-1}$.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>-</td>
<td>s</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>9611</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>10585</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>5691</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>8056</td>
</tr>
</tbody>
</table>

**Table (6.3):** The number of iterations required for a network to converge to $e = 1 \times 10^{-1}$ using the BP algorithm.

Table (6.4) shows the effect of the learning values on the number of iterations. The training process was terminated when $e = 5 \times 10^{-2}$. Figure (6.10) shows diagrammatically the effect of $\alpha$ values on convergence. These experiments were performed on networks with $H = 5, 10, 20$. 

187
Table (6.4): The effect of $\alpha$ values on the convergence of the GRBH algorithm for networks with $H = 5, 10$ and $20$.

<table>
<thead>
<tr>
<th>Learning rate</th>
<th>Number of iterations with $H = 5$</th>
<th>$H = 10$</th>
<th>$H = 20$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha_1$</td>
<td>$\alpha_2$</td>
<td>$\alpha_3$</td>
<td></td>
</tr>
<tr>
<td>800</td>
<td>80</td>
<td>8</td>
<td>1685</td>
</tr>
<tr>
<td>600</td>
<td>60</td>
<td>6</td>
<td>1309</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>4</td>
<td>2342</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>3</td>
<td>2401</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>2</td>
<td>2066</td>
</tr>
</tbody>
</table>

**Figure 6.10** The effect of the learning rates on the number of iterations using the GRBH algorithm with $H = 10, 20$.

The trained networks were tested using four different sets of images, these are described in Section 6.4. Table (6.5) shows the performance of these networks using a set of noiseless images. Tables (6.6), (6.7), and (6.8) shows the performance of the networks using noisy images with noise levels of 50dB, 25dB and 12dB respectively.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>96.2%</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>94.9%</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>97.4%</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>94.9%</td>
</tr>
</tbody>
</table>

Table (6.5): The performance of the networks under noiseless images.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
<td>68%</td>
</tr>
<tr>
<td>20</td>
<td>27</td>
<td>65.4%</td>
</tr>
<tr>
<td>30</td>
<td>26</td>
<td>66.7%</td>
</tr>
<tr>
<td>40</td>
<td>26</td>
<td>66.7%</td>
</tr>
</tbody>
</table>

Table (6.6): The performance of the networks under noisy images with noise levels of 50dB.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>35.9%</td>
</tr>
<tr>
<td>20</td>
<td>48</td>
<td>38.5%</td>
</tr>
<tr>
<td>30</td>
<td>47</td>
<td>39.7%</td>
</tr>
<tr>
<td>40</td>
<td>46</td>
<td>41%</td>
</tr>
</tbody>
</table>

Table (6.7): The performance of the networks under noisy images with 25dB noise levels.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>77</td>
<td>1.3%</td>
</tr>
<tr>
<td>20</td>
<td>76</td>
<td>2.6%</td>
</tr>
<tr>
<td>30</td>
<td>77</td>
<td>1.3%</td>
</tr>
<tr>
<td>40</td>
<td>76</td>
<td>2.6%</td>
</tr>
</tbody>
</table>

**Table (6.8):** The performance of the networks under noisy images with 12dB noise levels.

It can be seen from the results that the GRBH algorithm outperformed the BP algorithm in all cases. The speedup obtained from the GRBH algorithm was greater than 90%. A network with 5 hidden nodes could not converge to the required solution. The performance of the networks under noiseless images were around 95% recognition. On the other hand, when noisy images were used to test the trained networks the performance degraded as the SNR was reduced.

**6.5.4 EXPERIMENTAL RESULTS USING CMs**

By using CMs three different sets of features were generated. The first set consisted of all the features starting from $C_{11}$ upto $C_{44}$ ($C_{01}$, $C_{10}$ and $C_{00}$ are constant for all normalized images and therefore are not used as features). This gives 13 features in total. The second set consisted of all features upto $C_{66}$ which gives 26 different features. The third set consisted of all features upto $C_{88}$, which gives 43 different features. Higher orders of CMs was not considered for the reasons given earlier. The number of input nodes used are 13, 26 and 43 for the first, second and third sets respectively.
According to Equation (6.44) the minimum number of hidden nodes needed is 5 for all cases. The experiments were repeated for each feature set with $H = 5, 10, 20, 30$ and $40$. The networks were trained using the GRBH algorithm as well as the BP algorithm.

1. The first set of features

The chosen values of $\alpha$ for different ranges are the same as in Tables (6.1) and Figure (6.8). Figure (6.11) shows the maximum possible values of $\alpha_3$ against different number of hidden nodes. $\alpha_2$ and $\alpha_1$ are related to $\alpha_3$ according to Equation (6.47).

![Graph showing the maximum values of $\alpha_3$ against the number of hidden nodes $H$.](image)

**Figure 6.11** The maximum values of $\alpha_3$ against the number of hidden nodes $H$.

Table (6.9) gives the number of iterations required for a network to converge using the GRBH algorithm. The training process was terminated when $e =$
5 \times 10^{-2}.

<table>
<thead>
<tr>
<th>Number of hidden nodes H</th>
<th>Learning rate $\alpha_1$</th>
<th>Learning rate $\alpha_2$</th>
<th>Learning rate $\alpha_3$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>800</td>
<td>80</td>
<td>8</td>
<td>1114</td>
</tr>
<tr>
<td>10</td>
<td>800</td>
<td>80</td>
<td>8</td>
<td>183</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>40</td>
<td>4</td>
<td>161</td>
</tr>
<tr>
<td>30</td>
<td>400</td>
<td>40</td>
<td>4</td>
<td>148</td>
</tr>
<tr>
<td>40</td>
<td>300</td>
<td>30</td>
<td>3</td>
<td>167</td>
</tr>
</tbody>
</table>

**Table (6.9):** The number of iterations required for a network to converge to $e = 5 \times 10^{-2}$ using the GRBH algorithm.

Table (6.10) gives the number of iterations required for a network to converge using the BP algorithm. The training process was terminated when $e = 1 \times 10^{-1}$.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>9406</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>2439</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>2157</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>2358</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>2397</td>
</tr>
</tbody>
</table>

**Table (6.10):** The number of iterations required for a network to converge to $e = 1 \times 10^{-1}$ using the BP algorithm.

