Fast learning neural networks for classification

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/2134/25161

Publisher: © Tay Leng Phuan

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) licence. Full details of this licence are available at: https://creativecommons.org/licenses/by-nc-nd/4.0/

Please cite the published version.
FAST LEARNING ARTIFICIAL NEURAL NETWORKS FOR CLASSIFICATION.

By

Mr Tay Leng Phuan
B.Sc. (CSET) Magna Cum Laude, M. Eng. (EEE)

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

October, 1994

© by Tay Leng Phuan, 1994
To the Lord God Almighty who never fails,
from everlasting to everlasting, the great I AM ......

In memory of my Father Mr Tay Soo Hong
and
To my mother Mdm Teo Siew Eng,
whom I will always love, honour and respect.

Last but not least, to my beloved wife, Julie Ho Shu King,
who has suffered the days with me, in patience, love and hope.

Praise be to God .......
ACKNOWLEDGEMENTS

I would like to thank God for the opportunity and provisions of this Ph.D. candidacy at the Loughborough University of Technology. For me to come this far in academia is both a privilege and a miracle.

I would like to thank my beloved mother, Mdm Teo Siew Eng, for the financial support throughout the two years of my candidacy.

I would like to thank my beloved wife, Julie Ho Shu King, whom I sincerely love and cherish. Thank you for standing by my side in full support through all the suffering.

I would like to thank the support staff, who helped in the facilitation of the daily research. To Dr. Yousif, for providing the technical support, and to Judith, for her administrative role.

To my Supervisor, Professor C.F. Cowan, I would like to thank you for your support and assistance as a supervisor for my Ph.D. programme.

Last, but definitely not the least, I would like to thank my Director of Research, Professor D.J. Evans, for all the time and energy spent in the academic sparring sessions, which helped annihilate illogical hypotheses and solidify logical concepts. I have yet to find another man of comparable academic dedication and care. He bore the difficult role of providing possible directions, yet did not restrain, the obstinate student from heading off in another way. Although these sessions generated much heat and friction, the presence of his genuine care and concern always strengthened the relationship.
ABSTRACT

Neural network applications can generally be divided into two categories. The first involves function approximation, where the neural network is trained to perform intelligent interpolation and curve fitting from the training data. The second category involves classification, where specific exemplar classes are used to train the neural network. This thesis directs its investigations towards the latter, i.e. classification.

Most existing neural network models are developments that arise directly from human cognition research. It is felt that while neural network research should head towards the development of models that resemble the cognitive system of the brain, researchers should not abandon the search for useful task oriented neural networks. These may not possess the intricacies of human cognition, but are efficient in solving industrial classification tasks.

It is the objective of this thesis to develop a neural network that is fast learning, able to generalise and achieve good capacity to discern different patterns even though some patterns may be similar in structure. This eventual neural network will be used in the pattern classification environment.

The first model developed, was the result of studying and modifying the basic ART 1 model. The "Fast Learning Artificial Neural Network I" (FLANN I) maintains good generalisation properties and is progressive in learning. Although this neural network achieves fast learning speeds of one epoch, it was limited only to binary inputs and was unable to operate on continuous values. This posed a real problem because industrial applications usually require the manipulation of continuous values.

The second model, FLANN II, was designed based on the principles of FLANN I. It was built on the nearest neighbour recall principle, which allowed the network to operate on continuous values. Experiments were conducted on the two models designed and the results were favourable. FLANN II was able to learn the points in a single epoch and obtain exceptional accuracy. This is a significant improvement to other researcher's results.

A further study was conducted on the FLANN models in the parallel processing environment. The parallel investigations led to the development of a new paradigm; Parallel Distributed Neural Networks (PDNNs), which allows several neural networks to operate concurrently to solve a single classification problem. This paradigm is powerful because it is able to reduce the overall memory requirements for some classification problems.
CONTENTS

ACKNOWLEDGEMENTS ................................................................. i

ABSTRACT ...................................................................................... ii

CHAPTER 1 INTRODUCTION .............................................................. 1

1.1 The Biological Brain .................................................................. 1

1.2 The Artificial Neural Network ................................................. 5

1.3 Practical Applications of Neural Networks ......................... 6

1.3.1 Pattern Detection/ Recognition .................................. 8

1.3.2 Signal Filtering/ Processing ..................................... 8

1.3.3 Image Processing ..................................................... 10

1.3.4 Robotics .................................................................. 10

1.3.5 Control .................................................................. 11

1.3.6 Forecasting .............................................................. 12

1.4 Objectives of the Thesis ......................................................... 13

1.5 Organisation of the Thesis ...................................................... 15

CHAPTER 2 LITERATURE SURVEY .................................................... 18

2.1 Survey Objective ................................................................. 18

2.1.1 Network Learning Speed ......................................... 19

2.1.2 Network Accuracy ..................................................... 20

2.1.3 Network Capacity ................................................... 21

2.1.4 Network Generalization ........................................... 21

2.2 The Multilayered Perceptron Model (MLP) ......................... 22

2.2.1 Spiral Identification Problem .................................. 23

2.2.2 XOR Problem .......................................................... 23

2.2.3 Process Control System ........................................... 24

2.2.4 Conclusions for the MLP Model .............................. 25

2.3 The Kohonen Network ......................................................... 25

2.3.1 Vowel Coarticulation ............................................... 26

2.3.2 Speech Recognition ................................................ 26

2.3.3 Character Classification using Kohonen Networks ...... 27

2.3.4 Conclusions ............................................................. 27

2.4 The Adaptive Resonance Theory (ART) ............................ 28

2.5 The Hopfield Network ........................................................ 29

2.6 Survey Conclusions ............................................................. 30

CHAPTER 3 THE BACK PROPAGATION MODEL .................................. 31

3.1 Convergence in the Perceptron Model ............................... 32

3.1.1 The Learning Algorithm of the Perceptron .......... 33

3.1.2 Proof of Convergence .............................................. 34

3.2 The Relationship Between Error Reductions and Weight Changes .................................................. 36

3.3 The Kolmogorov Theorem ................................................... 41
3.4 The Generalised Delta Rule .......................................................... 43
  3.4.1 The Sigmoidal Function ...................................................... 44
  3.4.2 The Delta Values (δ_p and δ_p') in the New Perceptron Model ................. 45
3.5 The Implementation of an XOR Back Propagation Neural Net. .................. 47
  3.5.1 The Input Node Data Structure ........................................ 48
  3.5.2 The Programming model of the Hidden Node Data Structure ............... 48
  3.5.3 The Output Node Data Structure ....................................... 50
3.6 Results ....................................................................................... 51

CHAPTER 4 OTHER NEURAL NETWORK MODELS ..................................... 58
  4.1 Self-Organising Feature Maps .................................................. 58
  4.2 The Kohonen Architecture ..................................................... 60
  4.3 The Kohonen Algorithm ........................................................ 62
  4.4 Implementation ....................................................................... 66
    4.4.1 The Input Interface ............................................................ 66
    4.4.2 The Output Interface .......................................................... 67
  4.5 Results and Observations of the Implementation ............................. 68
  4.6 Adaptive Resonance Theory .................................................... 73
  4.7 The Architecture of the ART1 Network ..................................... 73
  4.8 The ART1 Algorithm ............................................................. 75
  4.9 Learning behaviour of ART1 ................................................... 76
  4.10 The Hopfield Network .......................................................... 78
  4.11 The Architecture of the Hopfield Network ................................ 78
  4.12 The Hopfield Network Algorithm ......................................... 82
  4.13 The Feasibility of the Storage Capacity .................................... 85
    4.13.1 Experiment 1 ................................................................. 85
    4.13.2 Experiment 2 ................................................................. 86
    4.13.3 Experiment 3 ................................................................. 86
  4.14 Conclusions for the Hopfield Model ....................................... 87

CHAPTER 5 FAST LEARNING ARTIFICIAL NEURAL NETWORKS
(FNN) .................................................................................................. 89
  5.1 The Architecture .................................................................... 92
  5.2 Differences in Vigilance Factors ................................................ 94
    5.2.1 ART1 Vigilance Factor p ................................................ 95
    5.2.2 FLANN I Vigilance Factors p and α .................................. 96
    5.2.3 Forced Learning ............................................................. 97
    5.2.4 Immediate Re-Coding ..................................................... 99
    5.2.5 The ART1 within the FLANN I ....................................... 99
  5.3 Weight Manipulation .............................................................. 100
  5.4 Calculations ............................................................................ 102
  5.5 FLANN I Algorithm .............................................................. 105
  5.6 Comparison of Learning Speeds in ART1 and FLANN I ............... 106
  5.7 Training FLANN I for Accuracy ............................................. 109

iv
CHAPTER 6 FAST LEARNING ARTIFICIAL NEURAL NETWORK II (FLANN II) ............................................................ 134
6.1 Distance Metrics ............................................................. 134
6.2 Characteristics of FLANN II .......................................... 137
6.3 The Architecture ........................................................... 137
6.4 An Overview ............................................................... 138
6.5 Detail Description of FLANN II ....................................... 140
  6.5.1 Weight assignments .................................................. 141
  6.5.2 Tolerance Factors .................................................... 142
  6.5.3 Vigilance Factors ..................................................... 143
  6.5.4 Nearest Neighbour Competitive Selection .................. 143
6.6 Flexibility of Tolerance Values ....................................... 146
6.7 The Voronoi Diagram .................................................... 148
6.8 The Algorithm ............................................................. 152
6.9 Learning to Tell Two Spirals .......................................... 153
6.10 On-Line Control and Fault Diagnosis of a Mixing Process.... 160
6.11 Neural network based diagnosis of a Continuous Stirred
    Tank Reactor System ...................................................... 168
6.12 Noisy Trigonometric Wave .......................................... 174
6.13 Comparison of FLANN II with other models .................... 176
  6.13.1 Individual Element Comparisons and Resolutions .... 176
CHAPTER 7 PARALLEL DISTRIBUTED NEURAL NETWORKS (PDNNs)

7.1 Parallel Architectures and Terminology .................................................. 178
    7.1.1 SISD ................................................................... 179
    7.1.2 SIMD ..................................................................... 179
    7.1.3 MISD ..................................................................... 180
    7.1.4 MIMD ..................................................................... 180
7.2 Multiprocessor Systems ............................................................................ 182
    7.2.1 Tightly Coupled Multiprocessor System ........................................ 182
    7.2.2 Loosely Coupled Multiprocessor System ....................................... 183
7.3 Network Topologies ................................................................................ 184
7.4 Platforms used for FLANN model Implementations .................................. 186
    7.4.1 The Transputer MIMD Environment ............................................ 187
    7.4.2 The Sequent Balance MIMD Environment ..................................... 189
    7.4.3 The Heterogeneous PVM Environment ........................................ 190
7.5 Parallel Computing in Current Neural Network Architectures .................. 192
7.6 Parallel Neural Computing using FLANN .................................................. 194
7.7 The Implications of Parallel Distributed Neural Networks (PDNNs) .......... 197
    7.7.1 The Faster Response Speed ......................................................... 197
    7.7.2 Storage Capacity and Classification Capacity .................................. 199
    7.7.3 Scalability and Portability ............................................................. 200
7.8 The Multiple Trigonometric Waveforms Experiment ................................ 201
    7.8.1 Transputer Implementation .......................................................... 202
    7.8.2 Balance 8000 Implementation ...................................................... 205
    7.8.3 PVM Implementation .................................................................... 209
7.9 PDNN (FLANN I) Implementation of the Parity Problem ......................... 211
    7.9.1 Storage Calculations .................................................................. 211
    7.9.2 Parity Problem PDNN (FLANN I) Implementation on the Transputer System .......................................................... 212
    7.9.2 Parity Problem PDNN (FLANN I) Implementation on the Balance 8000 .......................................................... 214
    7.9.4 Parity Problem PDNN (FLANN I) Implementation on PVM ............... 216
7.10 Conclusions of the PDNN Paradigm ......................................................... 217

CHAPTER 8 CONCLUSIONS .............................................................................. 219
8.1 Identification of Major Contributions to Knowledge .................................. 219
8.2 Conclusions from the Three Major Areas of Contribution ......................... 221
    8.2.1 Conclusions for the FLANN I model ............................................ 221
    8.2.2 Conclusions for the FLANN II model .......................................... 223
    8.2.3 Conclusions for the PDNN paradigm ............................................. 225
    8.2.4 Meeting the Objectives ................................................................. 226
8.3 Future Work ............................................................................................ 227
Chapter 1

Introduction
CHAPTER 1 INTRODUCTION

Work on artificial neural networks has been motivated right from its inception, by the recognition that the brain computes in an entirely different way from the conventional digital computer. The ease with which the brain performs complex functions remains unmatched by any man-made machine. Although the conventional computer, has been credited for its assistance in complex computations, it has yet to achieve the intelligence comparable to a human child. Even with its ability to perform high speed and accurate computations, it is still unable to accomplish simple tasks of cognition that comes naturally for a human being.

1.1 The Biological Brain

In comparison to the human's cerebral cortex, the computer possesses high speed registers that perform computations with simple logic operations. These hard wired processors perform fixed number crunching activities with the pre-programmed micro-coding found within the ROM. Unlike the conventional computer architecture, the human brain is a conglomeration of low speed biological processors, i.e. neurons. The brain's operations at low level are slower in processing speeds but tightly coupled together in a parallel structure. It is estimated that within the human's central nervous system, there are approximately $10^{11}$ neurons (Kandel, E.R., et al, 1991). These neurons vary in shape and sizes, but can basically be classified into several types. Figure 1.1a shows a typical pyramidal neuron found in the grey matter of the cerebral cortex. The orientation of the neurons within the cerebral cortex seem to be aligned with the Apical dendrite branches extending close to the surface of the cerebral cortex. Figure 1.1b shows the collection of pyramidal neurons within the cerebral cortex. It is
estimated that a single neuron can make up to 1000 connections to other neurons (Kandel, E.R., et al, 1991), with some neurons making inhibitory connections and others making excitation connections.

Figure 1.1 The typical pyramidal neurons (Kohonen, 1989)

Although the pyramidal neurons have been described, it is not the only kind of neurons that exist within the cerebral cortex. Other neurons such as the star shaped neuron seen in Figure 1.2, can be found within the mass of tightly coupled neurons. Other types of neurons include the invertebrate neuron, the retinal bipolar cell, the spinal motor neuron and the Purkinje cell of the cerebellum. Neurons are not restricted to the cerebral cortex, but can be found throughout the body. Motor neurons are examples of neurons that exist outside the cerebral cortex. These are
responsible for the control of muscle flexing abilities. They are sometimes better known as nerves.

![Star shaped neuron diagram]

Figure 1.2 The star shaped neuron

Each neuron varies in shape and size, but generally consist of a cell body called a soma, several spine like extensions of the cell body called dendrites, and a single fibre called the axon, which branches out from the soma to form connections with other neurons. Figures 1.1 and 1.2 show, the various parts of the neuron. Although a single Axonal fibre issues out of the neuron, it can branch out to form multiple connections with other neurons.

The chemicals found within and around the soma, are ions including sodium (Na\(^+\)), calcium (Ca\(^+\)), potassium (K\(^+\)) and chloride (Cl\(^-\)). The potassium concentrates within the neuron and the sodium concentrates outside. Upon receiving an electrical stimulus, the soma's membrane becomes permeable, allowing the sodium and calcium ions to pass through into the soma. This influx of ions changes the state of the soma, which transmits signals to other neighbouring neurons, through the synaptic connections made by the axons. Much of the details regarding the chemical storage and encoding of information remains a phenomenon.
It has been observed that the axons in general make connections with dendrites (Axo-dendritic connection) or with somas (Axo-somatic connection). It is rare that an Axo-axonal is found, but such connections do exist. It is also common to see a dendrite tree so saturated with axonal terminals that there is hardly any room left for any more connections. Figure 1.3 shows the magnified cross sectional view of an Axo-dendritic terminal connection (Stevens, 1967).

It is interesting to note that within each neuron, there exists a constant potential difference between the cell membrane and the internal cell body. Upon receiving input stimuli from connecting axons, either through the Axo-dendritic or Axo-somatic connections, the cell body will either depolarise or polarise accordingly. It will then transfer the electrical disturbance to other cells through the axons. Axons within the body vary from a 50 micra in length to a few metres in length (Stevens, 1967). It can therefore transmit signals throughout the body, flexing the required muscle under the control of the motor neuron.
1.2 The Artificial Neural Network

The study of artificial neural networks concentrates on building models that emulate the functional brain. Some have taken the approach of closely analysing the brain to acquire clues for implementing such a system. Others have produced plain algorithms that mimic psychological patterns of the human behaviours.

The dramatic concentration of research into this area, in recent years, has led to many developments and trends which accelerate the study of artificial neural networks. To date, there exists many definitions of neural networks. Hecht-Nielsen (1988) offers a general, yet rigorous definition.

A neural network is a parallel, distributed formation processing structure consisting of processing elements (PE) interconnected together with unidirectional signal channels called connections. Each processing element has a single output connection which branches into as many collateral connections as desired. The processing element output signal can be of any mathematical type desired. All of the processing that goes on within each processing element must be completely local; i.e., it must depend only upon the current values of the input signal arriving at the processing element via impinging connections and upon values stored in the processing elements’ local memory. (Hecht-Nielsen 1988)

Figure 1.4 shows the general schematic of an artificial neuron. Note that the function signified by \( f(x) \) depends on the model used.
1.3 Practical Applications of Neural Networks

In recent years, confidence in neural computing is beginning to mature, with a larger influx of applications that use neural computational methods to gain the competitive edge. Many problems of classification, which were once difficult to solve with conventional computational methodologies, are solved using neural network techniques. Some of these are:

- Pattern detection
- Signal filtering
- Image processing
- Robotics
- Control
- Forecasting

Table 1.1 shows some of the common neural networks models and their uses. Each row contains a description of the advantages and the disadvantages of the model.
<table>
<thead>
<tr>
<th>Network / Inventor</th>
<th>Primary Applications</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back propagation by Rumelhart and Hinton</td>
<td>Pattern recognition, signal filtering, image, segmentation,</td>
<td>Fast operation, good at forming internal representations, well studied,</td>
<td>Long training times</td>
</tr>
<tr>
<td></td>
<td>classification, control and data compression</td>
<td>has been successful.</td>
<td></td>
</tr>
<tr>
<td>Learning Vector Quantisation (LVQ) by</td>
<td>Data compression, Auto associative recall. (Pattern completion).</td>
<td>Able to self organize. Rapid execution after training is performed</td>
<td>Unresolved issues on number of training cycles, Slow training.</td>
</tr>
<tr>
<td>T Kohonen.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hopfield Network by J.J. Hopfield</td>
<td>Auto associative recall, optimization</td>
<td>Simple concept, easy to implement on VLSI</td>
<td>Unable to learn new states, poor memory storage, many unwanted spurious states</td>
</tr>
<tr>
<td>Adaptive Resonance Theory by G.Carpenter</td>
<td>Pattern recognition</td>
<td>Able to learn new patterns, form new categories and retain learned categories.</td>
<td>Nature of categorical exemplars may change with learning</td>
</tr>
<tr>
<td>and G.Grossberg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neocognitron by K.Fukushima</td>
<td>Recognition of hand drawn characters and other linear-outline</td>
<td>Able to perform scale translation and rotation invariant pattern recognition</td>
<td>Requires many processing elements, complex structures, and scaling issues for</td>
</tr>
<tr>
<td></td>
<td>figures</td>
<td></td>
<td>real world use need to be resolved.</td>
</tr>
</tbody>
</table>

Table 1.1 The common neural networks and their application areas
1.3.1 Pattern Detection/ Recognition

A popular office tool is the Optical Character Reader (OCR), which transforms a page of typed information into a computer readable ASCII format. The engine which performs the transformation is usually a neural network. It is becoming increasingly common to obtain packaged software that can provide up to 95% accuracy. Sharp Corporation demonstrated an OCR processing 200 Japanese characters per second with 99% accuracy (Hammerstrom, D., 1993). The application has to be extremely robust to allow inaccuracies caused by ink smudges, crinkled paper or poor sample copies. Other conventional techniques tend to fail when exposed to poor conditions.

Sonar Processing is another application that requires robust pattern recognition. Ghosh J., et. al. (1992) discusses the problems of sonar signal variability, making it a complex problem to resolve using conventional technologies. In his paper, he proposes the use of a hybrid system, based on neural network paradigms to perform characterisation and classifications of short duration oceanic signals.

1.3.2 Signal Filtering/ Processing

Sonar signals are acoustic energy that propagate through the water. They are used both in the military and commercial areas. In the commercial application, sonar signals are used to locate shoals of fish or even depths of the ocean floor for oil exploration. The actual signals are typically buried within excessive levels of oceanic noise. A typical signal to noise ratio may reach minus 15 decibels. This value yields a noise to signal ratio of 32 in the sonar sampling (Machado R.Z., et. al., 1992). The sonar signal is often passed through 4 consecutive stages, noise cancellation, feature extraction, detection/ classification and finally the signal display. The systematic flow is shown in Figure 1.5. Neural networks can be
applied to the first 3 stages and the sonar signals can be used with a greater accuracy.

![Sonar processing flow chart](image)

Ontario Hydro is investigating the use of sonar based methods in collecting data for environmental fisheries (Ramani N. et al., 1992). The neural network is trained using the data obtained from the sonar signals. Since the signals represent the size of the fish's swim bladder, such information would allow fishermen to target fishes of a specific size.

Other applications of signal filtering/processing are signal error correction by (Machado R.Z., et al., 1992) and sonar processing by (Speidel, S.L., 1992)
1.3.3 Image Processing

Image processing is another area that has benefited from neural network techniques. Neural networks are utilised in data compression, and even image representation.

Research centres such as NASA, receive millions of bytes of information from its remote probes in space. The information received often requires analysis and is usually archived as compressed formats. Since not all data is necessary, some of it is quantised for faster transmission rates over the information highway. Networks are trained to quantise enough information for the smooth transmission rate, yet not too drastically that vital information is lost. The back propagation networks and LVQ networks have been used successfully to compress data.

In the paper by (Niemann H., et. al, 1993), a neural network known as Learning with respect to experiences and perspectives (LEP) was used to perform adaptive image coding. LEP was used in texture mapping problems which provided computer graphics programmers with a library of surfaces.

1.3.4 Robotics

The study of robotics has been an interesting field studied by many researchers. (Venugopal K.P., et. al., 1992), (Walter J.A., et al. 1993). With the use of Multilayered Perceptron (MLP) models, Purdue University, USA has been able to achieve model based navigation along corridors (Meng M. et al., 1993). Like most compound applications, the neural network is only used at a certain portion of the navigational process. The problem was divided into several sub-parts which consists of the image capturing phase, the down sampling phase, the edge detection phase, the Hough mapping and finally the interpretation of the Hough
mapping to navigate the robot. The MLP is used in the final stage, where it is difficult to implement a robust navigational system using conventional methods.

An interesting application of robotics is found in (Walter J.A, et. al, 1993), where an implementation of neural networks for visuo-motor control of an industrial robot is presented. In the system, the PUMA 562 robot derives its position based on visual information provided by two cameras. With the use of the cameras, the robot can position its end effector to approximately 0.1% of the linear dimension of the work space. Such applications are becoming extremely useful in the manufacturing industry and automated factory lines.

Another area receiving interest is the development of Autonomous Underwater Vehicles (AUVs). These unmanned underwater explorers are equipped with neural network technology, aiding in the control and operation of the systems (Zorpette, 1994).

1.3.5 Control

In the paper by Widrow B., et al., (1963), it was shown that a single McCulloch-Pitts type artificial neuron can control a cart to balance a pole mounted on it. This feat was done by training the neural network with the physical values obtained from the experimental set up. With information such as angle of tilt, cart velocity, force of the motor that was used to drive the cart and the angle of tilt of the pole, the neural network was able to learn the control states. As shown in Figure 1.6, the cart-pole system consists of an inverted pendulum which is pivoted on a cart that traverses only the x axis. The pendulum is also pivoted such that it can only swivel parallel to the x axis. Without the equations of motion, and armed only with the failure signals when the pole exceeds a given angle, the system was trained to eventually perform balancing. Such a system as described in R.H. Cannon, (1967)
is highly unstable. Making the system balance implies a relevance of such a controller in the real world engineering adaptive control systems. Actual implementations include Barto A.G., et al, (1983), Anderson J.A., (1989). All of the experiments required the use of physical inputs to train the neural network or an existing controller which could perform the balancing feat.

![Diagram of cart pole setup]

Figure 1.6 The physical cart pole set up.

1.3.6 Forecasting

Artificial neural networks have the ability to detect some form of hidden organisation within the data that may often escape the human eye. One such application lies in the forecasting of short term load requirements in the power industry. In order that high quality electric energy is supplied to the customers, the electric company has to determine when the loading would occur and how much power will be consumed. This process would need to consider, profit maximisation and minimise excess power output. Such industrial problems require previously obtained data to model and forecast the future trends. (Lee K.Y. et, al, 1991). Another paper by (Chen S.T., et. al., 1992) discusses the use of a neural network to perform the same task, with the additional input element, the weather.
1.4 Objectives of the Thesis

The brief descriptions outlined indicate that the usage of neural networks is applicable to a wide variety of fields. With a closer observation, the application areas can be subdivided into two basic categories. The first involves function approximation, where the neural network is trained to perform interpolation and curve fitting from the training data. The second category involves classification, where specific exemplars are used to train a neural network to recognise classes.

Unfortunately, regardless of whether the neural network is built for function approximation or classification, the current technology is still plagued with the inability to learn fast. Since this thesis is concerned for classification problems, the discussions hence forth will pertain only to classification problems. Most classification problems require the neural network to learn distinct class characteristics from a training data set, and this data set is usually in abundance for the training. With the supply of a well defined training set, one will expect the neural systems to learn with ease, unfortunately, this is not true. Instead, the learning process is usually tedious and slow training times persist.

This situation perhaps exists due to the emphasis of the current day neural network models. Much of the researched models emphasize on modelling the human learning behaviour, with less interest in the industrial setting. The MLP models learn by reducing the energy (e.g. gradient descent), with much of its roots in Hebbian learning. The Kohonen network was a result of studies made on the neuron’s ‘Mexican hat shaped’ response to stimuli. The Adaptive Resonance Theory was Grossberg’s contribution from the cognitive psychologist’s point of view, where the top-down processing and bottom-up processing concepts are used. Little has been done to design networks for use in industrial applications. Most of the designs are results of modifications of existing models, and inevitably
retain the slow learning speeds. Industrial applications usually do not need a full set of human decision skills, but only a subset that provides a basic set of operations for it to function efficiently. In such an environment, training times may become more crucial than absolute accuracy.

In some simple applications, the neural network may require as many as 20,000 training cycles before the network achieves acceptable accuracy. This single problem of excessive training times has perhaps restrained the growth of neural network applications. Investors of factory automation lines may resort to less effective algorithms because the training for neural networks is marred with the slow learning. Flexible automation lines still persist in conventional methods because they are faster to implement than neural network methodologies.

It is felt that while neural network research should head towards the development of models that resemble the cognitive processes of the brain, researchers should not abandon the search of useful task oriented neural networks. These may not possess the intricacies of human cognition, but are sufficient to solve the task. There are many problems where a complete finite set of input-output pairs are available, which can be solved using neural network solutions, but with the current prevalent models, too much time is spent on network training.

It is the objective of this thesis to develop a neural network that is fast learning, able to generalise and achieve good capacity to discern different patterns even though some patterns may be similar in structure. This eventual neural network will be used in the pattern classification environment.
The objectives will focus on problems with a complete and finite set of input-output training pairs. If the finite training pairs are not available, then the conventional neural networks will be useful. This research is undertaken with the hope that it would complement, and not replace, the currently available neural networks in the area where fast learning is required for classification of patterns.

1.5 Organisation of the Thesis

This thesis has been organised into three general sections. The first section spans from Chapter 1 to Chapter 4 and provides a discussion on the common network architectures currently available. Chapter 2 is the literature survey of the networks and provides an outline of the basic neural network models currently available. In Chapters 3 and 4, the architectures are studied in more detail. This provides better support for the survey material of Chapter 2. Some of the common architectures studied are the Multilayered Perceptron (MLP) model, the Kohonen network, the Hopfield network and the ART1 network. For the MLP model, a solution for the XOR problem is provided in Appendix A. The character recognition problem is solved using the Kohonen network and the ART1. Discussions are also provided for the Hopfield network.

The main theme gathered from the study of the four common networks, is the need for faster training to counter the inflexibility of current networks in order to operate in a fast changing environment (with an exception of the ART1).

The second section of the thesis encompasses Chapters 5 and 6. It aims to improve the problem of training times without the loss of accuracy. This has been achieved in many varying degrees by other researchers. e.g., (Sanossian, 1992), (Fahlman S.E., 1988). Their improvement in training times are mostly with reference to the MLP models. Their solutions were rather successful and some training times can
be reduced by up to 50%. Unfortunately, 50% of 50 000 epochs still amount to a large 25 000 epochs.

Chapter 5 presents a Fast Learning Artificial Neural Network (FLANN I) that operates on binary input vectors. This system was developed as a result of studying the ART1 network. The resultant neural network can learn any given pattern and generalise well. In addition, it can maintain a constant learning speed of 1 epoch. The chapter unlocks the network operations and performs a comparison with the ART1.

Chapter 6 delves into a second novel architecture, (FLANN II). This second model was designed to solve systems using continuous input data. It is based on the nearest neighbour classifier, a common metric used in many networks such as the Kohonen network. Similar to the earlier FLANN I model, FLANN II also learns in a single epoch, but has the added flexibility to perform operations on continuous values. It can also be modified to operate on binary information. In the chapter, theories behind the network operations are explained, providing a lucid understanding of the model. Many application examples and implementations are provided to show the robustness of the new algorithm.

Well known problems such as classifying two intertwined spirals, classification of noisy wave forms, character recognition, parity bit problems are used as authenticators of the neural network models. In addition, two industrial process control diagnostic problems, extracted from a paper by (Zhang J., et. al., 1992) are used as tests for FLANN II.

The third part of this thesis involves parallel computing. Chapter 7 takes the FLANN models a step further into the Parallelisation environment. Here, a new
form of parallelisation is introduced. This is a significant contribution to the field of neural networks, where the new paradigm, Parallel Distributed Neural Networks (PDNNs), is developed. Several completely independent neural networks are executed concurrently on a loosely coupled parallel architecture to solve a single pattern recognition problem. The independence is maintained from the start, where the input vector is divided into independent sub-vectors for individual neural classifications in the distributed architecture. It is only in the later stages where the independent neural networks combine their classifications that the final result is obtained. The details of PDNN implementations on 3 different parallel platforms is provided, showing that the paradigm is a reliable and robust solution that can be used to reduce memory requirements and increase the classification performance.

Chapter 8 concludes the thesis and summarises the contributions and results obtained from Chapters 5 to 7. Several concluding comparisons are made between FLANN and common existing neural models. The impact of PDNNs is reiterated and a short section for future research is found towards the end of the chapter.
Chapter 2

Literature Survey
CHAPTER 2  LITERATURE SURVEY

In recent years, the number of neural network architectures have increased greatly. Most of the additions are modifications and improvements of existing architectures, as such, the main structure of the original network generally remains unchanged. To reduce the scope of the survey, and streamline the thesis, the investigation is conducted on the following neural network architectures.

- The Multilayered Perceptron (MLP) basic gradient descent model
- The Kohonen model
- The Adaptive Resonance Theory I (ART) model
- The Hopfield model

2.1 Survey Objective

This literature survey was conducted with the view that the thesis is focusing on classification problems. It aims to identify and evaluate specific characteristics, which are of interest, in classification neural network models. These characteristics can be categorised into the following:

- Network Learning Speed
- Network Accuracy
- Network Capacity
- Network Generalisation

The sections that follow elaborate on each of these categories.
2.1.1 Network Learning Speed

In general the network learning speed is defined as the total number of epochs required to train the network until it can accurately produce the desired output upon presentation of the input data used during training. An epoch is further defined as a complete cycle of presentations made using the training data set (Rumelhart, D.E., et. al., 1986).

Upon analysis of some applications, it quickly becomes acceptable that in certain situations, long training times can be tolerated if the final network produces a perfect classification (99%). This argument is legitimate and is the case with many actual applications (Hammerstrom, 1993), (Zhang J., et. al., 1992). As mentioned in the former reference, the optical character recognition (OCR) system created by SONY may receive as long as 2 months of training, but its precise performance during operations compensates for the long training periods.

In the light of many industrial applications, excessively long training times may sometimes discourage potential users from utilising neural networks. In the modern day manufacturing environment, some assembly lines no longer have a permanent manufacturing process (Whitney D., 1993), but possess a modifiable assembly line to manufacture different products, which can be achieved by changing the programs that govern the robotic assembling process. The following is a scenario which could possibly happen when neural networks are employed in a robot automated plant decides to change the manufacturing process.

First assume that with the slow learning rates in current neural network technology, a robot requires an average of 48 hours to be trained with the new operational procedures. This also runs along the assumption that the training set is readily available (which is seldom the case). In a manufacturing plant, the assembly
line may sometimes contain as many as 100 robots. To train all the robots would take 4800 hours. This is equivalent to more than half a year of training. With such involving time scales, there is reasonable doubt that any user would consider the application of such systems.

The essence of the argument is, although the slow learning characteristic of neural networks can be tolerated in some applications, its presence can lead to a great loss of confidence from other parts of the commercial world.

2.1.2 Network Accuracy

Network accuracy plays an important role in the field of neurocomputing. In many applications, (Zhang J., et. al., 1992), (Ramani N., et. al., 1992), (Chen S.T., 1992), any doubts with the network accuracy would cause drastic losses in revenue. In the previous reference by Chen S.T., (1992), the neural network is used to forecast short-term power requirements. If the neural network fails to produce a correct output supply, either an over supply would lead to waste or an under supply would force the power company to perform immediate rectification which is often costly. If an inaccurate classification network was used in a fault identification process (Zhang J., et. al., 1992), frequent false alarms may create extra work, over and above the original problem to be solved. Worse still, actual faults may go undetected.

In general, the accuracy of the neural network is of crucial importance. Little can be tolerated in an inaccurate neural network. It is therefore imperative that the network is consistently accurate and reliable.
2.1.3 Network Capacity

Acceptable network storage capacity is another important aspect of neural network architectures. The network should be versatile to permit any form of input vectors to be stored as exemplar patterns. This means that even a set of closely related patterns can be stored within the network and retrieved accurately as separate exemplars. Figure 2.1 shows two closely related patterns that can cause problems with networks with poor storage capacity.

Figure 2.1 Good storage capacity must accept closely related patterns without loss of generality.

Another good test for network capacity is the parity bit problem mentioned in Rumelhart D.E., et. al., (1986). Although network capacity can be overlooked in restricted domain applications, it is generally viewed as an important characteristic sought after in a neural network. In industrial applications such as (Machado R.Z., et. al., 1992), the vast sets of binary coded information relies heavily on the network capacity of the model. A network with poor information storage capacity would create confusion in such applications.

2.1.4 Network Generalization

The final characteristic of concern in this survey is the neural network's ability to generalise patterns. This is by far the greatest motivation for the spread of the
technology. If a neural network could only produce what it was taught and was unable to generalise, the industry would turn to expert systems or simple look-up tables. The generalization properties of neural networks have enhanced the robustness of systems to overcome noisy environments (Zhang J., et. al., 1992), (Fukushima, 1988), (Zhao Z., et. al., 1992), (Drucker H., et.al., 1992). This is the edge that the neural network technology has over expert systems, and it is therefore an important aspect of neurocomputing.

As mentioned in control related papers (Zhang J., et. al., 1992), expert systems that were built to perform systems diagnostics tend to fail when exposed to imperfect input readings. It was also noted that sensor readings were at times unavailable or maintain a margin of error. Yet, the neural network is able to provide accurate and reliable approximations. The generalization abilities of neural networks have provided the technology the leading edge from conventional knowledge based methods. If this property was absent from a neural network model, then the model is very restricted, reducing its capabilities to a simple look-up table.

2.2 The Multilayered Perceptron Model (MLP)

The MLP architecture is perhaps the most celebrated model in the field of neurocomputing. Some reasons for its persistent existence is its proven convergence, and the availability of good documentation. Furthermore, everyone appears to be using it.

At a 1993 conference in Liverpool, an informal interview was conducted to determine the reasons behind the popularity of MLP models. The unanimous answers echoed throughout the interview; "It has been proven to converge." "It is
Chapter 2

Literature Survey

well documented." "Everybody is using it." "Why change?" Perhaps the question that should now be answered is: "What is so good about the MLP model?"

2.2.1 Spiral Identification Problem

In a paper by Lang, K.J., et al., (1989), the authors used the MLP model to solve a spiral identification problem. Details of the problem are explained in Chapter 6, where the experiment is repeated using the Fast Learning Artificial Neural Network II (FLANN II). With modified versions of the back propagation model, the author managed to train the model to recognise the spirals with a mean training time of 20 000 epochs using a Vanilla Back Propagation model (Fahlman S.E., 1988). For the Cross Entropy Back Propagation model, they achieved a mean of 10 000 epochs whilst for the Quickprop model, they used 8000 epochs. In terms of accuracy, they obtained a mean of about 91%. The generalization factor was good, considering that the system was trained only with the exact spiral data points. The system did not encounter problems with storage capacity. The results obtained from the paper are tabulated in Table 2.1

<table>
<thead>
<tr>
<th>Run</th>
<th>Vanilla BP (Iterations)</th>
<th>Cross Entropy BP (Iterations)</th>
<th>Quickprop (Iterations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>18 900</td>
<td>16 200</td>
<td>4 500</td>
</tr>
<tr>
<td>B</td>
<td>22 300</td>
<td>8 600</td>
<td>12 300</td>
</tr>
<tr>
<td>C</td>
<td>19 000</td>
<td>7 600</td>
<td>6 800</td>
</tr>
<tr>
<td>Mean</td>
<td>20 000</td>
<td>10 000</td>
<td>8 000</td>
</tr>
</tbody>
</table>

Table 2.1 Results obtained from Lang, K.J. et. al, (1989)

2.2.2 XOR Problem

In Rumelhart, D.E., et al, (1986), the Parallel Distributed Processing (PDP) group worked on the original back propagation model to solve the XOR problem. The
input layer consisted of 2 nodes and the output layer had a single node. The hidden layers were varied between 1 to 32. In a single case with 8 hidden layers, the number of epochs required varied between 68 and 450, depending on the parameters set. Although certain cases encountered local minima, these cases became rare as the number of hidden layers were increased (Experiments with the XOR problem were repeated and the program can be found in Appendix A). In the same book (Rumelhart, D.E., et. al, 1986), the PDP group trained a network to perform a 4 bit parity checker. The solution was reached after 2 825 epochs.

### 2.2.3 Process Control System

In another paper by Zhang J., et. al., (1992), an MLP model was used to train a process control diagnostic system. Table 2.2 summarises the convergence of the experiments. The diagnostic network was accurate, even in cases when the partially obscured input vectors were deliberately used. Again, there was no encounter with capacity limitations. More details of this experiment are found in Chapter 6, where the experiment is repeated with the Fast Learning Artificial Neural Network II, (FLANN II).

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\eta$</th>
<th>Iterations 1 hidden layer</th>
<th>Iterations 2 hidden layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>0.6</td>
<td>no convergence</td>
<td>no convergence</td>
</tr>
<tr>
<td>0.9</td>
<td>0.1</td>
<td>550</td>
<td>1681</td>
</tr>
<tr>
<td>0.8</td>
<td>0.5</td>
<td>274</td>
<td>no convergence</td>
</tr>
<tr>
<td>0.8</td>
<td>0.4</td>
<td>380</td>
<td>907</td>
</tr>
<tr>
<td>0.8</td>
<td>0.2</td>
<td>534</td>
<td>2240</td>
</tr>
<tr>
<td>0.6</td>
<td>0.9</td>
<td>828</td>
<td>1361</td>
</tr>
<tr>
<td>0.6</td>
<td>0.2</td>
<td>1324</td>
<td>3898</td>
</tr>
<tr>
<td>0.5</td>
<td>0.4</td>
<td>1131</td>
<td>2297</td>
</tr>
<tr>
<td>0.5</td>
<td>0.2</td>
<td>1796</td>
<td>5215</td>
</tr>
</tbody>
</table>

Table 2.2 Results obtained from (Zhang J. et.al., 1992)
2.2.4 Conclusions for the MLP Model

From the survey performed on the characteristics of the MLP model, several attributes of this model can be deduced here. It is generally a good model that has not shown storage capacity problems. Although the network size may be determined by using Kolmogorov's theorem (pp 122, Hecht-Nielsen, 1990), most researchers seek more efficient convergence. Some resort to trial-and-error methods. Some papers proposed methods of estimating the size of networks, by utilising computational geometric structures such as the Voronoi diagram (Bose N.K., 1993), but these are just heuristical suggestions with no formal proof at the time of writing. Once the network is trained, the size of the network will remain fixed and any plans to accommodate new patterns may sometimes result in resizing the network, which normally requires total network re-training.