Table (6.11) shows the effect of the learning values on the number of iterations. The training process was terminated when $e = 5 \times 10^{-2}$. Figure (6.12) shows diagrammatically the effect of the $\alpha$ values on convergence.
These experiments were performed on networks with $H = 5$, 10 and 20.

<table>
<thead>
<tr>
<th>Learning rate</th>
<th>Number of iterations with $H = 5$</th>
<th>$H = 10$</th>
<th>$H = 20$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha_1$</td>
<td>$\alpha_2$</td>
<td>$\alpha_3$</td>
<td></td>
</tr>
<tr>
<td>800</td>
<td>80</td>
<td>8</td>
<td>1114</td>
</tr>
<tr>
<td>600</td>
<td>60</td>
<td>6</td>
<td>863</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>4</td>
<td>s</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>3</td>
<td>2809</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>2</td>
<td>3969</td>
</tr>
</tbody>
</table>

**Table (6.11):** The effect of $\alpha$ values on the convergence of the GRBH algorithm for networks with $H = 5$, 10 and 20.

**Figure 6.12** The effect of the learning rates on the number of iterations using the GRBH algorithm with $H = 5, 10, 20$.

The trained networks were tested using the four different sets of images which were described in Section 6.4. Tables (6.12), (6.13), (6.14) and (6.15) shows the performance of the networks using noiseless images, noisy images with 50dB, 25dB and 12dB noise levels respectively.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table (6.12):** The performance of networks under noiseless images.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>96.2%</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table (6.13):** The performance of networks under noisy images with noise levels of 50dB.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>38</td>
<td>51.3%</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>87.2%</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>93.6%</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>97.4%</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>97.4%</td>
</tr>
</tbody>
</table>

**Table (6.14):** The performance of networks under noisy images with noise levels of 25dB.
Table (6.15): The performance of networks under noisy images with noise levels of 12dB.

2. The second set of features

The chosen values of $\alpha$ for different ranges are the same as in Table (6.1) and Figure (6.8). Figure (6.13) shows the maximum possible values of $\alpha_3$ against different numbers of hidden nodes, $\alpha_2$ and $\alpha_1$ are related to $\alpha_3$ according to Equation (6.47).

![Figure 6.13](image)

**Figure 6.13** The maximum values of $\alpha_3$ against the number of hidden nodes $H$.

Table (6.16) gives the number of iterations required for a network to converge using the GRBH algorithm. The training process terminated when $\varepsilon = 5 \times 10^{-2}$

195.
Table (6.16): The number of iterations required for a network to converge to $\varepsilon = 5 \times 10^{-2}$ using the GRBH algorithm.

Table (6.17) gives the number of iterations required for a network to converge using the BP algorithm. The training process was terminated when $\varepsilon = 1 \times 10^{-1}$.

Table (6.17): The number of iterations required for a network to converge to $\varepsilon = 10^{-1}$ using the BP algorithm.

Table (6.18) shows the effect of the learning rates on the number of iterations. The training process was terminated when $\varepsilon = 5 \times 10^{-2}$. Figure (6.14) shows diagrammatically the effect of $\alpha$ values on convergence. These experiments were performed on networks with $H = 5, 10$ and 20.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1200 120 12</td>
<td>484</td>
</tr>
<tr>
<td>10</td>
<td>800 80 8</td>
<td>167</td>
</tr>
<tr>
<td>20</td>
<td>400 40 4</td>
<td>140</td>
</tr>
<tr>
<td>30</td>
<td>300 30 3</td>
<td>137</td>
</tr>
<tr>
<td>40</td>
<td>200 20 2</td>
<td>177</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td>6810</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>2246</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>1875</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>1908</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>2318</td>
</tr>
<tr>
<td>Learning rate</td>
<td>Number of iterations with</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------</td>
<td></td>
</tr>
<tr>
<td>$\alpha_1$</td>
<td>$\alpha_2$</td>
<td>$\alpha_3$</td>
</tr>
<tr>
<td>1200</td>
<td>120</td>
<td>12</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>800</td>
<td>80</td>
<td>8</td>
</tr>
<tr>
<td>600</td>
<td>60</td>
<td>6</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>2</td>
</tr>
</tbody>
</table>

Table (6.18): The effect of $\alpha$ values on the convergence of the GRBH algorithm for networks with $H = 5$, 10 and 20.

Figure 6.14: The effect of the learning rates on the number of iterations using the GRBH algorithm with $H = 5, 10, 20$.

The trained networks were tested using the four different sets of images which were described in Section 6.4. Tables (6.19), (6.20), (6.21) and (6.22) shows the performance of the networks using noiseless images, noisy images with 50dB, 25dB and 12dB noise levels respectively.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table (6.19):** The performance of networks under noiseless images.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>91%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table (6.20):** The performance of networks under noisy images with noise levels of 50dB.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>37</td>
<td>52.6%</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>89.7%</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>96.2%</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>97.4%</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>94.9%</td>
</tr>
</tbody>
</table>

**Table (6.21):** The performance of networks under noisy images with noise levels of 25dB.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>45</td>
<td>42.3%</td>
</tr>
<tr>
<td>10</td>
<td>47</td>
<td>39.7%</td>
</tr>
<tr>
<td>20</td>
<td>34</td>
<td>56.4%</td>
</tr>
<tr>
<td>30</td>
<td>32</td>
<td>59%</td>
</tr>
<tr>
<td>40</td>
<td>30</td>
<td>61.5%</td>
</tr>
</tbody>
</table>

Table (6.22): The performance of networks under noisy images with noise levels of 12dB.

3. **The third set of features**

The chosen values of $\alpha$ for different ranges are the same as in Table (6.1) and Figure (6.8). Figure (6.15) shows the maximum possible values of $\alpha_3$ against different numbers of hidden nodes, $\alpha_2$ and $\alpha_1$ are related to $\alpha_3$ according to Equation (6.47).

![Figure 6.15 The maximum values of $\alpha_3$ against the number of hidden nodes $H$.](image)

Table (6.23) gives the number of iterations required for a network to converge using the GRBH algorithm, the training process terminated when $e = 5 \times$
$10^{-2}$. Table (6.24) gives the number of iterations required for a network to converge using the BP algorithm. The training process was terminated when $e = 1 \times 10^{-1}$.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha_1$</th>
<th>$\alpha_2$</th>
<th>$\alpha_3$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>400</td>
<td>40</td>
<td>4</td>
<td>1541</td>
</tr>
<tr>
<td>10</td>
<td>800</td>
<td>80</td>
<td>8</td>
<td>174</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>40</td>
<td>4</td>
<td>173</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>30</td>
<td>3</td>
<td>138</td>
</tr>
<tr>
<td>40</td>
<td>300</td>
<td>30</td>
<td>3</td>
<td>137</td>
</tr>
</tbody>
</table>

**Table (6.23):** The number of iterations required for a network to converge to $e = 5 \times 10^{-2}$ using the GRBH algorithm

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$-$</td>
<td>$s$</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>1939</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>1547</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>1664</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>2775</td>
</tr>
</tbody>
</table>

**Table (6.24):** The number of iterations required for a network to converge to $e = 1 \times 10^{-1}$ using the BP algorithm

Table (6.25) shows the effect of the learning rates on the number of iterations. The training process was terminated when $e = 5 \times 10^{-2}$. Figure (6.16) shows diagrammatically the effect of $\alpha$ values on convergence. The experiments were performed on networks with $H = 5$, 10 and 20.
<table>
<thead>
<tr>
<th>Learning rate</th>
<th>Number of iterations with</th>
</tr>
</thead>
<tbody>
<tr>
<td>α₁</td>
<td>α₂</td>
</tr>
<tr>
<td>800</td>
<td>80</td>
</tr>
<tr>
<td>600</td>
<td>60</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
</tr>
</tbody>
</table>

Table (6.25): The effect of α values on the convergence of the GRBH algorithm for networks with H = 5, 10 and 20.

![Graph](image)

Figure 6.16 The effect of the learning rates on the number of iterations using the GRBH algorithm with H = 5, 10, 20.

The trained networks were tested using the four different sets of images which were described in Section 6.4. Tables (6.26), (6.27), (6.28) and (6.29) show the performance of the networks using noiseless images, noisy images with 50dB, 25dB and 12dB noise levels respectively.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table (6.26): The performance of the networks under noiseless images.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>96.2%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table (6.27): The performance of the networks under noisy images with noise levels of 50dB.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>50</td>
<td>35.9%</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>92.3%</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>96.2%</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>96.2%</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>93.6%</td>
</tr>
</tbody>
</table>

Table (6.28): The performance of the networks under noisy images with noise levels of 25dB.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>66</td>
<td>15.4%</td>
</tr>
<tr>
<td>10</td>
<td>31</td>
<td>60.3%</td>
</tr>
<tr>
<td>20</td>
<td>28</td>
<td>64.1%</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>60.3%</td>
</tr>
<tr>
<td>40</td>
<td>26</td>
<td>66.7%</td>
</tr>
</tbody>
</table>

**Table (6.29):** The performance of the networks under noisy images with noise levels of 12dB.

It can be seen from the results that the GRBH algorithm outperforms the BP algorithm in all cases. The speedup obtained from the GRBH algorithm is at least 90%. In some GRBH cases it was possible to use slightly larger values of $\alpha_3$ compared with the BP algorithm.

In general, when the same values of $\alpha$ are used, a network with a larger number of hidden nodes converges faster than a network with a smaller number of hidden nodes. However this can be changed by using larger learning rates for networks with a smaller number of hidden nodes. The results show that the networks with a different number of hidden nodes required a similar number of iterations when the values of $\alpha$ increased with reducing the number of hidden nodes. The results also show that the networks with minimum number of hidden nodes ($H = 5$) was not always possible to use larger learning rates, and they also required much larger numbers of iterations to reach the required solution.

It was also shown that the performance of the networks improved with increasing the number of hidden nodes, in fact when $H = 20$ the performance of the network improves considerably. This indicates that the minimum number of hidden nodes does not give the best results.
6.5.5 EXPERIMENTAL RESULTS USING ZMs

By using the ZMs four different sets of features were generated. The first set consisted of all the features starting from $A_{20}$ up to $A_{44}$ ($A_{00}$, $A_{10}$, $A_{21}$ are constant for all scaled images), this gives 7 features. The second set consisted of all features up to $A_{66}$ this gives 14 different features. The third set consisted of all features up to $A_{88}$ and this set has 23 different features. The forth set consisted of all features of order 10 (up to $A_{10,10}$), this set has 34 different features. Higher orders of ZMs are not used for the reasons given earlier. The number of the input nodes used are 7, 14, 23 and 34 for the first, second, third and fourth set respectively. According to Equation (6.44) the minimum number of hidden nodes needed is 5 for all cases. The experiments were repeated for each set with $H = 5, 10, 20, 30$ and 40. The networks were trained using the BP as well as the GRBH algorithm.

1. The First Set of Features

The chosen values of $\alpha$ for different ranges are the same as in Tables (6.1) and Figure (6.8). Figure (6.17) shows the maximum possible values of $\alpha_3$ against different numbers of hidden nodes. $\alpha_2$ and $\alpha_1$ are related to $\alpha_3$ according to Equation (6.47).

![Graph showing the maximum values of $\alpha_3$ against the number of hidden nodes $H$.](image)

**Figure 6.17** The maximum values of $\alpha_3$ against the number of hidden nodes $H$. 204
Table (6.30) gives the number of iterations required for a network to converge using the GRBH algorithm. The training process was terminated when $e = 5 \times 10^{-2}$.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\alpha_1$</td>
<td>$\alpha_2$</td>
</tr>
<tr>
<td>5</td>
<td>1600</td>
<td>160</td>
</tr>
<tr>
<td>10</td>
<td>800</td>
<td>80</td>
</tr>
<tr>
<td>20</td>
<td>600</td>
<td>60</td>
</tr>
<tr>
<td>30</td>
<td>400</td>
<td>40</td>
</tr>
<tr>
<td>40</td>
<td>300</td>
<td>30</td>
</tr>
</tbody>
</table>

Table (6.30): The number of iterations required for the networks to converge to $e = 5 \times 10^{-2}$ using the GRBH algorithm.