Its ability to generalise in all the cases surveyed, indicates its robustness. In the spiral problem in §2.2.1, the network was able to generalise positions of data points that it had not been trained with. The accuracy seen in §2.2.3 increases its value in industrial process control applications. Unfortunately, the only setback in this model is the learning speed. The worst case learning speed for a 6 input 7 output system in §2.2.3 was 5 215 iterations. For the 194 point spiral system in §2.2.1, the worst case was well into the 20 000 region. The slow learning might probably be the only unacceptable point in the MLP model.

2.3 The Kohonen Network.

Interestingly, the mass of researchers in neural networks have embraced the MLP models so much, that relatively few projects have used the Kohonen network model. This is a very unique model that has found its roots from the area of self-organisation models.
2.3.1 Vowel Coarticulation

In the paper by Leinonen L. et. al., (1993), the authors used the Kohonen self-organising map to perform an acoustical self-organisation of fricative vowel coarticulation. The results indicated that the self-organising map was able to achieve good results their psychoacoustic experiments. Due to the deep involvement into Finnish linguistic applications, the survey conducted, was brief, with an objective to seek out problems mentioned about the Kohonen network.

There was no mention of problems encountered when using the Kohonen network. From the results of the paper, it is seen that the Kohonen model has a very good ability to perform generalization. In the training of the network, several fricative vowels were used. The network produced a gradual transitive link between the vowels by performing, what seemed like an interpolation of two sounds. As a result, the authors could trace the path of sounds which uttered words would follow.

2.3.2 Speech Recognition

In yet another paper by Zhao Z., et. al., (1992) used the Kohonen network to perform speech recognition. The paper concentrated on using a novel method of smoothing the parameters of hidden Markov models (HMMs), which produces improved speech recognition results when only a limited amount of training data is available. In most of the literature that use the Kohonen network, modifications were usually made to the original network. This increased the difficulty for the survey to make direct comparisons and specific information related to the Kohonen characteristics could not be directly extracted. To extract the actual characteristics of the Kohonen model, most of the details were obtained from personal experiments conducted using the Kohonen network.
2.3.3 Character Classification using Kohonen Networks

The best source of extracting the characteristics of the Kohonen network is from the paper by Kohonen himself. In Kohonen T., (1988), it is understood that the training of the network may vary with the application usage. In an application where little noise is expected and the patterns do not receive much distortion, the number of training epochs may vary from 10 to 100. In an internal report, Tay L.P., (1993), the author trained the Kohonen network using 10 epochs. The resultant network could retrieve trained information to a good level. Although patterns with small Hamming distances occupied less space on the Kohonen grid, the storage capacity of the network was never a serious problem. If new patterns were required to be appended onto the training set, all previously stored patterns had to be re-trained. As mentioned in §2.3.2, the Kohonen network has a very good ability to perform generalization. With a training set of 10 numbers, 0 - 9, the trained network was able to recognize distorted images of the original patterns and perform generalization. More detailed work on the Kohonen network is found in Chapter 4.

2.3.4 Conclusions for the Kohonen Network

Several conclusions can be drawn from the survey conducted on the Kohonen network. Firstly, it is a relatively new network which is beginning to attract new researchers. The number of papers using the Kohonen network is not as numerous as the MLP model, but researchers are beginning to focus on it. From the papers surveyed, there is little mention of the number of training epochs required. This, unlike the MLP model, indicates the better training speed obtainable in Kohonen networks. Most of the papers utilize the self-organizational properties found in the network and the network's ability to generalize. From experiments, as documented in Tay L.P., et. al.,(1993) the Kohonen network is able to accommodate closely
related patterns, but these patterns will occupy smaller regions within the Kohonen grid.

The number of training epochs required by the Kohonen network is not specified by Kohonen. This is because there is no general number that can be provided and it will vary, depending on the application. Finally, the Kohonen network requires retraining once new input vectors are introduced to a fully trained network.

2.4 The Adaptive Resonance Theory (ART1)

Like the Kohonen network, there is a limited number of publications found on the ART1 neural network model. Most of the publications were by the original authors, Carpenter and Grossberg (Carpenter G.A., et.al., 1988, 1992), (Grossberg S., 1987, 1988). The survey was then conducted based on the publications of the original authors. It was found that the ART1 network has a fast learning rate, providing it with the ability to perform real time learning. It is also able to continue learning and yet not produce an unstable learning process, which is exhibited in the back propagation model.

Although the ART1 network is able to perform fast learning, relative to the other models, if new exemplars are introduced it may still require a small amount of retraining. This requirement depends on the Hamming distance and the similarity of the new pattern to the already stored exemplars. This means that there may be a slight amount of pattern confusion during the initial introduction stages of a new pattern (Grossberg S., et. al., 1987).

A good feature of the ART1 network is that it is able to continue learning as long as there is available memory within the hardware system. ART1 is therefore a growing network, adopting more patterns when the need arises. This provides the
model with a good capacity patterns. Chapter 4 and 5 will dwell on the ART1 model in more detail.

In general, the ART1 network is a good network. It is presented by Grossberg in a rather unique way, due to the psychological background of the research (Grossberg S., 1988). Although there are good storage capabilities, exemplars within the Hamming proximity may sometimes require several epochs of training before the two patterns can be distinguished. Sometimes this process cannot be done, except to increase the vigilance factor for a 100% match search and then relaxing the vigilance factor for generalization.

In terms of learning speed, the ART1 network is able to perform relatively quick learning. The number of epochs can be as low as 10. In a hybrid system by the same authors, the ART model was built with a fuzzy system. It produced rather fast learning results (Carpenter G. A., et. al., 1992). In an experiment with an intertwined spiral experiment, the fuzzy ART system was able to learn to classify the two spirals in 5 epochs, compared to the several thousand epochs required in the MLP counterparts.

2.5 The Hopfield Network

The network that is least encountered in pattern recognition applications is the Hopfield network. In the study of the Hopfield network during the initial survey, it seemed to provide the best form of training. All that was required was a matrix multiplication of the input vectors to create the weight matrix. Much was publicised about its ability, but after a detailed study and scrutiny of the algorithm, it was found that the Hopfield network would perform better in optimisation networks (Hopfield J.J., et. al., 1984), (Tank D.W., et. al.,1986). As a pattern classifier, the Hopfield network does not perform as well. This is explained later in Chapter 4, where the system is brought under scrutiny. Other literary support
comes from Hopfield himself (Hopfield J.J., 1982). In general, the Hopfield model is not suitable for the pattern recognition applications due to its poor capacity for input patterns. Many have claimed to provide information of the model's maximum storage capacity, but these have introduced biased conditions which are seldom prevalent in pattern recognition problems. The storage capacity drastically increases when patterns are deliberately separated by specific Hamming distances. But once this criteria is not met, the storage capacity becomes poor (Alianna M., et. al., 1991). Most of the applications which use the Hopfield network deal with optimisation. The travelling salesman problem is often associated with the Hopfield network. Since this thesis is concerned only with classification, the Hopfield network is not pursued further for its abilities.

2.6 Survey Conclusions

From the survey, it can be seen that there is still much room for improvements in the field of neural network. Models are still plagued with slow and laborious training speeds, which range from several epochs to thousands of epochs. The MLP models, though can be trained to provide quick response to exemplars, requires an enormous amount of time for its training. The Kohonen network begins as a fast learner but needs to contend with relatively slower response times after the network is trained, because it requires many nodes to make up the Kohonen grid. On the whole, the 4 neural networks discussed possess different qualities. In the 2 chapters that follow, these networks are discussed in more detail.

It would be desirable if a new neural network model could be designed to possess the good qualities of each network, allowing less epochs for training, yet providing good generalisations and acceptable response times.
Chapter 3

The Back Propagation Model
CHAPTER 3 THE BACK PROPAGATION MODEL

The back propagation neural network, or a more general term - Multilayered Perceptron, is perhaps the oldest model that has received much attention. Figure 3.1 shows a 2-5-1 back propagation neural network. The numbers indicate the number of nodes within each layer, where the input layer is on the left most position and the output layer is on the right most position. In this case, the hidden layer contains 5 nodes.

![Diagram of 2-5-1 Back Propagation Model](image)

Figure 3.1 A 2-5-1 Back Propagation model

Even today, researchers are studying various ways to enhance the model's capabilities. Several mathematical proofs which are the essential building blocks for the Delta rule and the theory of learning by back propagation are provided. An implementation of a basic back propagation neural net to solve the XOR problem
is included in Appendix A. Results of this implementation are documented in this chapter itself. The chapter is divided into seven distinct sections and these are listed as follows:-

- Convergence in the Perceptron Model
- Relationship between Error Reductions and Weight Changes
- The Kolmogorov Theorem
- The Generalised Delta Rule
- The Implementation of an XOR Back Propagation Neural Net
- Results

3.1 Convergence in the Perceptron Model

The governing principles of the original perceptron model is shown in equation (3.1). The eventual back propagation learning equations are outgrowths of this basic equation. As a result, a study of the convergence principles in the basic perceptron model is analogous to examining the reasons for the convergence achieved by the generalized Delta rule.

\[ a_{ij} = \sum_i w_{ij} x_i(t) \]  

(3.1)

where \( w_{ij} \) is the weight at presentation \( t \) between a neuron on layer \( i \) and layer \( j \). \( x_i \) is the input value of the presentation.

In the vector notation, (3.1) would be equivalent to,

\[ o_{pj} = W \cdot X \]  

(3.2)

where \( W \) is the weight vector and \( X \) is the input vector.
3.1.1 The Learning Algorithm of the Perceptron

Given that $x \in (D^+ \cup D^-)$, where $D^+$ and $D^-$ are the two classes from which the value $x$ is taken. The original perceptron learning algorithm proceeds as follows:

**Perceptron Learning Algorithm:**

1) Choose any value of $w$

2) Choose a value of $x$

3) if $x \in D^+$ and $w \cdot x > 0$ then GOTO 2
   if $x \in D^+$ and $w \cdot x \leq 0$ then GOTO 4
   if $x \in D^-$ and $w \cdot x < 0$ then GOTO 2
   if $x \in D^-$ and $w \cdot x \geq 0$ then GOTO 5

4) **ADD function**
   
   $w = w + x$
   
   GOTO 2

5) **SUB function**

   $w = w - x$
   
   GOTO 2

The above algorithm works only for a linearly separable space, (Minsky M, et. al., 1969). Nevertheless, in its simplicity, it serves as a platform for showing convergence in the perceptron model.
3.1.2 Proof of Convergence

Assume there exists a unit vector $W^*$ that discriminates the space in question such that,

$$ W^* \cdot X > k $$

where $k$ is a positive number.

To determine the relationship between the unit vector $W^*$ and the current weight vector $W$, consider the cosine of the angle between the two vectors, given by $G(W)$, where

$$ G(W) = \frac{W^* \cdot W}{\|W^*\| \|W\|} $$

(3.4)

$W^*$ is the targeted weight vector and $W$ is the current weight vector. Since $\|W^*\|$ is equal to 1 (unit vector), the expression in (3.4) can be rewritten as

$$ G(W) = \frac{W^* \cdot W}{\|W\|} $$

(3.5)

Note then that the value of $G(W)$ can never exceed 1 because it is a cosine of the angle between the two vectors. This condition results in the inequality below,

$$ G(W) \leq 1 $$

(3.6)

Consider the behaviour of $G(W)$ during step 4 of the algorithm in section 3.1.1, i.e. the $\text{ADD}$ function $w_{(t+1)} = w_t + x_t$. 
Chapter 3 The Back Propagation Model

The numerator of $G(W)$ in (3.5) would yield

$$W^* . W_{(t+1)} = W^* . (W_t + X)$$

$$= W^* . W_t + W^* . X$$

Note that from the assumption in (3.3)

$$W^* . W_{(t+1)} \geq W^* . W_t + k$$

after $n$ ADD functions $W^* . W_n \geq nk$ (3.7)

Consider now, the denominator of (3.5). Since the program is going through an ADD function, then from the algorithm provided in §3.1.1, $w^* . x$ must possess a value less than 0.

$$|w_{t+1}|^2 = w_{t+1} . w_{t+1}$$

$$= (w_t + x)(w_t + x)$$

$$= |w_t|^2 + 2w_t . x + |x|^2$$ (3.8)

Again, because the algorithm in section 3.1.1 is going through the ADD function, $x$ has to be 1 and the value of (3.8) must have been negative. Since the negative value could only have come from $2w_t x$, equation (3.8) can now be written as

$$|w_{t+1}|^2 \leq |w_t|^2 + 1$$ (3.9)

and after $n$ applications of the ADD function, equation (3.9) can be rewritten as
By combining equations (3.7) (the numerator) with (3.10), (the denominator), the expression below is obtained.

\[
|G(w_n)| = \frac{W^* \cdot W_n}{|W_n|} = \frac{nk}{\sqrt{n}} = k\sqrt{n}
\]

since \( G(W) \leq 1 \)

\[
\Rightarrow n \leq \frac{1}{k^2}
\]

From equation (3.11), since \( k \) is finite, \( n \) will also be finite. This proves that the perceptron model does in fact converge after \( n \) finite vector pair presentations.

3.2 The Relationship Between Error Reductions and Weight Changes

Before deriving the equations necessary for the generalized Delta rule, it would be useful to understand the relationship between the error reduction process and the changes in weights. \( \delta_{pj} \) is defined as the difference between the target output and the actual output. This is represented by,

\[
\delta_{pj} = (t_{pj} - o_{pj})
\]
Now examine the activation function of the simple perceptron given by equation (3.13):

\[ o_{pj} = \sum_k w_{kj}x_{pk} \quad (3.13) \]

The component \( x_{pk} \) is an input variable which is presented to the network. It is not a controlled variable. The only other variable is \( w_{kj} \), which in fact, is the only variable that can be altered to change the behaviour of the network.

The error of a single output perceptron \( p \) can be defined by being proportional to the square of the difference between the target output and the actual output. For the simplification of future derivations, let the constant of proportionality be \( \frac{1}{2} \).

Thus the equation can be written as,

\[ E_p = \frac{1}{2} \sum_j (t_{pj} - o_{pj})^2 \quad (3.14) \]

Figure 3.2 shows the relationship between the total error \( E \), which is \( \sum E_p \), and \( w_{ij} \) on a graphical plot.
The purpose of the Delta rule is to make a change in the weight, proportional to the negative of the derivative of the total error $E$. The derivation of this relationship begins with the chain rule, i.e.,

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ij}} \quad (3.15)$$

From the derivative of equation (3.14), we have

$$\frac{\partial E_p}{\partial o_p} = -(t_{pj} - o_{pj}) \quad (3.16)$$

Note that since $\delta_{pj}$ is defined as $(t_{pj} - o_{pj})$, from equation (3.12), then equation (3.16) becomes

$$\frac{\partial E_p}{\partial o_p} = -\delta_{pj} \quad (3.17)$$
From equation (3.13), the second half of the chain rule in equation (3.15) becomes

\[
\frac{\partial o_{pj}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_k w_{kj} x_{pk}
\]

(3.18)

Since \(\frac{\partial w_{kj}}{\partial w_{ij}} = 0\) except when \(k = i\), equation (3.18) becomes

\[
\frac{\partial o_{pj}}{\partial w_{ij}} = x_{pi}
\]

(3.19)

By combining the two parts of the chain rule, equation (3.15) becomes

\[
\frac{\partial E_p}{\partial w_{ij}} = -\delta_{pj} x_{pi}
\]

(3.20)

Note that by definition, the change in weight, \(\Delta_p w_{ij}\), after each presentation is given by the equation

\[
\Delta_p w_{ij} = \eta \delta_{pj} x_{pi}
\]

(3.21)

where \(\eta\) is the learning rate.

Thus based on (3.20) and (3.21)

\[
\Delta_p w_{ij} \propto -\frac{\partial E_p}{\partial w_{ij}}
\]

(3.22)

A relationship between the Error \(E\) and the change in weight \(w_{ij}\) can now be established. The total error \(E\) is given by equation (3.23)
\[ E = \sum_p \sum_j \frac{1}{2} (t_{pj} - o_{pj})^2 \]  

(3.23)

and combining equation (3.22) with (3.23), we obtain

\[ \Delta w_{ij} \propto -\frac{\partial E}{\partial w_{ij}} \]  

(3.24)

This result in essence states that the change in weight, \( \Delta w_{ij} \), is proportional to the negative derivative of the error \( E \). This is consistent with the graphical representation in Figure 3.2, where the gradients of the tangents on the curve exhibit these qualities. The weights that are away from the target value would converge towards the target value because of the Delta rule, bringing the error to a minimum. Thus, the Delta rule is proved to converge.
3.3 The Kolmogorov Theorem

The Kolmogorov theorem states that a network of perceptrons does not require more than 3 layers to perform a successful discrimination of any arbitrary complex shape (Beale R., et. al., 1991). For the linearly inseparable XOR problem, it is noted that the single perceptron is unable to create the spatial discriminations (Minsky M., et.al., 1969). This is due to the non-clustering spatial domains of the XOR problem. This effect is illustrated in Figure 3.3 Since the single perceptron has only a single linear discriminatory line, it is unable to separate the classes within the XOR problem.

![Figure 3.3](image)

Figure 3.3 The figure shows the failure of the single perceptron in classifying the XOR problem.

Note that when a second layer is added, as shown in Figure 3.4, the first layer would now behave with similar properties as a single perceptron would. Thus it is able to perform a class discrimination with \( n \) discriminating lines, where \( n \) is the number of units in the first layer.
Chapter 3 The Back Propagation Model

Figure 3.4 The two layer network

The expected graphical representation would thus change from that represented in Figure 3.3 to that of Figure 3.5.

Figure 3.5 The addition of another layer increases the discriminating power of the network

The function of the perceptron in the final layer would then be that of a simple logic gate. As the number of nodes increase within the two layers, the possibility of classifying convex regions comes into play. Unfortunately this does not give the network the capability of classifying concave regions within convex regions. In order to classify concave regions within convex regions, an additional layer has to be introduced into the network. Figure 3.6 shows a summary of the different networks with varied layers and the associated decision regions.
3.4 The Generalised Delta Rule

Based on Kolmogorov's theorem in §3.3, a 3 layer network is required to successfully perform spatial discriminating functions. The reason for the partial collapse of Neural Networks Research in the 1960s was due to their inability to determine the Delta values within the hidden layers of a multiple layered network. The failure of the early perceptron model perhaps lies the inflexibility of the output values, i.e. "0"s and "1"s. If these outputs were replaced by a continuous or step change in values instead of a hard limiting 1 or 0, then the network would be able to determine the proportional change needed on each set of weights. The two different threshold functions are illustrated in Figure 3.7.
Chapter 3  The Back Propagation Model

3.4.1 The Sigmoidal Function

The output of the new perceptron model given by the sigmoidal threshold function is shown in equation (3.25)

\[ o_{pj} = f( net_{pj} ) = \frac{1}{1 + e^{-net_n}} \]  \hspace{1cm} (3.25)

where \( net_{pj} \) is the activation function of the old perceptron model given by

\[ net_{pj} = \sum_i w_{ij} x_{pi} \] \hspace{1cm} (3.26)

The derivative of \( f(net_{pj}) \) is given by

\[ f'(net_{pj}) = \left( 1 - e^{-net_n} \right)^2 \left( -e^{-net_n} \right) \] \hspace{1cm} (3.27)

\[ = \frac{e^{-net_n}}{(1 + e^{-net_n})^2} \]
\[ = \frac{e^{-net_n} + 1 - 1}{(1 + e^{-net_n})^2} \]
\[ = \frac{1 + e^{-net_n}}{(1 + e^{-net_n})^2} - \frac{1}{(1 + e^{-net_n})^2} \]
\[ \begin{align*} 
\frac{1}{1 + e^{-\text{net}_n}} & = \frac{1}{1 + e^{-\text{net}_n}} \frac{1}{1 + e^{-\text{net}_n}} \\
& = \frac{1}{1 + e^{-\text{net}_n}} \left( 1 - \frac{1}{1 + e^{-\text{net}_n}} \right)
\end{align*} \]

\[ f' (\text{net}_{pj}) = f (\text{net}_{pj})(1 - f (\text{net}_{pj})) \quad (3.28) \]

since \( o_{pj} = f(\text{net}_{pj}) \)

\[ f' (\text{net}_{pj}) = o_{pj}(1 - o_{pj}) \quad (3.29) \]

3.4.2 The Delta Values (\( \delta_{pj} \) and \( \delta_{pi} \)) in the New Perceptron Model

Now define the new value of \( \delta_{pj} \) in the new perceptron model as

\[ \delta_{pj} = - \frac{\partial E_p}{\partial \text{net}_{pj}} \quad (3.30) \]

This is consistent with that of the old model since (new model) \( \text{net}_{pj} = \) (old model) \( o_{pj} \). Again, like the old model, the equation governing the weight changes remains as

\[ \Delta w_{ij} = \eta \delta_{pj} o_{pj} \quad (3.31) \]

by the chain rule, we have

\[ \delta_{pj} = - \frac{\partial E_p}{\partial \text{net}_{pj}} = - \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial \text{net}_{pj}} \]

Consider the second part of the chain rule,
\[ \frac{\partial o_{pj}}{\partial net_{pj}} = f'(net_{pj}) \]  (3.32)

The first part of the chain rule will yield,

\[ \frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) \]  (3.33)

completing the chain rule,

\[ \delta_{pj} = f'(net_{pj})(t_{pj} - o_{pj}) \]
\[ \delta_{pj} = (1 - o_{pj})o_{pj}(t_{pj} - o_{pj}) \]  (3.34)

Equation (3.34) denotes the Delta value of the output layer, which is directly involved in changing the values of weights found between the hidden layer and the output layer. As mentioned earlier, there was a breakdown in the research efforts when the Delta value responsible for changing the weights between the input layer and the hidden layer could not be found due to the unknown target value of the hidden layers. To differentiate the two delta values, denote the delta value for the hidden layer as \( \delta_{pi} \).

Then

\[ \delta_{pi} = - \frac{\partial E_p}{\partial net_{pi}} \]  (3.35)

and using the chain rule,

\[ - \frac{\partial E_p}{\partial net_{pi}} = - \frac{\partial E_p}{\partial o_{pi}} \frac{\partial o_{pi}}{\partial net_{pi}} \]  (3.36)

\[ \frac{\partial o_{pi}}{\partial net_{pi}} = f'(net_{pi}) \]  (3.37)
By combining equation (3.37) and (3.39) into (3.36), the Delta value for the hidden layer is given by,

\[
\frac{\partial E_p}{\partial o_{pi}} = \sum_j \frac{\partial E_p}{\partial \text{net}_{pj}} \frac{\partial \text{net}_{pj}}{\partial o_{pi}}
\]

\[
= -\sum_j \delta_{pj} \frac{\partial}{\partial o_{pi}} \sum_i w_{ij} o_{pj}
\]

\[
= -\sum_j \delta_{pj} w_{ij}
\]  

(3.38)  

(3.39)

By combining equation (3.37) and (3.39) into (3.36), the Delta value for the hidden layer is given by,

\[
\delta_{pi} = f' (\text{net}_{pi}) \sum_j (\delta_{pj} w_{ij})
\]  

(3.40)

The derivation of this Delta value is the most important breakthrough in neural network research. By using this, the back propagation network can be modelled.

### 3.5 The Implementation of an XOR Back Propagation Neural Net

The neural net to be implemented has two input neurons, one output neuron and a user defined layer consisting of a number of hidden neurons. The structure of the network is shown in Figure 3.8.

![Diagram of the neural network](image)

**Figure 3.8** The general structure of the network
3.5.1 The Input Node Data Structure

The neural network was built specifically to solve the XOR problem. This limits the network to two input neurons. The two neurons were modelled as a single dimensional array with two elements, In[0] and In[1]. These variables can be found within the program itself.

3.5.2 The Programming model of the Hidden Node Data Structure

The data structure of the hidden node was the most complex of the three kinds of nodes. This was because the essence of the entire network were contained in these nodes. Figure 3.9 shows the model of a hidden neuron derived from the data structure.

![Diagram of a hidden neuron with weights and connections](image)

**Figure 3.9** The hidden neuron contains additional structures that belong to other layers.

The actual C program structure used in the program is as follows,
typedef struct node
{
    float Delta_Value;  /*Hidden Layer Delta Value */
    float Net_Value;    /*Hidden Layer Net Value */
    float Output_Value; /*Hidden Layer Output Value */
    float Change_In_Bias, Bias; /*Hidden Layer Biases */
    float Weight_In1, Weight_In2; /*Input Weight Variables */
    float W_In_Change1, W_In_Change2; /*Delta Weight Changes Hidden Layers */
    float Weight_Out, W_Out_Change;
    struct node *next;     /*Creates a linked list */
} node *next;

Note that the data structures provided allows flexibility in the determination of the number of hidden neurons required. Since most of the required variables were found within the hidden node structure itself, any requirements of extra hidden neurons could be arranged by specifying additional hidden node data structures.

The additional nodes can be dynamically allocated as the data structure permits a construction of a linked list (last line "struct node *next" is a potential pointer to the next hidden node). The calculations involved in each hidden node could be done recursively since it was possible to traverse the linked list with short recursive programs. For this reason, most of the functions within the program are short recursive programs.

Note that the bias value for each node is included within the data structure. During the execution of the program, the bias values are changed in the same way as a weight would be changed. The only difference between a weight value and a bias value is that the weight changes are dependent on the input values, i.e. if the input value was a "1", a weight change would occur. If the Input value was a "0", no weight changes would occur. As for the bias value, it will always be altered
according to the Delta values calculated. Thus, the system could afford to have a random beginning and work its way toward a stable state.

3.5.3 The Output Node Data Structure

The output node data structures were fairly straightforward. This is because there was only a single output node to consider. No recursion was necessary as most of the functions involved the manipulation of the hidden nodes.
3.6 Results

The study of the back propagation model was successful and the program written learnt to solve the XOR problem with little difficulty. There was no encounter with any local minima because the networks generated always had 5 or more hidden nodes (This was in line with the suggestions made in Beale R. et. al., (1991).

The program was used to create various network configurations by varying the number of hidden nodes and the value of the random number seed. The system was then used with 36 different cases where the values of $\alpha$, $\eta$ and the number of hidden nodes were varied. These results are tabulated in Tables 3.1 to 3.9. Again, the general trend of the network behaviour was consistent to the findings made by the PDP group (Rumelhart D.E., et. al., 1986).

As the momentum value, $\alpha$, was increased beyond 0.9, the learning system became unstable. While as the momentum value contained a low value, the system took a much longer time to reach its stable states.

It was also noted that as the number of hidden nodes reached beyond 15, the learning performance of the system slowed down, i.e. the number of presentations had to be increased for the system to reach a stable state. This can clearly be seen in the graphical plots shown in Figure 3.10 to Figure 3.12.

Notice also that the momentum term $\alpha$ has a larger influence in the timings for convergence than the learning rate $\eta$. (See results charted in Figure 3.10 to Figure 3.12.)
### Table 3.1: XOR simulator training results for values of $\eta = 0.25$ and $\alpha = 0.50$. The approximation of 1 is any value greater than 0.85 and 0 is any value less than 0.11.

<table>
<thead>
<tr>
<th>XOR Problem with values of $\eta = 0.25$ and $\alpha = 0.50$</th>
<th>Number of Hidden Nodes</th>
<th>Number of Iterations</th>
<th>Random Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1418</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1358</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>939</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1010</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.2: XOR simulator training results for values of $\eta = 0.50$ and $\alpha = 0.50$. The approximation of 1 is any value greater than 0.85 and 0 is any value less than 0.11.

<table>
<thead>
<tr>
<th>XOR Problem with values of $\eta = 0.50$ and $\alpha = 0.50$</th>
<th>Number of Hidden Nodes</th>
<th>Number of Iterations</th>
<th>Random Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>814</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>602</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>390</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>488</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.3: XOR simulator training results for values of $\eta = 0.75$ and $\alpha = 0.50$. The approximation of 1 is any value greater than 0.85 and 0 is any value less than 0.11.

<table>
<thead>
<tr>
<th>XOR Problem with values of $\eta = 0.75$ and $\alpha = 0.50$</th>
<th>Number of Hidden Nodes</th>
<th>Number of Iterations</th>
<th>Random Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>526</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>372</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>279</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>307</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Chart 1: Simulation Results for $\alpha = 0.50$

Figure 3.10  Results plotted from Tables 3.1 - 3.3
Chapter 3  
**The Back Propagation Model**

<table>
<thead>
<tr>
<th>XOR Problem with values of $\eta = 0.25$ and $\alpha = 0.75$</th>
<th>Number of Hidden Nodes</th>
<th>Number of Iterations</th>
<th>Random Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>864</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>647</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>495</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>510</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: XOR simulator training results for values of $\eta = 0.25$ and $\alpha = 0.75$. The approximation of $I$ is any value greater than 0.85 and 0 is any value less than 0.11.

<table>
<thead>
<tr>
<th>XOR Problem with values of $\eta = 0.50$ and $\alpha = 0.75$</th>
<th>Number of Hidden Nodes</th>
<th>Number of Iterations</th>
<th>Random Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>385</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>303</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>203</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>389</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5: XOR simulator training results for values of $\eta = 0.50$ and $\alpha = 0.75$. The approximation of $I$ is any value greater than 0.85 and 0 is any value less than 0.11.

<table>
<thead>
<tr>
<th>XOR Problem with values of $\eta = 0.75$ and $\alpha = 0.75$</th>
<th>Number of Hidden Nodes</th>
<th>Number of Iterations</th>
<th>Random Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>254</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>214</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>154</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>335</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6: XOR simulator training results for values of $\eta = 0.75$ and $\alpha = 0.75$. The approximation of $I$ is any value greater than 0.85 and 0 is any value less than 0.11.
Chart 2: Simulation Results for $\phi^\circ = 0.75$

Figure 3.11  Results plotted from Tables 3.4 - 3.6
<table>
<thead>
<tr>
<th>XOR Problem</th>
<th>Number of Hidden Nodes</th>
<th>Number of Iterations</th>
<th>Random Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>with values of η = 0.25</td>
<td>5</td>
<td>249</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>257</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>186</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>360</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.7: XOR simulator training results for values of η = 0.25 and α = 0.90. The approximation of 1 is any value greater than 0.85 and 0 is any value less than 0.11.

<table>
<thead>
<tr>
<th>XOR Problem</th>
<th>Number of Hidden Nodes</th>
<th>Number of Iterations</th>
<th>Random Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>with values of η = 0.50</td>
<td>5</td>
<td>175</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>114</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>113</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>230</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.8: XOR simulator training results for values of η = 0.50 and α = 0.90. The approximation of 1 is any value greater than 0.85 and 0 is any value less than 0.11.

<table>
<thead>
<tr>
<th>XOR Problem</th>
<th>Number of Hidden Nodes</th>
<th>Number of Iterations</th>
<th>Random Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>with values of η = 0.75</td>
<td>5</td>
<td>106</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>87</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>85</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.9: XOR simulator training results for values of η = 0.75 and α = 0.90. The approximation of 1 is any value greater than 0.85 and 0 is any value less than 0.11.
Chart 3: Simulation Results for $\infty = 0.90$

Figure 3.12  Results plotted from Tables 3.7 - 3.9
Chapter 4

Other Neural Network Models
CHAPTER 4 OTHER NEURAL NETWORK MODELS

The Kohonen network, named after its originator, Teuvo Kohonen (1982), was the result of an inspiration generated through the study of Biological Cybernetics. Unlike the back propagation network of Chapter 3, the Kohonen network is self-organising and requires no target value from which the values of weight changes are derived. The Kohonen network is very different from the normal MLP models, and uses a scheme called learning vector quantization (LVQ). Kohonen's studies involved detailed research into different parts of the brain and biological structures of cortical cell masses. (Kohonen, 1982), (Kohonen, 1989).

4.1 Self-Organising Feature Maps

Recent advances in neural biology have seen the breakthrough of examining the brain without performing surgical operations. With the help of the Positron Emission Tomography (PET) Scan, researchers have been able to observe neural activity of the subject (Kendal, et al., 1991). Evidence shows that the brain functions are generally clustered, and operational when specific parts of the body are activated. To apply this concept to the network, Kohonen makes two assumptions about the biological neural net. These are follows :-

- The formation of an activity cluster is found around the unit at which the activation is maximum (NB: the maximum norm is just one of the possibilities. A minimum norm could also be used because the functions are essentially similar.)

- The change in the input weights of those units is confined within the region where the activity resides.
In his survey of the central nervous system (CNS), Kohonen found that most of the neuron structures were arranged in an orderly two-dimensional fashion. It was also noticed that the cells were stratified within several laminae. Some investigators into the biological neural structure also hold the belief that the cortical cell mass is functionally organised in vertical columns. It seems that such columns are organised around specific afferent axons so that they perform the basic input/output transformation of signals.

There has also been anatomical and physiological evidence that for a given group of neurons, there exists a type of lateral interaction. This interaction is plotted and shown in Figure 4.1. Neighbouring cells closer to the cell in question will experience a positive excitation interaction. Short range lateral excitation reaches up to a radius of 50 - 100 µm and the excitation area is surrounded by a penumbra of inhibition that reaches up to a radius of 200 - 500 µm. A weaker excitation action surrounds the penumbra and reaches up to a radius of several centimetres.

![Interaction Diagram](Figure 4.1 The lateral interaction about an arbitrary point of excitation.)
Thus it is with this Mexican hat lateral interaction curve upon which the Kohonen network bases its learning algorithm. Given that a single node within the mass of neurons obtains superiority based on the given norm, this winning neuron will then generate a lateral interaction which is similar to that shown in Figure 4.1. This form of interaction when operated on a given space would create an interactive effect upon its neighbours. If applied periodically, this coercing stimulus would then form a definite organisation within the mass of neurons which leads to Kohonen's discovery of self-organising maps.

Kohonen then continued to show the self-organisational properties in his network. A set of two dimensional input vectors were selected such that their probability density function was uniform over the area demarcated by its borderlines. The vectors were then randomly selected and presented to the Kohonen network. After some training, the vectors began to organise themselves within the weight vector grid even though the input vectors were randomly presented to the network (Kohonen, 1988).

4.2 The Kohonen Architecture

The generalised architecture of the Kohonen network is shown in Figure 4.2. It consists of an $m$ by $n$ grid array of neurons, where $m$ and $n$ are integers greater than 0. It is therefore possible to have a Kohonen network on a linear array, where either $m$ or $n$ is 1. From Figure 4.2, there are three input nodes in the network and each input node has 20 connections to all 20 nodes on the Kohonen grid. It would thus have 60 different connections between the input nodes and the output nodes. To avoid complex diagrams, the connections of only one input node has been illustrated. Each of the lines shown in the figure represents a weighted connection.
with a randomly initialised value, and each input node has the input value of a given presentation.

![Kohonen grid](image)

**Figure 4.2** The connection architecture between the input nodes and the output nodes in the Kohonen network.

Within the Kohonen grid itself, there exists an inter-nodal connection. This connection is determined by the nearest neighbour topology as seen in Figure 4.3. Note that the architecture of the Kohonen grid may not be limited to that shown in Figure 4.3, Another possible structure is the hexagonal grid shown in Figure 4.4.

![Hexagonal grid](image)

**Figure 4.3** The inter-nodal connections within the Kohonen grid.
4.3 The Kohonen Algorithm

The Kohonen algorithm requires the presentation of an $n$ dimensional input vector, $X$. The input vector $X$ is then connected to each output node on the Kohonen network by a weight vector $W$, of a similar dimension as $X$. It can thus be pictured as a one-to-one relation when comparing vectors in both $X$ and $W$.

The first step of the algorithm is to initialise the weight vectors with small random numbers. This initialisation will provide faster convergence if the weight values were chosen near to the expected convergent value. An example is to have small decimal initial values if the final weights are expected to converge towards either $0$ or $1$. Choosing large integers will not create devastating problems, but only slow the convergence of the system.

Upon initialisation of the weight vectors, a resultant value for each output node is calculated using the minimum Euclidean function. The value of each output node is determined by equation (4.1).

$$d_o = \sum_{i=0}^{n-1} (x_i(t) - w_{io}(t))^2$$  \hspace{1cm} (4.1)

where $d_o$ is the discriminating value, $x_i(t)$ is the $i$th element of the input vector at presentation $(t)$ and $w_{io}(t)$ is the corresponding $i$th element of the weight vector connected to the output node $o$ at presentation $(t)$. 
The node bearing the minimum Euclidean function will then be chosen as the 'winner'. The learning process then begins at this winning node.

Since the aim of the discriminating algorithm is to obtain the node with the minimum Euclidean distance value, it would be sensible that the reinforced node or a neighbourhood node, would be a 'winner' the next time the same input vector is presented. This implies that the reinforcement function would need to reduce the discriminating value between the weight vector $W$ and the input vector $X$. The learning function is shown in equation (4.2) as

$$w_j(t+1) = w_j(t) + \eta(t)(x_i(t) - w_j(t))$$

(4.2)

where $w_j(t+1)$ is the new weight value of the $i$th element on the output node $j$.

The factor introduced by $\eta(t)$ is a decimal value between 0 and 1.

Instead of fully incrementing the value of $w_j(t+1)$ to be equal to the value of $x_i(t)$, $\eta(t)$ controls the strength of reinforcement ($\eta(t)$ shall also be known as the coefficient of reinforcement). Is this coefficient necessary? The answer is yes. Assume that the value of $\eta(t)$ was not placed in the equation, then equation (2) would simply be an assignment statement. With a 100 percent learning rate, the network would then respond only to the specific input vector. This is not beneficial to the generalisation characteristics of any neural net, as a close representation of a specific presentation would not be classified.

To create a generalisation scheme in the network, Kohonen introduced a signal process which performs lateral signal generations. These lateral signals cause the neighbouring nodes of the 'winner' to be updated, veering their vectors closer in the direction of the input vector. The only difference between the neighbouring updates is that a smaller reinforcement coefficient is used as the distance of the node gets further from the winner. If a graph of the reinforcement coefficients was
plotted, verses the distance from the winner, it may look like that shown in Figure 4.5. Again, as the neighbourhood distance increases, the coefficient of reinforcement decreases.

![Coefficient of reinforcement](image)

Figure 4.5 The proportional degrading reinforcement coefficients of the lateral signals forms the key to generalisation in the Kohonen Network.

The effect of these proportional degrading reinforcement coefficients is the formation of a less accurate representation of the input vector. It would thus cause a generalisation effect which provides the network with the ability to classify signals it has never been learnt. To implement this effect, Kohonen devises a very simple, yet effective algorithm. Assuming that there are 10 presentation cycles involved in training the network. At cycle (1), the neighbours with distance less than \( n \) away from the winner will be updated with \( T_1(t) \) where \( 0 \leq T_1(t) \leq 1 \). As the presentation cycles increase to 10, the value of \( n \) would decrease, and the value of \( \eta(t) \) would also be reduced. The effect after 10 cycles, is a terrain which has varying levels of accuracy in the representation of the input vector, with the 'winner' bearing the most precise representation of the input vector. The following section provides a summary of the algorithm. (Lippmann, 1987)
Chapter 4  Other Neural Network Models

Step 1: Initialise Weights
Initialise weights from $N$ inputs to the $M$ output nodes to small random values, as shown in Figure 4.1. Set the initial radius of the neighbourhood.

Step 2: Present New Input

Step 3: Compute Distance of all Nodes
Compute distances $d_\sigma$ between the input and the output nodes $\sigma$ using

$$d_\sigma = \sum_{o=0}^{n-1} (x_i(t) - w_{io}(t))^2$$

where $d_\sigma$ is the discriminating value, $x_i(t)$ is the $i$th element of the input vector at presentation $t$, and $w_{io}(t)$ is the corresponding $i$th element of the weight vector connected to the output node $\sigma$ at presentation $t$.

Step 4: Select Output Node with Minimum Distance
Select node $\sigma^*$ as the output node with minimum $d_\sigma$.

Step 5: Update Weights to Node $\sigma^*$ and Neighbours
Weights are updated for node $\sigma^*$ and all nodes in the neighbourhood defined by $NE_{\sigma^*}(t)$ as shown in Figure 4.6. New weights are

$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)(x_i(t) - w_{ij}(t))$$

For $j \in NE_{\sigma^*}(t) \quad 0 \leq j \leq N-1$

The term $\eta(t)$ is the coefficient of reinforcement ($0 \leq \eta(t) \leq 1$) which decreases in time.

Step 6: Repeat by going to Step 2.

Figure 4.6  The neighbourhoods decrease as $t$ changes
4.4 Implementation

The Kohonen network was used to implement a simple character recognition system for Arabic numerals (i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9). The system constructed had a 4 by 5 grid matrix to capture the bitmap image of the Arabic numeral. The user interface for this capability was written in X Windows, and Figure 4.7 shows the input portion of the system.