Table (6.31) gives the number of iterations required for a network to converge using the BP algorithm. The training process was terminated when $e = 1 \times 10^{-1}$.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>7921</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>1975</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>1642</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td>1926</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>2182</td>
</tr>
</tbody>
</table>

Table (6.31): The number of iterations required for the networks to converge to $e = 1 \times 10^{-1}$ using the BP algorithm.

Table (6.32) shows the effect of the learning rates on the number of iterations. The training process was terminated when $e = 5 \times 10^{-2}$. 

205
<table>
<thead>
<tr>
<th>Learning rate</th>
<th>Number of iterations with</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha_1 )</td>
<td>( \alpha_2 )</td>
</tr>
<tr>
<td>1600</td>
<td>160</td>
</tr>
<tr>
<td>1200</td>
<td>120</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
</tr>
<tr>
<td>800</td>
<td>80</td>
</tr>
<tr>
<td>600</td>
<td>60</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
</tr>
</tbody>
</table>

Table (6.32): The effect of \( \alpha \) values on the convergence of the GRBH algorithm for the networks with \( H = 5, 10 \) and 20.

Figure (6.18) shows diagrammatically the effect of \( \alpha \) values on the convergence rate. These experiments were performed on networks with \( H = 5, 10 \) and 20.

Figure 6.18 The effect of the learning rates on the number of iterations using the GRBH algorithm with \( H = 5, 10, 20 \).

The trained networks were tested using the four different sets of images which were described in Section 6.4. Tables (6.33), (6.34), (6.35) and (6.36) shows
the performance of the networks using noiseless images, noisy images with 50dB, 25dB and 12dB noise levels respectively.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table (6.33):** The performance of the networks under noiseless images.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>30</td>
<td>61.5%</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>94.9%</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>96.2%</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>97.4%</td>
</tr>
</tbody>
</table>

**Table (6.34):** The performance of the networks under noisy images with noise levels 50dB.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>30</td>
<td>61.5%</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>94.9%</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>96.2%</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>97.4%</td>
</tr>
</tbody>
</table>

**Table (6.35):** The performance of the networks under noisy images with noise levels 25dB.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>59</td>
<td>24.4%</td>
</tr>
<tr>
<td>10</td>
<td>36</td>
<td>53.9%</td>
</tr>
<tr>
<td>20</td>
<td>29</td>
<td>62.8%</td>
</tr>
<tr>
<td>30</td>
<td>32</td>
<td>59%</td>
</tr>
<tr>
<td>40</td>
<td>31</td>
<td>60.3%</td>
</tr>
</tbody>
</table>

**Table (6.36):** The performance of the networks under noisy images with noise levels 12dB.

2. **The Second Set of Features**

The chosen values of $\alpha$ for different ranges are the same as in Table (6.1) and Figure (6.8). Figure (6.19) shows the maximum possible values of $\alpha_3$ against different numbers of hidden nodes. $\alpha_2$ and $\alpha_1$ are related to $\alpha_3$ according to Equation (6.47).

![Figure 6.19](image)

**Figure 6.19** The maximum values of $\alpha_3$ against the number of hidden nodes $H$.

Table (6.37) gives the number of iterations required for a network to converge using the GRBH algorithm. The training process was terminated when $\epsilon = $
$5 \times 10^{-2}$. Table (6.38) gives the number of iterations required for a network to converge using the BP algorithm, the training process was terminated when $e = 1 \times 10^{-1}$.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha_1$</th>
<th>$\alpha_2$</th>
<th>$\alpha_3$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1000</td>
<td>100</td>
<td>10</td>
<td>396</td>
</tr>
<tr>
<td>10</td>
<td>800</td>
<td>80</td>
<td>8</td>
<td>136</td>
</tr>
<tr>
<td>20</td>
<td>600</td>
<td>60</td>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>30</td>
<td>3</td>
<td>131</td>
</tr>
<tr>
<td>40</td>
<td>300</td>
<td>30</td>
<td>3</td>
<td>129</td>
</tr>
</tbody>
</table>

**Table (6.37):** The number of iterations required for a network to converge to $e = 5 \times 10^{-2}$ using the GRBH algorithm.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Learning rate $\alpha$</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>7267</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>1742</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>1569</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>1928</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>2028</td>
</tr>
</tbody>
</table>

**Table (6.38):** The number of iterations required for a network to converge to $e = 1 \times 10^{-1}$ using the BP algorithm.

Table (6.39) shows the effect of the learning rates on the number of iterations, the training process was terminated when $e = 5 \times 10^{-2}$. Figure (6.20) shows diagrammatically the effect of $\alpha$ values on the convergence rate. The experiments were performed on networks with $H = 5$, 10 and 20.
<table>
<thead>
<tr>
<th>Learning rate</th>
<th>Number of iterations with</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha_1$</td>
<td>$\alpha_2$</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
</tr>
<tr>
<td>800</td>
<td>80</td>
</tr>
<tr>
<td>600</td>
<td>60</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
</tr>
</tbody>
</table>

Table (6.39): The effect of $\alpha$ values on the convergence of the GRBH algorithm for the networks with $H = 5, 10$ and $20$.

Figure 6.20 The effect of the learning rates on the number of iterations using the GRBH algorithm with $H = 5, 10, 20$.

The trained networks were tested using the four different sets of images which were described in Section 6.4. Tables (6.40), (6.41), (6.42) and (6.43) shows the performance of the networks using noiseless images, noisy images with 50dB, 25dB and 12dB noise levels respectively.