4.4.1 The Input Interface

![Image of the input interface](image_url)

Figure 4.7  The user interface input system

In the example showed in Figure 4.7, the system is ready to capture the bitmap image of a '6' on the 4 by 5 grid matrix. Note that on the extreme right of the input section, there are Arabic numeral buttons. The button with the '6' has been highlighted. When the "WRITE" button is pressed, a file called 'Train' will be updated with a one dimensional array of '1's and '0's. This array is read from the 4 by 5 grid matrix, in a left to right direction, beginning at the top left corner and finally making its way down to the bottom right corner. At the end of the array is the actual Arabic value, in this case, 6 is added in. The data expected to be written into 'Train' would be

1111100011110011116
The file 'Train' is first initialised with the bitmap data beginning from 0 and consecutively increasing to 9. A complete sequence from 0 to 9 would be considered as an epoch. In the experiment, 15 epochs were used, which means that 15 sets of 0-9 were used.

**4.4.2 The Output Interface**

The output system was then designed around the input system, utilising the input grid matrix as a means to present test data to evaluate the network. This system is shown in Figure 4.8. However, before the process begins, the network requires initialisation. This is done by pressing the 'INITIALISE' button with the cursor. The network will then go into an initialisation phase, which is indicated by the change of the cursor icon. Upon initialisation, marked by the return of the arrow icon, the system can then begin training. Again, using the cursor, press the button labelled 'TRAIN'. Immediately, coloured squares will begin to populate the 'Winner Migration Screen'. Then portions of the 'Output Node Representation Screen' will be flooded with colour corresponding to the Arabic numeral.
Note, that towards the end of the training, the screen may not show anymore updating, but the cursor will remain at the watch icon. This is because the training area is getting smaller and the update processes are within the bounds of the regions bearing the same colour. It is imperative not to interrupt the process.

Upon the completion of the training, use the cursor to toggle the input grid matrix to obtain the desired bit mapped Arabic numeral. Press the button labelled 'TEST'. The network will then return the trained representation onto the output grid. Within the 'Output Node Representation Screen', a grey square will appear at the region where the winner for the input representation was found.

### 4.5 Results and Observations of the Implementation

The final state of the 'Output Node Representation Screen' after the training process has been completed is shown in Figure 4.9. The colours that were used to represent the numbers are shown in Table 4.1.

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>COLOUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Black</td>
</tr>
<tr>
<td>1</td>
<td>Red</td>
</tr>
<tr>
<td>2</td>
<td>Orange</td>
</tr>
<tr>
<td>3</td>
<td>Yellow</td>
</tr>
<tr>
<td>4</td>
<td>Green</td>
</tr>
<tr>
<td>5</td>
<td>Blue</td>
</tr>
<tr>
<td>6</td>
<td>Medium Purple</td>
</tr>
<tr>
<td>7</td>
<td>Light Blue</td>
</tr>
<tr>
<td>8</td>
<td>Dark Green</td>
</tr>
<tr>
<td>9</td>
<td>White</td>
</tr>
</tbody>
</table>

Table 4.1 The assigned colours for the various Arabic numerals.
Note that in Figure 4.9, the area commanded by the values 5, 6 and 8 are very close in proximity. This is due to the close resemblance of their bitmap images. In fact, the system seems to prefer recognising 5s rather than 6s. To solve problems of this type, it would best be done by increasing the input grid size, thereby adding extra components to discriminate between the two numbers. Note also that the area that represents 8 has been split in two parts. The smaller section still has its uses when it recognises bitmaps in the area.

Of the many results obtained from the individual number sets, only some are documented here. The images show that all have been successful in identifying the number shown in the box on the bottom right corner of each diagram. The training set used for all 15 training cycles is shown in Figure 4.10.
Figure 4.10  The training set used in 15 cycles.

Figure 4.11  Successful identifications of 0.

Figure 4.12  Successful identifications of 1

Figure 4.13  Successful identifications of 2

Figure 4.14  Successful identifications of 3

Figure 4.15  Successful identifications of 4
Figure 4.16 Successful identifications of 5

Figure 4.17 Successful identifications of 6

Figure 4.18 Successful identifications of 7

Figure 4.19 Successful identifications of 8

Figure 4.20 Successful identifications of 9
Although the Kohonen network operates on such simple principles, it has proved its effectiveness as a self-organising network. The results of this study show the following advantages.

- The training process requires considerably less training cycles than that expected by the back propagation network.

- The Kohonen training process is quick and can be left to learn by itself, unlike the back propagation counterpart that has to be carefully trained.

- The results of the tests performed on the system showed that the network had developed a substantial generalisation capability.

- The simplicity of the algorithm and equations both help in the reduction of implementation and training times.
4.6 Adaptive Resonance Theory (ART1)

The Adaptive Resonance Theory was a novel contribution into the field of neural networks, by Professor Steven Grossberg who created the network based on the Psychological aspects of the brain. The network he modelled had signals resonating back and forth between the architectural layers. As the signals did so, the weight system within the network would change its values to suit the prevailing input patterns, hence the name Adaptive Resonance Theory (Grossberg S, 1987), (Lippmann. R.P., 1987).

In the actual paper, Professor Grossberg vividly relates the ART1 system to the psychological studies of his field, providing naming conventions such as the "Long Term Memory" traces (LTM traces) and the "Short Term Memory" traces (STM traces). The original discussions are complex and tedious due to the use of unfamiliar terminology. Seeking out the actual algorithm within the literature can be quite illusive. The algorithm was eventually deciphered and simplified by Richard Lippmann (Lippmann. R.P., 1987). This thesis avoids the complex terminology used by Grossberg and utilises the notation introduced by Richard Lippmann. Lippmann's notations are of a more familiar convention which simplifies the understanding of the model. Full details of the original ART1 model can be found in Grossberg, S, (1987).

4.7 The Architecture of the ART1 Network

The most unique structure of the ART1 model is perhaps its ability to incorporate new patterns by increasing the number of storage nodes. The input vector of the system is of a fixed length, while the output nodes have the capability to increase in numbers. The structure of the simplified system is shown in Figure 4.21. This
simplified version omits several features explained in the original ART1 model, but the basic principles of the model still remain.

The input vector $I$ is first introduced to the network. This input signal will then be converted into the network representation, $T$ and transmitted to the output layer. In the output layer, each node contains a stored pattern which produces an output vector $Y$. The transmitted signals $T$ are then multiplied, in the bitwise form, by $Y$. Each output node would then produce a summed value obtained from the multiplication, which will be used as a means to a competition amongst all the output nodes. The winner of the competition will be used in a vigilance comparison to determine its validity. A ratio comparison between the bitwise summation of the gated function, in the input-output layers and the output winning node is used to determine if the threshold of difference has been breached. If the threshold has been exceeded, the node is temporarily disabled and the next winning node is selected for the comparison. In the following section, the ART1 network algorithm is presented to support the explanation provided above.
4.8 The ART1 Algorithm

The initialisation of the ART1 network begins with the \( n \) element input node \( I \). There are no output nodes to begin with. Since there are no output nodes, the initial inputs must not be disturbed by the top down comparisons. The initial values of the top down signals are then set to 1.

Step 1: Initialise

\[
\begin{align*}
    t_{ij} &= 1 \\
    w_{ij} &= \frac{1}{1+N}
\end{align*}
\]

for all \( 0 \leq i \leq N-1 \) and \( 0 \leq j \leq M-1 \)

Set \( \rho \), where \( 0 \leq \rho \leq 1 \)

Here \( t_{ij} \) is the top down signal between node \( i \) and node \( j \) at time \( t \) and \( w_{ij} \) is the bottom up connection weight between node \( i \) and node \( j \) at time \( t \). \( M \) is the number of output nodes and \( N \) is the number of input nodes.

Step 2: Get next input node

Step 3: Compute sum \( \mu_j \) defined as,

\[
\mu_j = \sum_{i=0}^{N-1} w_{ij} x_i
\]

for all \( 0 \leq j \leq M-1 \)

Here \( \mu_j \) is the summed value obtained in the output node \( j \) and \( x_i \) is the \( i \)th element of the input vector \( I \).

Step 4: Based on the maximum selection criteria, select \( \mu_{\text{winner}} = \max_j [\mu_j] \)

Step 5: Evaluate the quantities

\[
\begin{align*}
    \| X \| &= \sum_{i=0}^{N-1} x_i \\
    \| T \cdot X \| &= \sum_{i=0}^{N-1} t_{ij} \times x_i
\end{align*}
\]
and test if \[ \frac{\|T-X\|}{\|X\|} > \rho \] if YES, go to Step 7 otherwise go to Step 6

Step 6: Disable winning node and Goto 3.

Step 7: Adapt best match into system by evaluating

\[ t_{ij \text{ winner}} = t_{ij} \cdot x_i \]
\[ w_{ij \text{ winner}} = \frac{t_{ij} \cdot x_i}{0.5 + \sum_{i=0}^{N-1} t_{ij} \cdot x_i} \]

Step 8: Repeat by enabling any disabled nodes, then go to 2.

4.9 Learning behaviour of ART1

From the algorithm, it is still difficult to perceive the learning behaviour of the ART1 model. The pattern learning sequence in Figure 4.22 can be found in Grossberg's paper. (Grossberg S.,1987). Four unique patterns are used to train an ART neural network. The exemplars stored after each presentation is recorded and the ART learning characteristics are clearly seen.
From the practical implementation of the algorithm, it was found that the ART network was an efficient learning model. The desirable characteristics of the model are listed below.

- Able to accommodate new patterns by increasing the number of output nodes
- Able to generalise using the vigilance testing mode
- Weight modification process was not as complex as the other MLP models

Although the learning process was considerably faster than the MLP models, the ART model had some features that were not appealing.

- The learning process would sometimes overwrite stored exemplars.
- The vigilance factor is in direct conflict with generalisation. Sometimes, this cannot be balanced. Such a case was highlighted by (Lippmann R.P., 1987).
Several methods of overcoming the dilemma of the vigilance and generalisation have been suggested. One such method was to increase the vigilance when unique patterns with similar structures are processed. Once the patterns are learnt, the vigilance is then relaxed for better generalisation. As for the problem of overwriting stored patterns, this is an operational characteristic of ART1.

This section serves as an introduction to the ART1 network. A further discussion on the ART1 is provided in the next chapter to compare it to the new architecture of Fast Learning Artificial Neural Network (FLANN) model.

4.10 The Hopfield Network

In 1982, John Hopfield, a distinguished physicist proposed a design of a delocalised content addressable memory or categoriser using extensive asynchronous parallel processing. In his paper (Hopfield, 1982), he provided the following example. Suppose an item stored in memory is "H. A. Kramers and G. H. Wannier Phys. Rev. 60, 252 (1941)". A general content addressable memory would be capable of retrieving this entire memory item on the basis of sufficient partial information. The input "Wannier (1941)" may suffice. He also mentioned that an ideal memory could deal with errors such as "Vannier (1941)". Hopfield builds his model based on the bi-stable neuron similar to that of McCulloch and Pitts (McCulloch et. al, 1943).

4.11 The Architecture of the Hopfield Network

The Hopfield network consists of a fully connected network with bi-directional links that contain the weight values. This is shown in Figure 4.23. The individual neurons are very similar to the perceptrons. Each has a hard limiting threshold, making the output values take on two states. The system can be designed to take
on a binary value (0 and 1) or a bipolar value (-1 and +1). In the system, only the states of the neurons change. The weights are kept at a constant value. The state of the system is specified by listing the states of the neurons represented by a binary word of length N, where N is the number of neurons in the system.

![A fully connected Hopfield Net with weighted bi-directional links](image)

Figure 4.23 A fully connected Hopfield Net with weighted bi-directional links

Referring to Figure 4.23, activation of any single node $x_i$ is determined by the value of $\sum_{j \neq i} w_{ij}x_j$. Like the McCulloch and Pitts model, activation occurs when this value exceeds a threshold value $T_i$. Two rules can then be formalised and they are as follows:

\begin{align*}
\text{If } \sum_{j \neq i} w_{ij}x_j > T_i, \\
\text{then } x_i \text{ becomes 1 } & \quad \text{(4.3)} \\
\text{or } \sum_{j \neq i} w_{ij}x_j < T_i, \\
\text{then } x_i \text{ becomes 0 } & \quad \text{(4.4)}
\end{align*}

With these basic activation equations at the node level operations, further equations were used to perform the learning properties, at the network level. The central feature to the Hopfield analysis is the association of each network state, which he called $E$ (for energy). With similar objectives of the error back
propagation neural network models, Hopfield networks have the task of minimising the total energy of the network each time it experiences a state change.

Denoting a change in the energy \( E \) by the symbol \( \Delta E \). Each time a firing rule is applied, there are two possibilities. The first is when \( x_i \) changes to 1 and the other is when \( x_i \) changes to 0 (see (4.3) and (4.4)).

In order for \( x_i \) to become a 1, from equation (4.3), a change in \( x_i \), denoted by \( \Delta x_i \), must be positive and the activation \( \sum_{j \neq i} w_{ij}x_j - T_i \) must be positive. In order for \( x_i \) to become a 0, \( \Delta x_i \) must be negative and the activation \( \sum_{j \neq i} w_{ij}x_j - T_i \) must also be negative. In both cases, the product of \( \Delta x_i \left( \sum_{j \neq i} w_{ij}x_j - T_i \right) \) will result in a positive value. Hopfield then chooses minimising \( \Delta E \) where

\[
\Delta E = -\Delta x_i \left( \sum_{j \neq i} w_{ij}x_j - T_i \right).
\]

The total energy in the system would then be given by;

\[
E = -\frac{1}{2} \sum_{i} \sum_{j \neq i} w_{ij}x_j x_i + \sum_{i} x_i T_i \quad (4.5)
\]

Storage of patterns into the Hopfield network would imply that any introduction of new patterns would require a generation of minima points in the energy landscape created by equation (4.5). Since the generation of minima points on the landscape is equivalent to minimising the function, the equation has to be further minimised upon the introduction of new patterns. This will involve making the value of equation (4.5) to become more negative. The second term does not help in this
minimising process, and to remove it from the equation, the threshold, $T_i$, can be set to 0. This leaves the term:

$$E = -\frac{1}{2} \sum_i \sum_{j \neq i} w_{ij} x_j x_i$$

which can also be written as

$$E = -\frac{1}{2} \sum_i \sum_{j \neq i} w_{ij}^* x_j^* x_i^* - \frac{1}{2} \sum_i \sum_{j \neq i} w_{ij}^s x_j^s x_i^s$$

where the terms with the superscript "*" indicate other patterns and the term with the superscript "s" indicate the special pattern of interest. Since there is little control over the "*" terms, each new pattern has to be responsible for making its own minima on the energy landscape.

Concentrating on the special term of interest,

$$E_s = -\frac{1}{2} \sum_i \sum_{j \neq i} w_{ij}^s x_j^s x_i^s$$

should be made as small as possible, which means that the value of

$$\sum_i \sum_{j \neq i} w_{ij}^s x_j^s x_i^s$$

must be as large as possible. Since the $x_j$ and $x_i$ are bipolar, these terms can be negative. To maximise the result, the appropriate step to take, is to set the equation equivalent to

$$\sum_i \sum_{j \neq i} w_{ij}^s x_j^s x_i^s = \sum_i \sum_{j \neq i} (x_j^s)^2 (x_i^s)^2$$
which in fact equates the weight factor $w_{ij}$ to be equal to the summed product of the two pattern vectors. Thus the explanation for the weight assignment equation is derived.

$$w_{ij} = \sum_{s=0}^{n-1} x_i^s x_j^s$$  \hspace{1cm} (4.6)

where $w_{ij}$ is the weight connection between the $i$th node and the $j$th node. $s$ is the exemplar pattern, beginning at 0 and $n$ is the number of exemplar patterns. $x_i^s$ and $x_j^s$ are the output states of the $i$th node and $j$th node at the presentation cycle of the $s$th exemplar. The values of $i$ and $j$ are such that $i \geq 0$ and $j \leq n$. If $i = j$, then $w_{ij} = 0$.

Unfortunately, although equation (4.6) does minimise the total energy $E$ for the pattern $s$, it does not prevent the addition of spurious states into other patterns. The problem with these spurious states are examined by analysing some examples in § 4.13.

4.12 The Hopfield Network Algorithm

The Hopfield algorithm is presented here. The entire algorithm is quite simple to implement, but the effectiveness of the network as a classifier needs to be studied. This is done in the sections that follow.

Step 1: Assign connection weights using equation (4.6)

$$w_{ij} = \sum_{s=0}^{n-1} x_i^s x_j^s$$  \hspace{1cm} (4.6)  \hspace{1cm}  \text{(copy of 4.6)}

Step 2: The second step of the algorithm initialises the nodes with an unknown pattern. The output of each individual node $i$ can be written with respect to the $t$th time presentation. This is shown in equation (4.7).
\[ \mu_i(t) = x_i \]  \hspace{1cm} (4.7)

where \( \mu_i(t) \) is the output of node \( i \) at presentation \( t \).

Step 3: The final step of the algorithm is iterative in nature. It requires the output value of the pattern to be iterated until a stable state is reached. The next output state is given by the function in equation (4.8). Figure 4.24 shows the hard limiting function \( f_h() \) reflected in the equation, where \( f_h(x_i) = 1 \) if \( x_i > 0 \) and \( f_h(x_i) = -1 \) if \( x_i < 0 \).

\[ \mu_i(t+1) = f_h \left( \sum_{j=0}^{n-1} w_{ij} \mu_j(t) \right) \]  \hspace{1cm} (4.8)

where \( n \) is the number of nodes in the network, and \( f_h() \) is the hard limiting function shown in Figure 4.23. The other variables are similar to the labelling conventions adopted in equations (1) and (2).

![The hard limiting function used in the discrete Hopfield Net](image)

The important issues now encompass the storage of information within the weight matrix of equation (4.6). Given an exemplar pattern \([1 1 0 0 1]\), the resultant weight matrix is shown in Figure 4.25. Being a 5 bit word, this would constitute to a 5 node Hopfield network. Note that because the weight links are bi-directional, the matrix then becomes symmetric.
At the positions of the matrix where $i = j$, the value is set to 0. During the evaluation of equation (4.6), the zeros are transformed to bipolar values to avoid zeroing of products. The exemplar's binary value has now been stored within the weight matrix. Thus, the system will be expected to converge toward the value $[11001]$ after iterating through the algorithm explained in § 4.12. To test the system, the binary word $[11000]$ is used as an unknown pattern to be tested on the matrix.

$$
[11000] \times \begin{bmatrix}
0 & 1 & -1 & -1 & 1 \\
1 & 0 & -1 & -1 & 1 \\
-1 & -1 & 0 & 1 & -1 \\
-1 & -1 & 1 & 0 & -1 \\
1 & 1 & -1 & -1 & 0
\end{bmatrix} = [1 \ 1 \ -2 \ -2 \ 2]
$$

Performing the simple multiplication, the raw results are $[1 \ 1 \ -2 \ -2 \ 2]$. This is before the values are passed through the hard limiting function shown in Figure 4.23. After using the hard limiting function, the next state values are $[1 \ 1 \ 0 \ 0 \ 1]$. If this value is again iterated as required by the algorithm, the raw results are $[2 \ 2 \ -3 \ -3 \ 2]$. The hard limiting system will produce the next state values as $[1 \ 1 \ 0 \ 0 \ 1]$. This in fact is a stable state, which means that the system will always remain in $[1 \ 1 \ 0 \ 0 \ 1]$. This is how Hopfield stores exemplar patterns within the weight vector. The weight vectors are therefore fixed and will not be altered throughout the process of iterations.
There are many ways of implementing the Hopfield net and the procedure described here is the synchronous model. Whether it is the synchronous or asynchronous model, the basic underlying principles are similar. The next question to consider is whether the storage principles used in the Hopfield model suffice in a pattern recognition environment.

### 4.13 The Feasibility of the Storage Capacity

Hopfield in his paper (Hopfield, 1982) mentions the limitation of the pattern storage capacity. In his experiments, he provides a value of 0.15N at 90% accuracy (where N is the number of nodes and the value 0.15N is the number of exemplar patterns expected to contain). This value is extremely small, providing a very poor storage capacity. For storing the 10 Arabic numerals, there must be at least 70 nodes. Even if the 70 nodes were provided, there was still no guarantee of reaching the correct stable states. Some experiments were conducted to examine the severity of this storage problem.

#### 4.13.1 Experiment 1

In the examples that follow, exemplars are used to create the weight matrices, after which, an exhaustive search is made for the stable states. The exhaustive search is done by multiplying every possible combination of the 5 bit binary word with the matrix to see if it resulted in the same binary word. The exemplar used was [1 1 0 0 1]. The weight matrix was shown in Figure 4.25 of the previous section.

After the search was completed, a total of 3 stable states were discovered. These were [1 1 0 1 0], [0 0 1 1 0] and [0 0 0 0 0]. The last stable state appears in every matrix. Interestingly, the second stable state is the inverse bitmap image of the original exemplar. This means that the inverse image cannot be used as another
exemplar. The existence of additional stable states imply that the system has actually created unwanted hollows. For an accurate pattern recognition neural network, this characteristic cannot be compromised.

4.13.2 Experiment 2

In this experiment, two 5 bit binary words were stored into a 5x5 matrix as specified in Hopfield's algorithm. They were \([1 \ 1 \ 0 \ 0 \ 1]\) and \([1 \ 0 \ 1 \ 1 \ 1]\). The resultant matrix was

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 2 \\
0 & 0 & -2 & -2 & 0 \\
0 & -2 & 0 & 2 & 0 \\
0 & -2 & 2 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

After performing the exhaustive search for stable states, a set of 4 stable states were discovered. These were, \([1 \ 0 \ 1 \ 1 \ 1]\), \([1 \ 0 \ 0 \ 0 \ 1]\), \([0 \ 0 \ 1 \ 1 \ 0]\), \([0 \ 0 \ 0 \ 0 \ 0]\). A confusion occurs within the matrix which prevents the first binary word exemplar, \([1 \ 1 \ 0 \ 0 \ 1]\) from ever being recognised again. There were now 3 other stable states which have 'crept' into Hopfield's exemplar storage matrix. This shows that the system fails to operate correctly as a pattern classifier.

4.13.3 Experiment 3

In the third experiment, three binary words were chosen to reside within a 5x5 matrix. These were \([1 \ 0 \ 0 \ 1 \ 1]\), \([1 \ 1 \ 0 \ 0 \ 0]\) and \([1 \ 0 \ 1 \ 0 \ 1]\). The resultant matrix was

\[
\begin{bmatrix}
0 & -1 & -1 & -1 & 1 \\
-1 & 0 & -1 & -1 & -3 \\
-1 & -1 & 0 & -1 & 1 \\
-1 & -1 & -1 & 0 & 1 \\
1 & -3 & 1 & 1 & 0 \\
\end{bmatrix}
\]
This time, none of the original exemplars were found within the stable states of the resultant weight matrix. The stable states were \([1 \ 0 \ 0 \ 0 \ 1], [0 \ 0 \ 0 \ 1 \ 1], [0 \ 0 \ 1 \ 0 \ 1] \) and \([0 \ 0 \ 0 \ 0] \).

Hopfield does mention some of these pitfalls within the matrix storage structure. Other researchers in the area have also discovered similar faults and some have found ways of improving the storage efficiency (Wilson et. al, 1988), (Bruck et. al, 1988). Some have introduced a continuous Hopfield system, which makes its learning appear very much like the conventional MLP model. A statement quoted from a book (Alianna Maren et. al, 1990) states "The primary reason that the Hopfield network has attracted so much attention is that several researchers have shown that it is a relatively easy network to build in hardware using currently available VLSI technology." This would probably be one of the reasons for the ready acceptance of the algorithm.

### 4.14 Conclusions for the Hopfield Model

Initial studies of the Hopfield network show that the results, reflect a less promising image of the Hopfield network than what is prevalent in the field of neural computing. For this reason, the efforts of this research was redirected to other more promising models in neural computing for pattern recognition.

The deficiencies of the pattern storage function is probably the most detrimental of the faults in the Hopfield network. The inconsistent results in simple simulations make it difficult to entrust the network with more complex systems. Taking the simple example mentioned in experiment 1, by initialising the storage matrix with one exemplar, 3 stable states were created. Two of which were local minima. Such unstable behaviour cannot maintain an accurate and reliable pattern recognition system.
The 0.15N storage factor for the Hopfield network, cannot even be guaranteed as it would very much depend on the inter-pattern relationships. This uncertainty blurs the hope further, of a stable classification system. For these reasons, the studies pertaining to the Hopfield network for pattern recognition were terminated.

Although the Hopfield has not worked in the area of pattern recognition, it has been known to be useful for the area of Global Optimisation, which is not in the scope of this thesis. This perhaps accounts for the persistence of Hopfield researchers in the field neural networks.
Chapter 5

Fast Learning Artificial Neural Network I (FLANN I)
CHAPTER 5  FAST LEARNING ARTIFICIAL NEURAL NETWORKS (FLANN I)

Before proceeding with the discussion of the new neural network models, the objective of this thesis is recalled from chapter 1. As stated in Section 1.4,

"It is the objective of this thesis to develop a neural network that is fast learning, able to generalise and achieve good capacity to discern different patterns even though some patterns may be similar in structure. This eventual neural network will be used in the pattern classification environment."

The first inspirations of developing a fast learning neural network came during the analysis of the original ART1 neural network in Chapter 4 (see §4.6 - §4.9). Several functional aspects of ART1 were extracted and a modification of the original algorithm was necessary to stabilise the network's learning behaviour. This eventually produced a binary input neural network model that behaved differently from the original ART1 model. The resultant neural network model was able to function like ART1 and in addition, maintain a constant learning rate of 1 epoch. Due to this expedient learning behaviour, it was given the name "Fast Learning Artificial Neural Network" (FLANN I).

The process of examining the ART1 algorithm involved the actual coding of an ART1 neural network model to perform a pattern recognition task on a simple model problem. From this study, it was an opinion that the most fragile section in the ART1 learning algorithm was the one sided comparison between the vector \( \mathbf{I} \) in the input system and the vector \( \mathbf{O} \) in the output system (§ 4.7). This comparison is better known as vigilance testing. This comparison takes place during the "Top-Down" pattern comparison stage of the ART1 algorithm. Using the vigilance
testing shown in equation (5.3), the input vector $I$ is tested for similarities with the stored pattern $Y$. If the input vector $I$ was a subset of the output vector $Y$, the vigilance testing will allow $I$ to overwrite the storage space of $Y$. If the overwritten vector, $Y$, was a required exemplar, a second epoch is needed to train the network again with another presentation of $Y$ to reinstate its presence as a stored pattern.

This overwriting effect is illustrated in the sequence shown in Figure 5.1. Step 1 in the sequence shows the learning of the first vector, where the pattern is stored in position 1. In step 2, a new pattern is presented, but since this new vector is a subset of the first vector, the one-sided comparison is unable to realize that the patterns are vastly different. The result is the overwriting of the vector stored in step 1. The first pattern no longer exists in the system. To bring the first pattern back into the system, step 3 is required, allowing the first presented pattern to become reinstated. Since the reinstated pattern is no longer a subset, but a super set of the existing pattern in storage 1, it does not perform any overwriting, but occupies a new storage space in 2.

![Figure 5.1](image-url)

**Figure 5.1** One-sided comparisons in ART1 results in the overwriting of unique exemplar pattern
The Fast Learning Artificial Neural Network (FLANN I) takes on the likeness of the ART1 neural network model, but incorporates an extra vigilance testing stage. In addition, new weight assignment equations are also introduced. Thus the resultant FLANN I model bears resemblance to the ART1 model, but possesses distinct differences in the system as a whole.

Using another example, the network is tasked to learn the alphabet from A - J. Results show that the FLANN I was able to learn all 10 patterns after being taught once (See § 5.6). FLANN I can also instantly re-learn previously stored patterns without affecting other stored patterns. In addition to the fast learning speed, it has a noise tolerant mechanism that operates without jeopardising the learning speed.

The differences between the FLANN I model and the ART1 network are brought out in the description of the operational functions and the architectural structures of the two networks. Some of the basic qualities, which will be discussed later in § 5.11, found in FLANN I are listed as follows :-

- Fast learning, instant teachability
- Stable pattern storage
- Simple implementation
- Direct addressing of patterns
- Able to perform immediate re-coding
- Addresses noise problems
- Parallel implementation
5.1 The Architecture

The architecture of FLANN I is shown in Figure 5.2. The input layer \( I \), has a fixed number of nodes, which is determined in the early stages of the network design. The size of the input layer should always be determined prior to programming the neural network so that the applications programmer/ engineer can create the correct input data sizes in the software/ hardware. The output layer, \( Y \), is a constant growing layer with its capacity limited only by the amount of hardware memory available. The neural network begins operations with an initial state of no output nodes. As the patterns are presented by the neural network trainer, the network will dynamically create the output nodes to accommodate new patterns. This process of dynamic node allocation will continue as long as there is available hardware memory space within the computing system.

![Diagram of FLANN I Architecture](image)

Figure 5.2 The Architecture of FLANN I

Unlike the Kohonen network and the MLP models, where the information is spread through an entire region of the neural network's processing nodes, FLANN I relies heavily on individual nodes as storage elements. This dependence on a single node
is sometimes known as a 'grandmother cell' network system (Beale R., et. al., 1991). (Each output node could perhaps be visualised as a cluster of nodes tuned for a specific operation). Connecting the two layers of input and output nodes, are weighted connections that are used to determine the winning node using the equation,

\[
\text{winning\_node} = \max \left[ C_j \sum_{i=0}^{n-1} W_{ij} I_i \right].
\]  

(5.1)

where \( W_{ij} \) is the weight between input node \( i \) and output node \( j \) and \( I_i \) is the state of the input node \( i \). \( C_j \) is the comparator response for node \( j \).

Based on the winning node selected, the input vector, \( I \), and the selected output node vector, \( Y \), are used to generate a filtered response \( V \). The function governing the filtered response \( V \), is a bitwise AND function performed on the vectors \( I \) and \( Y \). This is expressed mathematically in (5.2).

\[
V = I \cap Y
\]  

(5.2)

where \( I \) is the input vector and \( Y \) is the output response. Note that this is a bitwise operation.

Vigilance factors \( \rho \) and \( \alpha \) are values originally set by the network designer. These ratios are used as instruments to check the data consistency of \( I \) and \( Y \). Data consistency is checked by comparing the user set ratios with the ratios obtained from \( |I| \) divided by \( |I| \) and also \( ||I| \) divided by \( |I| \). The details of these tests will be described later. Eventually, based on equation (5.1), the system will determine if a match is found. If the vigilance tests are successful, the bit value of \( C_j \) is set to 1, otherwise, it is set to 0. \( C_j \) takes the role of a qualifying variable for the equation. During initialisation, it is set to 1, providing fair competition between all nodes on the output layer.
5.2 Differences in Vigilance Factors

As mentioned earlier in the introduction of this chapter, the ART1 system uses a single vigilance factor to perform vector comparisons, which causes the ART1 network to overwrite other unique vectors. The consequences of this is seen in Figures 5.8 and 5.9. In Figure 5.8, the vigilance factor was set at 0.5. The resulting stored images after the first cycle of 10 patterns reveals that only 3 patterns are retained. In Figure 5.9, when $p$ was set at 0.8, 5 out of 10 patterns are stored after the first epoch. From our experiments, it was noted that some patterns just cannot be uniquely learnt.

Such an example is to teach the system to recognise A and B with a vigilance factor of $p = 0.5$. Presenting A will make the system learn A. Presenting B will make the system alter A into a B. When the system is again presented with an A, it will re-code the stored prototype of B as an A again, making it a vicious cycle. (Grossberg proposed to increase the vigilance until the exemplar is accepted and then reduce the vigilance again.) The learning system may require several cycles to stabilise and the introduction of new patterns may cause a disruption of stored patterns, requiring the network to be re-trained. Although the ART1 has a relatively fast training speed, it nevertheless has to be re-trained if conflicting patterns are introduced. A more serious consequence happens when one becomes uncertain of what the network has and has not learnt.
5.2.1 ART1 Vigilance Factor $\rho$

The vigilance testing procedure of the ART1 is now introduced with the help of an example. The pattern vector in Figure 5.3 is presented to a newly initialised ART1 neural network. The filled bits represent "1" and the blank bits represent "0". Since this is the first pattern, the ART1 network will learn the pattern as a novel exemplar.

![Pattern 1](image)

$|Y| = 4$

Figure 5.3 1st pattern to ART1

After the first pattern has been absorbed into the ART1 neural network, a second vector shown in Figure 5.4 is presented to the ART1 neural network. By now, the first stored vector will be in the output node $Y$ and the current new vector will be in the input node $I$.

![Pattern 2](image)

$|I| = 1$

Figure 5.4 2nd pattern to ART1

The value of $V$ is given by $Y \cap I$, which results in 1. The vigilance testing mode requires a testing using equation (5.3), where $\rho$ is a preset value $0 \leq \rho \leq 1$ (Lippmann, R.P., 1987).
\[
\frac{|V|}{|I|} \geq \rho 
\]  
\hspace{1cm} (5.3)

\(\rho\) is the vigilance threshold, \(I\) is the input vector, \(Y\) is the output stored vector, and \(V\) is the result obtained with the function \((I \cap Y)\).

Note that the calculation of \(\rho\) from the two pattern sequences results in the value of \(\frac{|V|}{|I|}\) to be equivalent to 1. Since the vigilance factor is set to \(0 \leq \alpha \leq 1\), this second pattern is accepted as a similar pattern to the first. The resonating algorithm in the ART1 causes the first pattern to be overwritten by the second. This is a problem because the two patterns differ with a Hamming distance of 3 (For explanation of Hamming distances, see § 6.1), which is 50\% of the vector size, yet the vigilance testing maintains that the patterns are 'perfect' matches. This illustrates the ineffectiveness of the one-sided comparison.

5.2.2 FLANN I Vigilance Factors \(\rho\) and \(\alpha\)

In comparison, the vigilance testing in the FLANN I is quite different. The single vigilance comparison made by the ART1 leaves room for frequent oscillations within the stored patterns of the neural network. To restrict this "learning and forgetting" syndrome, FLANN I introduces another vigilance factor to stabilise the system. This additional vigilance factor is shown in equation (5.4). The neural system now takes the output vector \(Y\) into consideration during the comparison of the filtered response \(V\). This results in a double-sided comparison which is more accurate. The additional vigilance factor restricts the network from making one-sided comparisons, which causes the volatile changes apparent in the ART1 learned prototypes. The basic functions of the system are quite similar to the ART1, but now, the system has a more stringent form of comparison. A match would have to show consistency between the filtered response \(V\) and the two other vectors \(I\) and \(Y\).
\[ \frac{|V|}{|Y|} \geq \alpha \] (5.4)

\( \alpha \) is the vigilance threshold, \( I \) is the input vector, \( Y \) is the output stored vector, and \( V \) is the result obtained with the function \( (I \cap Y) \).

By using the vectors shown in Figures 5.3 and 5.4, the values of \(|V|\), \(|Y|\) and \(|I|\) remain unchanged, but the vigilance testing performed in (5.4) results in \( \frac{|V|}{|Y|} = \frac{1}{4} \).

This will fail the tests if \( \alpha \) was set correctly (It is normal practice to set vigilance values greater than 0.5 and lesser than 1.) Unlike the ART1 network, FLANN I prevents any re-modification of stored inputs. Thus, even if a new pattern is similar to an already established pattern, the system will not change the already stored pattern. This leaves the system stable with no possibilities of unmonitored overwriting.

### 5.2.3 Forced Learning

During the training of the ART1 neural network, exemplars possessing close proximity Hamming distances to patterns already stored, could not be retained as new exemplars. To overcome this problem, Grossberg proposed a tightening of the vigilance factor, so that the system would become sensitive to the slightest difference in patterns, causing a retention action in the network. In essence, the trainer is required to intervene and allow the system to learn the pattern. Once the network is trained, the vigilance factors are relaxed again to allow the network to perform better generalisations. This problem of balancing the learning threshold and generalisations are sometimes known as the "Learning verses Generalisation" dilemma. This is caused by the inconsistent structure of patterns that humans satisfy themselves with. To state simply by using examples, the alphabets from A - Z were not created so that each character was a fixed Hamming distance apart
Chapter 5  
Fast Learning Artificial Neural Network (FLANN I)

from another. The full set of Chinese characters were not based on any difference measures. The neural network should therefore have the ability to latch onto any application and perform efficiently. Such a need for user intervention is not caused by a flaw in the neural network, but the inconsistent environments the neural network is required to operate in. The ART I therefore possesses the excellent ability to bend its learning rules to accommodate these special cases.

FLANN I possesses similar controls over the "Learning versus Generalisation" dilemma. The basic assumption is that the trainer is knowledgeable in these matters, and provides the relevant information when the neural network fails to accept a novel pattern. Unlike the ART I network, where the vigilance factors are manipulated before and after training, FLANN I bypasses the vigilance factors during a trainer intervention cycle. Although this process bypasses the vigilance checks, the neural network still maintains a recollection of all stored patterns. If the new pattern is not present in the pattern storage, the neural network receives it as a new node, adding it to the pattern library. This pattern audit is necessary to assure that no two 100% identical patterns are stored in the neural network as different exemplars.

This feature of forced learning is equivalent to the classroom learning scenario where the student learns a new concept, but confuses it with closely related concepts. The teacher with this knowledge can clarify the matter and the student learns to separate the concepts. FLANN I allows the user intervention to tell the network to distinguish closely related patterns. FLANN I is thus a hybrid of a self-learning system and a teachable system. Due to this teachable feature, the vigilance factors can remain as low as desired, allowing the system to handle high noise intensities.
5.2.4 Immediate Re-Coding

With a neural network like FLANN I, the training tasks of the user is simplified. One does not need to worry that stored patterns are overwritten and by comparing with the MLP training scheme, FLANN I training is extremely simple. Unfortunately, it is possible that wrong pattern training may occur due to human error. Occasionally, the trainer may accidentally associate the image pattern with a wrong assignment during the training process. In networks such as the MLP model, this may require a slightly longer process of training to arrive at the correct global minima. In the Kohonen Network, one may require a full retraining of the network. The ART1 model may require a thorough examination of the stored exemplars to confirm the eradication of the error. In the FLANN I system, immediate re-coding is available. This is done by presenting the exact pattern as it was stored and a corrected assignment is accompanied with the correction. Upon receiving an exact same pattern, the neural network will replace the old information with the new updated information. This option provides the trainer with the flexibility of correcting errors. In certain applications, patterns may possess different meanings and different points in time. This immediate recoding feature again becomes a unique asset of the FLANN I network.

5.2.5 The ART1 within the FLANN I

The similarities between the ART1 model and the FLANN I model are numerous, but as seen from the previous sections, FLANN I learns in a way that is very different from the ART1 model. Some of the literature surveyed have shown that the ART1 model is designed for applications that require sensitivity to the prevailing environment. As such, the volatility of stored exemplars in the ART1 can be justified. If indeed there exists a need for FLANN I to perform similar to the ART1 model, it can be achieved by reducing the second vigilance factor (§ 5.2.2)
in the FLANN I model to 0. This modification would change FLANN I into the ART1 model, although mathematically some calculations differ, functionally, the models perform identically.

5.3 Weight Manipulation

The ART1 system uses the weight manipulation scheme shown in equation (5.5) (Lippmann R.P., 1987).

\[
W(t + 1) = \frac{T(t) * I}{0.5 + \sum T(t) * I}
\]  

(5.5)

W is the weight assigned, T is the top down LTM traces, I is the input. The decimal value is placed in the equation to perform subset and superset discrimination.

A closer examination of the numerator of equation (5.5) shows that the value of \( W(t+1) \) is only valid when \( I \) is "I". This implies that the equation will only assign weights if \( I \) is active and all inactive nodes do not contribute directly into the final weight value. It is felt, that this scheme of operation does not fully represent the status of the network as all the nodes should be taken into account irrespective of the weights. Based on the Biological Cybernetics study conducted by Kohonen (see § 4.1), he found that there were both excitation signals as well as inhibitory signals. These signals would create a form of Mexican hat shaped response around the neuron, forming an umbra of excitation response and a penumbra of inhibition. Based on this principle, a new scheme for weight assignments was adopted in FLANN I. The FLANN I uses all the weight connections within the system regardless of the input states. Although in a slightly altered form of the Mexican hat principle, FLANN I uses the inactive nodes to generate a small negative value, proportional to the summation of individual elements of the entire
vector. With the active nodes, FLANN I generates a large positive value, proportional to the actual input vector. The equation applied to the weights is shown in equation (5.6).

\[
W_{ij} = \frac{I_i}{\sum_{k=0}^{n-1} I_k} + \frac{I_i - I}{(\sum_{k=0}^{n-1} I_k)(n - \sum_{k=0}^{n-1} I_k)}
\]  

(5.6)

\(W\) is the weight assigned, \(I\) is the input value and \(n\) is the number of input nodes. \(i\) is the \(i^{th}\) input node and \(j\) is the \(j^{th}\) output node. \(k\) is the input node sequence beginning with 0 and ending at \(n\).