210
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table (6.40): The performance of the networks under noiseless images.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>94.9%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table (6.41): The performance of the networks under noisy images with noise levels 50dB.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>30</td>
<td>61.5%</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>97.4%</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>97.4%</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>97.4%</td>
</tr>
</tbody>
</table>

Table (6.42): The performance of the networks under noisy images with noise levels 25dB.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>53</td>
<td>32.1%</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
<td>59%</td>
</tr>
<tr>
<td>20</td>
<td>22</td>
<td>71.8%</td>
</tr>
<tr>
<td>30</td>
<td>24</td>
<td>69.2%</td>
</tr>
<tr>
<td>40</td>
<td>25</td>
<td>68%</td>
</tr>
</tbody>
</table>

Table (6.43): The performance of the networks under noisy images with noise levels 12dB.

3. The Third Set of Features

The chosen values of $\alpha$ for different ranges are the same as in Table (6.1) and Figure (6.5). Figure (6.21) shows the maximum possible values of $\alpha_3$ against different numbers of hidden nodes. $\alpha_2$ and $\alpha_1$ are related to $\alpha_3$ according to Equation (6.47).

![Figure 6.21](image)

Figure 6.21 The maximum values of $\alpha_3$ against the number of hidden nodes $H$.

Table (6.44) gives the number of iterations required for a network to converge using the GRBH algorithm. The training process was terminated when $e = 5 \times 10^{-2}$.
Table (6.44): The number of iterations required for a network to converge to $e = 5 \times 10^{-2}$ using the GRBH algorithm.

Table (6.45) gives the number of iterations required for a network to converge using the BP algorithm. The training process was terminated when $e = 1 \times 10^{-1}$.

Table (6.46): The number of iterations required for a network to converge to $e = 1 \times 10^{-1}$ using the BP algorithm.

Table (6.46) shows the effect of the learning rates on the number of iterations. The training process was terminated when $e = 5 \times 10^{-2}$.
<table>
<thead>
<tr>
<th>Learning rate $\alpha_1$</th>
<th>$\alpha_2$</th>
<th>$\alpha_3$</th>
<th>$H = 5$</th>
<th>$H = 10$</th>
<th>$H = 20$</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>80</td>
<td>8</td>
<td></td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>60</td>
<td>6</td>
<td>684</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>4</td>
<td>1199</td>
<td>259</td>
<td>118</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>3</td>
<td>1569</td>
<td>354</td>
<td>144</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>2</td>
<td>9391</td>
<td>540</td>
<td>213</td>
</tr>
</tbody>
</table>

**Table (6.46):** The effect of $\alpha$ values on the convergence of the GRBH algorithm for the networks with $H = 5$, 10 and 20.

Figure (6.22) shows diagrammatically the effect of $\alpha$ values on the convergence rates. The experiments were performed on networks with $H = 5$, 10 and 20.

**Figure 6.22** The effect of the learning rates on the number of iterations using the GRBH algorithm with $H = 5, 10, 20$. 

214
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table (6.47): The performance of the networks under noiseless images.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table (6.48): The performance of the networks under noisy images with noise levels 50dB.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>19</td>
<td>75.6%</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>97.4%</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>97.4%</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>96.2%</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>94.9%</td>
</tr>
</tbody>
</table>

Table (6.49): The performance of the networks under noisy images with noise levels 25dB.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>32</td>
<td>59.0%</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>48.8%</td>
</tr>
<tr>
<td>20</td>
<td>26</td>
<td>66.7%</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>60.3%</td>
</tr>
<tr>
<td>40</td>
<td>25</td>
<td>68%</td>
</tr>
</tbody>
</table>

**Table (6.50):** The performance of the networks under noisy images with noise levels 12dB.

4. **The Fourth Set of Features**

The chosen values of $\alpha$ for the different ranges are the same as in Table (6.1) and Figure (6.8). Figure (6.23) shows the maximum possible values of $\alpha_3$ against different numbers of hidden nodes. $\alpha_2$ and $\alpha_1$ are related to $\alpha_3$ according to Equation (6.47).

![Figure 6.23](image)

**Figure 6.23** The maximum values of $\alpha_3$ against the number of hidden nodes $H$.

Table (6.51) gives the number of iterations required for a network to converge using the GRBH algorithm. The training process was terminated when $e =$
$5 \times 10^{-2}$. Table (6.52) gives the number of iterations required for a network to converge using the BP algorithm. The training process was terminated when $e = 1 \times 10^{-1}$.

<table>
<thead>
<tr>
<th>Number of hidden nodes</th>
<th>Learning rate</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>$\alpha_1$</td>
<td>$\alpha_2$</td>
</tr>
<tr>
<td>5</td>
<td>1200</td>
<td>120</td>
</tr>
<tr>
<td>10</td>
<td>800</td>
<td>80</td>
</tr>
<tr>
<td>20</td>
<td>600</td>
<td>60</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>40</td>
<td>300</td>
<td>30</td>
</tr>
</tbody>
</table>

**Table (6.51):** The number of iterations required for a network to converge to $e = 5 \times 10^{-2}$ using the GRBH algorithm.

<table>
<thead>
<tr>
<th>Number of hidden nodes</th>
<th>Learning rate</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>$\alpha$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>4435</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>2393</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>1453</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>1552</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>1849</td>
</tr>
</tbody>
</table>

**Table (6.52):** The number of iterations required for a network to converge to $e = 1 \times 10^{-1}$ using the BP algorithm.

Table (6.53) shows the effect of the learning rates on the number of iterations. The training process was terminated when $e = 5 \times 10^{-2}$. Figure (6.24) show diagrammatically the effect of $\alpha$ values on convergence. The experiments were performed on networks with $H = 5, 10$ and 20.
Learning rate | Number of iterations with $H = 5, 10, 20$
--- | --- | --- | --- | ---
$\alpha_1$ | $\alpha_2$ | $\alpha_3$ | $H = 5$ | $H = 10$ | $H = 20$
1200 | 120 | 12 | 533 | - | -
1000 | 100 | 5 | $s$ | - | -
800 | 80 | 8 | 538 | 181 | -
600 | 60 | 6 | 1056 | 160 | 383*
400 | 40 | 4 | 1280 | 238 | 118
300 | 30 | 3 | 1550 | 302 | 150
200 | 20 | 2 | 3502 | 490 | 214

Table (6.53): The effect of $\alpha$ values on the convergence of the GRBH algorithm for the networks with $H = 5, 10$ and 20.

Figure 6.24 The effect of the learning rates on the number of iterations using the GRBH algorithm with $H = 5, 10, 20$.

The trained networks were tested using the four different sets of images which were described in Section 6.4. Tables (6.54), (6.55), (6.56) and (6.57) shows the performance of the networks using noiseless images, noisy images with 50dB, 25dB and 12dB noise levels respectively.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table (6.54):** The performance of the networks under noiseless images.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>96.2%</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table (6.55):** The performance of the networks under noisy images with noise levels 50dB.

<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>36</td>
<td>53.9%</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>97.4%</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>98.7%</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>98.7%</td>
</tr>
</tbody>
</table>

**Table (6.56):** The performance of the networks under noisy images with noise levels 25dB.
<table>
<thead>
<tr>
<th>Number of hidden nodes $H$</th>
<th>Number of errors</th>
<th>Percentage of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>47</td>
<td>39.7%</td>
</tr>
<tr>
<td>10</td>
<td>19</td>
<td>75.6%</td>
</tr>
<tr>
<td>20</td>
<td>22</td>
<td>71.8%</td>
</tr>
<tr>
<td>30</td>
<td>19</td>
<td>75.6%</td>
</tr>
<tr>
<td>40</td>
<td>23</td>
<td>70.5%</td>
</tr>
</tbody>
</table>

**Table (6.57):** The performance of the networks under noisy images with noise levels 12dB.

The results show that the GRBH algorithm improves the convergence rate considerably compared with the BP algorithm. By using larger learning rates for networks with smaller hidden nodes, the convergence rate increases. The maximum possible values of $\alpha_3$ is proportional to the number of hidden nodes.

### 6.6 Conclusion

From the results it can be seen that the GRBH algorithm could improve the convergence rate considerably. It was also shown that there is a link between the smallest learning rate $\alpha_3$ and the fan-in of a node. Larger values of $\alpha_3$ can be used as the number of connections to a node is reduced.

All the trained networks performed well when noiseless images were used for testing. On the other hand, the networks that were trained using GMs performed badly when some noise was added to the testing images. This is due to the fact that the GM values are correlated. The performance of the networks that were trained using ZMs and CMs under noisy images improved as the number of hidden nodes increased.
CHAPTER VII

Conclusion
The Backpropagation learning algorithm was implemented on the parallel Balance machine. Two different approaches of the partitioning algorithm were used. For the On-line parallel simulator the network was partitioned vertically into slices with each processor possessing one part of the network. On the other hand, the Batch parallel simulator used partitioning on the vector set with each processor having the entire network in its possession and only one part of the vector set. The speedup of the parallel simulators approached the ideal values when the network contained enough nodes. In general the parallel Batch simulator performed slightly better than the On-line simulator when the same size of network was used. The performance of the Batch simulator was also improved by increasing the size of the training set. On the other hand, there was very slight improvement with the On-line simulator when the training set size increased. As the network size was increased the training time increased, hence there is a larger demand for parallel machines.

The simulators are designed to be expanded for further improvement in the future. This can be done by adding new moduli. New node functions as well as error functions can be added to the simulators. Also, different learning algorithms can be added to the simulators and compare them with the existing ones. By using a graphical terminal and windows it is possible to display the network architecture, connection weights and the state of the nodes. The development of the connection weights during training using graphical display can give an insight and understanding of these changes. A graphical plot of the error surface can be a great help too.

For further work, the implementation of ANNs on different parallel machines such as the Transputer is needed for developing new machines based on
ANNs. Also it is important to try to implement ANNs on Optical computers as they can pass large amounts of information.

The BP algorithm is very slow and it gets slower as the task gets larger and more complex. Therefore improvement of the learning algorithm is important to study large and complex tasks. A new Heuristic learning algorithm for the BP was introduced. This algorithm is called the GRBH algorithm, whereby the learning rate $\alpha$ is different for different weights and its value is dependent on the range of the modulus of the error gradient for that particular connection. The gradient is divided into a number of groups and each group is associated with a fixed value of $\alpha$. It was shown that the best number of groups are 3 and 6 for the On-line simulator and 3,5 and 6 for the Batch simulator. It was shown that the GRBH algorithm out performed the BP algorithm. This algorithm was tested on a number of tasks where a speedup of around 80% was achieved using the On-line method and a speedup of 99% was achieved using the Batch simulator.

The BP algorithm was also accelerated using deterministic gradient methods, the Steepest Descent and Conjugate Gradient methods. These algorithms require a line search for calculating the learning rate $\alpha$. Therefore more calculations are needed per cycle compared with the GRBH as well as the usual BP algorithm. It was shown that the SD method required less iterations than the usual BP algorithm. It was also shown that the CG method improves the convergence rate even further. For all cases the GRBH algorithm appeared to do better than the other algorithms and without extra calculations.

A number of additional experiments is needed to see if further improvement in accelerating the GRBH algorithm is possible. One approach is to try
a number of different error functions and test if any of them can have any improvement on the convergence rate.

The GRBH algorithm was tested on the character recognition task by using moment invariant features. Three different moments were used, Geometric, Complex and the Zernike moment. Also a different number of features were used for training the networks. It was shown that in all cases the GRBH algorithm performed much better than the usual BP algorithm where again the speedup was around 99%. Also it was shown that the Zernike as well as the Complex Moments performed well under noise. It was also shown that the values of the learning rate was dependent on the number of the nodes in the network. As the number of the connections to a node increases then a smaller value of $\alpha$ should be used.

For further work, it is important to try different preprocessing techniques and compare their results. Even though the GRBH algorithm proved to be much better than any of the existing acceleration techniques used for the BP algorithm, other tasks should be tested using this algorithm to evaluate its performance.
References


[60] Parker, D., [1982], *Learning Logic*, Invention report, S81-64, File 1, Office of Technology Licensing, Stanford University.


[77] Stone, H.S., [1980], An Introduction to Computer Architecture, SRA.


[90] Zeidenberg, M., [1990], Neural Networks in Artificial Intelligence, Ellis Harwood.