From equation (5.6), when the input element \(I_i\) is 1, the second term becomes 0. The equation effectively becomes (5.7) and the value of \(W_{ij}\) becomes the reciprocal of the summation of the elements of the input vector

\[
W_{ij} = \frac{I_i}{\sum_{k=0}^{n-1} I_k} + 0
\]  

(5.7)

If input \(I\) is 0 then the first term reduces to 0 and due to \(I_i\) being 0, the second term becomes a small negative value. This is reflected in equation (5.8).

\[
W_{ij} = 0 + \frac{I_i - I}{(\sum_{k=0}^{n-1} I_k)(n - \sum_{k=0}^{n-1} I_k)}
\]  

(5.8)

In this way, FLANN considers all weights important to the operation of the network. The resultant assigned weights in FLANN I are therefore a function of the entire input vector and all weights possess a value and are never left at 0.
5.4 Calculations

The following example is used to comprehend the effects of equation (5.6). Two simple bit mapped patterns of the characters A and B shown in Figure 5.5 are used. The bitwise representation of the characters are reflected directly below each pattern.

\[ \begin{align*}
W_{ON} &= \frac{1}{20} + \frac{0}{\left(\sum I_k\right)(20 - \sum I_k)} \\
W_{OFF} &= 0 + \frac{-1}{\left(\sum I_k\right)(20 - \sum I_k)} \\
W_{ON} &= \frac{1}{14} + 0 = 0.07143. \\
W_{OFF} &= 0 + \frac{-1}{(14)(6)} = -0.011905.
\end{align*} \]

For any bits that are "OFF" in the character image of A, the weight value will be given by,

Using the same method of calculations, the \( W_{ON} \) and \( W_{OFF} \) for character B are \( \frac{1}{13} \) (which is 0.076923) and \( -\frac{1}{(13)(7)} \) (which is -0.010989). Assume the character B is presented to the network that has stored both the images. At one point in the
algorithm, the input vector of B will be checked with the template of A. The calculations would then go through the following sequence.

1) A bit-wise multiplication is performed between the input vector and the weights stored in the A character template. This is illustrated in Figure 5.6.

![Figure 5.6 Input vector B comparing with template A](image)

2) By performing a bitwise multiplication, the weights (not shown) on the "1" connections of the A template are 0.07143. This value will only be valid if the corresponding element in the B vector is a 1.

3) On the "0" connections of the A template, there exists the small negative value of -0.01191. This value will become valid if the corresponding element in the B vector is a 1.

4) In this way, correct matches for "1"s will gain more weights for the final summation and any incorrect matches will always lose weights which penalise the vector in the final comparison. The total template calculations using a B on an A will yield $11 \times 0.07143 - 2 \times 0.01191 = 0.76191$.

5) If the input B vector is compared with the B template, the exact match will yield a perfect summation score of approximately 1 (approximately due to floating point calculations).

6) The final competition using equation (5.1) will be used to choose the closest pattern found in the neural network's memory.
From this example, it can be seen how equation (5.6) successfully allows FLANN I to distinguish different patterns. But what about patterns that are closely related?

This section describes how FLANN I distinguishes between similar prototypes, stressing on the similar structures. The bitmap in Figure 5.7a shows the numeric 5 and Figure 5.7b shows the numeric 6. Each bitmap is represented on 4 by 5 grid, resulting in a 20 bit word when the 5 rows of 4 bits are concatenated as a single word. From equation (5.6), the character 5 would create weights of 1/14 (based on 14 'ON' bits, resulting in 0.07142) for the 'ON' bits and -(1/(20-14)) x 1/14) or (-0.0119) for the 'OFF' bits. The character 6 will have weights of 1/15 (based on 15 'ON' bits, resulting in 0.0667) for the 'ON' bits and (-0.01333) for the 'OFF' bits. Again, unlike the ART1 network that discards the off bits with no assignments to the 'OFF' bits, FLANN I considers all nodes, be it 'ON' or 'OFF'. Thus pattern 6 would not be considered a pattern 5 because the summation result would yield a value of 1 - 0.0119 (0.9881). The -0.0119 is from the extra bit matched with the 'OFF' bit found in the stored pattern 5. This would eventually lose the competition to the value created by the stored pattern 6 as the summed value would be 1. This additional term in the equation is small enough, to be stable in noisy input systems. An intuitive question to ask is "Whether a character 5 would be recognised as a 6 in a noisy environment?" The exact character in a noisy input system will always have the maximum value, less the negative values caused by the noise. The worst case possible for noise will only cause a maximum loss of one 1/(Sum of Elements) [for the pattern 6 is 1/15 and the pattern 5 is 1/14]. This will always bring the system to the nearest represented pattern, keeping it stable in noisy conditions.
5.5 FLANN I Algorithm

After understanding the individual parts that form FLANN I, it is now necessary to view the entire algorithm in its entity. The algorithm of FLANN I is summarised as follows.

1. Initialise Network with $p$ and $\alpha$ as vigilant values between 0.5 and 1.0 (Too low a value of $\alpha$ and $p$ is of no consequence to the network, but becomes meaningless when the network interprets patterns.

2. Set all $C_j$ values to 1. Present next pattern to the input nodes. If there are no output nodes present, GOTO 8.

3. Determine the winning node in the network using the stated criteria. If no output nodes left, GOTO 8.

$$\text{winning\_node} = \max \left(C_j \sum_{i=0}^{n} W_{ij} I_i \right)$$

4. Perform vigilance testing with $\alpha$ using equation (5.4) and $p$ using equation (5.3). If vigilance testing is less than $\alpha$ and $p$, then set $C_j$ to 0 and goto 3.

5. If $\frac{|v|}{|I|} < 1$ or $\frac{|v|}{|V|} < 1$ GOTO 7

6. Re-code winning node. GOTO 2 /* when input pattern is 100% match*/

7. Match is found. GOTO 2.

8. Assign weights to the system using equation and GOTO 2.

$$W_{ij} = \frac{I_i}{\sum_{k} I_k} + \frac{I_i - I}{(\sum_{k} I_k)(n - \sum_{k} I_k)}$$  (copy of 5.6)
Chapter 5  
Fast Learning Artificial Neural Network (FLANN I)

The recoding ability of FLANN is encountered in step 6 of the algorithm. Recoding is a very powerful tool in environments that constantly change. Thus FLANN is able to adapt itself to changes in old patterns. This recoding should not be done automatically by the system, and should always be authenticated by the trainer. The program can be written to skip over this feature if it is not required.

5.6 Comparison of Learning Speeds in ART1 and FLANN I

Several basic experiments were performed on FLANN I to examine the reliability of the network and compare it to the ART1 network. A basic test using characters from A to J was done, teaching the network whenever necessary and allowing the network to form new patterns upon identifying a novel input pattern.

The results that follow, show the speeds with which each network learns. The ART1 had varying degrees of learning, depending on the value of $p$. In a single epoch of 10 presentations, the ART1 model with a $p$ value of 0.5 could only retain 3 patterns. This is due to the one-sided comparisons made in the vigilance testing. This comparison, as mentioned earlier, is suitable for some applications, but not for all. The FLANN I neural network showed a positive ability to retain all that it is taught. As a result, it remembered all 10 characters, forming the complete set of patterns in a single epoch, without overwriting stored patterns. Figure 5.8 and 5.9 show the learning rate of the ART1 network in a single epoch. Figure 5.10 shows the learning rate of FLANN I.
Figure 5.8 ART1 after 1 epoch $\rho = 0.5$

Figure 5.9 ART1 after 1 epoch $\rho = 0.8$
Figure 5.10  FLANN after 1 epoch
Chapter 5  
Fast Learning Artificial Neural Network (FLANN I)

5.7 Training FLANN I for Accuracy

Although it is true that the model has the ability to learn patterns quickly, it does not discount the need for careful training of the network before it becomes a useful and accurate system. It is recommended that training should always begin with the most accurate form of exemplars. The network should then be tested to determine if further training is required. This testing should be a systematic application of test data to check for accuracy.

In certain applications, training is complete upon presentation of the most accurate form of exemplars. In others, where the structure of the problem is less predictable (this is usually due to the inconsistency in the human's creative skills), training may take the form of presenting intermediate variant patterns that represent a particular class. Some examples of structural inconsistency is the alphabet set A - Z or the Chinese character set. To illustrate such an inconsistency caused by the fickle mind, Figure 5.11 shows a pattern of sequences.

![Pattern of sequences](image)

Figure 5.11 To illustrate the inconsistency of human decisions?

Upon initial inspection, one would be confused, because the sequence seems perfectly legitimate. Focusing at the third pattern, most will say that it is a 5 because it has only one extra bit added to the first pattern. But there are a few who will say that it is a 6 because it is the same pattern as the 6, except for a bit missing from the original 6. This inconsistency in human decisions creates extreme difficulty in training accurate systems. Due to this inconsistency, neural network models need to maintain a degree of flexibility to allow the trainer to intervene with
the learning process (Assuming that the trainer knows best). (Self-learning systems can seldom accommodate such inconsistencies.) FLANN I allows the user to intervene, and it is capable of adapting to the trainer's commands. The responsibility of FLANN I's accuracy is fully dependent on the trainer's understanding of the problem and not on the FLANN I algorithm.

In the example shown in Figure 5.11, the trainer would need to determine which number the third pattern looks like. During the training process, the training should begin with the most accurate exemplars. This would mean training the network with the first 2 patterns in Figure 5.11. Upon training the system, the testing procedure may involve presenting the third pattern, where conflicting opinions may occur, i.e. the trainer may think it is a 6 but the neural network may think it is a 5. Of course the trainer is always right and the neural network never gets its own way. The trainer needs to over-rule the answer by using the forced learning mode to train the FLANN I system to register the variant pattern. In this way, FLANN I can be trained to handle simple variants.

Although this example was cited, it may not be an efficient way of handling character recognition systems. It serves only as an example to describe the concepts of variant training.

5.8 The XOR Problem

The most celebrated (exclusive-or) XOR problem was also used as a test for the FLANN I network. To briefly explain, the XOR problem is to classify the following inputs correctly, as shown in Table 5.1.
Table 5.1 The XOR Truth Table

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</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>0</td>
</tr>
</tbody>
</table>

As mentioned in section 3.3, the XOR problem was an impossible problem to solve using a single perceptron since it is linearly inseparable. In the book by Rumelhart, D.E. et. al, (1986), the results of the XOR solution, solved by the back propagation method, indicated a need for several hundred cycles of training. The experiment was also conducted and presented in Chapter 3 of this thesis. From these experiments, the learning process of the MLP model was slow, requiring up to 1400 cycles. Apart from the slow learning speeds, Rumelhart mentions the possible encounters with local minima.

The XOR problem was solved using FLANN I in a single epoch of presentations. The total number of presentations was 4, 1 for each pattern found within the truth table shown in Table 5.1. Unlike the problems mentioned in § 5.7, where inconsistencies may arise, the XOR was a straightforward true or false problem. There was no need to perform variant pattern training. In addition to the fast learning speed, FLANN I does not encounter any local minima problems, which has been a devoted topic of discussion for most MLP users.

Since the XOR solution contains all permutation possibilities on the truth table, the values for p and α were set close to the limit, i.e. 1. As a result of this tight limit, there was no need for generalisation properties, and the system could be allowed to train itself from a truth table.
5.9 The Parity Bit Problem

Another well known model problem was used by Minsky and Papert (1969). The truth table for the parity bit problem which Rumelhart used is shown in Table 5.2.

<table>
<thead>
<tr>
<th>BIT 1</th>
<th>BIT 2</th>
<th>BIT 3</th>
<th>BIT 4</th>
<th>PARITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.2 The Parity Bit Truth Table

This problem poses difficulty to most neural networks, because of the need to identify and distinguish similar patterns that differ by only a single bit. As discussed in (pp 334 - 335, Rumelhart, D.E. et. al, 1986), this parity problem with 4 bits required 2825 epochs of training on a back propagation neural network model. The additional training time required, in an MLP model, for additional input vector elements does not increase linearly.

The parity bit problem was implemented on the FLANN I architecture with 8 bits. Again, like the XOR counterpart, the $\alpha$ and $\rho$ values were set close to 1, making the system identify and learn patterns with the slightest difference. In a single epoch...
of 256 presentations, the network was fully trained. The identification of the parity bit sequences used to test the network indicated an accuracy of 100%. The section that follows describes the implementation of the parity problem using the FLANN I model.

The initialisation phase of FLANN I required the setting of \( \alpha \) and \( \rho \) to the desired values. Since the unique patterns may only differ by a single bit, it was imperative to set \( \alpha \) and \( \rho \) to a high value. In doing so, there is a stringent vigilance that fails upon the slightest difference in pattern comparisons. A total of 8 input bits are required, and the C program structure of the output node appears as follows:

```c
struct output {
    int input_bit[8];
    int parity;
    struct output *next_node;
};
```

The first declaration of the 8 bit array is the space needed for each output node to possess storage space for the binary pattern. Parity is used to store the parity bit. If the pattern has never been classified before, the program will classify the pattern and store the parity bit in the new node, otherwise it will search the matched node for the parity value.

The final component of the `struct` statement is used for a linked list structure creation. This will point to the next node containing a classified pattern. If there are no other patterns remaining, this will be a NULL pointer. The program flowchart is shown in Figure 5.12.
After the initialisation phase has been performed and before the actual program begins (i.e. in the header file), the file containing all the bit patterns are read sequentially. Each pattern is then passed through the FLANN I algorithm, signified in the flowchart by the diamond shaped decision box, "Is Pattern New?". If FLANN I has not classified the pattern before, it will create a new node, link it to the existing list and calculate the parity for the new pattern. If the pattern is an old pattern, FLANN I will print the parity bit based on the match and the stored parity bit. The actual parity program is found in the disk attached.
5.10 Number Recognition System

In the two model examples discussed previously in sections 5.8 and 5.9, both require FLANN I to be very stringent in its vigilance testing. Although this showed that FLANN could be trained to recognise minute differences in patterns, it failed to reflect the generalisation properties of FLANN I. In the next system, FLANN I is implemented in a more complex binary classification system for bit-mapped numbers. Numbers from 0 to 9 were drawn on a 7 X 10 grid and FLANN I was trained using some samples. This experiment is different from the first two since there is now ample room for generalisation. The results of the initial stages of a trained network are provided. FLANN I was trained from 0 to 9 in a single epoch of 10 presentations.

The training process began with the number 0 and is increased sequentially to 9. During the 1 epoch training the bitmap of 3 was classified as a 0. User intervention by the trainer was required to rectify the situation by forcing the system to learn the 3 as a unique pattern. Similarly, the pattern 6 was confused with a 5 and the pattern 8 was confused with a 6. Since FLANN I allowed the user to intervene using the forced learning mode, the system maintained its single epoch learning status. The values for $\alpha$ and $\rho$ were set to 0.7. The figures that follow show the actual input patterns on the first grid and subsequent 8 distorted patterns. The result of the pattern classification is labelled below each individual pattern, stating a pass or a fail. If a pass is mentioned, it implies a correct classification and if the system failed to recognise, a fail is placed below the figure. An analysis is performed on the general kinds of pattern failures and the noise immunity in the experiment.
5.10.1 Classification Results for Numeric Zero

Original Pattern | Passed Pattern | Passed Pattern
--- | --- | ---
Failed (noise thres.) | Classed as a 3 | Classed as an 8
Failed (signal thres.) | Passed Pattern
5.10.2 Classification Results for Numeric One

Original Pattern  Passed Pattern  Passed Pattern
Passed Pattern  Passed Pattern  Passed Pattern
Passed Pattern  Passed Pattern  Passed Pattern
Failed (noise thres.)  Failed (signal thres.)  Failed Pattern
5.10.3 Classification Results for Numeric Two

Original Pattern Passed Pattern Passed Pattern
Passed Pattern Passed Pattern Passed Pattern
Passed Pattern Passed Pattern Passed Pattern
Failed (noise thres.) Failed (signal thres.) Failed Pattern
5.10.4 Classification Results of Numeric Three

Original Pattern  Passed Pattern  Passed Pattern

Passed Pattern  Passed Pattern  Passed Pattern

Failed (noise thres.)  Failed (signal thres.)  Classed as a 9
5.10.5 Classification Results of Numeric Four

Original Pattern  Passed Pattern  Passed Pattern

Passed Pattern  Passed Pattern  Passed Pattern

Failed (noise thres.)  Failed (signal thres.)  Failed (signal thres)
5.10.6 Classification Results of Numeric Five

Original Pattern  Passed Pattern  Passed Pattern

Passed Pattern  Passed Pattern  Passed Pattern

Failed (noise thres.)  Failed (signal thres.)  Classed as a 6
5.10.7 Classification Results for Numeric Six

Original Pattern     Passed Pattern     Passed Pattern
Passed Pattern       Passed Pattern     Passed Pattern
Passed Pattern       Passed Pattern     Passed Pattern
Passed Pattern       Failed (noise thres.)     Failed (signal thres.)     Classed as a 5
5.10.8 Classification Results for Numeric Seven

Original Pattern

Passed Pattern

Passed Pattern

Passed Pattern

Passed Pattern

Passed Pattern

Passed Pattern

Failed (noise thres.)

Failed (signal thres.)

Failed Pattern
5.10.9 Classification Results for Numeric Eight

Original Pattern  | Passed Pattern  | Passed Pattern  
Passed Pattern  | Passed Pattern  | Passed Pattern  
Passed Pattern  | Failed (signal thres.)  | Failed (noise thres.)  | Classed as a 9
5.10.10 Classification Results for Numeric Nine

Original Pattern  Passed Pattern  Passed Pattern

Passed Pattern  Passed Pattern  Passed Pattern

Failed (noise thres.)  Failed (signal thres.)  Classed as a 7
From the results reflected in sections 5.10.1 to 5.10.10, FLANN I exhibits
generalisation properties compatible with other neural networks. With just a single
epoch of 10 presentations, it is able to recognise a large set of the generalised
exemplars.

The number of nodes required for training the network, to recognise 0 - 9, was ten.
One might suggest that as the number of patterns increases, FLANN I will lower
its efficiency due to an exponential increase of complexity in the searching
algorithm. However, although there is an exponential increase in the search
timings, such a situation may never occur. This is because FLANN I is a form of
nearest neighbour search algorithm, working in the binary system which utilises the
Hamming distance measure. Since generalised exemplars can be approximated by
close proximity comparisons, all that is required, is a few representative exemplars
to be stored. It is therefore expected, that the number of actual stored exemplars
would remain manageable. The exponential increase in computational comparisons
can still be tolerated for small exemplar sets. In addition, current applications,
seldom maintain a large set of exemplars, because it is a difficult task to train
common available networks. As a result, most applications maintain a small feasible
set of exemplars, which suits FLANN I.

In the experiments, test data was deliberately sought to trigger a failure in the
system. There are two basic ways of triggering a failure. The first is to increase the
noise ratio excessively and the other is to reduce the signals beyond the minimum
signal threshold. The minimum signal threshold is 70% of the actual image signal.
This is reflected in the failed patterns, by the label "Failed (signal thres.)". The
maximum noise threshold is 70% above the actual signal size.
5.11 FLANN and Other Architectures

In Chapter 2, four characteristics were used to gauge the abilities of the neural network. These were:-

- Learning Speed
- Accuracy
- Capacity
- Generalisation

In addition to these, there are several others that were not mentioned previously. Although these were not mentioned earlier, it does not imply that they are less important. These were omitted because they apply to the practical aspects of the neural networks. These are :-

- Trained network's response speed
- Network Initialisation process
- Network complexity
- Secondary storage
- Closely similar pattern identifications
- Forced learning and immediate recoding.

The various sections that follow compare 3 main architectures with the FLANN I model. These architectures are the MLP model, the Kohonen Network model and the ART1 model.
5.11.1 The Learning Speed

The MLP learning model requires the minimisation of the energy function within the neural model. This process of gradient descent is both tedious and slow. Already the simplest XOR problem required several hundred epochs to train the network. In the Kohonen system, the learning speed of the system is dependent on the learning ratio and the speed at which the area of influence would decrease. An incorrect manipulation of these variables may cause the system to oscillate, creating poor learning results. The number of epochs required for fully training the Kohonen network is also not specified. The user would have to form a trial-and-error estimation for the time needed. In the ART1 system, training is fast, with the capability to learn new patterns as quickly as FLANN I, unfortunately, the system occasionally overwrites desired exemplars. This may cause the erasure of desired exemplars without the knowledge of the trainer. The system is required to be trained using several epochs of presentation until it is certain that all exemplars are stored. In the FLANN I system, all learning is done in a single epoch. This learning even includes patterns that are closely similar.

5.11.2 The Accuracy

In both the Kohonen model and the ART1 model, the network accuracy can be adjusted to be extremely sensitive, making the resultant network very accurate. This implies that the network, although accurate, may be less able to generalise. The ART network can be taught with a stringent vigilance, but after training, this vigilance can be relaxed, allowing the system to generalise. Unfortunately, the Kohonen system cannot modify its generalisation properties once trained. The MLP model can usually achieve such accuracy to solve the parity problem, but it would require much training. As mentioned, a 4 bit parity system required more
than 2000 training epochs. FLANN I handled the parity system with a single learning epoch. It was able to identify each pattern correctly with 100% accuracy.

5.11.3 Capacity

The Kohonen model has a fixed capacity when the network is created. Given an \( n \times n \) grid, the number of patterns would then be limited by the storage capacity of the grid. In the MLP model, this is again limited to the size of the layers that were initially defined. Both the ART1 model and the FLANN I models have capacities limited by the available memory in the computer. The network itself allocates extra nodes for new exemplars as the need arises. In terms of efficient usage of the nodes, the MLP model will probably require less memory space. This is because it spreads its information over the entire network, while the other 3 networks do not. This is perhaps still a strong positive factor that the MLP model has over the other 3 newer models.

5.11.4 Generalisation

Once trained, the Kohonen model, the ART1 model and the MLP models can all perform good generalisation and not one of the systems lacked this property. FLANN I also maintains an efficient generalisation property which is exhibited in the numeric recognition experiment. Even though there were 10 nodes which contained the most accurate patterns, FLANN I is able to recognise deformed patterns of a similar class.

5.11.5 Trained Network Response Speed

The MLP model will probably perform the best in terms of network response speed. This is due to the architecture of the model. The results are immediately
obtained from the summation process and there are no comparisons required. In the other 3 architectures, there is a need for both summation processes and comparisons. It is expected that the MLP model will be the fastest to respond to inquiries. The FLANN I response times tend to be slower than the MLP models but will probably be faster than the Kohonen Network's response time.

5.11.6 Initialisation Process

The initialisation process for the MLP model uses a random assignment of weight connections. This form of assignment is more crucial in the Kohonen network to obtain a random spread of weights. Uniform weight assignments in the Kohonen model may cause it to diverge. The ART I and FLANN I models are initialised by only determining the number of input nodes required. The output nodes are dynamically allocated during training.

5.11.7 Network Complexity

The process of training the MLP model requires the calculation of individual weights and propagating the results backwards to alter the weights within the hidden layers. This complicates the algorithm and makes implementation difficult. In the Kohonen model, training requires the reduction of the areas of influence. While reducing the area of influence, the degree of accuracy is increased in the winning nodes. This process needs to be monitored to prevent over training of the system, causing the degrading of generalisation properties. In the actual ART I model, there are other complex structures that govern the network. Discussions which involve attention priming, top down comparisons, etc. complicate the entire network. FLANN I retains the simple algorithm and maintains only two main equations. The others are merely comparisons. The simple algorithm is not
essential in the current context, but if the neural network is to migrate onto a parallel computing architecture as is intended, simplicity in the algorithm is crucial.

5.11.8 Secondary Storage

In all neural network applications, it is necessary to save the network weights in some form of external storage. This requires the storing of information such as the tolerance values and the individual weight connections. The Kohonen network with an $n \times n$ grid and an input vector of $i$ elements would require $n \times n \times i$ weight connections to be stored. The FLANN I and the ART1 models are required to store $n \times i$ nodes, where $i$ is the number of input elements in the input vector and $n$ is the number of output nodes. The FLANN I, in comparison with the Kohonen network requires less nodes because the generalisation properties are not stored in the nodes, but they are determined in the algorithm. The neural network with the most efficient storage system is the MLP model. This model embeds parts of multiple patterns within each node. As a result, the storage capacity for an MLP neural network is usually less than that needed for a FLANN I model. Note that the FLANN I model does not store the weights of the network, but only the output. The weights are immediately calculated from the output nodes. In this aspect, it is easier to reconstruct the weights of the FLANN I system, whereas it is difficult to remember the weights of the MLP model if the weights are overwritten.

5.11.9 Closely Similar Pattern Recognition

In the MLP architecture, there is a weak influence over the exemplars that the network is to learn. With patterns that are grossly different, the network may eventually converge, but with patterns that are differing only by a single bit, training becomes difficult. Such problems also exist in the Kohonen Network. If the Kohonen system fails to make fine distinctions, fine tuning is necessary. The
ART1 model will need to be adjusted to raise the vigilance factors. At the same time, more training is required because similar patterns may overwrite old exemplars. The FLANN I model accommodates closely similar patterns by using the forced learning mode. The trainer has full control over the network's learning behaviour. As a result, the single epoch training time is maintained.

5.11.10 Forced Learning and Immediate Recoding

The inability to directly control the learning behaviour of the neural network model is sometimes frustrating. In the training experience of an MLP model, teaching the system to learn the image of a 1 may take as many as 20 cycles. Teaching the system a new pattern of a 2 may take another 20 cycles. But by the time the system learns the second pattern, it probably has forgotten the first. The Kohonen network cannot be directly trained to improve on one exemplar without jeopardising the others. ART1 can learn new patterns by increasing the vigilance to 1 and lowering it later. FLANN I achieves it by forcing the pattern into the new nodes. This forcing action is not done blindly. The system will check if the exemplar already exists. In the case where the exemplar exists, the system will perform an immediate recoding. This recoding is useful to correct wrongly trained exemplars. After all, such a fast learning model should still accommodate for human errors.

5.12 Conclusions for FLANN I model

FLANN I is very much similar to other network topologies, in that it has to be fine tuned according to the application. The value of $\alpha$ and $\rho$ can be varied according to the application specific needs. An example of such manipulations would be to set $\alpha$ close to 1 and $\rho$ at 0, making the network more sensitive to the new input patterns more than the stored patterns. Such manipulations need experimentation.
to determine the best value for the particular application. FLANN I's properties are summarized as follows:

- Fast learning
- Accurate
- Dynamic memory allocating
- Able to generalise
- Maintains a set of stable patterns
- Provides a mechanism for recoding
- Facile implementation algorithm
- Adaptable to noisy inputs

Its ability to learn fast is an asset which has not been achieved by other networks. In addition to the fast learning ability, it has maintained the important properties of a neural network.
Chapter 6

Fast Learning Artificial Neural Network II (FLANN II)
CHAPTER 6 FAST LEARNING ARTIFICIAL NEURAL NETWORK II (FLANN II)

In the previous chapter, the novel development of FLANN I was presented and the experimental results were discussed. The results indicate that the FLANN I model is able to function as a basic neural network, comparable to the older established models. Unfortunately, FLANN I models are restricted to operate on binary coded problems. This inevitably casts doubts about the feasibility of FLANN I in a wider spectrum of applications which involve continuous inputs; after all, a large proportion of potential industrial applications operate with continuous values.

On closer inspection, it was realised that the FLANN I algorithm could not be easily manipulated to operate directly on continuous input values. Since such modifications for FLANN I are both difficult and inefficient, a completely different neural network had to be designed for processing continuous inputs. The new neural network model has to maintain similar qualities found in the first model, and yet operate with continuous input values. In order to design a new neural network with similar capabilities, it was necessary to extract the basic concepts from the original FLANN I model.

6.1 Distance Metrics

The FLANN I model utilises the comparison of Hamming distances of exemplars to perform its classification. It is on the Hamming distance proximity that the best matches are selected. The following explanation describes Hamming distances.

Given two vectors \( \mathbf{A} = [a_1 \ a_2 \ a_3 \ \ldots \ a_n]^T \) and \( \mathbf{B} = [b_1 \ b_2 \ b_3 \ \ldots \ b_n]^T \), their Hamming distance is defined as the number of positions in which \( \mathbf{A} \) and \( \mathbf{B} \) differ. This relation can be denoted by \( d(\mathbf{A}, \mathbf{B}) \).
\[ d(A,B) = \text{number indices } i = 1,2,3,...,n, \text{ which } a_i \neq b_i. \]

Based on this relationship, several expressions can be derived. These are as follows:

\[ d(A,A) = 0 \]
\[ d(A,B) > 0 \text{ if } A \neq B \]
\[ d(A,B) = d(B,A) \]

Based on this Hamming distance relationship, the edges of the cube in Figure 6.1 signify the neighbours of a vector with the Hamming distance of 1.

![Figure 6.1 The edges of the cube possess Hamming distances of 1, relative to the nearest neighbours.](image)

To create a new algorithm for the continuous world, a new distance measure is required. The most accurate form of measure is the Euclidean distance metric which is evaluated by the formula,

\[ d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \]
where $d(x, y)$ denotes the distance between the two vectors $x$ and $y$. Some other forms of distance metrics are city blocks and square distances. Although these distance metrics require less time for computations, they are less accurate. Since accuracy is higher in priority than speed, the most accurate calculation is chosen. Thus the actual network implementation discounted the use of the city block and square distance calculations. It used the Euclidean measure because it was the most accurate. To reduce the time in the calculations, the square root function was discarded. This was possible because the algorithm uses the resultant value as a form of comparison. The square root effect is cancelled during comparisons with other distances obtained using the same formula, and can be removed from the competition equation.

In the surveys conducted, it is apparent that the Kohonen network model also uses the Euclidean distance measure. The Kohonen network is a learning vector quantisation (LVQ) neural network that utilises the nearest neighbour recall to perform operations. The effectiveness of the Kohonen network has previously been discussed in Chapter 4.

FLANN II was designed based on this Euclidean distance metric and like the Kohonen network, it is an LVQ neural network. Although both the Kohonen and the FLANN II use the same distance metric, their resultant properties differ in many aspects. The most significant difference is in the training speeds of the two neural networks. The Kohonen network training cycles are determined by the trainer but FLANN II has a constant learning rate of 1 epoch.
6.2 Characteristics of FLANN II

Unlike its predecessor that can only operate on binary systems, FLANN II can be used on both the binary and continuous systems. FLANN II's main characteristics are cited as follows:-

- It can be completely trained in a single epoch and operates on continuous values.
- It has the flexibility to determine individual vector element resolutions and constraints.
- It has a stable system of storage.
- It has a simple architecture, as a result, the implementation is not complicated.
- It can be designed to re-code previously learned patterns (allowing correction for human errors during training.)
- It can perform well even in noisy environments.
- The output layer grows dynamically during training.
- It can be readily mapped onto a parallel architecture (see Chapter 7).

The FLANN II appears to resemble FLANN I in several operational aspects, but the actual operations in the algorithm are quite different. (Careful understanding of the algorithm is still required, as there are functions that can be easily misunderstood. e.g. the discerning function \( D \) does not operate the same way as in most MLP models). FLANN II has the added flexibility of controlling the sensitivity of individual elements in the input vector.

6.3 The Architecture

The basic architecture of FLANN II is delineated in Figure 6.2. It consists of two layers, similar to the basic Kohonen network model (Kohonen T., 1982) and the
ART1 model (Grossberg S., et. al., 1987). Connecting the output layer and the input layer are a set of weight vectors. Each output node has weights connected to each element of the input vector. The weight assignment system is unlike that of the Kohonen network or ART1. If a novel pattern is to be stored, it performs a direct copy of the input vector into the weight vector. No other form of equations are needed for evaluating the weight vectors.

As in FLANN I, the output layer can be viewed as a single dimensional layer which grows dynamically as more novel patterns are encountered. As these novel patterns are encountered, the output system will allocate memory space for the pattern. This implies that the design stages of the network would not require a detail plan of the output layer, but only a correct configuration of the input layer. The output layer will determine its own size, as the system is trained. Classification results are eventually obtained directly from the winning output node.

![Figure 6.2 The FLANN II architecture.](image)

6.4 An Overview

Before explaining the details of the FLANN II operations, it is necessary to understand a brief overview of the network. FLANN II performs its selections through two distinct levels. The first level of selection is the vigilance testing, which uses the following equation (6.1).
\[ \frac{1}{n} \sum_{i=1}^{n} D\left( \delta_i^2 - (W_{ij} - I_i)^2 \right) \geq \rho \]  

(6.1)

where \( D \) is the discerning function found in Figure 6.3. \( \delta_i \) is the tolerance value for the \( i \)th element of the input vector. \( W_{ij} \) is the weight joining the \( i \)th element of the input node to the \( j \)th output node. \( I_i \) is the \( i \)th element of the input vector \( I \). \( \rho \) is the vigilance factor and \( n \) is the number of elements in the input vector \( I \).

It allows the network to focus on pattern sets that qualify for a second level of competition. This vigilance testing level performs the two basic functions:

- Individual vector element tolerance testing.
- General pattern vigilance testing.

The final selection for the winning node uses the equation in (6.2). This equation is similar to that utilised in the Kohonen network and it creates a form of inter-nodal competition.

\[ \text{winner} = \min \left[ \sum_{i=0}^{n} \left( W_{ij} - I_i \right)^2 \right] \]  

(6.2)

where \( W_{ij} \) is the weight vector \( W \) between the \( i \)th input node and the \( j \)th output node, and \( I_i \) is the \( i \)th element of the input vector \( I \).

Of the two, only equation (6.1) derives the decision to learn a new pattern. It uses the linear tolerance measure \( \delta_i \) to gauge each individual element for acceptability. Note that \( \delta_i \) can be a different entity for each \( i \)th element, or it could be designed as a common threshold for the entire vector. This is dependent on the problem and it provides the network designer with flexibility.

The selection through the vigilance testing can yield two possibilities. The first, is the failure to locate any match and the other is to locate at least a possible match.
Chapter 6  Fast Learning Artificial Neural Network II (FLANN II)

To encounter the first case would mean that the network has encountered a novel pattern. This pattern can then be absorbed into the network by the weight assignments made onto a new output node. If the second possibility arises, the set of possible matches are submitted to the second level competition. With equation (6.2), the network will select the best match. Note that in certain applications where several best matches are permitted, the algorithm can be modified to obtain several possible results (e.g. medical diagnosis of illness symptoms).

Figure 6.3 shows the discerning function $D$ which is used by equation (6.1) to perform the $\delta_i$ threshold test. Note that the function does not perform the way MLP models do. It is a mere threshold function that determines if individual elements of a vector are within the stipulated tolerance.

![Figure 6.3 The discerning function $D$](image)

6.5 Detail Description of FLANN II

With the understanding of the general operations in FLANN II, the details of the system can now be described. The following sections describe the detail functions of each component in the algorithm. FLANN II generally consists of four components. These are as follows:-
6.5.1 Weight assignments

The weight assignment on the FLANN II system is probably the simplest of all neural network models. The code segment that follows shows the embedding of the weights in each node structure.

```c
struct node {
    float weight[n];
    float tolerance[n];
    int class;
    struct node *next_node;
}
```

Upon inspection, one could deduce that the array `weight[n]` is for the storage of the weight vector, the `tolerance[n]` is for individual tolerances set for each element, the `int class` is the classified result of the pattern and the final pointer `struct node *next_node` is used to generate a linked list. With a linked list system, the program can dynamically allocate more nodes as novel patterns are introduced. When a new novel exemplar is encountered, a new node is created using the structure. The connecting weights are assigned with a direct replication of the input vectors. This simple replication process removes any form of special coding required, reducing the calculations required for weight assignments. This is unlike the FLANN I where weight values for all nodes are calculated based on the
equation (5.6). Figure 6.4 shows the weight assignment for a FLANN I system and Figure 6.5 shows the weight assignments for FLANN II. (Black = 1 and white = 0)

\[ \text{Figure 6.4 FLANN I weight assignment for } 10101 \]

\[ \text{Figure 6.5 FLANN II weight assignment for } 10101 \]

Since the storage of the weight vectors are equivalent to the storage of the original input vectors, the network trainer can now observe and keep track of the trained state of the neural network. This is usually difficult in the case of an MLP model, where the trainer cannot directly deduce the stored exemplars from the weight values because the solutions are coded into all the nodes of the system.

### 6.5.2 Tolerance Factors

The FLANN II uses a new variable, \( \delta \), called the tolerance factor. This is an important component that gives it flexibility and generalisation properties. This tolerance factor can be embedded in the program as a global variable or it could be a variable unique to each element in the vector (as shown in the code segment in...
Chapter 6  Fast Learning Artificial Neural Network II (FLANN II)

the previous section). FLANN II determines if an element in the continuous input vector is within tolerance by comparing it with the predetermined \( \delta \). This is done using equation (6.1) which requires the calculated difference between the input vector element and the output vector element to be squared. This assures that the value is always a positive number. It is later subtracted from the square of the tolerance value \( \delta \). With the help of the mapping represented by the discerning function \( D \), shown in Figure 6.3, a result of 0 or 1 is obtained. The overall contribution of the tolerance factor is further described in § 6.5.4.

6.5.3 Vigilance Factors

In addition to the tolerance factor, FLANN II continues to use the concept of vigilance factors to determine the proximity of the pattern as a whole. There are in total, two sources of approximations. The first is in the tolerance factor that determines individual proximity of each element and the second is global proximity of the entire vector, which is determined by the vigilance factor, \( \rho \).

Unlike FLANN I which maintains two vigilance factors, FLANN II retains only one vigilance factor. Like FLANN I, \( \rho \) is set to \( 0 \leq \rho \leq 1 \). It is normally given a value larger than 0.5. If \( \rho \) is set to 0, it implies that all vector inputs will be classified, and the mechanism for selection is purely based on the competition stage.

6.5.4 Nearest Neighbour Competitive Selection

The previous three sections have dealt with each individual component that makes FLANN II, but have not related them together to form the neural network. The main selection of the winning exemplar is determined by the use of equation (6.2).
Equation (6.2) is the minimum distance function that calculates the simplified Euclidean distance discussed in § 6.1 (i.e. the Euclidean distance without the square root.). But prior to this selection, it is necessary to determine if the stored exemplar passes the vigilance tests. Failure to precede the competition with the vigilance testing may result in erroneous classifications.

Using the example shown in Table 6.1, assume that a FLANN II network is trained with a tolerance value, \( \delta \), of 1.5 and a vigilance factor \( \rho \), of 0.8. Each element of the stored exemplars are shown in their respective positions of the table. The last pattern in the table is a test pattern which FLANN II needs to process.

<table>
<thead>
<tr>
<th></th>
<th>Element 1</th>
<th>Element 2</th>
<th>Element 3</th>
<th>Element 4</th>
<th>Element 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exemplar 1</td>
<td>2.0</td>
<td>2.5</td>
<td>5.0</td>
<td>1.6</td>
<td>7.0</td>
</tr>
<tr>
<td>Exemplar 2</td>
<td>3.0</td>
<td>4.8</td>
<td>5.7</td>
<td>6.7</td>
<td>7.7</td>
</tr>
<tr>
<td>Exemplar 3</td>
<td>5.4</td>
<td>6.7</td>
<td>8.9</td>
<td>3.6</td>
<td>3.2</td>
</tr>
<tr>
<td>Test Pattern</td>
<td>2.3</td>
<td>2.6</td>
<td>6.8</td>
<td>2.0</td>
<td>6.0</td>
</tr>
</tbody>
</table>

Table 6.1 3 Exemplars showing elements stored in FLANN II
The last pattern is a test pattern

The evaluation of the pattern vigilance testing begins with a tolerance comparison between the test pattern and the stored exemplars by using equation 6.1,

\[
\sum_{i=1}^{n} D(\delta_i^2 - (W_{ij} - I_i)^2) \geq \rho \quad \text{(copy of 6.1)}
\]

The first stages of the calculations will involve the difference between corresponding elements within each vector. This is reflected in the \((W_{ij} - I_i)^2\)
portion of the equation. Table 6.2 shows the results of this subtraction and the squaring effect of the first stage calculations.

<table>
<thead>
<tr>
<th></th>
<th>Element 1</th>
<th>Element 2</th>
<th>Element 3</th>
<th>Element 4</th>
<th>Element 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exemplar 1</td>
<td>0.09</td>
<td>0.01</td>
<td>3.24</td>
<td>0.16</td>
<td>1.0</td>
</tr>
<tr>
<td>Exemplar 2</td>
<td>0.49</td>
<td>4.84</td>
<td>1.21</td>
<td>22.09</td>
<td>2.89</td>
</tr>
<tr>
<td>Exemplar 3</td>
<td>9.61</td>
<td>16.8</td>
<td>4.41</td>
<td>2.56</td>
<td>7.84</td>
</tr>
</tbody>
</table>

Table 6.2 The results after the first stages of calculating $(W_{ij} - I_i)^2$

The next stage of the equation requires the $\delta^2$ term, which has the value 2.25, to subtract individual elements in Table 6.2. This is then mapped into the function $D$ represented as a graph in Figure 6.3. The resultant value of the numerator

$$\sum_{i=1} D(\delta^2 - (W_{ij} - I_i)^2)$$

before the summation of the individual elements is shown in Table 6.3.