APPENDIX A

Backpropagation Parallel Simulation User Manual
This manual is an introduction to how to use the Parallel Backpropagation Simulator on the Balance machine. The simulator can be used to construct, train and test a network. It is written in Pascal, the program is divided into a number of units, each unit executes one part of the simulator. These units are compiled and linked together. The simulator is menu driven and therefore very easy to use.

**How to compile the simulator?**

To compile the program type the following:

```
```

- `-u` indicates that there are a number of units that are required to be linked together
- `-e` allows the compiler to link Pascal program with C program together.
- `-mp` allows the compiler to use the parallel library.
- `-o` allows you to choose any output file name, so the name of the output file should follow `-o`.

The following is a brief description of all the Pascal files.

- `npp1.p`: All the global variables are declared in this file, also the parallel routines are declared in this file. Some of the common functions such as the random and logistic functions are placed in this file.
- `prop2.p`: This file contains the code for setting the parameters.
- `prop4.p`: This file contains the code for setting the connection weights.
- `prop7.p`: This file contains the code for constructing the linked list as described in Chapter 4.
prop5.p: This file contains the code for setting up the networks, as well as loading and storing the network.

prop6.p: This file contains the code for showing the network.

prop9.p: This file contains the code for setting up the vector set including loading and storing.

prop12.p: This file contains the code for testing the network.

nnp.p: This file contains the code for training the network.

prop10.p: This file contains the code for the main menu.

bp.p: This file calls the main menu.

clock.c: This file contains a C code for calculating execution time.

How to run the Simulator?

Type the execute file.

neural.out

After you have typed the execute file, the following main menu appears:

Main menu

1: Input parameters
2: Input network
3: Input vector set
4: Learn network
5: Show network
6: Test network
7: Test number of errors
8: Enter frame
9: Test a number of networks
10: Calculate moments
0: Exit

Option

You have to choose an item between 0-10. The options 1-3 should be chosen before the other option as we cannot train or test the network without having the network or the vector set. So type option 1 to input the parameters.

The following menu appears:

network parameters
1: input the number of learning cycles
2: input the number of ranges
3: input the learning factors
4: input the gradient range of each group
5: input the value of alpha
6: input the seed value
7: show the parameter values
0: exit

Option 1, allows you to input the number of cycles needed to train the network.
Option 2, input the number of groups of the gradient. It can be one group, hence the usual BP algorithm. If the number of groups is larger than one then the GRBH algorithm is used.

Option 3, you have to input the values of the learning rate $\alpha$.

Option 4, input the range of the gradient for each group.

Option 5, input the value for the momentum term $\beta$.

Option 6, when we construct a new network the weights are chosen randomly so by choosing different seed value we will have different weights for a network.

Option 7, displays the parameters of option 1, 2, 3, 4, 5 and 6.

Option 0, allows you to exit the network parameter menu and go back to the main menu.

Option 2 of the main menu allows you to construct, load or save a network. So if you type 2 the following menu appears:

network menu

1: input network from keyboard

2: load network

3: store network

4: repeat network structure

5: load layer network

6: store layer network

0: exit

$\dagger$ option
The difference between 2 and 5 is that in option 5 only feed-forward networks can be loaded but in option 2 all types of networks can be loaded. The same applies to option 3 and 6. Let us assume you want to train the XOR with one hidden node as shown in Figure 1a.

![XOR network diagram](image)

<table>
<thead>
<tr>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$a_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

(a) (b) XOR vector set.

Figure A.1 (a) The XOR network.
(b) XOR vector set.

Type 1. The program types the following:

- enter the total number of nodes
  (type) 4
- Is it a multilayer network (y/n)
  (type) y
- enter the number of layers
  (type) 3
- enter the number of nodes in layer 1
  (type) 2
- enter the number of nodes in layer 2
enter the number of nodes in layer 3

Is it only feedforward net? (y/n)

Is it only adjacent layers? (y/n)

multilayer connection menu

1: fully connected network
2: randomly connected network
3: pixel connections
0: exit

In the XOR case of Figure 1a the connections are fully connected, as all the possibilities of feedforward net connections are there, so you have to choose option 1. On the other hand if you want to have less connections you have to choose option 2. By choosing 2 you have to give the percentage of connectivity. Option 3 allows you to choose the number of connections per node.

After you have typed your option, type 0 to exit from the multilayer connection menu. The program types the following:

do you want new network connectivity? (y/n)

After you have typed your option, type 0 to exit from the multilayer connection menu. The program types the following:

do you want new network connectivity? (y/n)
The connection weights can be set to zero by choosing option 1. It can be set to random values by choosing option 2. Option 3 allows you to enter from the keyboard the connection weights for every connection. For the XOR problem type 2, then exit the menu by typing 0. The network menu appears. The network has now been constructed, you can save the network on a file and can be retrieved whenever it is needed. To save the network type 6, input the name of the file.

(type)  XOR.net

do you want to store cw

(type)  y

Option 3 can be used too but if the network is very large then option 6 should be used provided that the network is feedforward net. Option 4 is used to repeat the same network a number of times as required. For example, the network in Figure 1a can be repeated say 3 times as shown in Figure 2.
Type 0 to exit the network menu, and go back to the main menu. Before training the network you require to input the vector set. This can be done by choosing the option 3. So type 3, the following menu appears:

vector set menu

1: enter from the keyboard
2: load from a file
3: repeat the vector set
4: expand the vector set
5: correct vector pair
6: change to negative
7: add noise to the vector pair

The following applies only for the input vectors:

8: enter from the keyboard
9: load a file
10: expand vectors
11: calculate the geometrical moment

12: in-out

13: add two files

14: store vector pairs

0: exit

# option

To input the vector set from the keyboard type 1. The program types the following:

size of the vector set

You have to type the number of the vector pairs that are needed to train on the network. For the XOR problem the number of the vector pairs are 4 as shown in Figure 1b. So type 4,

enter vector pair 1

2 input values?

(type) 0 0

1 output value?