By the complete evaluation of equation (6.1), only exemplar 1 passes the vigilance testing. The full summation of exemplar 1 is 4 and since the number of elements in the vector is 5, the final value is 0.8, which is just within the vigilance factor. The other two exemplars fail the vigilance test since they each possess a factor of 0.4.

<table>
<thead>
<tr>
<th></th>
<th>Element 1</th>
<th>Element 2</th>
<th>Element 3</th>
<th>Element 4</th>
<th>Element 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exemplar 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Exemplar 2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Exemplar 3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.3 Results of the individual elements after the mapping into $D$
In the example used, the first exemplar will become the winner of the next selection by default, since there are no other competing exemplars for the next stage. When more than one exemplars passes the vigilance test, then equation (6.2) is used to determine the eventual winner of the final competition.

**NB:** It is imperative, that the vigilance factors should precede the competition stage, as there may be occasions when exemplars possess the minimum value during the competition, yet fail the vigilance tests.

### 6.6 Flexibility of Tolerance Values

In section 6.5.2, a short discussion was provided for the use of the tolerance factor. It was also explained that the tolerance factor $\delta$ can be programmed as a global variable. Though this is true, not all application solutions can be designed by maintaining a single global tolerance factor. This is due to the difference in measurement units between input elements or scale resolutions.

A character recognition problem explained in § 5.10 can be implemented using a global tolerance factor. This is because the binary system is used throughout all vector elements and the scale between input elements are similar. To implement the character recognition system using FLANN II, the binary "1" no longer possesses a value of exactly 1, but $1 \pm 0.05$. This is very different from the FLANN I solution of the problem, where the binary values are represented by exact "1"s and "0"s. In this sense, the solution provided by FLANN II is very similar to the MLP solution (See the XOR example in Chapter 3).

Some problems in signal processing may require different resolutions. Figure 6.6 shows a conceptual example, where a signal classification problem requires different resolutions for individual vector elements.
The magnified area depicted in the larger circle shows the minute details required from the entire signal. In addition to the details of the magnified area, the larger portion of the signal may also be required. As a result, the final input information units may range between $x$ units and $\mu x$ units. Using a global tolerance unit is not feasible in such a case. The tolerance factors must be set for individual elements, providing a flexible system to accommodate variations in elements.

The code segment shown in § 6.5.1 provides a way of allowing tolerance factors to become part of the created node. Although this is possible, it may not be effective, as it will eventually occupy too much memory space. An alternative solution is to store the global tolerance factors for each element in the header file of the program. Each element can then possess individual tolerance factors.
6.7 The Voronoi Diagram

The previous sections have presented FLANN II as an algorithm. An alternative way to view FLANN II, is to use a Voronoi diagram. The Voronoi diagram is a fundamental concept in computational geometry (Preparata F.P., et. al., 1985), (Fortune S., 1987), (Sugihara K., et. al., 1992) and can be constructed for visualisation in a $\mathbb{E}_2$ Euclidean space. The basic concept behind the Voronoi diagram is as follows:

Given two points $p$ and $q$, let $d(p,q)$ denote the Euclidean distance between $p$ and $q$. For a finite set $P = \{p_1, ..., p_n\}$ of points in the plane, region $R(p_i)$ is defined by

$$R(p_i) = \{ p \mid d(p,p_i) < d(p,p_j) \text{ for any } j \neq i \}$$

$R(p_i)$ is called the Voronoi region of $p_i$. The Voronoi regions $R(p_1), \ldots, R(p_n)$ make a partition of the plane and this is the Voronoi Diagram for $P$. An element of $P$ is called a generator of the Voronoi diagram. Figure 6.7 shows a Voronoi diagram with 7 generators.

![Voronoi Diagram](image)

Figure 6.7 The Voronoi diagram of 7 generators

The Kohonen minimising equation (4.1) is a distance function to minimise the distance vector between the pattern to be recognised and the previously learned patterns. The pattern vector stored within a cell of the Kohonen network is a
generator within the $n$ dimensional Voronoi diagram, where $n$ is the dimensionality of the Kohonen input vector. It would be correct to say that the Kohonen grid will have a fixed number of Voronoi generators that will eventually be trained to classify patterns within the space. Each node of the Kohonen grid corresponds to a Voronoi generator and the vector stored in the weights represent the position of the generator within the $n$ dimensional space. As the Kohonen algorithm proceeds to introduce variant forms of the original pattern, by multiplying each element of the input vector with the variable $\eta$ and alters the position of the Voronoi generators, spreading the area of influence around the original input vector. This spread of the influence of the input vector corresponds to the "tugging" and "pushing" experienced during the training of the Kohonen grid. A Kohonen grid of 10 X 10 would have a fixed quantity of 100 Voronoi generators within the $n$ dimensional Voronoi diagram.

FLANN II uses a different scheme to introduce the Voronoi generators into the $n$ dimensional space. Instead of fixing the number of Voronoi generators and setting up variants, as the Kohonen system does, FLANN II introduces a tolerance value which replaces the need for variant calculations. The storage of patterns would only require a single node, and variants are considered by the use of the $\delta_1$ tolerance. Since the size of the Voronoi diagram is not defined, FLANN II permits dynamic additions of Voronoi generators into the $n$ dimensional space. This corresponds to the learning of novel input patterns even after the network has been trained. Figure 6.8 and 6.9 illustrate the different models as viewed on a $\mathbb{E}_2$ Voronoi diagram. Two classes are in question and a data point is used from each of the two classes. The Kohonen would introduce variants while FLANN II uses single generators for each data point but harness the help of the tolerance value.
The Voronoi diagram for a FLANN II classification of two patterns, is an infinite line passing through the half plane denoted by $H(p_1,p_2)$. Note that the Kohonen system creates and occupies the entire Voronoi area. The test pattern $X$ will always be classified as one of the points within the Kohonen grid, based on the nearest neighbour criteria. FLANN II on the other hand creates the Voronoi generator and uses the tolerance values $\delta_i$ as the first criteria for the search. Possibilities of no classifications may arise, implying the encounter with a novel pattern. FLANN II would then introduce the test pattern as a new Voronoi generator and the Voronoi diagram expands. Note that the tolerance criteria $\delta_i$ can extend across the edge of the Voronoi half plane, $H(p_1,p_2)$. Any Voronoi generator within the area of tolerance satisfies the tolerance criteria and possesses the chance of competing in the final selection.

Occasionally, due to the tolerance, FLANN II may form overlaps due to closely related patterns. This is illustrated in Figure 6.10. The final match will then be determined by the nearest neighbour recall performed on the set of points that pass the first level selection.
Figure 6.10  \( X \) lies within the bounds of \( R(B) \), but \( B \) is not within the tolerance of \( X \). Instead, the match is contended by \( R(A) \) and \( R(C) \).

Note that if a Voronoi generator violates a boundary of tolerance, it may still continue in competition as long as the left hand side (LHS) value of equation (6.1) is still larger than the vigilance factor \( \rho \). The reason for this is that although \( \delta_i \) is a tolerance test for each individual element of the input vector, an error caused by a single element may not affect the overall performance of the vector in question. This allowance in \( \rho \) provides flexibility to search for closely related patterns.

The difference between the Kohonen network and FLANN II thus becomes clear. As the number of new training patterns increases, FLANN II increases the number of Voronoi generators in the pattern space. This ability to cope with ad hoc learning is beneficial to many real life applications.
6.8 The Algorithm

FLANN II has the flexibility to solve various neural classification problems. The basic algorithm for FLANN II as an on-line learning/classifying neural network is presented here.

Step 1: Initialise Network with \( p \) between 0.5 and 1.0. Determine and set all tolerance values \( \delta_i \). The value of \( p \) and \( \delta_i \) affects the behaviour of the classification and learning process.

Step 2: Present next pattern to the input nodes. If there are no output nodes present, GOTO 6.

Step 3: Determine the set of nodes that are possible matches using equation (1). If no output nodes match, GOTO 6.

\[
\sum_{i=0}^{n} D[\delta_i^2 -(W_{ij} - l_i)^2] \geq p
\]

(copy of 6.1)

Step 4: Use the criteria in equation (2) to determine the winning node from the selected set from Step 3.

\[
winner = \min \left[ \sum_{i=0}^{n} (W_{ij} - l_i)^2 \right]
\]

(copy of 6.2)

Step 5: Match is found. Extract result and GOTO 2.

/*The system can perform forced pattern learning here. Forced learning is a form of supervised learning which copies the weight vector into a new node, regardless of the existence of a match. The only instance forced learning should not create a new node is when the input vector is 100% similar to the output match. In which case, the existing node must not be duplicated.*/

Step 6: Create a new output node and perform direct mapping of input vector into weight vector of new output node and GOTO 2.
6.9 Learning to Tell Two Spirals

In 1988, Alexis P. Wieland of Mitre Corporation officially proposed a benchmark test for artificial neural networks (Lang K.J., et. al., 1988). Two intertwined spirals as shown in Figure 6.11 are used to generate a training set of 194 points. Of which, half the points belonged to one spiral, A, and the other half belonged to the other spiral, B. The desired neural network model was trained with the 194 points and tested for classification properties after the network has been fully trained.

Wieland pointed out several features of this task which make it an interesting test for neural network algorithms. Not only does it require the network to learn a highly non-linear separation of the input space, which is difficult for most current algorithms, its 2 dimensional input space makes it facile to plot the network’s transfer function in order to study the inner workings and development during learning.

![Figure 6.11 Two inter-twining spirals A and B.](image-url)
This benchmark challenge was taken up by Kevin J. Lang and Michael J. Witbrock. Their results were published in (Lang K.J., et. al., 1988). The experiments that they carried out were based on (MLP) models. Their eventual publication provided a clear and accurate account of the capabilities of the MLP models.

Their initial attempts at the problem were undertaken using the conventional back propagation networks, containing few layers of hidden units. These experiments failed, and it convinced them of the difficulty of the task as well as the need for a specific design to solve the problem. In his paper, Lang discussed the benefits from the computational power of many layers, which unfortunately, back propagation learning generally slows down by an order of magnitude each time a layer is added to the network. This is caused by the attenuation of the error signal each time it flows through a layer, and learning progress is limited by the slow adaptation of units in the early stages of the multilayer network.

The eventual network design which Lang used is shown in Figure 6.12. It has 5 hidden nodes in each of the 3 hidden layers with a total of 138 weights. This value of 138 was obtained by an heuristic approximation, that each weight would store 1.5 bits of information (Lang K.J., et. al., 1989). Given a set of 194 training points, there should be approximately 194/1.5 weights. This division yields approximately 130.
Table 6.4 shows the results of the training runs conducted using 3 kinds of MLP models. The first of which was the ordinary back propagation with momentum, followed by the Vanilla back propagation with a cross-entropy error function (Rumelhart, et. al., 1986) and the final model was the Quickprop (Fahlman, 1988). Lang trained his networks until the output unit activation was within 0.4.

<table>
<thead>
<tr>
<th>Run</th>
<th>Learning Algorithm</th>
<th>Vanillia BP</th>
<th>X Entropy BP</th>
<th>Quickprop</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>18 900</td>
<td>16 200</td>
<td>4 500</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>22 300</td>
<td>8 600</td>
<td>12 300</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>19 000</td>
<td>7 600</td>
<td>6 800</td>
</tr>
</tbody>
</table>

Table 6.4 Results from (Lang K.J., et. al., 1988)

The results provided in Table 6.4 show the training epochs required by the network before it was able to recognise a majority of the training points. This does not include test points taken randomly from the universal set of points within the region. They observed that the best representation of this spiral property was near the centre of the picture. This was the region where there was a greater concentration of training points. They continued to enhance the accuracy of the
network by increasing the density of training points by 4 times. This test run with 4 times the point density required 64 000 epochs. Lang's eventual evaluation was based on a test set of 770 cases. Of these 770 cases, there were 76 errors using the run A and 56 errors in run B.

The same experiment was repeated using FLANN II and comparisons were made with the results provided by Lang. A minimum training set of 194 points was used, to train FLANN II in a single epoch. When the network completed its learning in a single epoch, a maximum relaxation of the vigilance factor was done by setting it to 0. This is to allow the network to begin classifications and cease its learning process. A total relaxation of the constraints was necessary to allow the network to classify every test point, no matter how far it is from the next closest Voronoi generator within the stored pattern set. This is not a usual practice, but is used for the purpose of testing its effectiveness in classifying any point in the region. The classifications would therefore use the minimum result obtained from equation (6.2). The network had a total of 2 input nodes and 194 output nodes Each of the 194 output nodes was used to represent a unique point on the spiral. A total of 388 connections were made between the two layers.

After completing the single epoch training, the system was tested with 10 000 points, evenly spread within the square grid defined by the bottom left corner co-ordinate of (-7.0, -7.0) to the top right corner co-ordinate of (7.0, 7.0). The results showed that FLANN II was able to classify almost 100% of the 10 000 points. Due to the accuracy of the neural network, the ASCII image of the results could be printed out and the two distinct spirals can be observed. Unlike the results obtained from the MLP models, the edges of the spirals are clear and distinct (See Lang K.J., et.al., 1988).
The errors in the results are not immediately noticeable, but on closer inspection, one will find several occur at certain boundary points of the spirals. From Figure 6.13, one can safely approximate these errors amount to less than 50 occurrences. The percentage of erroneous results is 0.5%. The accuracy of the network is about 99.5%.

The superiority of FLANN II in this example is clear. The MLP models required several thousand epochs of training, and still could not correctly learn the basic set of training data. When additional points were used to increase the accuracy of classifications, a total of 64,000 epochs were needed to train the network. This is a vast difference from the single epoch training provided by FLANN II. Comparing the accuracy of the network, out of 10,000 training points, less than 50 misclassifications were detected, giving an accuracy level of about 99.5%. The MLP systems were tested with 770 cases and out of these, 76 were incorrect. The error percentage was about 10%, giving the network an accuracy of 90%.
Figure 6.13 ASCII print out of spiral classification results where "0" represents spiral A and "8" represents spiral B. (0 and 8 were used because the character image sizes are similar, allowing a perfect rectangular image.)

Although the accuracy of the system is 99.5%, it is still important to understand the causes of a 0.5% error rate occurrence. In section 6.7, the basic principles of the Voronoi diagram has been explained. Consider the two sets of points in Figure 6.14. There are two classes present in the figure; the first class is represented by the black dots and the second class is represented by white dots. Notice that the
Voronoi diagram for the points meet with rectangular edges. The individual point separators eventually join to form the combined edges that make up the straight line AB. Here, a distinct straight line boundary is present. If random points were used to test the FLANN II classification with this example, one would expect 100% accuracy because of the clear distinction between the two classes.

Unfortunately, examples like the spiral system do not possess such perfect straight line boundaries. A conceptual example of such a system is delineated in Figure 6.15. Here, the points are shifted slightly to form an off-phased training set. Notice that the Voronoi edges that forms the boundary AB is no longer a straight line, but a series of ridged lines joined together. A test point at the point X will cause problems of ambiguity, since it crosses the original boundary but is undetected due to the shift in the training data positions. The 0.5% of errors occurring in the spiral example stems from a similar problem.
6.10 On-Line Control and Fault Diagnosis of a Mixing Process

Control and fault diagnostics is one of the areas where there is a growing usage of neural networks. The paper by Zhang J., et. al., (1992) is one of the few publications that provides adequate information and data to repeat the experiments using the FLANN II system. This is a rare opportunity because industrial application data is often unavailable due to the commercial sensitivity of the projects.

Zhang built his fault-diagnosis system structure with training pairs of specific symptoms, S, and corresponding faults, F. The neural network learns using the training pairs and upon completion is used to classify on-line measurements. Since the raw data is not suitable as direct inputs to the neural network diagnostic system, they have to be pre-processed. Eventually, the trained neural network diagnoses the system for any faults. This scheme is illustrated in Figure 6.16. The neural network based on-line diagnostic system was applied by Zhang on a pilot-scale mixing process for which several different knowledge based systems have been developed and tested. The mixing process is shown in Figure 6.17, where two tanks in cascade receive hot water and cold water supplies. Both streams enter tank 1 and mixing takes place. The contents of tank 1 is passed to tank 2 and subsequently out to a pool tank (not shown in Figure 6.17).
Figure 6.16  The typical neural network based diagnostics

Figure 6.17  The mixing process used by Zhang J., et. al. (1992)
The information pre-processor shown in Figure 6.16 is a quantitative-to-qualitative value converter, which converts the quantitative changes of measurements and controller outputs into a qualitative form. Zhang uses 4 states to classify any kind of raw measurement taken. These states are:

- 0 for unknown
- 1 for increasing
- 2 for steady
- 3 for decreasing

He considers the unknown state as important because there are times when the measuring instruments fail, making data unavailable. The possible faults that Zhang considers are as follows:

- sensor failures
- hand valve 1 is blocked
- hand valve 2 is blocked
- cold water control valve has failed and is giving a high output
- cold water control valve has failed and is giving a low output
- hot water control valve has failed and is giving a high output
- hot water control valve has failed and is giving a low output

The network used an output range value of 0 to 1. Any value greater than 0.6 was considered a fault. Table 6.5 shows the results of the MLP model used to operate the diagnostic system. Each hidden layer possessed 5 hidden nodes and the target system accuracy was 95%.
Table 6.5  Training steps for two networks

<table>
<thead>
<tr>
<th>α</th>
<th>η</th>
<th>Epochs with 1 hidden layer</th>
<th>Epochs with 2 hidden layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>0.6</td>
<td>Fail to converge</td>
<td>Fail to converge</td>
</tr>
<tr>
<td>0.9</td>
<td>0.1</td>
<td>550</td>
<td>1681</td>
</tr>
<tr>
<td>0.8</td>
<td>0.5</td>
<td>274</td>
<td>Fail to converge</td>
</tr>
<tr>
<td>0.8</td>
<td>0.4</td>
<td>380</td>
<td>907</td>
</tr>
<tr>
<td>0.8</td>
<td>0.2</td>
<td>534</td>
<td>2240</td>
</tr>
<tr>
<td>0.6</td>
<td>0.9</td>
<td>828</td>
<td>1361</td>
</tr>
<tr>
<td>0.6</td>
<td>0.2</td>
<td>1324</td>
<td>3898</td>
</tr>
<tr>
<td>0.5</td>
<td>0.4</td>
<td>1131</td>
<td>2297</td>
</tr>
<tr>
<td>0.5</td>
<td>0.2</td>
<td>1796</td>
<td>5215</td>
</tr>
</tbody>
</table>

The training data which Zhang used was obtained from previously developed AI systems. Training pairs consists of two vectors, the symptom vector S and the target vector T. There were 6 elements in S and 6 elements in T. Table 6.6 shows the training set which was used.

Table 6.6  Training pairs S and T and the corresponding faults.

<table>
<thead>
<tr>
<th>Training Pairs</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>S: 1</td>
<td>Hot-water control valve fails low</td>
</tr>
<tr>
<td>T: 0</td>
<td></td>
</tr>
<tr>
<td>S: 2</td>
<td>Cold-water control valve fails low</td>
</tr>
<tr>
<td>T: 0</td>
<td></td>
</tr>
<tr>
<td>S: 2</td>
<td>Hand valve 1 is blocked</td>
</tr>
<tr>
<td>T: 0</td>
<td></td>
</tr>
<tr>
<td>S: 3</td>
<td>Hand valve 2 is blocked</td>
</tr>
<tr>
<td>T: 0</td>
<td></td>
</tr>
<tr>
<td>S: 3</td>
<td>Hot-water control valve fails high</td>
</tr>
<tr>
<td>T: 0</td>
<td></td>
</tr>
<tr>
<td>S: 2</td>
<td>Cold-water control valve fails high</td>
</tr>
<tr>
<td>T: 1</td>
<td></td>
</tr>
<tr>
<td>S: 2</td>
<td>No fault</td>
</tr>
<tr>
<td>T: 0</td>
<td></td>
</tr>
</tbody>
</table>
The MLP model that Zhang used had 6 nodes at the input layer, 5 hidden nodes in the hidden layer and 6 output nodes. The diagnosis of the system would yield a 6 element output vector, which would correspond to the trained target values. The system was tested with partially incorrect data and Zhang's results are shown in Table 6.7. Activation is reached when the diagnostic value is larger than 0.6. Values in the table with "*" indicate an incorrect value in the element. Values with "?" indicate the unavailability of the value.

<table>
<thead>
<tr>
<th>Symptom vectors, Diagnosis vectors, Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>S: 0 1 1 1 3 2*</td>
</tr>
<tr>
<td>D: 0.0108 0.0002 0.0013 0.1356 0.0481 0.7257</td>
</tr>
<tr>
<td>Fault: Hot-water control valve fails low</td>
</tr>
<tr>
<td>S: 2 2 0 2* 3 1</td>
</tr>
<tr>
<td>D: 0.0001 0.0773 0.0863 0.0043 0.8554 0.0075</td>
</tr>
<tr>
<td>Fault: Cold-water control valve fails low</td>
</tr>
<tr>
<td>S: 2 1* 3 1 3 1*</td>
</tr>
<tr>
<td>D: 0.0177 0.0315 0.0000 0.9412 0.0252 0.0109</td>
</tr>
<tr>
<td>Fault: Hand valve 1 is blocked</td>
</tr>
<tr>
<td>S: 2* 2 2 3 1 1*</td>
</tr>
<tr>
<td>D: 0.0179 0.2686 0.6050 0.0005 0.0287 0.0013</td>
</tr>
<tr>
<td>Fault: Hand valve 2 is blocked</td>
</tr>
<tr>
<td>S: 3 3 2* 2 0 1</td>
</tr>
<tr>
<td>D: 0.0051 0.8508 0.1028 0.0041 0.0791 0.0001</td>
</tr>
<tr>
<td>Fault: Hot-water control valve fails high</td>
</tr>
<tr>
<td>S: 0 2 3 2* 1 3</td>
</tr>
<tr>
<td>D: 0.8003 0.0012 0.0007 0.5242 0.0001 0.2958</td>
</tr>
<tr>
<td>Fault: Cold-water control valve fails high</td>
</tr>
</tbody>
</table>

Table 6.7 Results from (Zhang J., et. al., 1992)

Zhang's data was used with a similar diagnostic implementation using FLANN II. Since there were 7 possible faults in the system, the trained network eventually possessed 7 output nodes. The flowchart in Figure 6.18 illustrates the loop that was implemented in the actual program. In such an example, where the input vectors are a fixed distance apart, FLANN II can be designed to run in an automatic mode to perform self-learning.
To determine the vigilance factor, one must first decide the extent which the system will tolerate a variation. Zhang's results, in Table 6.7, indicate a maximum tolerance of a difference of 2 element positions containing wrong values. Using this information, the vigilance factor cannot exceed the fraction given by

$$\rho \leq \left(1 - \frac{N_e}{N_t}\right),$$

where $N_e$ is the number of element positions with an incorrect value and $N_t$ is the total number of vector elements. The value of $\rho$ in Zhang's example is $\rho \leq 0.6667$.

The tolerance value, $\delta$, works out to be 0.5, since the average distance between two states is 0.5. The tolerance should not overlap the different states. The FLANN II program was designed to read a single file which contained both the
training data and the test data. The training data precedes that test data, so that FLANN II can learn the correct responses before the actual classification. Once the parameters for $p$ and $\delta$ are set, FLANN II would automatically learn new vectors and classify learnt exemplars. To facilitate the program, a number code was appended to the end of each training vector and a 0 was appended to the end of the test vectors. If FLANN II encountered a new vector, it would use the number code as a class. Since the test vectors were to be classified, an arbitrary number 0 was used. Table 6.8 shows the data file used for training FLANN II.

<table>
<thead>
<tr>
<th>Vector Input</th>
<th>Class code</th>
<th>Faults (Not fed as program data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 3 3</td>
<td>1</td>
<td>Hot-water control valve fails low</td>
</tr>
<tr>
<td>2 2 1 1 3 1</td>
<td>2</td>
<td>Cold-water control valve fails low</td>
</tr>
<tr>
<td>2 2 3 1 3 2</td>
<td>3</td>
<td>Hand valve 1 is blocked</td>
</tr>
<tr>
<td>3 2 2 3 1 2</td>
<td>4</td>
<td>Hand valve 2 is blocked</td>
</tr>
<tr>
<td>3 3 3 2 1 1</td>
<td>5</td>
<td>Hot-water control valve fails high</td>
</tr>
<tr>
<td>2 2 3 3 1 3</td>
<td>6</td>
<td>Cold-water control valve fails high</td>
</tr>
<tr>
<td>2 2 2 2 2 2</td>
<td>7</td>
<td>No fault</td>
</tr>
<tr>
<td>0 1 1 1 3 2 @</td>
<td>0</td>
<td>@ results can also be compared with examples found in Table 6.7.</td>
</tr>
<tr>
<td>2 2 0 2 3 1 @</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 1 3 1 3 1 @</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 2 2 3 1 1 @</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3 3 2 2 0 1 @</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0 2 3 2 1 3 @</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1 1 1 2 3 3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1 1 2 2 3 3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 2 1 1 3 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 3 1 1 3 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 2 3 1 2 3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 2 2 3 1 2</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.8 Vector inputs for FLANN II

The sample test data was used on FLANN II and the results are tabulated in Table 6.9. Two sets of test data were extracted from Zhang's paper. The first set is from Table 6.7, and are marked with the "@" symbol in Table 6.8. Since the second set of results are exactly similar to Zhang's, it is not presented in this thesis, but Zhang's results are reflected on the right column of the last 6 vectors of Table 6.8.
The execution of FLANN II was smooth and did not encounter any problems. The training data was learnt and the system was able to automatically differentiate between new exemplars and stored exemplars. The results were 100% similar to those obtained by Zhang, except that FLANN II does not provide decimals for the diagnostic vector, but the exact training pairs found in Table 6.6. (i.e. The diagnostic vector $D$ is exactly the replica of the target vector $T$.)

<table>
<thead>
<tr>
<th>Input Vector</th>
<th>Class code (L) = Learned (C) = Classified</th>
<th>Fault decoded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 3 3</td>
<td>1(L)</td>
<td>Hot-water control valve fails low</td>
</tr>
<tr>
<td>2 2 1 1 3 1</td>
<td>2(L)</td>
<td>Cold-water control valve fails low</td>
</tr>
<tr>
<td>2 2 3 1 3 2</td>
<td>3(L)</td>
<td>Hand valve 1 is blocked</td>
</tr>
<tr>
<td>3 2 2 3 1 2</td>
<td>4(L)</td>
<td>Hand valve 2 is blocked</td>
</tr>
<tr>
<td>3 3 3 2 1 1</td>
<td>5(L)</td>
<td>Hot-water control valve fails high</td>
</tr>
<tr>
<td>2 2 3 3 1 3</td>
<td>6(L)</td>
<td>Cold-water control valve fails high</td>
</tr>
<tr>
<td>2 2 2 2 2 2</td>
<td>7(L)</td>
<td>No fault</td>
</tr>
<tr>
<td>0 1 1 1 3 2</td>
<td>1(C)</td>
<td>Hot-water control valve fails low</td>
</tr>
<tr>
<td>2 2 0 2 3 1</td>
<td>2(C)</td>
<td>Cold-water control valve fails low</td>
</tr>
<tr>
<td>2 1 3 1 3 1</td>
<td>3(C)</td>
<td>Hand valve 1 is blocked</td>
</tr>
<tr>
<td>2 2 2 3 1 1</td>
<td>4(C)</td>
<td>Hand valve 2 is blocked</td>
</tr>
<tr>
<td>3 3 2 2 0 1</td>
<td>5(C)</td>
<td>Hot-water control valve fails high</td>
</tr>
<tr>
<td>0 2 3 2 1 3</td>
<td>6(C)</td>
<td>Cold-water control valve fails high</td>
</tr>
<tr>
<td>1 1 1 2 3 3</td>
<td>1(C)</td>
<td>Hot-water control valve fails low</td>
</tr>
<tr>
<td>1 1 2 2 3 3</td>
<td>1(C)</td>
<td>Hot-water control valve fails low</td>
</tr>
<tr>
<td>2 2 1 1 3 1</td>
<td>2(C)</td>
<td>Cold-water control valve fails low</td>
</tr>
<tr>
<td>2 3 1 1 3 1</td>
<td>2(C)</td>
<td>Cold-water control valve fails low</td>
</tr>
<tr>
<td>2 2 3 1 2 3</td>
<td>3(C)</td>
<td>Hand valve 1 is blocked</td>
</tr>
<tr>
<td>2 2 2 3 1 2</td>
<td>4(C)</td>
<td>Hand valve 2 is blocked</td>
</tr>
</tbody>
</table>

Table 6.9  Results from FLANN II execution.

The training was completed within a single epoch and the classification began when the program read the test data section of the file. The automatic learning system was possible because there was a well defined distance between the states and prior knowledge of the types of data expected.
6.11 Neural network based diagnosis of a Continuous Stirred Tank Reactor System

In another example by Zhang, a neural network was developed to diagnose faults in a continuous stirred tank reactor (CSTR) system. Figure 6.19 shows a hypothetical exothermic reaction, which takes place within the reactor vessel. This system is cooled by an external heat exchanger. The temperature and level in the reactor as well as the recycle flow rate are controlled by feedback control systems. Zhang's results are again used for the purpose of demonstrating the capability of FLANN II to operate as an effective neural network model.

![Diagram of CSTR system](image)

Figure 6.19 The Continuous Stirred Tank Reactor (CSTR)

The qualitative model based diagnostic system for the CSTR system is similar to that of the mixing process, where measurements are again based on a 0 - 3 system of measurements. Unlike the mixing system, the CSTR system has 11 possible faults and the input vector now has 14 elements. Considering "No fault" as a single state, there are 12 states.
Zhang built the diagnostic neural network with 14 input nodes, and 12 output nodes. There are 10 nodes within a single hidden layer. The architecture of the neural network is shown in Figure 6.20. The design of the system was such that each node on the output layer corresponds to a fault detected. Each time the inputs were presented to the system, the output layer will reflect the diagnosed single fault.

![Figure 6.20](image)

Figure 6.20 The MLP model used in Zhang's experiment

To obtain the training data, individual faults were deliberately induced on the actual CSTR system and the qualitative deviations were recorded. The training data is again formed into the symptom vector, S, and the target vector, T. The neural network response is the diagnostic vector D. The faults that were possible are listed as follows:

- Pipe 1 is blocked
- External feed-reactant flow too high
- Pipe 2 or 3 is blocked or pump fails
- Pipe 10 or 11 is blocked or control valve 1 fails low
- External feed reactant temperature abnormal
- Control valve 2 fails high
- Pipe 7, 8 or 9 is blocked or control valve 2 fails low
- Control valve 1 fails high
- Pipe 4, 5 or 6 is blocked or control valve 3 fails low
- Control valve 3 fails high
- External feed-reactant concentration too low
- No fault.

The vector training pairs shown in Table 6.10 were used to train the MLP neural network.

<table>
<thead>
<tr>
<th>Training pairs</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>S: 12213113232223</td>
<td>Pipe 1 is blocked</td>
</tr>
<tr>
<td>T: 000000000001</td>
<td></td>
</tr>
<tr>
<td>S: 32231331212221</td>
<td>External feed reactant flow too high</td>
</tr>
<tr>
<td>T: 000000000010</td>
<td></td>
</tr>
<tr>
<td>S: 22231112112231</td>
<td>Pipe 2 or 3 is blocked or pump fails</td>
</tr>
<tr>
<td>T: 000000001000</td>
<td></td>
</tr>
<tr>
<td>S: 22231131212221</td>
<td>Pipe 10 or 11 is blocked or control valve fails low</td>
</tr>
<tr>
<td>T: 000000010000</td>
<td></td>
</tr>
<tr>
<td>S: 22232222232223</td>
<td>External feed reactant temperature abnormal</td>
</tr>
<tr>
<td>T: 000000100000</td>
<td></td>
</tr>
<tr>
<td>S: 22212222232221</td>
<td>Control valve 2 fails high</td>
</tr>
<tr>
<td>T: 000001000000</td>
<td></td>
</tr>
<tr>
<td>S: 22223222212223</td>
<td>Pipe 7, 8 or 9 is blocked or control valve 2 fails low</td>
</tr>
<tr>
<td>T: 000010000000</td>
<td></td>
</tr>
<tr>
<td>S: 22133232322223</td>
<td>Control valve 1 fails high</td>
</tr>
<tr>
<td>T: 000100000000</td>
<td></td>
</tr>
<tr>
<td>S: 22223222132232</td>
<td>Pipe 4, 5 or 6 is blocked or control valve 3 fails low</td>
</tr>
<tr>
<td>T: 001000000000</td>
<td></td>
</tr>
<tr>
<td>S: 22222223222212</td>
<td>Control valve 3 fails high</td>
</tr>
<tr>
<td>T: 010000000000</td>
<td></td>
</tr>
<tr>
<td>S: 22121212212221</td>
<td>External feed reactant concentration too low</td>
</tr>
<tr>
<td>T: 100000000000</td>
<td></td>
</tr>
<tr>
<td>S: 22222222222222</td>
<td>No fault</td>
</tr>
<tr>
<td>T: 00000000000000</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.10  The CSTR network training pairs

The results of Zhang's experiment are found in Table 6.11. The diagnostic faults made by the MLP model never gives the exact value of 0 or 1, but provide decimal values between 0 and 1. The activation value for a 1 is taken at 0.6 and any value less than 0.6 is considered a 0.
### S = Symptom vectors, D = Diagnosis vectors

<table>
<thead>
<tr>
<th>S:</th>
<th>1</th>
<th>2</th>
<th>0^2</th>
<th>1</th>
<th>3</th>
<th>0^2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3*</td>
<td>3</td>
<td>2</td>
<td>3*</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>D:</td>
<td>0.0023</td>
<td>0.0113</td>
<td>0.0061</td>
<td>0.0110</td>
<td>0.0053</td>
<td>0.0112</td>
<td>0.0325</td>
</tr>
<tr>
<td>Fault: Pipe 1 is blocked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S:</td>
<td>3</td>
<td>0^2</td>
<td>2</td>
<td>2*</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2*</td>
<td>1</td>
<td>3*</td>
<td>2</td>
<td>0^2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>D:</td>
<td>0.0844</td>
<td>0.2207</td>
<td>0.0001</td>
<td>0.1529</td>
<td>0.0002</td>
<td>0.3512</td>
<td>0.0003</td>
</tr>
<tr>
<td>Fault: External feed-reactant flow is too high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S:</td>
<td>0^2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1*</td>
</tr>
<tr>
<td>1</td>
<td>2*</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0^2</td>
<td></td>
</tr>
<tr>
<td>D:</td>
<td>0.0231</td>
<td>0.0000</td>
<td>0.0151</td>
<td>0.0000</td>
<td>0.0048</td>
<td>0.0113</td>
<td>0.0001</td>
</tr>
<tr>
<td>Fault: Pipe 2 or 3 is blocked or pump fails</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S:</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2*</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0^2</td>
<td>1*</td>
<td>1</td>
<td>2</td>
<td>3*</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>D:</td>
<td>0.0002</td>
<td>0.0041</td>
<td>0.0008</td>
<td>0.0000</td>
<td>0.0013</td>
<td>0.0279</td>
<td>0.0445</td>
</tr>
<tr>
<td>0.9089</td>
<td>0.0430</td>
<td>0.0106</td>
<td>0.0006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault: Pipe 10 or 11 is blocked or control valve 1 fails low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S:</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>0^2</td>
<td>2*</td>
<td>2</td>
<td>3*</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3*</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>D:</td>
<td>0.0000</td>
<td>0.0658</td>
<td>0.0180</td>
<td>0.1835</td>
<td>0.0001</td>
<td>0.2600</td>
<td>0.7801</td>
</tr>
<tr>
<td>0.0002</td>
<td>0.0000</td>
<td>0.0009</td>
<td>0.0223</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault: External feed reactant temperature abnormal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S:</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0^2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1*</td>
<td>3</td>
<td>1*</td>
<td>2</td>
<td>2</td>
<td>0^2</td>
<td></td>
</tr>
<tr>
<td>D:</td>
<td>0.0022</td>
<td>0.0017</td>
<td>0.0189</td>
<td>0.0131</td>
<td>0.0000</td>
<td>0.9757</td>
<td>0.0028</td>
</tr>
<tr>
<td>0.0023</td>
<td>0.0005</td>
<td>0.0090</td>
<td>0.0032</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault: Control valve 2 fails high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6  Fast Learning Artificial Neural Network II (FLANN II)

The experiment conducted by Zhang was repeated using the FLANN II model. Since the average difference between two states is again 0.5, the value for $\delta$ was also set to 0.5. The value for $\rho$ is now set to 0.7, since there is a possibility that 4 out of 14 elements are incorrect. The results of the FLANN II implementation are in Table 6.12. The program flowchart is similar to that shown in Figure 6.18. A single file containing the training pairs and the test data was used. Instead of using a 11 element diagnostic vector, a single integer code was used, beginning from 1 and ending at 12.
<table>
<thead>
<tr>
<th>Input Vector</th>
<th>Code (L) = Learned (C) = Classified</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 2 1 3 1 1 3 2 3 2 2 2 3</td>
<td>1(L)</td>
<td>Pipe 1 is blocked</td>
</tr>
<tr>
<td>3 2 2 3 1 3 3 1 2 1 2 2 2 1</td>
<td>2(L)</td>
<td>External feed reactant flow too high</td>
</tr>
<tr>
<td>2 2 2 3 1 1 2 1 1 1 2 2 3 1</td>
<td>3(L)</td>
<td>Pipe 2 or 3 is blocked or pump fails</td>
</tr>
<tr>
<td>2 2 2 3 1 1 3 1 2 1 2 2 2 1</td>
<td>4(L)</td>
<td>Pipe 10 or 11 is blocked or control valve 1 fails low</td>
</tr>
<tr>
<td>2 3 2 3 2 2 2 2 2 2 2 2 2 2 3</td>
<td>5(L)</td>
<td>External feed reactant temperature abnormal</td>
</tr>
<tr>
<td>2 2 2 2 1 2 2 2 2 2 2 2 2 2 1</td>
<td>6(L)</td>
<td>Control valve 2 fails high</td>
</tr>
<tr>
<td>2 2 2 3 2 2 2 2 1 2 2 2 3</td>
<td>7(L)</td>
<td>Pipe 7, 8 or 9 is blocked or control valve 2 fails low</td>
</tr>
<tr>
<td>2 2 1 3 3 2 3 2 3 2 2 2 2 3</td>
<td>8(L)</td>
<td>Control valve 1 fails high</td>
</tr>
<tr>
<td>2 2 2 3 2 2 2 2 1 3 2 2 3 2</td>
<td>9(L)</td>
<td>Pipe 4, 5 or 6 is blocked or control valve 3 fails low</td>
</tr>
<tr>
<td>2 2 2 2 2 2 2 2 2 2 2 1 2 1 2</td>
<td>10(L)</td>
<td>Control valve 3 fails high</td>
</tr>
<tr>
<td>2 2 1 2 1 2 1 2 2 1 2 2 2 1</td>
<td>11(L)</td>
<td>External feed reactant concentration too low</td>
</tr>
<tr>
<td>2 2 2 2 2 2 2 2 2 2 2 2 2 2 2</td>
<td>12(L)</td>
<td>No fault</td>
</tr>
<tr>
<td>1 2 0 1 3 0 1 3 3 3 2 3 2 3</td>
<td>1(C)</td>
<td>Pipe 1 is blocked</td>
</tr>
<tr>
<td>3 0 2 2 1 3 3 1 2 1 3 2 0 1</td>
<td>2(C)</td>
<td>External feed reactant flow too high</td>
</tr>
<tr>
<td>0 2 2 3 1 1 1 1 2 1 2 2 3 0</td>
<td>3(C)</td>
<td>Pipe 2 or 3 is blocked or pump fails</td>
</tr>
<tr>
<td>2 2 2 2 1 1 3 0 1 1 2 3 2 1</td>
<td>4(C)</td>
<td>Pipe 10 or 11 is blocked or control valve 1 fails low</td>
</tr>
<tr>
<td>2 3 2 0 2 2 3 2 2 3 2 3 2 3</td>
<td>5(C)</td>
<td>External feed reactant temperature abnormal</td>
</tr>
<tr>
<td>2 2 2 2 0 2 2 2 1 3 1 2 2 0</td>
<td>6(C)</td>
<td>Control valve 2 fails high</td>
</tr>
<tr>
<td>2 3 2 2 3 2 0 2 3 1 2 0 2 3</td>
<td>7(C)</td>
<td>Pipe 7, 8 or 9 is blocked or control valve 2 fails low</td>
</tr>
<tr>
<td>2 3 1 0 3 3 2 3 2 3 2 0 2 3</td>
<td>8(C)</td>
<td>Control valve 1 fails high</td>
</tr>
<tr>
<td>0 2 2 2 3 2 3 2 1 3 1 2 3 2</td>
<td>9(C)</td>
<td>Pipe 4, 5 or 6 is blocked or control valve 3 fails low</td>
</tr>
<tr>
<td>2 0 2 2 3 2 2 2 3 2 2 2 1 2</td>
<td>10(C)</td>
<td>Control valve 3 fails high</td>
</tr>
<tr>
<td>2 2 0 2 1 2 0 2 1 1 2 2 2 0</td>
<td>11(C)</td>
<td>External feed reactant concentration too low</td>
</tr>
</tbody>
</table>

Table 6.12 The results obtained from FLANN II
The results obtained from the second control diagnostic experiment shows that the FLANN II model was able to correctly identify all the faults presented to it. The execution of the neural network was fast, as it could differentiate the 12 states from the input file and began classification upon the encounter of any repeated patterns.