(type) 0

This is repeated for the 4 vector pairs, after you input the 4th vector pair the program prints the following:

store vector set

input the name of the file

(type) XOR.vec

input the value of n
n is a value that is multiplied by all the elements of the vector pair, it can be used if you need to change the range of the vector pairs, otherwise type 1. Once the vector set is stored, you can retrieve it by choosing option 2. Option 3 can be used to repeat the same number of pairs over a number of networks so if you want to repeat it for 3 networks then vector pair, say 2, would have 6 elements rather than 2 and 3 outputs rather than one so 01 01 01 is the input and 1 1 1 is the output. Option 4 is used to increase the size of the vector set, say you want to add two more vector pairs then type 4, the program prints the following:

input the number of the new vector pairs

You have to type the number of the vector pairs to be added to the vector set, then enter the input vector followed by its corresponding output vector. This process is repeated until all the new vector pairs are exhausted.

Option 5 allows you to correct any of the vector pairs in the vector set, let us assume that vector pair 2 in your example is incorrect so you can modify it by choosing option 5.

(input) 5

input the number of pairs to be corrected

(input) 1

input the vector pair number

(input) 2

the input vector is 242
11  do you want to change it? (y/n)
(type) y
enter the input vector
(type) 0 1
the output vector is
1  do you want to change it? (y/n)
(type) n
store vector set
input the name of the file
(type) XOR.vec
input the value of n
(type) 1

Option 6 changes the range of the vector pairs from 0 – 1 to -.5,.5. Option 7 adds noise to the input vectors, the signal to noise ratio and the level of noise can be chosen as needed. The S/N can be controlled by choosing the number of pixels in error. When you type 7 the following is printed:

input the number of pixels to be changed
(type) 50
input the value of the noise for a single pixel
(say) 1

Option 8, 9 and 10 are the same as options 1, 2 and 4 except that they deal with only input vectors and not output vectors.
Option 11 calculates the geometrical moments of an image. These moments are invariant to rotation and also made invariant to scale and position. Let us assume that we have a file called *char.vec* that consists of 78 input vectors, each input is $16 \times 16$ pixels representing the English character 3 examples per character. To find the moments of these characters type 11, then the following is printed:

```
input the size of the image
(type) 16
load vector set
input the name of the file
(type) char.vec
input the number of examples for each class
(type) 3
input the number of output nodes
(type) 26 {as we have 26 letters}
would you like to load stat values? (y/n)
(type) n
store input vector
input the name of the file
(type) char1.vec
```

Option 12 adds output vectors to the inputs.

Option 13 adds two files together

Option 14 stores the vector pairs
Type 0 to exit the vector set menu and go back to the main menu.

After you have loaded the network and the vector set and chosen the required parameters, then the learning process can start by typing 4. The following is printed:

input the number of processors

(type) 4

**** time = ****

The execution time is printed.

The connection weights, network connections and the nodes’ states can be examined by typing 5. The following menu appears:

examine network menu

1: show network connections

2: show network weights

3: show network states

4: show input vector

0: exit

$ option

Let us assume that 1 is chosen then the following table appears:

The network connections are:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
This table indicates that there are only 5 connections in the network.

Option 4 prints the input vectors, so if 4 is typed then the following is printed:

input the number of pixels to be printed per line

(type)  16

do you like the values to be integer? (y/n)

(type)  y

To exit the examine network menu, type 0 and you are back to the main menu.

Option 6 of the main menu tests the network by applying any example from the keyboard. This can be done as follows:

(type)  6

   enter the input values: they are 2 values

(type)  0 1

1

Option 7 tests the network by going through the vector set that is used for training or testing. So by typing 7 the following is printed:

vector pair 3

vector pair 4

- 
- 
- 
- 

number of the vectors in error is - 

the number of errors = - out of -
total error = -

the total square error = -

After going through all the vector pairs, it prints all the vectors in error, the number of vector pairs in error, the total number of elements in the vector set that are in error, the sum of the absolute error and also the sum of the square errors are printed.

Option 8 of the main menu allows you to test a network with one or two input images, this is mainly used for visual depth perception, see Evans and Sanossian [1990]. Type 8 and the following menu is printed:

frame menu

1: enter frame from the keyboard
2: load frame
3: store frame
4: show frame
5: show output
6: store output
0: exit

Option 1 of the frame menu allows you to enter the images from the keyboard. Let us assume that you want to test a network that has two sets of inputs, one representing the left hand side eye and the other representing the right hand side eye. Then you require to enter two images, let us assume that each frame (or image) consists of 4 by 4 then by typing 1 the following is printed

enter the frame size x, y
is it \textit{XOR} operation? (y/n)

\textit{y} \quad \{\text{for two images, n for one image}\}

enter frame one

0 1 1 0
0 0 0 1
1 0 1 0
0 1 0 1

enter frame two

1 1 1 1
0 1 0 1
0 0 0 0
1 1 0 0

This result will be printed according to the function of the network.

These input frames can be stored by choosing option 3 and they can be retrieved any time by choosing option 2. To display the input frames choose option 4, and to display the output result choose option 5. The output can be stored using option 6. To exit the frame menu type 0 and you will go back to the main menu.

Using the simulator it is possible to calculate the Complex and Zernike moments of an image, these can be done by choosing option 10 of the main menu. So type 10 and the following is printed:
moment invariants menu

1: scale the image
2: calculate Complex moment
3: add output vectors
4: calculate Zernike moment
0: exit

# option

After you input the vector set, the images should be scaled so that they will be invariant to position and size. So type 1, the following is printed:

input the value of betta

(type) 50

do you want to save the file? (y/n)

(type) n

The scaled images can be stored on a file if you want, betta is the number of pixels that are active per image. To calculate the Complex moment or Zernike moment choose 2 or 4 respectively and the following is printed:

input the number of processors

(type) 2

input the order of complex moment

(type) 10

time = –

input the number of examples for each class

249
(type) 3
input the number of output nodes

(type) 26
would you like to load the stat values? (y/n)

(type) n
would you like to save the stat? (y/n)

(type) n
store vector input
input the name of the file

(type) cm10.vec

This shows that after calculating the moments they can be stored. Option 3 can be used to add new outputs.
APPENDIX B

The Training Set of the English Letters
APPENDIX C

The Testing Set of the English Letters