6.12 Noisy Trigonometric Wave

In the previous 3 experiments, FLANN II was tested on its effectiveness of the vigilance factor \( \rho \) and the nearest neighbour search paradigm. The next example described was specifically designed to test the effectiveness of the tolerance factor, \( \delta \). Four distinct trigonometric waveforms were used in this experiment. These are,

- \( \sin(A) \)
- \( \sin(2A) \)
- \( \cos(A) \)
- \( \cos(2A) \)

The FLANN II network is trained with an accurate sample of each waveform. Immediately after a single presentation, test data with random amplitude distortions are presented to the neural network. Figure 6.21 shows the diagram of the 4 waveforms used for training. (These have been superimposed on each other, but in the experiment, they were presented one at a time.) Each waveform was sampled 100 times between 0 and 4\( \pi \). The actual training vector is not tabulated in this thesis due to its extreme length, but the vector generator program can be found in the accompanying disk. With the same program, the waveforms can be generated with a random noise distortion.
Figure 6.21 The 4 trigonometric waveforms used for testing the tolerance factor $\delta$ in FLANN II.

The results of the noisy waveform identification showed that FLANN II could recognise correctly 100% of the test data. A 10 element sample is provided in Table 6.11, where a sine wave data set is reproduced. The first vector presented is the sine wave data. The others that follow are from the distorted waveform. In the actual experiment, 4 of the waveforms were trained together at the beginning and the distorted waveforms were presented in random sequence. FLANN II was successful in identifying to which function the distorted waveforms belonged.

<table>
<thead>
<tr>
<th></th>
<th>0.0000</th>
<th>0.125</th>
<th>0.248</th>
<th>0.368</th>
<th>0.481</th>
<th>0.587</th>
<th>0.684</th>
<th>0.770</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0187</td>
<td>0.175</td>
<td>0.231</td>
<td>0.439</td>
<td>0.430</td>
<td>0.630</td>
<td>0.653</td>
<td>0.849</td>
<td></td>
</tr>
<tr>
<td>0.0205</td>
<td>0.204</td>
<td>0.183</td>
<td>0.416</td>
<td>0.402</td>
<td>0.640</td>
<td>0.645</td>
<td>0.841</td>
<td></td>
</tr>
<tr>
<td>0.0218</td>
<td>0.144</td>
<td>0.223</td>
<td>0.392</td>
<td>0.462</td>
<td>0.651</td>
<td>0.636</td>
<td>0.833</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.13 Partial data from the actual $\sin(A)$ sampling with distortions introduced into the last 3 rows
6.13 Comparison of FLANN II with other models

All the comparison results between FLANN I and the other architectures can be applied to the FLANN II model. The most distinct feature to both neural networks is the single epoch learning speed of the network. The power of this fast learning capability is seen when the neural network is trained by reading sample data and immediately begins to classify the test data appended to the end of the training data set. The points mentioned in this section are additional features not found in FLANN I.

6.13.1 Individual Element Comparisons and Resolutions

The ability to perform comparisons based on different resolutions is an important aspect of the neural network. In the conventional MLP solution, it is often difficult to train a single network with vastly different resolutions. To overcome such problems, MLP designs require a pre-processing stage, which manipulates the data to suit the network. In Zhang's experiments, this is done in the information pre-processing stage, illustrated in Figure 6.16. For the FLANN II to handle this, it has to switch to the correct resolution for the particular element. This is done by performing calculations with the corresponding tolerance factor of the element. This ability to control various resolutions provides a smoother operation in the FLANN II model.

6.13.2 Real Number Operations

FLANN II is unlike the FLANN I and ART1 network, where the input vectors are binary. It is very similar to the Kohonen network operations, but the differences have been discussed in section 6.6. By comparison with the MLP model, FLANN II is able to learn extremely fast. In Zhang's results shown in Table 6.5, some of the
timings takes up to 5000 epochs. Occasionally, the training becomes hampered by divergent runs. Furthermore, the MLP is sometimes plagued with local minima, which prevents the system from reaching a solution.

6.13.3 System Accuracy

As illustrated in the twin spiral problem, the MLP solution required up to 64 000 epochs for the system to begin to recognise the training set. The results of the classification, based on 770 test samples was about 90%, which means 1 in 10 test sets are wrongly classified. In addition, the authors, (Lang J., et. al., 1988) found it difficult to increase the accuracy of the MLP network. The FLANN II network was trained in a single epoch and the classification success rate was about 99.5% based on a test set of 10 000 points (1 in 100 are wrongly classified). The power of the nearest neighbour paradigm is harnessed within this neural network system and allows smooth training conditions.

6.13.4 Individual Element Tolerance

Unlike the FLANN I model and the ART1 system, where input data has to be specifically "1"s and "0"s, FLANN II can operate on real numbers. The input vectors do not need to be exactly similar to the trained data. The tolerance factor $\delta$ for each element permits noisy data to be classified successfully. In the fourth experiment, where noisy trigonometric data was classified, the waveforms were totally distorted by random changes in the amplitudes. FLANN II was still able to classify these with a 100% accuracy.
Chapter 7

Parallel Distributed Neural Networks (PDNN)
In Chapters 5 and 6, two novel neural network models were developed with fast learning speeds of 1 epoch. The experiments conducted, show that these models were efficient learning systems with reliable classification properties. One may interpret these developments as a successful achievement of the original objective stated in § 1.4. This chapter ventures a step further to demonstrate the potential usefulness of FLANN in the parallel environment. These investigations, of FLANN in the parallel environment, eventually produced a new paradigm involving parallel neural computing. Before proceeding further, a brief introduction of parallel computing is provided. Section 7.1 deals with Flynn's taxonomy, explaining the four categories of computer architectures. Section 7.2 elaborates on the differences found in shared memory systems and distributed memory systems and section 7.3 discusses several different network topologies.

7.1 Parallel Architectures and Terminology

In general, digital computers may be classified into four categories, according to the multiplicity of instruction and data streams. This scheme of classification was introduced by Michael J. Flynn and is often referred to as Flynn's Taxonomy (Flynn, 1966). The essential computing process is the execution of a sequence of instructions on a set of data. The term stream is used to denote a sequence of items (instructions or data) as executed or operated on by a single processor. The four classes of Flynn's taxonomy are as follows:

- Single Instruction stream - Single Data stream (SISD)
- Single Instruction stream - Multiple Data stream (SIMD)
- Multiple Instruction stream - Single Data stream (MISD)
- Multiple Instruction stream - Multiple Data stream (MIMD)
7.1.1 SISD

Most serial computers fall into this category. Although the instruction execution may be pipelined, computers in this category can decode a single instruction in unit time. An SISD computer may have multiple functional units, but these are under the control of a single control unit. Figure 7.1 shows the organisation of a SISD computer.

![The SISD computer](image)

7.1.2 SIMD

The SIMD category includes processor arrays with a common control unit. This control unit is responsible for broadcasting instructions to some or all processors to perform similar operations on different data sets. Hence in a single time unit, a single operation is in the same state of execution on multiple processing units, each manipulating different data. Figure 7.2 illustrates the concept of SIMD computers.
Chapter 7 Parallel Distributed Neural Networks (PDNNs)

Figure 7.2 The SIMD computer

7.1.3 MIMD

The MIMD category is a concept in Flynn's taxonomy, but no successful embodiment of this class exists as a product. The MIMD class possesses \( n \) processors, each receiving distinct instructions operating over the same data stream and its derivative. This structure has received less attention and has been challenged as impractical by some architects.

7.1.4 MIMD

The MIMD category embraces multiprocessors capable of executing several independent programs using different sets of data simultaneously. They constitute the more general type of processors. These can sometimes be regarded as independent arrays of Von Neuman processors, capable of communicating with one another either by sharing all or part of their storage space by message passing techniques. Two possible forms of MIMD systems are shown in Figures 7.3 and
7.4. Figure 7.3 illustrates the MIMD with a shared memory module and the system in Figure 7.4 shows a distributed memory MIMD system.

Figure 7.3 The MIMD Computer (Shared Memory)

Figure 7.4 The MIMD Computer (Distributed Memory)
7.2 Multiprocessor Systems

A multiprocessor system can be classified as being either tightly coupled or loosely coupled. Enslow (1977) originated the definition for tightly coupled multiprocessor systems which gained a wide acceptance. A year later, Enslow (1988) proposed the definition for loosely coupled multiprocessor systems.

7.2.1 Tightly Coupled Multiprocessor System

Enslow defines a tightly coupled multiprocessor system to satisfy the following properties:

- It must contain 2 or more processors of approximately comparable capabilities.
- All processors share access to a common memory. This does not preclude the existence of local memories for each or some of the processors.
- All processors share access to I/O channels, control units and devices. This does not preclude the existence of some local I/O interface and devices.
- The entire system is controlled by one operating system.

Tightly coupled multiprocessor systems are also known as shared memory machines or bus-oriented systems. Figure 7.5 illustrates a tightly coupled multiprocessor system. Their communication network consists of one or more system buses, to which all system components are interconnected. The advantage of such a bus structure is that it offers flexibility in the processor topology. (Topologies will be discussed in section 7.3.)
Another advantage of this system is the ease of adding and removing components and subsystems to and from the bus. The two main disadvantages of such an architecture are:

- The system bus which allows communications between 2 devices can become a bottle neck when excessive traffic occurs.
- Failure of the system bus will cause a total failure of the machine

To reduce the traffic over the system bus, some tightly coupled systems utilise cache memory modules by strategically placing them between the system bus and the individual processors. In this way, the frequently accessed data which may still exist within the cache memory is reused without further access through the system bus. This then reduces the chances for bus contention in the system. An example of a tightly coupled multiprocessor system is the Sequent Balance 8000.

### 7.2.2 Loosely Coupled Multiprocessor System

Enslow defined the loosely coupled system as having the following properties:
- A multiplicity of general purpose, physical and logical resources that can be assigned to specific tasks on a dynamic basis.

- A physical distribution of the above resources, interacting through a communications network.

- A high level operating system that unifies and integrates the control of distributed components. Individual processors may have their own local operating systems.

- System transparency which permits services to be requested by name only, without having to identify the serving resources.

Loosely coupled systems are sometimes known as local memory machines or distributed systems. Figure 7.6 shows the typical architecture of a loosely coupled system.

![Figure 7.6 The Architecture of a Typical Loosely Coupled System](image)

7.3 Network Topologies

In the previous two sections, the discussions have centred about the classification of machines and the distinctions between tightly coupled systems and loosely coupled systems. Another factor that affects the parallelisation of a system is the network topology on which the processors are built. This is significant because the network topology utilised determines the number of hops the communication
message needs in order to reach the destination processor. The number of hops in a path from source to destination node is equal to the number of point-to-point links a message must traverse to reach its destination. Figure 7.7 shows some of the possible network topologies.

![Diagram of network topologies: Linear Network, Toroidal Mesh, Star Network, Tree Network, Hypercube.](image)

Figure 7.7 Various Network Topologies a) Linear Network b) Toroidal Mesh c) Star Network d) Tree Network e) Hypercube
Chapter 7 Parallel Distributed Neural Networks (PDNNs)

The messages passed within the linear network in Figure 7.7a has a maximum of 3 hops. Although the toroidal network found in Figure 7.7b has 4 times as many processors, the network has a maximum of 4 hops. Similarly, the network topologies in Figure 7.7c and 7.7d both require a maximum of 2 hops. The Hypercube is also commonly utilised, and it is represented in Figure 7.7e. It requires only a maximum of 3 hops.

The kind of topology selected for any parallel system is very much dependent on the problem requirements and the hardware processor. Shared memory multiprocessor systems such as the Balance 8000 have the added advantage of forming various topologies because the inter-processor communications is done through the system bus supporting the shared memory system. For processors like the Transputer, which is built with 4 to 6 communications links, some topologies cannot be physically built. Figure 7.7c shows the star network which cannot be built using a Transputer network. This is mainly due to the physical constraints of the individual processor. In the case of the T800 Transputer, where 4 links are available, the star topology can only be a connection with a maximum of four other Transputers but the topology in Figure 7.7c requires 8 links.

7.4 Platforms used for FLANN model Implementations

At the Parallel Algorithms Research Centre, three parallel computing environments were available. As such, the FLANN models were tested on all 3 platforms. The objective of the implementations were to study the feasibility of FLANN models in the parallel environment, and if they were practically efficient, it was necessary to understand the portability of FLANN between different parallel environments. The 3 environments available are listed as follows:-
Chapter 7 Parallel Distributed Neural Networks (PDNNs)

- Transputer T800 MIMD environment
- Sequent Balance 8000 MIMD environment
- Parallel Virtual Machine (Heterogeneous PVM) MIMD environment

7.4.1 The Transputer MIMD Environment

The Transputer environment is a set of multiprocessors with local memory units. Transputers usually come in 3 possible basic processing units. These are the T414 series with 4 inter-processor communication links and no built-in coprocessor, the T800 series with 4 communication links and a built-in coprocessor and finally the recent T9000 series with 6 communications links and a built-in coprocessor. The Transputers used in the experiments were of the T800 series, connected in a 4 by 4 array. Since there is local memory on each processor, it is classified as a loosely coupled system.

Figure 7.8 shows the block diagram of the internal structure of a single T800 Transputer. The T414 Transputers are quite similar to the T800 Transputers, except for the absence of a floating point unit (FPU) in the T414. Both processors possess 4 communications links which are used for message passing and system organisation.
In the network topology discussion of section 7.3, it was briefly stated that the topology is dependent also on the processor involved. The star network in Figure 7.7c is an example of a topology that the Transputer is unable to cope with. This is due to the constraint of possessing only 4 communications links. The makers of the Transputer system have continued to pursue the need for more links. As a result, they are now introducing the newest range of Transputers, the T9000 processors. These Transputers possess 6 links and parallel computers built from the T9000 Transputers can now be configured using software links, where the topology is constructed by software specifications and virtual links are made. This new range of Transputers can allow the simulation of topologies such as the star network.

Although the star network topology cannot be physically built using the Transputers, other effective topologies such as the tree or the toroidal structure are well within the design capabilities of the Transputers with 4 links.
A set back in these systems is the inability for all processors, except the root Transputer, to perform I/O services through the host computer. Screen operations and device access is limited only to the root Transputer. As a result, it is difficult to monitor the processes being executed in the slave Transputers.

7.4.2 The Sequent Balance MIMD Environment

The Balance 8000 environment uses a shared memory system that serves 12 general purpose 32 bit microprocessors. This is a true multiprocessor environment which represents a tightly coupled system. This system has the advantage of performing input/output on the same memory space and slave processors can be easily monitored. Some characteristics of the Sequent Balance 8000 parallel computer are as follows:-

- It is a tightly coupled system, and all the processors share a single pool of memory to enhance resource sharing and communication among different processors.

- All processors, memory models and I/O controllers are plugged into a single high speed bus, making it simple to add processors, memory and I/O bandwidth.

- Processors automatically perform load balancing to ensure that the load is distributed between all available processors

A significant advantage of the Sequent shared memory system is the ability for any processor to access the I/O ports. This feature allows the programmers to monitor intermediate states of the parallel program, making error detection more effective. The structure of the Sequent machine is generally similar to that illustrated in Figure 7.5.
7.4.3 The Heterogeneous PVM Environment

The Heterogeneous Parallel Virtual Machine (PVM) environment is a novel concept which utilises the power of several UNIX workstations connected through the computer network to form a virtual parallel computer. PVM is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single computational resource. It is an experimental system developed in the USA that supports FORTRAN and C. The resultant parallel machine is quite like the Transputer system, except that the Ethernet connections possess a slower inter-processor connection but a large bandwidth. Figure 7.9 illustrates the connection topology.

![Figure 7.9 PARC's DEC Alpha network configuration and PVM resides only on the two DEC Alphas](image)

This environment is emerging strongly in the field of parallel computing where it becomes a practical resource for both parallel programmers and conventional sequential programmers. The Heterogeneous PVM environment provides the power of an MIMD system with less restrictions, compared to the Transputer programming environment. This is due to the advances of networking technology.
Unfortunately the delay times currently experienced in the network protocols reduce the efficiency of the parallel technology. As such, these systems are expected to have slower computational speeds, compared to true parallel computers.

Figure 7.10 shows another configuration suitable for connecting PVM platform workstations. This is done with a megaswitch that interfaces the systems. The advantage of this system is the isolation from the unwanted traffic experienced in the normal Ethernet connection. As highlighted in § 7.8.3, the results of the PVM execution over the Ethernet indicate erratic timings which ranges from 1.03 seconds to 1.31 seconds. This difference is almost 25% of the average timing, which is caused by the prevailing traffic on the Ethernet. The megaswitch increases the performance of the virtual machine by keeping the message passing tasks to the isolated fibre optic network.
7.5 Parallel Computing in Current Neural Network Architectures

In the attempts to speed up processing times in neural computing, researchers have resorted to parallelising some neural network architectures. An example of such a procedure can be found in Sanossian (1992). In her experiments, the MLP model was used and the hidden node calculation process was parallelised. The method of parallelisation is illustrated in Figure 7.11, where the elements of the input vector are shared between the partitioned hidden layers. The data is processed in parallel and transferred to the third layer which combines the separately processed data, to eventually classify the input vector.

![Figure 7.11 A typical MLP parallelisation technique](image)

This section investigates a different form of parallelism. The concept introduced, is one that has not been widely exploited for neural networks. Instead of introducing the partitioning process at the hidden layer, parallelisation begins at the input layer. Although this may seem like a simple shift of parallelising positions, the explanation that follows will show that it is a non-trivial task to design using most neural network models, and is extremely difficult to implement.

The input pattern is first divided into independent sub-patterns where each subsequent parallelised embedded node will receive only a portion of the full
vector. Classification of the full vector is achieved when each independent sub-vector is recombined in the deeper layers of the network structure.

If the MLP network was designed to work on such a parallelisation scheme, it would probably look like the model shown in Figure 7.12. A closer look at Figure 7.12 will reveal that the number of calculations, estimated by the number of connections, have been reduced. This reduction in the number of connections also implies a natural reduction of the processing time per classification. Unfortunately, this concept pivots on the false assumption that the technique can be implemented on the MLP model. The MLP model cannot support this concept because each node in the hidden layer requires the full information of the input pattern (See Chapter 3).

![Parallel MLP Model Diagram](image)

**Figure 7.12**  A theoretical parallel MLP model

The sacred binding relationship that prevents the separation of input patterns is not only inherent to the MLP model, but also present in most of the architectures studied in this thesis. This is not to say that building such a parallel system is impossible for these architectures. It is possible for a neural network with such a parallel architecture to be built, but the effort used to design the system would be too complex for practical purposes.
Chapter 7 Parallel Distributed Neural Networks (PDNNs)

The Neocognitron to a large extent, is an attempt to implement cascading levels of cell like structures to perform pattern recognition (Fukushima, 1988). Although the classification results were impressive, the high overheads during the training process, and a massive usage of computer memory posed real problems. Fukushima's network eventually used 9 input layers and up to 70,000 cells.

7.6 Parallel Neural Computing using FLANN

Although the form of parallelism introduced in § 7.5 was not feasible in most of the common architectures, it was implemented using the FLANN models. This was feasible for the FLANN models because of their ability to learn patterns in a single epoch and yet maintain a stable storage of exemplars. A simple representation of the experimental network is delineated in Figure 7.13.

From Figure 7.13, observe the break down of the 10 element input vector into two 5 element input vectors. The original relationship of the vectors can be perceived of as being partitioned at the first level of processing. The outputs of the two FLANN classifications are then brought together by another neural network in the next layer, combining the two vectors to form a correct classification.

![Figure 7.13 Multiple FLANN networks](image)
The assumption in this strategy is that the input vector is a result of a concatenation from several sources of inputs, such as the experiments encountered in section 6.10 and 6.11. This concatenation usually implies the independence of relationships between input elements. In which case, the input elements have no logical significance to neighbouring elements, but each play a small part in forming the pattern as a whole.

Referring to the network found in Figure 7.13, begin with the left input vector (Input Vector 1). This is fed into a FLANN model, which could either be of a FLANN I or FLANN II neural network, depending on the input specification. From the characteristics of FLANN in chapters 5 and 6, the neural network would immediately learn the vector or classify it, depending on whether it is a new pattern or a stored pattern. In either case, the class will immediately be reflected in the winning node of the output layer. While the left input vector is being processed by the left neural network, a concurrent classification process is occurring in the right neural network, which would result in a classification of the input vector on the right (Input Vector 2). The output nodes of the two neural networks would then reflect the final classifications of the first level neural networks. These are then fed to the third neural network in the deeper layer. The two classifications can be thought as propagating into the final layer to be recombined as a single entity, or it could be viewed as inputs to new neural network. Which ever the case, the result, is a successful classification which works through the many layers. The resultant learning rate of the network of neural networks is still 1 epoch.

Note that this system of divide and conquer can only function if the vector input partitioning is legitimate, i.e. the problem allows the vectors to be partitioned in this manner. Some of these partitions form naturally into a composite structure, much like the human face being made of a pair of eyes, a nose, a mouth and a pair
of ears etc. These organs form the naturally occurring composite set within the facial recognition context. Similarly, data coming from a process control system may sometimes be partitioned by the location of the instrument readings, such as the fluid level, temperature and pressure within a particular tank found in the whole process.

If the assumption of data partitionability is true for a particular problem, then FLANN is able to achieve this feat of parallel divide and conquer. With these possibilities, it is now feasible to implement FLANN as a hierarchical network of neural networks. These could be designed and mapped into the loosely coupled MIMD environment. In the other models such as the MLP model, immediate learning is not possible. As such, it is difficult to train the deeper layers with such efficiency and the concept of cascaded networks becomes difficult to achieve. In summary, the characteristics of the two FLANN models that facilitate the application of parallelism into the models are as follows:

- Fast Learning Capability (Learns in 1 epoch)
- Weight values are direct functions of the input vector
- Compact algorithm
- Capacity of the final network is independent of original configurations
- Capable of self-learning

Unlike other models, FLANN is able to derive its weights directly from the training vector. This aids the learning process by not requiring extra cycles to descend to a global minima. The compactness of the algorithm means that the actual neural engine is small, requiring little space in each processor of the processor array. As the number of training patterns increase, FLANN will be able to handle the influx by allocating extra system memory space. Since FLANN can be set on an automatic learning mode, the deeper layers can function autonomously.
These characteristics provide a platform for a parallelisation strategy that involves the segmentation of the input vectors into independent sub-vectors. The segmentation would depend on the data partitionability explained previously in this section. This form of parallelisation introduces a new and powerful paradigm to modularise neural computing: Parallel Distributed Neural Networks (PDNNs). The ability to obtain input vector independence and cascaded networks of neural networks is an extremely powerful paradigm. This concept which can be applied in any composite problem, can unleash a new set of possibilities. It is at this juncture that the newly developed FLANN models achieve a parallel status as Parallel Distributed Neural Networks (PDNN).

7.7 The Implications of Parallel Distributed Neural Networks (PDNNs)

The concept of PDNN is not just another neural network architecture that simply performs pattern recognition with generality and accuracy. There are several other important properties found in such a system and these are listed as follows:-

- Faster Response Speed
- Larger Storage Capacity and Classification Capacity
- Better Scalability and Portability

7.7.1 The Faster Response Speed

The typical operations of the FLANN model perform searches for existing stored patterns. With the help of the weights and an inter-nodal competition, best matches are selected. It is therefore conceivable that as the number of patterns continue to increase, a threshold limit for the neural network's efficiency may be reached. (Seldom has this limit been reached as most neural network applications do not require many unique patterns.)
With the ability to divide the input vector into independent sub-vectors, the processing speeds may be increased by an exponential time. Using a parity problem as an example, this point can illustrated with clarity.

A test problem may possess 10 binary input nodes. This would provide a worst case of $2^{10}$ output possibilities and produce 1024 output nodes with the correct parity. Figure 7.14 illustrates what the network may be like using a 10 element input vector. A sequential implementation may be slow due to the massive search space.

![Figure 7.14](image)

Figure 7.14 A single 10 element input vector will generate $2^{10}$ output possibilities on FLANN models.

If the system was parallelised and the input vectors were broken into independent sub-vectors of 2 elements, as seen in Figure 7.15, the worst case competition for the two parity bits is 4. Since there are 5 sub-divisions in a 10 bit vector, the next re-combining layer will have another $2 \times 2^2$ competition steps. The worst time taken for the whole system would be $4 \times 2^2$ or 16 competition steps, since there are 4 layers. The total number of competitions that take place in the entire model becomes $9 \times 2^2$ which is of an exponential reduction from the original $2^{10}$ order of complexity. Figure 7.15 illustrates the same parity problem solved using a network of 9 FLANN models.
7.7.2 Storage Capacity and Classification Capacity

Current day neural network applications may be required to operate on large sets of image data. As the image size reaches 512 X 512 pixel sizes, the total number of pixels to process becomes too huge for practical image recognition applications, even though there may only be a small set of actual patterns. Most applications reduce the number of bits by methods such as blending, but by doing so, it jeopardises the information integrity and accuracy of the final result.

By the application of the PDNN concept, images may now be segmented into smaller clusters, reducing not only the speed of processing, but the size of the memory requirements within the processor. Processors such as the T800s and i860s are able to provide the services of local memory to perform its own localised calculations.
Referring to the parity problem, if the 10 element vector was divided into 5 equal segments, the storage requirement for the entire network would also reduce by an exponential value. Let the number of possible patterns be $2^{10}$. By dividing the vector and permuting the 2 element vectors, the same $2^{10}$ patterns can be represented in $9 \times 2^2$ patterns. This reduction saves storage space within the system, increasing the search efficiency of the network.

### 7.7.3 Scalability and Portability

With the ability to sub-divide input vectors, not only can problems be broken down into finer partitions of binary digits, but they can also be scaled up to solve larger problem sets. Work on medical diagnosis may begin by training a single neural network to recognise symptoms around the head region. Examples like inflamed eyes, running nose etc. Once the system is effective in learning the symptoms around the head region, a totally separate neural network can be trained to learn the symptoms in another region. A whole set of neural networks may be independently trained, but combined using the PDNN principle, where additional FLANN layers are used to combine these learnt symptoms to form a complete diagnostic system.

The advantage of such a system is the reduced risk of failure of a single system, because the deeper layers will have the ability to decide from the general consensus of the neural networks in the earlier layers. Assume that a fictitious system has been built to identify faces with the natural partitions of the nose, eyes, mouth, cheeks and ears. Each of the partitions can then be fed into the initial layer of neural networks. If the subject had been involved in a tiff and received a broken nose, the neural network responsible for identifying his nose will probably perform a misclassification. Fortunately, the deeper layer neural network is able to make a further generalisation from the results obtained in the first layer neural networks.
With no loss of generality, this forms a robust system which may extend through as many layers as is required to solve the problem. The sections that follow are model application examples used to support the PDNN concept.

7.8 The Multiple Trigonometric Waveforms Experiment

The concepts introduced in sections 7.6 and 7.7 were implemented on a model problem to show its feasibility. The trigonometric waveforms in § 6.12 are revisited in this section. A set of waveforms are generated using the basic trigonometric functions. Each generated input vector is produced by concatenating 3 waves randomly generated by the sin(A), sin(2A), cos(A) and cos(2A) functions. In all, there are 64 possible permutations for the creation of unique waveforms. An example of such a waveform is shown in Figure 7.16.

![Figure 7.16](image)

A sample of the waveforms generated

The PDNN concept is applied here by feeding the waveform into an initial layer of 3 independent FLANN II networks. The first layer systems will classify or learn the waveform and produce 3 separate output results. These 3 results will then be fed
into the second layer FLANN II as a concatenated input vector. Figure 7.17 shows the physical division of the waveform.

![Figure 7.17](image_url) The division of the input vector for Parallel Distributed Neural Networks (PDNN)

### 7.8.1 Transputer Implementation

The PDNN (FLANN II) was implemented on a set of 4 Transputers, with each FLANN II module resident in each Transputer (See Figure 7.18). The second layer FLANN II was resident within the root Transputer (PE I), performing final classifications after the three first level networks (PE 2, 3, 4) had completed processing. Figure 7.18 shows the scheme of the actual implementation.
Each time the first level neural networks (PE 2, 3, 4) obtained a new classification, it would generate a new node, and provide it with an unique identification number. This identification number is unique at each new class detected. This index number will then be fed into the network of the second layer (PE 1). The second layer network will then classify the waveforms from the results of the first level networks. Again, the classification process can be done by allotting unique index numbers, but since the neural network in the final layer is most important to the user, it can also be made to accept an interactive classification by the trainer.

Since the slave processors (PE 2, 3, 4) do not have access to the I/O devices which are held by the host computer, the intermediate results of the slave classification process were not available. The Transputer server program residing in the host computer provides only the root Transputer (PE 1) with I/O access. This is illustrated in Figure 7.18.

The sample of the data used for the multiple waveform experiment is shown in Table 7.1. The intervals for A are 36 degrees apart or $\pi/5$, beginning at 0 and the final column is $9\pi/5$. 
Chapter 7  Parallel Distributed Neural Networks (PDNNs)

Table 7.1  Multiple Waveform data used for PDNN (FLANN II) testing

<table>
<thead>
<tr>
<th>Combination</th>
<th>Sample</th>
<th>Training</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>cos 2A</td>
<td>1.0000</td>
<td>0.3090</td>
<td>-0.8090</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>sin A</td>
<td>0.0000</td>
<td>0.5878</td>
<td>0.9511</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>sin 2A</td>
<td>0.0000</td>
<td>0.9511</td>
<td>0.5878</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>cos 2A</td>
<td>1.0000</td>
<td>0.3090</td>
<td>-0.8090</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
<tr>
<td>cos A</td>
<td>1.0000</td>
<td>0.8090</td>
<td>0.3090</td>
</tr>
</tbody>
</table>

The networks in the initial levels are left to perform automatic classification. Table 7.2 shows the timings achieved by the Transputer implementation, along with the sequential execution speeds obtained from a single Transputer processor.

<table>
<thead>
<tr>
<th>PDNN Execution</th>
<th>Storage space required</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>608 bytes</td>
<td>7 seconds</td>
</tr>
<tr>
<td>Sequential Execution</td>
<td>7680 bytes</td>
<td>11 seconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PDNN is</th>
<th>Storage space required</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.6 times more efficient</td>
<td>1.57 times faster</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2  Results of the Multiple Waveform PDNN (FLANN II) Experiment using 4 Transputers to process the 4 neural networks

As the Transputer development tool kit did not provide for finer measurements, the timings were all taken in seconds. 10 execution runs of the program were conducted on the same program to determine if there was any time difference. Since the Transputer is an isolated parallel machine, all 10 executions time were consistent and the PDNN (FLANN II) execution time remained at 7 seconds. The increase in execution speed for the Transputer implementation was about 1.57
times faster than the sequential execution. This numerical figure of 1.57 is obtained by dividing the sequential time of 11 seconds by the parallel time of 7 seconds. This speed increase, compared to the ratio of storage savings of 12.6 times is relatively unproportional, because storage space calculations do not directly translate to speed increases. Other factors such as the physical structure of the system, the speed of links and synchronisation also affect the execution times. The Transputer system probably uses most of the execution time for message passing communications that creates an overhead which narrows the execution time gap between the PDNN (FLANN II) program with less data and the sequential program with more data. The sequential program seems to possess a faster speed per storage byte because its execution is void of any message passing whereas the PDNN (FLANN II) implementation needs 6 message passing activities for each pattern (See the slave Transputer of Figure 7.18). Although the speed increase is proportionally small, the overall execution time of the PDNN (FLANN II) is still 4 seconds faster than the actual sequential program.

7.8.2 Balance 8000 Implementation

The multiple waveform problem was also implemented on the Balance 8000 parallel computer. The architecture of the PDNN (FLANN II) was mapped onto 4 separate parallel processors on the shared memory system. The advantage of using the Balance 8000 machine, is the ability for each processor to access I/O devices. As such, the intermediate values of the experiment were available for monitoring.

Although the actual physical allocation of processors is not known, the program for the Balance 8000 used only 4 processors on the multiprocessor machine. Figure 7.19 shows the expected system configuration when the program is executed.
The first layer FLANN II networks would reside within PEs 2, 3 and 4. Once the classifications from the 3 PEs are ready, PE 1 will retrieve the information from the shared memory. Unlike the Transputer system which required links to send information through PE 1 to the slaves in PE 2, 3 and 4 (Figure 7.18), the PEs 2, 3 and 4 in Figure 7.19 obtain their data straight from the shared memory system, bypassing the extra communication overheads. The Balance 8000 results tabulated in Table 7.5 reflect more favorably on the speed increases. The initial section of the data presented to the PDNN (FLANN II) is found in Table 7.3. Due to the limited space on a page, the actual values are not presented, but the function from which the waveforms were derived are shown in each partition.
Chapter 7 Parallel Distributed Neural Networks (PDNNs)

Waveform number | Partition 1 | Partition 2 | Partition 3
--- | --- | --- | ---
1 | cos A | cos A | cos A
2 | cos 2A | cos 2A | cos 2A
3 | sin A | sin A | sin A
4 | sin 2A | sin 2A | sin 2A
5 | cos A | cos A | cos 2A
6 | cos A | cos A | sin A
7 | cos A | cos A | sin 2A
\* | permute | permute | permute
\* | permute | permute | permute
\* | permute | permute | permute
64 | sin 2A | sin 2A | sin A

Table 7.3 Permuted data from 4 basic waveforms of sin A, sin 2A, cos A and cos 2A

The sequence of the input data is shown in Table 7.3. The intermediate results obtained from the Balance 8000 are shown in Table 7.4. Referring to waveform number 1, the neural network in the initial layer assigns the index number 1 to the partitions because cos A is the first waveform encountered in the learning process. Similarly, cos 2A is assigned "2", sin A is assigned "3" and sin 2A is assigned "4". The initial 4 patterns were not permuted but presented as triplets of the same waveform. This is to assist the monitoring of the experiment. Once these initial waveforms are learnt, subsequent identifications are made easy as a classification of a "1" would indicate a cos A function, a "2" being a cos 2A function etc. After all 4 initial patterns have been assigned, no more new patterns are formed in the first layer neural networks. New patterns are only formed in the final layer as permutations of the 4 basic classes are encountered. The technique of permuting sub-patterns is useful for identification problems such as human facial recognition.
system which can help in criminal identification. All the segmented partitions can be organs such as the mouth, the nose, the ears and the eyes etc. Instead of storing the large individual image of a person, the coding of the compound image is made possible by using a database that contains the common segmented facial features.

<table>
<thead>
<tr>
<th>Waveform position</th>
<th>Partition 1 classification</th>
<th>Partition 2 classification</th>
<th>Partition 3 classification</th>
<th>Overall classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>permute</td>
<td>permute</td>
<td>permute</td>
<td>7</td>
</tr>
<tr>
<td>64</td>
<td>permute</td>
<td>permute</td>
<td>permute</td>
<td>•</td>
</tr>
</tbody>
</table>

Table 7.4 Results of the intermediate and final classification.

Shown in Table 7.4 is the actual results obtained from the first epoch of the PDNN (FLANNII) learning the waveforms. Subsequently, distorted waveforms which are not shown in the table were classified with a 100% accuracy. The timings tabulated in Table 7.5 show that the PDNN (FLANN II) implementation on the Balance 8000 machine is approximately 4 times faster than the sequential implementation. Several executions were performed on the Balance 8000 machine and the timings varied by only a negligible amount, i.e. 0.02 seconds. The ratio of the sequential time over the parallel time provides the speed increase of the parallel implementation over the sequential implementation.
Chapter 7 Parallel Distributed Neural Networks (PDNNs)

<table>
<thead>
<tr>
<th></th>
<th>Storage space required</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDNN Execution</td>
<td>608 bytes</td>
<td>13.16 seconds</td>
</tr>
<tr>
<td>Sequential Execution</td>
<td>7680 bytes</td>
<td>51.74 seconds</td>
</tr>
<tr>
<td>PDNN time is</td>
<td>12.6 times more efficient</td>
<td>3.93 times faster</td>
</tr>
</tbody>
</table>

Table 7.5 Results of the Multiple Waveform PDNN (FLANN II) Experiment using the Balance 8000.

7.8.3 PVM Implementation

The multiple waveform experiment was repeated under a PVM programming environment. Two Digital Alpha machines were connected over a TCP/IP network to form the parallel machine. Although the program executed correctly, the parallel timings did not reflect any speed increases over the sequential timings. In fact, the parallel implementation seems to be slower than the sequential execution code. The reason for this slow speed is probably attributed to the extensive software layers required in the TCP/IP protocols and the traffic existing on the Ethernet. This traffic causes a substantial delay which is not present in true parallel computers. Table 7.6 shows the experimental results obtained from the PDNN (FLANN II) implementation on the PVM platform. The intermediate results were similar to those shown in Table 7.4.

Several executions were performed on the PDNN (FLANN II) implementation to determine the average timing of the system. During the times when heavy traffic is experienced over the Ethernet line, the PDNN (FLANN II) system may take up to 1.31 seconds to execute the program. When executed during the off peak times, the PDNN (FLANN II) system took about 1.01 seconds. The cause for this erratic behaviour in execution times is due to the traffic loading on the Ethernet which the PVM system uses for communications between the two DEC Alphas machines.
The discussion to use a FDDI (fibre optic) megaswitch in § 7.4.3 provides a solution that should speed up these inter-processor communication times.

<table>
<thead>
<tr>
<th>PDNN Execution</th>
<th>Storage space required</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>608 bytes</td>
<td>1.24 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.04 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.21 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.03 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.19 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.09 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.23 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.31 seconds</td>
</tr>
<tr>
<td>Sequential Execution</td>
<td>7680 bytes</td>
<td>0.46 seconds</td>
</tr>
</tbody>
</table>

Table 7.6 Results of the Multiple Waveform PDNN (FLANN II) Experiment using PVM.

When the speeds of PDNN (FLANN II) execution are compared, the DEC Alpha parallel executions may not have achieved any increases in speeds over the sequential counterparts, but amongst the platforms utilising the PDNN paradigm, the overall execution times on these PVM workstations were the fastest. The Transputer systems required 7 seconds (See Table 7.2), the Balance 8000 used 13.16 seconds (See Table 7.5) and the PVM implementation needed approximately 1.2 seconds for the execution using the same data set.
7.9 PDNN (FLANN I) Implementation of the Parity Problem

In Chapter 5, the FLANN I model was used to solve the parity problem. The results indicate that FLANN I was able to learn the new patterns in a single epoch and provide a 100% accuracy in its learning abilities.

This section extends the FLANN I model using PDNN structures. Instead of using a single FLANN I model to obtain the solution, several FLANN I networks are used concurrently to solve the problem. Like the experiments introduced in § 7.8, this experiment will utilise four separate FLANN I networks, configured in a cascading sequence as shown in Figure 7.20. The input layer will consist of a 9 bit binary pattern, but because it is processed by 3 separate FLANN I networks, each individual network will only process 3 input bits.

![Figure 7.20 PDNN (FLANN I) Implementation](image)

7.9.1 Storage Calculations

As seen from Figure 7.20, the original 9 element input pattern is partitioned into 3 segments of 3 elements each. The left most FLANN I network will process 3 input elements, allowing a possible set of 8 patterns. This is the same for the other 3
neural networks in the first layer. Since each have 8 possibilities, the first layer will require enough storage space for 24 possibilities. The last layer possesses another 8 possibilities so the 4 networks will require only 32 separate memory spaces to classify all possibilities contained in the problem.

Unlike the single FLANN I solution which processes 9 elements in a single input resulting in a larger storage requirement, the PDNN (FLANN I) implementation will reduce that storage requirement as described in §7.7.2. This reduction in storage requirements effectively reduces the search space of the algorithm, reducing the overall computation time.

7.9.2 Parity Problem PDNN (FLANN I) Implementation on the Transputer System

The first of the proposed PDNN (FLANN I) implementation from section 7.8.1 was done on the Transputer system. The results showed that the classification was 100% accurate with no exceptions, indicating that the concept of PDNNs was correct. The speed of the PDNN (FLANN I) system was also faster than the sequential implementation. A total of 4 Transputers were used and the implemented system was mapped into the physical architecture similar to that explained in §7.8.1.

Maintaining the exact configuration setup, the three slaves executed the classification for the 3 segments of the input vector. When the classification of the inputs was complete, the resultant classification is routed back to the root Transputer which contains the final FLANN I classifying system. When complete, the root Transputer will possess the results which will be sent back into the host computer, providing the user with the PDNN (FLANN I) processed results. Table 7.7 shows the timing results of the parity problem when solved using PDNNs.
Again, like the PDNN (FLANN II) implementation in §7.8.1, the timings were consistent throughout all execution tests.

<table>
<thead>
<tr>
<th></th>
<th>Storage space required</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDNN Execution</td>
<td>32 X 2 bytes</td>
<td>1 second</td>
</tr>
<tr>
<td>Sequential Execution</td>
<td>512 X 2 bytes</td>
<td>6 seconds</td>
</tr>
<tr>
<td>PDNN (FLANN I) is requires 16 times less space</td>
<td>6 times faster</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.7 Results of the Parity Problem PDNN (FLANN I) Experiment using Transputers.

For the parity implementation, the system seems to be 6 times faster than the multiple waveform implementation. This ratio is obtained from the division of the sequential execution time with the parallel execution time.

The tree structure used in PDNNs is a powerful paradigm that can break down many large problems into smaller and manageable portions. Unfortunately, these structures may not always be feasible for array processors such as Transputers. Consider the structure in Figure 7.21. Although the network structure is a scale-up model of the basic structure used in the PDNN (FLANN I) experiments, the fixed link connections on some Transputer boards will not permit such a construction. This problem is caused by the hardware limits on the array processor.
7.9.3 Parity Problem PDNN (FLANN I) Implementation on the Balance 8000

The Parity problem executed on the Balance 8000 Multiprocessor platform provided the best speed increase, when compared with all the PDNN (FLANN I) implementations. The speed increase was about 20 times faster than the sequential implementation. This value was obtained by dividing the sequential execution time of 77.86 seconds with the PDNN execution time of 3.87 seconds. Table 7.8 shows the execution times.

<table>
<thead>
<tr>
<th></th>
<th>Storage space required</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDNN Execution</td>
<td>512 X 2 bytes</td>
<td>3.87 seconds</td>
</tr>
<tr>
<td>Sequential Execution</td>
<td>32 X 2 bytes</td>
<td>77.86 seconds</td>
</tr>
<tr>
<td>PDNN (FLANN I)</td>
<td>Require 16 times less memory</td>
<td>is about 20 time faster</td>
</tr>
</tbody>
</table>

Table 7.8 Results of the Parity Problem PDNN (FLANN I) Experiment using the Balance 8000.
The advantage of parallel implementations on the multiprocessor systems is the ability to map any topology on to the hardware architecture. This means that the topology that poses a problem for the Transputer systems (Figure 7.21) can be easily mapped onto the Balance 8000 multiprocessor machine.

A sample of the parity training data is tabulated in Table 7.9. Note that the first layer FLANN I classifiers are only required to classify 8 patterns and the entire 512 patterns are mere permutations of these 8 patterns. Table 7.10 shows the intermediate results obtained from the Balance 8000 machine.

<table>
<thead>
<tr>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>2 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>3 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 1 1</td>
</tr>
<tr>
<td>4 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>5 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>6 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>7 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>8 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>9 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

Table 7.9 Sample data for testing PDNN (FLANN I)

<table>
<thead>
<tr>
<th>Partition 1 Classification</th>
<th>Partition 2 Classification</th>
<th>Partition 3 Classification</th>
<th>Final Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.10 Sample of the results obtained from PDNN (FLANN I) Classification
7.9.4 Parity Problem PDNN (FLANN I) Implementation on PVM

Similar experiments were conducted using the PVM platform. The Virtual machine was made up of two DEC Alpha workstations, connected through an Ethernet system. The PVM platform, illustrated in Figure 7.13, was not expected to produce any significant speed increases because of the slow inter-processor communications interface. In the PVM environment, the results again showed the lack of speed reductions when using PDNNs. However, the overall execution speeds were still faster than the experiments conducted on the Transputer and Balance 8000 platforms.

The PVM system was able to map the PDNN (FLANN I) architecture into the two DEC Alphas to generate the correct neural network. The results showed that the accuracy of the system was maintained at 100%, but the parallel timings were slower than the sequential timings. It was deduced that PVM is not suitable for this application as it relies on a constant, uninterrupted flow of data traffic over the Ethernet. Table 7.11 shows the timings generated by PVM for the PDNN (FLANN I) implementation of the parity problem. The times were taken over several separate executions. The variations in the execution times are attributed to the traffic on the Ethernet.
7.10 Conclusions for the PDNN Paradigm

From the experimental results, the Parallel Distributed Neural Networks concept seems to possess similar properties as the single FLANN network. It is able to learn within the single epoch and perform generalised classifications. Although the implementation speed increase ratios were not consistent, it can be seen that the PDNN implementation can indeed provide a considerable amount of speed increases.

In addition to the speed increases without loss of accuracy and generalisation, PDNN implementations can reduce the space required for large compound problems. The experimental results show that implementations of the multiple waveform problem and the parity problem have actually achieved an overall reduction in storage requirements. On the Transputer implementation of the waveform problem, there was a speed increase of 1.5 times that of the sequential implementation and an increase of 6 times for the parity implementation. The Balance implementations provided a better representation of the speed increases because there was less communications necessary to pass messages. The timings
for the waveform problem experienced a speed increase of 4 times and the parity implementation was 20 times faster.

The timings obtained from the PVM implementation were rather misleading because the prevailing Ethernet traffic formed an external factor which slowed the system down. To show that the Ethernet was indeed a disturbance for the program, several execution timings were taken and were shown to be inconsistent. Since the data and programs were fixed, these inconsistencies could only have been caused by the traffic over the Ethernet which linked the two DEC Alpha stations together. A proposed solution that may solve this problem is to connect the DEC Alpha stations together using an isolated FDDI megaswitch. This would ensure that interworkstation communications is not hampered by external network traffic.

PDNN concepts are not specific to any parallel computer architecture. As seen from the implementations, 3 separate platforms were used and PDNNs was easily mapped onto the existing hardware.
Chapter 8

Conclusions
CHAPTER 8 CONCLUSIONS

To provide a clear and systematic abstraction of the work completed in this thesis, this concluding chapter is divided into three distinct sections. The first of which deals with the contributions made to knowledge and the second reiterates the conclusions found in the 3 major areas of contribution, i.e. FLANN I, FLANN II and PDNNs. The final section provides suggestions for avenues from which further investigations can be undertaken.

8.1 Identification of Major Contributions to Knowledge

The first four chapters in this thesis studies the existing neural network architectures, namely the back propagation model, the Kohonen network, the ARTI network and the Hopfield network. The tree diagram in Figure 8.1 shows the overall structure of the thesis and labelled beside the chapters are the experiments conducted. The Hopfield network was omitted because it was not suitable for neural network classification applications.

The documentation of the major contributions made in this thesis begin from Chapter 5, where the first FLANN model was introduced. Here, the XOR problem, the parity problem and a character recognition system were used to illustrate the expedient learning capabilities of the model. The weakness of the FLANN I model to cope with continuous input values caused the need to design yet another new neural network model, the FLANN II.
Chapter 6 introduced the FLANN II network as a second major contribution found in the thesis. Four classification experiments were conducted on the model to determine its ability to classify patterns as a neural network. These are as follows:
Chapter 8  Conclusions

- The Twin Spiral Experiment
- The CSTR control diagnostic experiment,
- The Mixing process diagnostic experiment
- The Waveform Classification experiment

The models developed in Chapters 5 and 6 differ in many aspects, but possess the unique ability to learn patterns in a single epoch.

Chapter 7 builds on the models designed from Chapters 5 and 6. It uses the FLANN I and FLANN II models to produce the new Parallel Distributed Neural Network (PDNN) paradigm. This unique concept of cascading networks of neural networks provide a new platform for parallel neural computing. The concept of PDNNs was implemented on three different parallel processing platforms, namely the T800 Transputer system, the Balance 8000 multiprocessor parallel computer and a cluster of DEC Alpha workstations operating under the PVM environment.

8.2 Conclusions from the Three Major Areas of Contribution

Section 8.1 provided the overall perspective of the contributions made in this thesis. The sections that follow provide the individual conclusions.

8.2.1 Conclusions for the FLANN I model

The FLANN I was modelled after the ART1 neural network, which was originally designed by Grossberg. Although the FLANN I model had its roots in the ART1 model, it exhibited many qualities which differed from the ART1 model. Three experiments were documented to examine the novel FLANN I model.
The results of the XOR and parity experiments showed that the FLANN I model was able to consistently learn all the input patterns in a single epoch. Comparing this to the MLP model, which struggled with several hundred epochs for the XOR problem, the FLANN I model was very efficient in learning the patterns for correct classification. Furthermore, FLANN I maintained a stable pattern storage characteristic which allowed patterns to be learnt without the disturbance of previously learnt information. The accuracy of the network to identify closely similar patterns was also evident from the results. In the 8 bit parity training processes, only a single epoch of 256 patterns was needed to fully train the neural network. As compared to over 200 epochs for training an MLP model with 4 bits.

In the character recognition experiment, FLANN I exhibited its ability to generalise stored pattern inputs. In a single epoch, it was presented with the bitmapped pattern of 0 to 9, and it could recall all the patterns. In addition to the perfect recall, it was able to generalise the 10 patterns learnt. As a result of this generalisation, it was able to recognise deformed images of the original patterns. These results are reflected in section 5.10. In comparing with the ART1 model, the FLANN I maintained a consistently fast learning speed of a single epoch. Although the ART1 system could occasionally learn just as fast, it encountered the unwarranted recoding process which overwrites previously stored patterns with newer closely related patterns. This poses difficulty in the learning process because the user may not realise the overwriting of previously stored patterns.

In the area of classifying binary patterns, FLANN I seems to be an effective substitute for other neural network models. It is a simple model with several desirable properties, which are listed as follows:-
• Able to learn fast (1 Epoch)
• Operates on binary inputs
• Able to generalise
• Able to function in noisy environments
• Has a facile implementation
• Able to re-code patterns

8.2.2 Conclusions for the FLANN II model

The FLANN I model was an efficient learning model which had many desirable properties, unfortunately, it was only able to perform neural network operations on binary numbers. This places the model at a slight disadvantage because most real world applications require real numbers. Due to this deficiency, another model had to be designed. The next model designed was the Fast Learning Artificial Neural Network II.

The FLANN II model was no longer designed using the Hamming distance measure, which could maintain an ordered system of selection in the binary world. Instead it was designed around similar metrics found in the Kohonen network. This was the basic Euclidean distance measure. As a result of these different metric operations, the FLANN II algorithm vaguely resembles the FLANN I system.

The capabilities of the FLANN II model was tested with 4 separate experiments. These experiments were as follows:-

• Identifying two spirals
• Fault diagnostics of a mixing process
• Fault diagnostics of a continuous stirred tank reactor
• Identification of noisy sinusoidal waveforms
In the two spirals experiment, FLANN II was tested for its learning speed and generalisation properties. In a single epoch, FLANN II was trained with 194 points, of which 97 points belonged to one spiral and 97 points belonged to the other. The network was able to classify 10,000 test points with an estimated accuracy of 99.5%. In a similar experiment conducted with the MLP models (Lang K.J., et. al. 1988), Lang used 64,000 epochs and achieved an accuracy of 90% with 770 test patterns.

In the two fault diagnostic problems, the FLANN II model was again tested for speed of learning and generalisation. The data was obtained from a paper by (Zhang J., et. al., 1992) where the authors used an MLP model to train their industrial system to perform diagnostics. They were successful, with their network learning in 500 to 5000 epochs. FLANN II was able to learn in a single epoch and when tested with the test data, it maintained 100% correct classifications for both problems. Zhang's paper was extremely helpful in providing the data required to test the FLANN II model, as most authors do not provide any form of test data due to the sensitivity of industrial information.

The final experiment was to test the FLANN II network under noisy conditions. The FLANN II was trained to recognise 4 different waveforms, \( \sin(A) \), \( \sin(2A) \), \( \cos(A) \) and \( \cos(2A) \). Once the network was trained, waveforms with random distortions were presented to the neural network. The FLANN II produced a 100% accuracy in the classification process.

The FLANN II bore similar fast learning abilities as FLANN I. In addition, it operates on both binary and continuous input vectors. This makes it a very valuable network for current day industry usage since it now has the ability to learn in a single epoch. The characteristics of FLANN II are listed as follows:-

---

224
• Ability to learn patterns in a single epoch
• Operates with both binary and continuous input values
• Simple implementation
• Ability to re-code patterns
• Ability to generalise
• Ability to operate under noisy environments

8.2.3 Conclusions for the PDNN paradigm

Chapter 7 was the result of taking a step beyond the objectives and exploring the potential of the FLANN models in a parallel processing environment. Most parallel implementations of neural networks require the input vectors to be duplicated in the various parallel tasks (see § 7.3). This results in two possibilities, either the use of a shared memory system, or the full transfer of the input vector to all sub tasks in the parallel program. The inability to separate the input vectors into sub-vectors is a drawback that hinders the full potential of parallel processing systems.

Thus a new way of using neural networks in parallel is introduced. Here, the original input vector was divided into portions and processed. Three different parallel computing platforms were used. These were the Transputer platform, the Balance 8000 platform, and the PVM platform. The accurate and consistent results obtained from the three platforms confirmed the feasibility of the Parallel Distributed Neural Network (PDNN) paradigm. The results of the multiple sinusoidal waveform experiment showed that such a network of neural networks could function together as a unit to classify patterns across the several layers. Such a paradigm is both efficient and powerful because it introduces the concept of scalability and portability to neural networks. From the theoretical discussions, it was shown how PDNNs can even reduce the total storage space required, compared to processing vectors as a single entity.
In addition to the accuracy of the PDNN implementations, the speed increases were also significant. Both the Transputer and Balance 8000 implementations provided results which were 1.5 times to 20 times faster than an equivalent sequential implementation. Although the Transputer system required excessive overheads in the message passing tasks, it was able to achieve 1.5 times faster in the multiple waveform problem and 6 times faster in the parity problem for both PDNN implementations. The Balance 8000 machine achieved 4 times the speed in the PDNN implementations over the sequential implementation of the waveform problem, and 20 times faster for the PDNN implementation of the parity problem, over the sequential counterpart. Although the PDNN implementations using PVM were slower, it was suggested that the PVM system used was dependent on the traffic over the Ethernet which linked the DEC Alpha machines together. The network traffic or the software layers in PVM may cause excessive overheads which slow the parallelisation process considerably.

8.2.4 Meeting the Objectives

The objective of this thesis as stated in section 1.4 was,

to develop a neural network that is fast learning, able to generalise and achieve good capacity to discern different patterns even though some patterns may be similar in structure. This eventual neural network will be used in the pattern classification environment.

This objective was fulfilled in the development of FLANN I and FLANN II. Their generalisation capabilities were shown in the numerous experiments conducted. The parity and twin spiral experiments tested the neural models for the ability to
discern closely similar patterns. The ultimate achievement in these experiments, was the constant network learning speed of a single epoch.

Over and above the stated objectives, additional work was done to develop a new paradigm for parallel neural computing, PDNNs. This innovation in the parallel independent sub-vector processing concept paves the way for a larger set of neural network problems. Problems that were once impossible due to their large input vectors, can now be solved with PDNNs. Neural networks which were trained separately may now be combined to form larger classification systems.

8.3 Future Work

The developments in this thesis can perhaps provide the gateway to a new revolution in the changing world of neural computing. FLANN I and FLANN II are models which operate at very high levels of efficiency. Accompanying these models, is the new PDNN paradigm which is valuable for parallel implementations. This opens the opportunity for applied research, where the industry can now rely on the fast learning characteristics of these models for classification.

Apart from the application aspects of FLANN models, it may be necessary to investigate this new approach to neural computing. For a long time, neural network researchers have been dwelling too deeply into Hebbian learning. Yes, Hebbian learning does exists in the micro structure of each neuron, but biological facts show that brain operations span across large clusters of neurons. The results from PET Scans show that clusters of neurons are involved in cognition, which apparently behaves differently from a single neuron. The conventional neural network learning paradigms may be too myopic by focusing solely on mimicking the neuron, and fail to study the brain from a higher perspective. Perhaps the secrets of the brain can be unlocked from visualisation at a much higher level. The FLANN models have
introduced learning that can perhaps be perceived as a higher dimension of cognition. Each "grandmother cell" in the output layer may be viewed as a cluster of neurons. In which case, FLANN models are not concerned with the modelling of the micro structure of Hebbian learning, but focuses on the macro form of learning. Formalisation and investigations at these levels need to be done, to broaden the scope of neural computing.

This thesis has advanced only the area of classification and provided an alternative solution for the slow learning speeds in neural networks. As for the formidable area of functional approximation, much has yet to be done to increase the network learning speed.
Bibliography


Bibliography


Appendix A
Appendix A  XOR Program

/***********************************************
| Program Title: The BP Neural Network XOR Simulator |
| Programmer   : Alex Tay Leng Phuan            |
| Institute    : Loughborough University of Technology |
| Dated        : 5th Nov. 1992                  |
\***********************************************/

#include <stdio.h>
#include <math.h>

/***********************************************
| Hidden Node (1)                          |
| Input(1) --> 0                           |
|   \ Output Node                           |
|   0 -----------------> Output            |
| /                                          |
| Input(n) --> 0                            |
| Hidden Node (n)                           |
\***********************************************/

typedef struct node
{
  float Delta_Value;    /* Hidden layer Delta Value */
  float Net_Value;      /* Hidden Layer Net Value */
  float Output_Value;   /* Hidden Layer Output Value */
  float Change_In_Bias, Bias;  /* Hidden Layer Bias' */
  float Weight_In1, Weight_In2;  /* Hidden Layer Input Weights */
  float W_In_Change1, W_In_Change2;  /* Delta Weight Changes */
  float Weight_Out;      /* Output Layer Weight */
  float W_Out_Change;    /* Output Layer Weight */
  struct node *next;    /* pointer to next node */
};

typedef struct out_node
{
  float Delta_Value;    /* Output Layer Delta Value */
  float Change_In_Bias, Bias;  /* Output Layer Bias' */
  float Net_Value;      /* Output Layer Net Value */
  float Output_Value;   /* Output Layer Output Value */
};

#define NODE struct node     /* Data type NODE */
#define OUT_NODE struct out_node  /* Data type OUT_NODE */

float In[2];          /* External Input Values */
float Delta_Sum, Nu, Alpha;  /* Nu = Learning Rate */
                          /* Alpha = Momentum */
main(argc, argv)
int argc;
char *argv[];
{
FILE *infile, *outfile; /* infile = input variables */
/* outfile = output values */

int Iterations = 0, i, Number_of_HNs;
int temp1 = 0, temp2 = 0, temp3 = 0; /* temp variable used to decode */
/* input presentation. After */
float target, In_Val1, In_Val2; /* decoding of temp variables, */
/* Network input values are */
/* placed in In_Val1 & In_Val2 */

void Get_Delta_Sum(); /* Summation of output Delta X hidden weights */
void See_Weight(); /* See all weights and bias' in the Network */
void See_Hidden_Net(); /* See all net values in the hidden layer */
void See_Hidden_Output(); /* See all output values in the HL */
void See_Output_Net(); /* See the net value of the output layer */
void See_Output_Output(); /* See the final o/p value of the o/p layer */
void See_Delta_HIDDEN(); /* See each Delta value of the HL */

void Sum_HIDDEN_Net(); /* Summation of the Input layer to give NET */
void Sum_Output_Net(); /* Summation of the Output layer to give Net */
void Cal_HIDDEN_Output(); /* Calculates the HL output value f(NET) */
void Cal_Output_Output(); /* Calculates the o/p output value f(NET) */
void Delta_Output(); /* Outer most delta value */
void Delta_HIDDEN(); /* HL Delta Value */
void Change_Weight_Output(); /* Changes weights in o/p layer */
void Change_Weight_HIDDEN(); /* Changes weights and bias' in HL */
void Change_Output_Bias(); /* Changes bias in o/p layer */

NODE *make_hidden_node(); /* creates a new hidden node */
NODE *head; /* head of linked list for hidden layer */
NODE *temp_node, *current_node;
OUT_NODE *output; /* output node */
OUT_NODE *make_output_node(); /* creates output node */
int seed; /* seed value for random generator */

if (argc != 3) {
printf("Usage: XOR <number of hidden nodes> <seed>\n");
exit(0); }

/***********************************************************/
datafile: contains 2 values on two separate lines. The
first value is for Nu and the second line's value is
for Alpha. Both are floating point numbers, between 0
and 1.
/***********************************************************/
if ((infile = fopen("datafile","r")) == NULL){
    printf("Sorry I can't find my datafile\n");
    printf("I'm not working without a datafile\n");
    printf("Good bye.\n");
    exit(0);
}

i test : contains the output values of the four input
l combinations, ie 00, 01, 10, 11. the input values for
l In_Val1 and In_Val2 are also put at the end of the line

if ((outfile = fopen("test","w")) == NULL){
    printf("Sorry I can't find my datafile\n");
    printf("I'm not working without a test file\n");
    printf("Good bye.\n");
    exit(0);
}

NumbecoCHNs = atoi(argv[1]); /* extracting size of HL */
seed = atoi(argv[2]); /* extracting seed value */
srand(seed); /* Initialize random generator with seed */
fscanf(infile,"%f\n",&Nu); /* Get Value of Nu from infile */
fscanf(infile,"%f\n",&Alpha); /* Get Value of Alpha from infile */

printf("Number of Hidden Nodes : %d\n",Number_of_HNs);
printf("Seed is : %d\n",seed);
printf("Nu is : %f\n",Nu);
printf("Alpha is : %f\n",Alpha);

head = make_hidden_nodeO; /* Initialize list by creating head */
currencnode = head; /* Prepare head for construction of linked */
* list

for (i = 1; i < Number_of_HNs; ++i) /* i = 1 includes head */
{ 
    current_node->next = make_hidden_nodeO();
    current_node = current_node->next;
}

printf("%d nodes have been successfully created in the Hidden Layer\n",Number_of_HNs);

output = make_out_nodeO; /* creating output node */
printf("The XOR output node has also been generated successfully\n");

See_Weight(head); /* check layer initialization */
while (temp3 < 500) /* prevent accidental keyboard mistakes */
{
    Iterations += 1; /* Iteration count */
    scanf("%d",&temp3);
    if (temp3 == 0) {temp1 = 0;temp2 = 0;} //**************************/
    if (temp3 == 1) {temp1 = 0;temp2 = 1;} /* Decoding temp3 to */
    if (temp3 == 2) {temp1 = 1;temp2 = 0;} /* values of temp1 & 2 */
    if (temp3 == 3) {temp1 = 1;temp2 = 1;} //**************************/
    In_Val1 = In[0] = (float) temp1; /* xfer of decoded temp1 to In_Val1 */
    In_Val2 = In[1] = (float) temp2; /* xfer of decoded temp2 to In_Val2 */
    if (temp1 == temp2) {target = 0.0;} /* decode target result */
    else {target = 1.0;}
    printf("Target is %f\n",target);
}

Sum_Hidden_Net(head);
Cal_Hidden_Output(head);
output->Net_Value = 0.0;
Sum_Output_Net(head,output);
Cal_Output_Output(output);
Delta_Output(target,output);
Delta_Sum = 0.0;
Get_Delta_Sum(output,head);
Delta_Hidden(head);
Change_Weight_Output(head,output);
Change_Weight_Hidden(head,Nu);
Change_Output_Bias(output);

/* *********************************************/
/* Systematic calculation of output values ONLY. No */
/* modification of weights, Deltas or biases are done. This */
/* is the process of writing each input permutation's o/p */
/* into the file "test". */
/* *********************************************/

In[0] = 0.0; In[1] = 0.0; /* 0 0 */
Sum_Hidden_Net(head);
Cal_Hidden_Output(head);
output->Net_Value = 0.0;
Sum_Output_Net(head,output);
Cal_Output_Output(output);
fprintf(outfile,"%f\n",output->Output_Value);
printf("%f\n",output->Output_Value);
In[0] = 0.0; In[1] = 1.0; /* 0 1 */
Sum_Hidden_Net(head);
Cal_Hidden_Output(head);
output->Net_Value = 0.0;
Sum_Output_Net(head,output);
Cal_Output_Output(output);
fprintf(outfile,"%f \n",output->Output_Value);
printf("%f \n");

In[0] = 1.0; In[1] = 0.0; /* 1 0 */
Sum_Hidden_Net(head);
Cal_Hidden_Output(head);
output->Net_Value = 0.0;
Sum_Output_Net(head,output);
Cal_Output_Output(output);
fprintf(outfile,"%f \n",output->Output_Value);
printf("%f \n");

In[0] = 1.0; In[1] = 1.0; /* 1 1 */
Sum_Hidden_Net(head);
Cal_Hidden_Output(head);
output->Net_Value = 0.0;
Sum_Output_Net(head,output);
Cal_Output_Output(output);
fprintf(outfile,"%f %d %d\n",output->Output_Value,temp1,temp2);
printf("%f \n",output->Output_Value);
}
/* *********************************************/
// Final print statements before "test" is closed. These
// write the Iterations and the final o/p bias into the file
// *********************************************
printf("%d Iterations were registered\n",Iterations);
fprintf(outfile,"%d Iterations were registered\n",Iterations);
See_Weight(head);
printf("The final Output Bias is %f\n",output->Bias);
fprintf(outfile,"The final output Bias is %f\n",output->Bias);
}
/* *********************************************/
// make_hidden_node() is type NODE *, returning the pointer
// to the address of the newly created hidden node. All
// unknown values are initialized to 0.0 or NULL. Weights
// and biases are randomly assigned values between -3.0 and
// 13.0.

A-5
NODE *make_hidden_node()
{
    NODE *new_node;
    float w1, w2, w3, thres;
    new_node = (NODE*) malloc(sizeof(NODE));
    new_node->next = NULL;
    w1 = rand();
    w2 = rand();
    w3 = rand();
    thres = rand();
    new_node->Weight_In1 = ((float) (37.0 * (float) w1/37.0 - (int) w1/37) - 13.8)/8.0); 
    new_node->Weight_In2 = ((float) (37.0 * (float) w2/37.0 - (int) w2/37) - 18.5)/5.0); 
    new_node->Weight_Out = ((float) (37.0 * (float) w3/37.0 - (int) w3/37) - 17.3)/7.0); 
    new_node->Bias = ((float) (37.0 * (float) thres/37.0 - (int) thres/37) - 15.3)/7.0); 
    new_node->Change_In_Bias = 0.0;
    new_node->W_In_Change1 = 0.0;
    new_node->W_In_Change2 = 0.0;
    new_node->W_Out_Change = 0.0;
    return (new_node);
}

OUT_NODE *make_out_node()
{
    float thres;
    OUT_NODE *new_node;
    new_node = (OUT_NODE*) malloc(sizeof(OUT_NODE));
    thres = rand();
    new_node->Bias = ((float) (37.0 * (float) thres/37.0 - (int) thres/37) - 13.3)/37.0); 
    new_node->Change_In_Bias = 0.0;
    return (new_node);
}

**************
See Weights()
A recursive printf of weights within the hidden layer.
Appendix A  XOR Program

Note that the values are in LIFO sequence from the head

```c
void See_Weight(cnode)
NODE *cnode;
{
  if (cnode->next != NULL)
  { See_Weight(cnode->next);
  }
  printf("In1 is %3.3f In2 is %3.3f Out is %3.3f Bias is %3.3f\n", 
    cnode->Weight_In1,cnode->Weight_In2,cnode->Weight_Out,cnode->Bias);
}
```

```c
void See_Hidden_Net(cnode)
NODE *cnode;
{
  if (cnode->next != NULL)
  { See_Hidden_Net(cnode->next);
  }
  printf("Net Value = %f\n",cnode->Net_Value);
}
```

```c
void See_Hidden_Output(cnode)
NODE *cnode;
{
  if (cnode->next != NULL)
  { See_Hidden_Output(cnode->next);
  }
  printf("Output Value = %f\n",cnode->Output_Value);
}
```

```c
void See_Output_Net()
```

Appendix A  XOR Program

/* Views the Output Net Value. Placed as a function for clarity in main() */
void See_Output_Net(output)
OUT_NODE *output;
{
    printf("Output Node Net Value is %f\n", output->Net_Value);
}

/* Views the final output after passing through the network */
void See_Output_Output(output)
OUT_NODE *output;
{
    printf("Output Node Output Value is %f\n", output->Output_Value);
}

/* Recursively sees all the Delta values in the HL */
void See_Delta_Hidden(cnode)
NODE *cnode;
{
    if (cnode->next != NULL)
        See_Delta_Hidden(cnode->next);
    printf("Delta Hidden is %f\n", cnode->Delta_Value);
}

/* Recursively sums the Net values for the hidden layer */
void Sum_Hidden_Net(cnode)
NODE *cnode;
{
    float Sum = 0.0;
    if (cnode->next != NULL)
        Sum_Hidden_Net(cnode->next);
    Sum = cnode->Weight_In1 * In[0] + cnode->Weight_In2 * In[1];
    cnode->Net_Value = Sum;
Appendix A  XOR Program

*****

Cal_Output_Output()

Sigmoid function to calculate final output value of the
network.

*************************************************/

/**

void Cal_Output_Output(output)

OUT_NODE *output;
{

double EPow;
float thres;
EPow = exp((double)(-(output->Net_Value + output->Bias)));
output->Output_Value = (float)(1.0 / (1.0 + EPow));
}

*************************************************/

Cal_Hidden_Output()

Recursive calculation of output values of the hidden
units. Uses the sigmoid function for calculations.

*************************************************/

/**

void Cal_Hidden_Output(cnode)

NODE *cnode;
{

double EPow;
float thres;
if (cnode->next != NULL){
    Cal_Hidden_Output(cnode->next);
}
EPow = exp((double)(-(cnode->Net_Value + cnode->Bias)));
cnode->Output_Value = (float)(1.0 / (1.0 + EPow));
}

*************************************************/

Sum_OutpucNet()

Recursively sums the hidden layer outputs X weights and
places these values into the output unit's Net value

*************************************************/

/**

void Sum_OutpucNet(cnode,out_node)

NODE *cnode;
OUT_NODE *out_node;
{

float temp;
if (cnode->next != NULL){
    Sum_OutpucNet(cnode->next,out_node);
}

}
Appendix A  XOR Program

out_node->Net_Value += cnode->Weight_Out * cnode->Output_Value;
}

/**********************
| Delta_Output() |
| Calculates the delta value of the output unit. |
\***************************/

void Delta_Output(target,o_node)
float target;
OUT_NODE *o_node;
{
o_node->Delta_Value = (target - o_node->Output_Value) * o_node->Output_Value
   * (1.0 - o_node->Output_Value);
}

/**********************
| Delta_Hidden() |
| Calculates the delta values of each hidden unit based on |
| the calculated value of the delta value of the o/p unit |
\***************************/

void Delta_Hidden(cnode)
NODE *cnode;
{
if (cnode->next != NULL){
   Delta_Hidden(cnode->next);
}
cnode->Delta_Value = cnode->Output_Value * (1.0 - cnode->Output_Value)
   * Delta_Sum;
}

/**********************
| Get_Delta_Sum() |
| Calculates the o/p layer delta X 2nd layer weights for |
| calculation of the hidden layer's delta value. |
\***************************/

void Get_Delta_Sum(o_node,cnode)
OUT_NODE *o_node;
NODE *cnode;
{
if (cnode->next != NULL){
   Get_Delta_Sum(o_node,cnode->next);
}
Delta_Sum += o_node->Delta_Value * cnode->Weight_Out;
}
Appendix A
XOR Program

/***/
| Change_Weight_Hidden() |
| Changes the weight values within the hidden layer. Note |
| the updating of the "previous change in weights", |
| W_In_Change1 and W_In_Change2. A change in Bias is also |
| required. Biases are treated like weights |
\/***/

void Change_Weight_Hidden(cnode,Nu)
NODE *cnode;
float Nu;
{
float Diff_Bias, Diff_Weight1, Diff_Weight2;
if (cnode->next != NULL){
    Change_Weight_Hidden(cnode->next,Nu);
}
Diff_Weight1 = (Nu * cnode->Delta_Value * In[0])
    + (Alpha * cnode->W_In_Change1);
Diff_Weight2 = (Nu * cnode->Delta_Value * In[1])
    + (Alpha * cnode->W_In_Change2);
Diff_Bias = (Nu * cnode->Delta_Value)
    + (Alpha * cnode->Change_In_Bias);
cnode->Change_In_Bias = Diff_Bias;
cnode->W_In_Change1 = Diff_Weight1;
cnode->W_In_Change2 = Diff_Weight2;
cnode->Bias += Diff_Bias;
cnode->Weight_In1 += Diff_Weight1;
cnode->Weight_In2 += Diff_Weight2;
}

/***/
| Change_Weight_Output() |
| This changes the weights of the output layer. Note that |
| since the output bias is only a single value, it is |
| updated separate from this routine, unlike that hidden |
| layer, the biases can be changed in the same routine |
\/***/

void Change_Weight_Output(cnode,o_node)
NODE *cnode;
OUT_NODE *o_node;
{
float Diff_Weight;
if (cnode->next != NULL){
    Change_Weight_Output(cnode->next,o_node);
}

Diff_Weight = (Nu * o_node->Delta_Value * cnode->Output_Value) + (Alpha * cnode->W_Out_Change);

cnode->W_Out_Change = Diff_Weight;
cnode->Weight_Out += Diff_Weight;
}

/*****************************************
| Change_Output_Bias()
| The function changes a single value of the output bias.
| The bias is treated as a weight would be.
|*****************************************/
void Change_Output_Bias(o_node)
OUT_NODE *o_node;
{
float Diff_Bias;
Diff_Bias = (Nu * o_node->Delta_Value) + (Alpha * o_node->Change_In_Bias);
o_node->Change_In_Bias = Diff_Bias;
o_node->Bias +=Diff_Bias;
}

/*******************************************
| End of XOR Simulation program (1992)
| ** This program is classified as freeware. **
| ** Program was solely written by Alex Tay Leng Phuan. **
|-----------------------------------------------
| It has shown characteristics of a good working program,
| but the author assumes no responsibility for any failures
| in the code. Feel free to use it, play with it, or
| modify it. Please exclude the author's name if you have
| modified any part of the program.
|*******************************************/
Appendix B
Appendix B  FLANN I Programs

 Programmed by: Alex Tay Leng Phuan
 Title: Parity checker using FLANN I

 This program uses FLANN I to learn the parity system by first deciding if the input vector has been learned before. Instead of implementing the entire FLANN I, only the summation function is required, because the vigilance factors have been set to 1. Thus if alpha and rou are 1, it implies that the summation function must be a 1 for the input vector to match a learned pattern. Note that the system has been implemented using a dynamic linked list, which grows as the need arises. To make the system operate on various bit sizes, all you have to do is change #define nbits to the size you want and compile again. The system is designed to read in the bits from a file and then do a learn/identify. Output is sent to the screen.

 Dated: 10 Nov 1993

 The programmer has done his best here, to ensure that the system is bug free. Any roaches within the program are definitely not the intention of yours truly. If you do find any, just kill them. But believe me, I did not see it. SERIOUS! HONEST! Scout's Honour!

 ps: I was never a scout.

 *****************************************
 Individual structure of each output node
 *****************************************

 struct output {
   float bit[nbits];
   int parity;
   struct output *next;
 }

#define OUTPUT struct output

#include <stdio.h>
#include <math.h>

#define nbits 8
#define alpha 0.98
#define rou 0.98

/* *****************************************
 Individual structure of each output node
 *****************************************/

*/
main(argc, argv)
int argc;
char *argv[];
{
FILE *infile;
int i, j;
int nbits; /* temp storage of input vector */
int start = 0; /* flag to indicate first node i.e. head */
int search_end; /* end of linked list, item not found */
int rou_sum, alpha_sum; /* not used here, but can be if you */
/* want to relax the vigilance for */
/* other applications */
int found; /* found a match */
int pattern = 0; /* checking number of new input nodes */

int sum;
float f_sum;
float ON, OFF;
OUTPUT *create();

if (argc < 2) {
    printf("Usage: function <input filename>\n");
    exit(1);
}

if ((infile=fopen(argv[1],"r")==(FILE*)NULL) {
    printf("Unable to open %s\n",argv[1]);
    exit(1);
}

while(fscanf(infile,"%d ",&t[0])!=EOF) {
    for (i=1; i<nbits; i++) fscanf(infile,"%d ",&t[i]);
}

| Creation of the head of the linked list |
| ****************************************|
if (start == 0) {
    head = create();
    sum = 0;
    for (i=0; i<nbits; i++) {sum+=t[i];}
    ON = 1.0/(float)sum;
    OFF = -1.0/(float)(sum*nbits);

    for (i=0; i<nbits; i++) {
        if (t[i]==1) {
            head->bit[i]=ON;
        }
        else {
            head->bit[i]=OFF;
        }
    }
head->bit[i]=OFF;
    }
}
if(sum/2 != (float)sum/2.0)
    head->parity = 1;
else {head->parity = 0;}
start = 1;
tail = head;
pattern ++;
printf("learning head at %d\n",pattern);
/*....................................................................
 | If the head has already been created, then
 |....................................................................*/
else {
    node = head;
    search_end = 0;
    found = 0;
    while (search_end == 0){
        f_sum = 0.0;
        for (i=0;i<nbits;i++){
            f_sum += (float)t[i] * node->bit[i];
        }
        if (f_sum < 0.98){ /* actually 1, but given and */
            /* take during floating point*/
            /* calculations. */
            if (node->next == NULL){ /* reached the tail end */
                search_end = 1;  }
            else{
                temp = node->next;
                node = temp;
            }
        }
    else{
        found = 1;       /* set found to 1 for next phase */
        search_end = 1;          /* get out of search loop */
        }
    }
    /*....................................................................
    | When the item is not found, make new output node
    | and store it into the system as a new node. Do
    | not forget to do the linked list updating such as
    | tail extension etc.
    |....................................................................*/
    if (found == 0){
        node = create();
        tail->next = node;
        tail = node;
        sum = 0;
        for(i=0;i<nbits;i++) {sum += t[i];}
ON = 1.0/(float)sum;
OFF = -1.0/(float) (sum*nbits);

for(i=0;i<nbits;i++)
{
    if (t[i]==1){
        node->bit[i]=ON;
    }
    else{
        node->bit[i]=OFF;
    }
}

if(sum/2 != (float)sum/2.0){
    node->parity = 1;
}
else {node->parity = 0;}

pattern ++;
printf("learning new pattern %d\n",pattern);

} /* If match is found, print out the contents and parity */

else {
    for (i=0;i<nbits;i++){
        printf ("%d ",t[i]);
    }
    printf (" parity is %d\n",node->parity);
}

} /* Creating a new node whenever the main program needs one. It then send back the pointer to the new node. */

OUTPUT *create()
{
    OUTPUT *new_node;
    new_node=(OUTPUT*) malloc (sizeof(OUTPUT));
    new_node->next = NULL;
    return(new_node);
}
This program is the character recognition example for FLANN I. It uses a 7 X 10 grid mentioned in chapter 5. The main bulk of the program is X Window programming. The FLANN I system starts in the X environment loop towards the middle of the program. All the colours are predefined and command labels are all preset.

If you are going to use this interface, I think it is easier if you went ahead to build another. It is probably easier and faster, because the techniques here are very specific to the program. You may not like the programming style either.

```
#include <stdio.h>
#include <string.h>
#include <X11/Xlib.h>
#include <X11/cursorfont.h>
#include <X11/Xutil.h>

#define DEFAULT_EXIT_TEXT "Exit"
#define DEFAULT_RUN_TEXT "Test"
#define DEFAULT_GO_TEXT "Go"
#define DEFAULT_TRAIN_TEXT "Train"
#define DEFAULT_LOAD_TEXT "Load"
#define DEFAULT_SAVE_TEXT "Save"
#define DEFAULT_STORE_TEXT "Store"
#define DEFAULT_LEARN_TEXT "Learn"
#define DEFAULT_INIT_TEXT "Init"
#define DEFAULT_ONE "1"
#define DEFAULT_TWO "2"
#define DEFAULT_THREE "3"
#define DEFAULT_FOUR "4"
#define DEFAULT_FIVE "5"
#define DEFAULT_SIX "6"
#define DEFAULT_SEVEN "7"
#define DEFAULT_EIGHT "8"
#define DEFAULT_NINE "9"
#define DEFAULT_ZERO "0"
#define BLUE "blue"
#define LBLUE "light blue"
#define BLACK "black"
#define RBLUE "royal blue"
#define BLACK "black"
#define WHITE "white"
#define RED "red"
#define YELLOW "yellow"
#define GREY "grey"
```
Appendix B  FLANN I Programs

#define DEFAULT_BDWIDTH 1
#define DEFAULT_FONT "helvb24"
#define DEFAULT_FILE "Train"
#define DEFAULT_OUT_FILE "Freezer"
#define row 10
#define column 7
#define in_node 70
#define out_node 500

/****************************
| Defining all the variables |
\******************************

char *main_bg = LBLUE,
*main_fg = BLACK,
*main_font = DEFAULT_FONT,
*W1_color = RBLUE,
*exit_bg = YELLOW,
*exit_fg = BLACK,
*exit_font = DEFAULT_FONT,
*Z_color = BLACK,
*O_color = WHITE,
*button_color = BLUE,
*outfgcolor = GREY,
*outbgcolor = GREY,
*display_name = NULL,
*exit_text = DEFAULT_EXIT_TEXT,
*test_text = DEFAULT_RUN_TEXT,
*go_text = DEFAULT_GO_TEXT,
*train_text = DEFAULT_TRAIN_TEXT,
*load_text = DEFAULT_LOAD_TEXT,
*save_text = DEFAULT_SAVE_TEXT,
*store_text = DEFAULT_STORE_TEXT,
*learn_text = DEFAULT_LEARN_TEXT,
*init_text = DEFAULT_INIT_TEXT,
*file = DEFAULT_FILE,
*ofile = DEFAULT_OUT_FILE,
*onetext = DEFAULT_ONE,
*twotext = DEFAULT_TWO,
*threetext = DEFAULT_THREE,
*fourtext = DEFAULT_FOUR,
*fivetext = DEFAULT_FIVE,
*sixtext = DEFAULT_SIX,
*seventext = DEFAULT_SEVEN,
*eighttext = DEFAULT_EIGHT,
*ninetext = DEFAULT_NINE,
*zerotext = DEFAULT_ZERO;

Display *p_disp;
unsigned int width, height, utemp;
char *app_name = "XART1";
XFontStruct *mfontstruct,
*efontstruct;
XWMHints xwmh;
XSizeHints xsh;
Window Main, Exit, W1, Zero, One, Two,
Three, Four, Five, Six, Seven, Eight,
Nine, Init, Test, Go, Load, Save, Store,
Learn;
GC theGC, exitGC, W1GC, OGC, ZGC,
inputBGC, inputWGC;
XEvent theEvent;
unsigned long main_bgpix, main_fgpix,
exit_fgpix, exit_bgpix,
button_pix, W1_pix, Z_pix, O_pix;
int x, y, xmin, xmax, ymin, ymax, itemp,
new_value, old_value;
int Done = 0, T_Value, O_Value, storage = 0,
vigil_test, init_flag=0, start_flag = 0;
float vigil = 0.70;
int ex, ey;
int IN[column][row],in[in_node];
int recode_num;
unsigned int width, height, utemp;

typedef struct output_node{
  float W[in_node];
  int T[in_node];
  int ctrlj;
  int value; /*number assigned for image*/
};

#define NODEJ struct output_node
NODEJ node_j[out_node];
NODEJ winner;

void initialize();
NODEJ execute();
int vigilance();
void remember();
void display();
void change_store();
void call_exit();
void recode();

FILE *outfile, *infile;
/************                       \  
| Beginning the program proper. This section sets the window \ 
| interface for the input system. It doesn't run FLANN I \ 
| until all parameters to the window are ready. \ 
\****************************************************/

main(argc, argv)
int argc;
char **argv;
{
    int i, j, t, k;
    float t2, t0;
    char *tmpstr;
    Colormap default_cmap;
    XColor color;
    int bitmask;
    XGCValues gcv;
    XSetWindowAttributes xswa;
    app_name = argv[0];

    for (i=0; i<column; i++){
        for (j=0; j<row; j++){
            IN[i][j]=0;
        }
    }

    if ((p_disp = XOpenDisplay(display_name)) == NULL)  
    {  
        fprintf(stderr,"%s: can't open display named %s\n",  
                argv[0], XDisplayName(display_name));
        exit(1);
    }

    if ((efontstruct = XLoadQueryFont(p_disp, exit_font)) == NULL)  
    {  
        fprintf(stderr,"%s: display %s cannot load font %s\n",  
                app_name, DisplayString(p_disp), exit_font);
        exit(1);
    }

    if ((mfontstruct = XLoadQueryFont(p_disp, main_font)) == NULL)  
    {  
        fprintf(stderr,"%s: display %s cannot load font %s\n",  
                app_name, DisplayString(p_disp), main_font);
        exit(1);
    }
}
Appendix B  FLANN I Programs

/***********************
| Beginning of colour map set up |
\**************************/
default_cmap = DefaultColormap(p_disp,DefaultScreen(p_disp));

if( XPARColor(p_disp,default_cmap,main_bg,&color) == 0 ||
    XAllocColor(p_disp,default_cmap,&color) == 0)
    main_bgpix = WhitePixel(p_disp,DefaultScreen(p_disp));
else
    main_bgpix = color.pixel;

if( XPARColor(p_disp,default_cmap,main_fg,&color) == 0 ||
    XAllocColor(p_disp,default_cmap,&color) == 0)
    main_fgpix = WhitePixel(p_disp,DefaultScreen(p_disp));
else
    main_fgpix = color.pixel;

if( XPARColor(p_disp,default_cmap,exit_bg,&color) == 0 ||
    XAllocColor(p_disp,default_cmap,&color) == 0)
    exit_bgpix = WhitePixel(p_disp,DefaultScreen(p_disp));
else
    exit_bgpix = color.pixel;

if( XPARColor(p_disp,default_cmap,exit_fg,&color) == 0 ||
    XAllocColor(p_disp,default_cmap,&color) == 0)
    exit_fgpix = BlackPixel(p_disp,DefaultScreen(p_disp));
else
    exit_fgpix = color.pixel;

if( XPARColor(p_disp,default_cmap,W1_color,&color) == 0 ||
    XAllocColor(p_disp,default_cmap,&color) == 0)
    W1_pix = WhitePixel(p_disp,DefaultScreen(p_disp));
else
    W1_pix = color.pixel;

if( XPARColor(p_disp,default_cmap,O_color,&color) == 0 ||
    XAllocColor(p_disp,default_cmap,&color) == 0)
    O_pix = WhitePixel(p_disp,DefaultScreen(p_disp));
else
    O_pix = color.pixel;

if( XPARColor(p_disp,default_cmap,Z_color,&color) == 0 ||
    XAllocColor(p_disp,default_cmap,&color) == 0)
    Z_pix = WhitePixel(p_disp,DefaultScreen(p_disp));
else
    Z_pix = color.pixel;

if ( XPARColor(p_disp,default_cmap,button_color,&color)==0 ||
    XAllocColor(p_disp,default_cmap,&color)==0)
button_pix = WhitePixel(p_disp, DefaultScreen(p_disp));
else
    button_pix = WhitePixel(p_disp, DefaultScreen(p_disp));
button_pix = color.pixel;

/***********************
| End color map set up |
***********************

xsh.flags = (PPosition | PSize | PMinSize | PMaxSize);
xsh.height = (DisplayHeight(p_disp, DefaultScreen(p_disp))
- (2*DEFAULT_BDWIDTH)-29);
xsh.min_height = xsh.height;
xsh.max_height = xsh.height;
xsh.width = (DisplayWidth(p_disp, DefaultScreen(p_disp))
- (2*DEFAULT_BDWIDTH)-8);
xsh.min_width = xsh.width;
xsh.max_width = xsh.width;
xsh.x = 0;
xsh.y = 0;

/***********************
| Begin initialising windows for interface |
***********************
Main = XCreateSimpleWindow(p_disp, DefaultRootWindow(p_disp),
xsh.x, xsh.y, xsh.width, xsh.height,
DEFAULT_BDWIDTH, main_fgpix, main_bgpix);
XSetStandardProperties(p_disp, Main, app_name, app_name,
None, argv, argc, &xsh);

xwmh.flags = (InputHint | StateHint);
xwmh.input = False;
xwmh.initial_state = NormalState;
XSetWMHints(p_disp, Main, &xwmh);

gcv.font = efontstruct->fid;
gcv.foreground = exit_fgpix;
gcv.background = exit_bgpix;
exitGC = XCreateGC(p_disp, Main,
(GCFont | GCForeground | GCBbackground), &gcv);

gcv.font = efontstruct->fid;
gcv.foreground = W1_pix;
gcv.background = W1_pix;
W1GC = XCreateGC(p_disp, Main,
(GCFont | GCForeground | GCBbackground), &gcv);
Appendix B  FLANN I Programs

gcv.foreground = Z_pix;
gcv.background = Z_pix;
ZGC = XCreateGC(p_disp, Main,
   (GCForeground | GCBackground), &gcv);

gcv.foreground = O_pix;
gcv.background = O_pix;
OGC = XCreateGC(p_disp, Main,
   (GCForeground | GCBackground), &gcv);

gcv.font = efontstruct->fid;
gcv.foreground = Z_pix;
gcv.background = button_pix;
inputBGC = XCreateGC(p_disp, Main, (GCFont|GCForeground|GCBackground), &gcv);

gcv.font = efontstruct->fid;
gcv.foreground = O_pix;
gcv.background = button_pix;
inputWGC = XCreateGC(p_disp, Main, (GCFont|GCForeground|GCBackground), &gcv);

Arrow = XCreateFontCursor(p Disp, XC_arlrow);
Clock = XCreateFontCursor(p Disp, XC_watch);

gcv.font = mfontstruct->fid;
gcv.foreground = main_fgpix;
gcv.background = main_bgpix;
theGC = XCreateGC(pDisp, Main,
   (GCFont | GCForeground | GCBackground), &gcv);

xswa.colormap = DefaultColormap(p Disp, DefaultScreen(p Disp));
xswa.bit_gravity = CenterGravity;
XChangeWindowAttributes(p Disp, Main,
   (CWColormap | CWBitGravity), &xswa);
XSelectInput(p Disp, Main, ExposureMask);
XMapWindow(p Disp, Main);

ex = 20;
ey = 50;
Zero = XCreateSimpleWindow(p Disp, Main, ex, ey,
   40, 40, DEFAULT BD_WIDTH,
   button_pix, button_pix);
XSelectInput(p Disp, Zero, ExposureMask | ButtonPressMask);

XMapWindow(p Disp, Zero);

ex = 70;
ey = 50;
One = XCreateSimpleWindow(p disp, Main, ex, ey,
40, 40, DEFAULT BDWIDTH,
button_pix, button_pix);
XSelectInput(p disp, One, ExposureMask | ButtonPressMask);
XMapWindow(p disp, One);

ex = 20;
ey = 100;
Two = XCreateSimpleWindow(p disp, Main, ex, ey,
40, 40, DEFAULT BDWIDTH,
button_pix, button_pix);
XSelectInput(p disp, Two, ExposureMask | ButtonPressMask);
XMapWindow(p disp, Two);

ex = 70;
ey = 100;
Three = XCreateSimpleWindow(p disp, Main, ex, ey,
40, 40, DEFAULT BDWIDTH,
button_pix, button_pix);
XSelectInput(p disp, Three, ExposureMask | ButtonPressMask);
XMapWindow(p disp, Three);

ex = 20;
ey = 150;
Four = XCreateSimpleWindow(p disp, Main, ex, ey,
40, 40, DEFAULT BDWIDTH,
button_pix, button_pix);
XSelectInput(p disp, Four, ExposureMask | ButtonPressMask);
XMapWindow(p disp, Four);

ex = 70;
ey = 150;
Five = XCreateSimpleWindow(p disp, Main, ex, ey,
40, 40, DEFAULT BDWIDTH,
button_pix, button_pix);
XSelectInput(p disp, Five, ExposureMask | ButtonPressMask);
XMapWindow(p disp, Five);

ex = 20;
ey = 200;
Six = XCreateSimpleWindow(p disp, Main, ex, ey,
40, 40, DEFAULT BDWIDTH,
button_pix, button_pix);
XSelectInput(p_disp,Six,ExposureMask | ButtonPressMask);
XMapWindow(p_disp,Six);

ex = 70;
ey = 200;
Seven = XCreateSimpleWindow(p_disp, Main, ex, ey,
40, 40, DEFAULT_BDWIDTH,
button_pix, button_pix);
XSelectInput(p_disp,Seven,ExposureMask | ButtonPressMask);
XMapWindow(p_disp,Seven);

ex = 20;
ey = 250;
Eight = XCreateSimpleWindow(p_disp, Main, ex, ey,
40, 40, DEFAULT_BDWIDTH,
button_pix, button_pix);
XSelectInput(p_disp,Eight,ExposureMask | ButtonPressMask);
XMapWindow(p_disp,Eight);

ex = 70;
ey = 250;
Nine = XCreateSimpleWindow(p_disp, Main, ex, ey,
40, 40, DEFAULT_BDWIDTH,
button_pix, button_pix);
XSelectInput(p_disp,Nine,ExposureMask | ButtonPressMask);
XMapWindow(p_disp,Nine);

ex = 20;
ey = 320;
Exit = XCreateSimpleWindow(p_disp, Main, ex, ey,
90, 40, DEFAULT_BDWIDTH,
exit_fgpix, exit_bgpix);
XSelectInput(p_disp,Exit,ExposureMask | ButtonPressMask);
XMapWindow(p Disp,Exit);

ex = 120;
ey = 320;
Init = XCreateSimpleWindow(p Disp, Main, ex, ey,
90, 40, DEFAULT_BDWIDTH,
exit_fgpix, exit_bgpix);
XSelectInput(p Disp,Init,ExposureMask | ButtonPressMask);
XMapWindow(p_disp, Init);
ex = 320;
ey = 100;
Go = XCreateSimpleWindow(p_disp, Main, ex, ey,
90, 40, DEFAULT_BDWIDTH,
exit_fgpix, exit_bgpix);
XSelectInput(p_disp, Go, ExposureMask | ButtonPressMask);
XMapWindow(p_disp, Go);
ex = 320;
ey = 150;
Test = XCreateSimpleWindow(p_disp, Main, ex, ey,
90, 40, DEFAULT_BDWIDTH,
exit_fgpix, exit_bgpix);
XSelectInput(p_disp, Test, ExposureMask | ButtonPressMask);
XMapWindow(p_disp, Test);
ex = 320;
ey = 50;
Store = XCreateSimpleWindow(p_disp, Main, ex, ey,
90, 40, DEFAULT_BDWIDTH,
exit_fgpix, exit_bgpix);
XSelectInput(p_disp, Store, ExposureMask | ButtonPressMask);
XMapWindow(p_disp, Store);
ex = 420;
ey = 50;
Learn = XCreateSimpleWindow(p_disp, Main, ex, ey,
90, 40, DEFAULT_BDWIDTH,
exit_fgpix, exit_bgpix);
XSelectInput(p_disp, Learn, ExposureMask | ButtonPressMask);
XMapWindow(p_disp, Learn);
ex = 220;
ey = 320;
Save = XCreateSimpleWindow(p_disp, Main, ex, ey,
90, 40, DEFAULT_BDWIDTH,
exit_fgpix, exit_bgpix);
XSelectInput(p_disp, Save, ExposureMask | ButtonPressMask);
XMapWindow(p_disp, Save);
Appendix B  FLANN I Programs

Load = XCreateSimpleWindow(p_disp, Main, ex, ey,
    90, 40, DEFAULT_BDWIDTH,
    exit_bgpix, exit_bgpix);
XSelectInput(p_disp,Load,ExposureMask | ButtonPressMask);
XMapWindow(p_disp,Load);

ex = 150;
ey = 50;
W1 = XCreateSimpleWindow(p_disp, Main, ex, ey,
    156, 222, DEFAULT_BDWIDTH,
    W1_pix, W1_pix);
XSelectInput(p_disp,W1,ExposureMask | ButtonPressMask);
XMapWindow(p_disp,W1);

/*****************************************************/
| Beginning of event loops. X Windows run the interface with |
| Event detections. Events are user interactions with the computer. |
| e.g. mouse movement is an event. Upon receiving specific |
| events, the endless loop that follows is programmed with the |
| right response. It will then do the appropriate function. |
/*****************************************************/

while (!Done)
{
    XNextEvent(p_disp,&theEvent);
    /*****************************************************/
    | Window with 7 X 10 grid for number input |
    /*****************************************************/
    if(theEvent.xany.window == Main)
    {
        if(theEvent.type == Expose &&
            theEvent.xexpose.count == 0)
        {
            if(XGetGeometry(p_disp,Main,&utemp,
                &itemp, &itemp, &width, &height,
                &utemp, &utemp) == 0) break;
            XClearWindow(p_disp,Main);
            XDrawString(p_disp,Main,theGC,60,400,zerotext,2);
            XDrawString(p_disp,Main,theGC,90,400,onetext,2);
            XDrawString(p_disp,Main,theGC,120,400,threetext,2);
            XDrawString(p_disp,Main,theGC,150,400,threetext,2);
            XDrawString(p Disp,Main, theGC, 180, 400, fivetext, 2);
            XDrawString(p Disp, Main, theGC, 210, 400, sixtext, 2);
            XDrawString(p Disp, Main, theGC, 240, 400, sevetext, 2);
            XDrawString(p Disp, Main, theGC, 270, 400, sextext2, 2);
        }
    }
}
XDrawString(p_disp,Main,theGC,300,400,eighttext,2);
XDrawString(p_disp,Main,theGC,330,400,ninetext,2);
if (start_flag == 0){
  for (i=0;i<10;i++){
    XFillRectangle(p_disp,Main,ZGC,(i*30)+55,430,20,20);
  } start_flag = 1;
}

Terminate program window
if (theEvent.xany.window == Exit){
  switch(theEvent.type){
    case Expose:
      if(theEvent.xexpose.count == 0){
        XClearWindow(p_disp,Exit);
        XDrawString(p_disp,Exit, exitGC, 19,30,exit_text, 4);
      }
      break;
    case ButtonPress:
      printf("%d %s",node_j[0].T[1], node_j[0].W[1]);
      Done = 1;
  }
}

if (theEvent.xany.window == W1){
  switch(theEvent.type){
    case Expose:
      if(theEvent.xexpose.count == 0){
        XClearWindow(p_disp,W1);
        for (i=0; i<column ; i++){
          for (j=0; j<row; j++){
            x = 2+(i*22);
            y = 2+(j*22);
            if (IN[i][j] == 0){
              XFillRectangle(p_disp,W1,ZGC,x,y,20,20);
            } else XFillRectangle(p_disp,W1,OGC,x,y,20,20);
          }
        }
      }
      break;
    case ButtonPress:
      {
  }
}


\[ x = \text{theEvent.xbutton.x}; \]
\[ y = \text{theEvent.xbutton.y}; \]
\[ i = 0; \]
\[ \text{while} (i < \text{column}) \{
    j = 0;
    \text{while} (j < \text{row}) \{
        \text{xmin} = 2 + (i \times 22);
        \text{xmax} = \text{xmin} + 20;
        \text{ymin} = 2 + (j \times 22);
        \text{ymax} = \text{ymin} + 20;
        \text{if} (((\text{xmin}<x)\&(\text{xmax}>x))\&(\text{ymin}<y)\&(\text{ymax}>y)) \{
            \text{if} (\text{IN}[i][j] == 0) \{
                \text{IN}[i][j] = 1;
                \text{XFillRectangle}(\text{p}_\text{disp}, \text{W1}, \text{OGC}, \text{xmin}, \text{ymin}, 20, 20);
            \} \text{else} \{
                \text{IN}[i][j] = 0;
                \text{XFillRectangle}(\text{p}_\text{disp}, \text{W1}, \text{ZGC}, \text{xmin}, \text{ymin}, 20, 20);
            \}
        \}
        j++;
    \}
    i++;
\}
\]
\[ \text{k} = 0; \]
\[ \text{for} (i=0; i<\text{column}; i++) \{
    \text{for} (j=0; j<\text{row}; j++) \{
        \text{in}[k] = \text{IN}[i][j];
        k++;
    \}
\}
\]
\[ \text{break; } \]

\[ \text{if} (\text{theEvent.xany.window} == \text{Init}) \{
    \text{switch(\text{theEvent.type})} \{
    \text{case Expose:} \{ \text{if(\text{theEvent.xexpose.count} == 0}) \{ \text{XClearWindow(\text{p}_\text{disp}, \text{Init});} \text{XDrawString(\text{p}_\text{disp}, \text{Init, exitGC, 20, 30, init_text, 5});} \} \text{break;} \text{case ButtonPress:} \{ \}
\]
XUndefineCursor(p_disp,Main);
XDefineCursor(p_disp,Main,Clock);
XFlush(p_disp);
initialize(0);
XUndefineCursor(p_disp,Main);
XDefineCursor(p_disp,Main,Arrow);
XFlush(p_disp);
break;
}
}

/****************************\
| Training Go ahead window |
\*****************************/

if (theEvent.xany.window == Go)
{
    switch(theEvent.type)
    {
        case Expose:
            if(theEvent.xexpose.count == 0)
            {
                XClearWindow(p_disp,Go);
                XDrawString(p_disp,Go,exitGC,25,30,go_text,3);
            }
            break;
        case ButtonPress:
            {
                XUndefineCursor(p_disp,Main);
                XDefineCursor(p_disp,Main,Clock);
                XFlush(p_disp);
                for (i=0;i<out_node;i++)
                {
                    node_j[i].ctrl_j = 1;
                }
                i = 0;
                vigil_test = 0;
                while (i<storage) 
                { 
                    winner = execute();
                    vigil_test = vigilance(winner);
                    if (vigil_test == 0) winner.ctrl_j = 0;
                    else (i = storage);
                    i++;
                }
                if(vigil_test == 1) display(winner);
                else if (vigil_test == 2){
                    printf("RE-CODING\n");
                    recode();
                    XFillRectangle(p_disp,Main,ZGC,
                    (winner.value*30)+55,430,20,20);
                    XFillRectangle(p_disp,Main,OGC,
                    (winner.value*30)+55,430,20,20);
(node_j[recode_num].value*30)+55,430,20,20);

}  
else { remember(storage);
    storage++;
    change_store(storage);
}
XUndefineCursor(p_disp,Main);
XDefineCursor(p_disp,Main,Arrow);
XFlush(p_disp);
break;
}
}

/*********************************************************
| Shows how many patterns have been stored             |
/**********************************************************/

if (theEvent.xany.window == Store)
{
    switch(theEvent.type)
    {
        case Expose:
            if(theEvent.xexpose.count == 0)
            {
                XClearWindow(p_disp,Store);
                XDrawString(p_disp,Store,exitGC,17,30,store_text,6);
            }
            break;
         case ButtonPress:
            {
                break;
            }
    }
}

/**********************************************************
| Forced learning window                               |
/**********************************************************/

if (theEvent.xany.window == Learn)
{
    switch(theEvent.type)
    {
        case Expose:
            if(theEvent.xexpose.count == 0)
            {
                XClearWindow(p_disp,Learn);
            }
Appendix B  FLANN I Programs

XDrawString(pDisp,Learn,exitGC,15,30,learn_text,6);
} break;
case ButtonPress:
{
    XUndefineCursor(pDisp,Main);
    XDefineCursor(pDisp,Main,Clock);
    XFlush(pDisp);
    remember(storage);
    storage++;
    change_store(storage);
    XUndefineCursor(pDisp,Main);
    XDefineCursor(pDisp,Main,Arrow);
    XFlush(pDisp);
    break;

} /*----------------------------------------------*/
{| Load stored values window               |
|------------------------------------------*/

if (theEvent.xany.window == Load)
{
    switch(theEvent.type)
    {
    case Expose:
        if (theEvent.xexpose.count == 0)
        {
            XClearWindow(pDisp,Load);
            XDrawString(pDisp,Load,exitGC,16,30,load_text,S);
        }
        break;
    case ButtonPress:
        {
            XUndefineCursor(pDisp,Main);
            XDefineCursor(pDisp,Main,Clock);
            XFlush(pDisp);
            if ((infile=fopen("art.dat","r")) == NULL)
            {printf("Error opening art.dat\n");
                exit(1);
            }
            t = 0; t2 = 0.0;
            fscanf(infile,"%d ",&storage);
            for (i=0; i<storage ;i++)
            {
                for (j=0; j<in_node ;j++)
                {
                    fscanf(infile,"%d ",&node[j][i].T[j]);
                    t = t + node[j][i].T[j];
                }
            }
Appendix B

FLANN I Programs

```c

t2 = 1.0/(0.001 + (float)t);
t0 = -1.0/((float)t * (float)(in_node-t));
printf("t is %d, t2 is %f and t0 is %fn",t,t2,t0);
for (k=0;k<in_node;k++){
    if (node_j[i].T[k] == 1) {
        node_j[i].W[k] = t2;
    } else node_j[i].W[k] = t0;
}
t = 0; t2 = 0.0; t0 = 0.0;
scanf(infile,"%d ",&node_j[i].value);
printf("value is %d\n", node_j[i].value);
change_store(storage);
initialize(storage);
XUndefineCursor(p_disp,Main);
XDefineCursor(p_disp,Main,Arrow);
XFlush(p_disp);
break;
}
}

/*****************************
| Save FLANN I states in file
|******************************/
if (theEvent.xany.window == Save) {
    switch(theEvent.type) {
    case Expose:
        if (theEvent.xexpose.count == 0) {
            XClearWindow(p_disp,Save);
            XDrawString(p_disp,Save,exitGC, 16,30,save_text,5);
        }
        break;
    case ButtonPress:
        { 
            XUndefineCursor(p_disp,Main);
            XDefineCursor(p_disp,Main,Clock);
            XFlush(p_disp);

            if ((outfile=fopen("art.dat","w")) == NULL) {
                printf("Error opening art.dat\n");
                exit(1);
            }
            if (init_flag == 0){
```
printf("Network not initialized\n");
call_exitO; }
fprintf(outfile,"%d\n",storage);
for (i=0;i<out_node;i++)
for (j=0;j<in_node;j++)
fprintf(outfile,"%d ",node_j[i].T[j]);
}
fprintf(outfile,"%d ",node_j[i].value);
fclose(outfile);
XUndefineCursor(p_disp,Main);
XDefineCursor(p_disp,Main,Arrow);
XFlush(p_disp);

/****************************************************
| Test without storing. This is to test if the pattern is already stored |
| or if it is a new pattern. It will not learn, but only test |
*****************************************************/
if (theEvent.xany.window == Test)
{
    switch (theEvent.type)
    {
    case Expose:
        if (theEvent.xexpose.count == 0)
        {
            XClearWindow(p_disp,Test);
            XDrawString(pDisp,Test,exitGC,20,30,test_text,5);
        }
        break;
    case ButtonPress:
    {
        XUndefineCursor(pDisp,Main);
        XDefineCursor(pDisp,Main,Clock);
        XFlush(p_disp);
        for (i=0;i<out_node;i++)
        {
            node_j[i].ctrlj = 1;
        }
        i = 0;
        vigil_test = 0;
        while (i<storage)
        {
            winner = executeQ;
            printf("winner is %d\n",winner.value);
            vigil_test = vigilance(winner);
            if (vigil_test == 0) winner.ctrlj = 0;
            else (i = storage);
            i++;
        }
    }
if((vigil_test == 1)||(vigil_test == 2)) display(winner);
    else {
        XFillRectangle(p_disp,Main,ZGC,(old_value*30)+55,430,20,20);
    }
    XUndefineCursor(p_disp,Main);
    XDefineCursor(p_disp,Main,Arrow);
    XFlush(p_disp);
    break;
}

/*************************************************************************
| Next 10 windows for indicating identification by FLANN I |
***************************************************************************/

if (theEvent.xany.window == Zero)
{
    switch(theEvent.type)
    {
    case Expose:
        if(theEvent.xexpose.count == 0)
        {
            XClearWindow(p_disp,Zero);
            if(T_Value == 0)
                XDrawString(p_disp,Zero, inputWGC,15,30,zerotext, 1);
            else
                XDrawString(p_disp,Zero, inputBGC,15,30,zerotext, 1);
        }
        break;
    case ButtonPress:
        O_Value = T_Value;
        T_Value = 0;
        XDrawString(p_disp,Zero, inputWGC,15,30,zerotext, 1);
        break;
    }
}

if (theEvent.xany.window == One)
{
    switch(theEvent.type)
    {
    case Expose:
        if(theEvent.xexpose.count == 0)
        {
            XClearWindow(p_disp,One);
            if(T_Value == 1)
                XDrawString(p_disp,One, inputWGC,15,30,onetext, 1);
        }
else
    XDrawString(p_disp,One, inputBGC,15,30,onetext, 1);
}
break;
case ButtonPress:
    O_Value = T_Value;
    T_Value = 1;
    XDrawString(p Disp,One, inputWGC,15,30,onetext, 1);
break;

if (theEvent.xany.window == Two)
{
    switch(theEvent.type)
    {
    case Expose:
        if(theEvent.xexpose.count == 0)
        {
            XClearWindow(p_disp,Two);
            if(T_Value == 2)
                XDrawString(p_disp,Two, inputWGC,15,30,twotext, 1);
            else
                XDrawString(p_disp,Two, inputBGC,15,30,twotext, 1);
        }
        break;
    case ButtonPress:
        O_Value = T_Value;
        T_Value = 2-
        XDrawString(p_disp,Two, inputWGC,15,30,twotext, 1);
    }
    break;
}

if (theEvent.xany.window == Three)
{
    switch(theEvent.type)
    {
    case Expose:
        if(theEvent.xexpose.count == 0)
        {
            XClearWindow(p_disp,Three);
            if(T_Value == 3)
                XDrawString(p_disp,Three, inputWGC,15,30,threetext, 1);
            else
                XDrawString(p_disp,Three, inputBGC,15,30,threetext, 1);
        }
        break;
    case ButtonPress:
        O_Value = T_Value;
        T_Value = 3-
        XDrawString(p_disp,Three, inputWGC,15,30,threetext, 1);
    }
O_Value = T_Value;
T_Value = 3;
XDrawString(p_disp,Three, inputWGC,15,30,threetext, 1);
break;
}

if (theEvent.xany.window == Four)
{
switch(theEvent.type)
{
case Expose:
    if(theEvent.xexpose.count == 0)
    {
        XClearWindow(p_disp,Four);
        if(T_Value == 4)
            XDrawString(p Disp,Four, inputWGC,15,30,fourtext, 1);
        else
            XDrawString(p Disp,Four, inputBGC,15,30,fourtext, 1);
    }
    break;

case ButtonPress:
    O_Value = T_Value;
    T_Value = 4;
    XDrawString(p Disp,Four, inputWGC,15,30,fourtext, 1);
    break;
}
}

if (theEvent.xany.window == Five)
{
switch(theEvent.type)
{
case Expose:
    if(theEvent.xexpose.count == 0)
    {
        XClearWindow(p Disp,Five);
        if(T_Value == 5)
            XDrawString(p Disp,Five, inputWGC,15,30,fivetext, 1);
        else
            XDrawString(p Disp,Five, inputBGC,15,30,fivetext, 1);
    }
    break;

case ButtonPress:
    O_Value = T_Value;
    T_Value = 5;
    XDrawString(p Disp,Five, inputWGC,15,30,fivetext, 1);
    break;
}
Appendix B FLANN I Programs

)

if (theEvent.xany.window == Six)
{
switch(theEvent.type)
{
case Expose:
    if(theEvent.xexpose.count == 0)
    {
        XClearWindow(p_disp, Six);
        if (T_Value == 6)
            XDrawString(p_disp, Six, inputWGC, 15, 30, sixtext, 1);
        else
            XDrawString(p_disp, Six, inputBGC, 15, 30, sixtext, 1);
    }
    break;
    case ButtonPress:
        O_Value = T_Value;
        T_Value = 6;
        XDrawString(p_disp, Six, inputWGC, 15, 30, sixtext, 1);
        break;
    }
    }
    if (theEvent.xany.window == Seven)
    {
switch(theEvent.type)
    {
case Expose:
        if(theEvent.xexpose.count == 0)
    {
        XClearWindow(p_disp, Seven);
        if (T_Value == 7)
            XDrawString(p_disp, Seven, inputWGC, 15, 30, seventext, 1);
        else
            XDrawString(p_disp, Seven, inputBGC, 15, 30, seventext, 1);
    }
    break;
    case ButtonPress:
        O_Value = T_Value;
        T_Value = 7;
        XDrawString(p_disp, Seven, inputWGC, 15, 30, seventext, 1);
        break;
    }
    }
    if (theEvent.xany.window == Eight)
    {
switch(theEvent.type)
    {
{  
case Expose:  
    if (theEvent.xexpose.count == 0)  
    {  
        XClearWindow(p_disp,Eight);  
        if (T_Value == 8)  
            XDrawString(p_disp,Eight, inputWGC,15,30,eighttext, 1);  
        else  
            XDrawString(p_disp,Eight, inputBGC,15,30,eighttext, 1);  
    }  
    break;  

case ButtonPress:  
    O_Value = T_Value;  
    T_Value = 8;  
    XDrawString(p_disp,Eight, inputWGC,15,30,eighttext, 1);  
    break;
}

if (theEvent.xany.window == Nine)  
{
    switch(theEvent.type)  
    {
    case Expose:  
        if (theEvent.xexpose.count == 0)  
        {  
            XClearWindow(p_disp,Nine);  
            if (T_Value == 9)  
                XDrawString(p_disp,Nine, inputWGC,15,30,ninetext, 1);  
            else  
                XDrawString(p_disp,Nine, inputBGC,15,30,ninetext, 1);  
        }  
        break;  
    case ButtonPress:  
        O_Value = T_Value;  
        T_Value = 9;  
        XDrawString(p_disp,Nine, inputWGC,15,30,ninetext, 1);  
        break;
    }
}

if (T_Value != O_Value)
{
    switch (O_Value)
    {
    case 0: XDrawString(p_disp,Zero, inputBGC,15,30,zerotext, 1);  
            break;

    case 1: XDrawString(p_disp,One, inputBGC,15,30,onetext, 1);  

    case 2: XDrawString(p_disp,Two, inputBGC,15,30,twotext, 1);  
            break;

    case 3: XDrawString(p_disp,Three, inputBGC,15,30,threetext, 1);  
            break;

    case 4: XDrawString(p_disp,Four, inputBGC,15,30,fourtext, 1);  
            break;

    case 5: XDrawString(p_disp,Five, inputBGC,15,30,fivetext, 1);  
            break;

    case 6: XDrawString(p_disp,Six, inputBGC,15,30,sixtext, 1);  
            break;

    case 7: XDrawString(p_disp,Seven, inputBGC,15,30,sevtext, 1);  
            break;

    case 8: XDrawString(p_disp,Eight, inputBGC,15,30,eighttext, 1);  
            break;

    case 9: XDrawString(p_disp,Nine, inputBGC,15,30,ninetext, 1);  
            break;
    }
}
Appendix B  FLANN I Programs

break;
case 2: XDrawString(p dispersion, Two, inputBGC, 15,30, twotext, 1);
    break;
case 3: XDrawString(p dispersion, Three, inputBGC, 15,30, threetext, 1);
    break;
case 4: XDrawString(p dispersion, Four, inputBGC, 15,30, fourtext, 1);
    break;
case 5: XDrawString(p dispersion, Five, inputBGC, 15,30, fivetext, 1);
    break;
case 6: XDrawString(p dispersion, Six, inputBGC, 15,30, sixtext, 1);
    break;
case 7: XDrawString(p dispersion, Seven, inputBGC, 15,30, seventext, 1);
    break;
case 8: XDrawString(p dispersion, Eight, inputBGC, 15,30, eighttext, 1);
    break;
case 9: XDrawString(p dispersion, Nine, inputBGC, 15,30, ninetext, 1);
    break;
}
}

call_exit();

\*/
End of main program. The functions are found in sub-program
\*/
Appendix B  FLANN I Programs

/******************************************
| This is the second half of the character recognition program
| which is used to compile both programs. When compiling, use
| the command:
|    cc xFLANN.c -lx11 -lm -o xFLANN
| and the compile will search for the other part of the program.
| This section of code contains all the functions. The flow is found
| in the other program.
\******************************************/
#include 'xFLANNa.c'
#include <XlllXatom.h>

/************************************************************************
| This section initialises the FLANN I output nodes. This is a
| primitive implementation, so arrays are used, instead of linked
| lists.
\************************************************************************/
void initialize(num)
int num;
{
int i,j;
float temp;
temp = (float) (1.0/(float)in_node); /*in_node is number of input nodes*/
for (i=num;i<out_node;i++) {
    for (j=0;j<in_node;j++) {
        node[j][i].T[j] = 1;
        node[j][i].W[j] = temp;
        node[j][i].ctrl[j] = 1;
        node[j][i].value = 0;
    }
}
/*initialise the screen and input grid*/
for (i=0;i<column ;i++)
    for (j=0;j<row ;j++)
    {
        IN[i][j] = 0;
    }
XClearWindow(p disp,W1);
for (i=0; i<column ; i++)
    for (j=0; j<row; j++)
    {
        x = 2+(i*22);
        y = 2+(j*22);
        if (IN[i][j] == 0){
            XFlllRectangle(p disp,W1,ZGC,x,y,20,20);
        }
        else XFlllRectangle(p disp,W1,OGC,x,y,20,20);
    }
init_flag = 1;
storage = num;
change_store(num);
}
/****************************************************/

\begin{verbatim}
\textbf{execute()}
\{
\textbf{int} \textbf{ij}, \textbf{win\_node}=0;
\textbf{float} \textbf{temp}=0.0, \textbf{win\_temp}=0.0;
\textbf{if (init\_flag} == 0) \{ \textbf{printfl("Network not initialized\n")} \textbf{call\_exit();} \}
\textbf{for (i=0;i<out\_node;i++)}{
\textbf{for (j=0;j<in\_node ;j++)}{
\textbf{temp} = \textbf{temp} + (\textbf{node\_j[i].W}[j] \ast \textbf{(float)}\textbf{in}[j] \ast \textbf{(float)}\textbf{node\_j[i].ctrl});
\}
\textbf{if (win\_temp < temp)}{
\textbf{if (vigilance(node\_j[i])!=0)} \{ \textbf{printf("%f %f\n",win\_temp,temp);} \textbf{win\_temp} = \textbf{temp};
\textbf{win\_node} = \textbf{i};
\}
\textbf{temp} = 0.0;
\}
\textbf{recode\_num} = \textbf{win\_node};
\textbf{return(node\_j[win\_node]);}
\}
\end{verbatim}

| ***********************************************

\begin{verbatim}
\textbf{vigilance(node)}
\{
\textbf{int} \textbf{i}, \textbf{t}=0,\textbf{x}=0, \textbf{T}=0;
\textbf{float} \textbf{temp1, temp2};
\textbf{for (i=0;i<in\_node;i++)}{
\textbf{x} = \textbf{x} + \textbf{in}[i];
\textbf{t} = \textbf{t} + (\textbf{node.T}[i]\ast\textbf{in}[i]);
\textbf{T} = \textbf{T} + \textbf{node.T}[i]/\textbf{*new vigil guard*/}
\}
\end{verbatim}

| ***********************************************

\begin{verbatim}
/*The next line is a new vigil guard implementation*/
/*Original guard does not have upward T comparison */
\textbf{temp1} = ((\textbf{float})\textbf{t}/(\textbf{float})\textbf{x});
\textbf{temp2} = ((\textbf{float})\textbf{t}/(\textbf{float})\textbf{T});
\textbf{if (temp1==temp2) \&\& (temp1 == 1.0)}\{\textbf{return(2);} \}
\textbf{if ((vigil > temp1)||(vigil > temp2))}\{\textbf{return(0);}\}
\end{verbatim}

|
else return(1);

void remember(num)
int num;
{
int i;
float temp=0.0,temp2,temp3;
if (init_flag == 0){
printf("Network not initialized\n");
call_exit();
}
nodej[num].value = T_Value; /*T_Value is assigned number*/
/*can be 1,2,3,4,5,6,7,8,9,0*/
/*Performing summation*/
for (i=0;i<in_node;i++)
{ temp = temp + (nodej[num].T[i]*in[i]);
}
temp2 = 1.0/(0.001+temp);
temp3 = -1.0/(temp * ((float)(in_node)-temp));
printf("temp2 is %f\n",temp2);
printf("temp3 is %f\n",temp3);
/*Assigning weights for individual connections*/
for (i=0;i<in_node;i++)
{ if (in[i]==1){
 nodej[num].T[i]=1;
 nodej[num].W[i]=temp2;
 }
else {
 nodej[num].T[i] = 0;
 nodej[num].W[i] = temp3;
 }
}
XFillRectangle(p_disp,Main,ZGC,(new_value*30)+55,430,20,20);
}

void display(winner)
NODEJ winner;
{
printf("winner.value is %d\n",winner.value);
new_value = winner.value;
XFillRectangle(p_disp,Main,ZGC,(old_value*30)+55,430,20,20);
XFillRectangle(p_disp,Main,OGC,(new_value*30)+55,430,20,20);
old_value = new_value;
printf("Winner\n");
}

/********************************************
| Keeps track of how many output nodes   |
| have been created.                   |
********************************************
void change_store(num)
int num;
{
char number[6];
sprintf(number,"%d",num); /*itoa*/
while (strlen(number)<5){ /*pad blank till 5 spaces*/
    strcat(number," "); }
strcpy(store_text,number);
XClearWindow(p_disp,Store);
XDrawString(p_disp,Store,exitGC,20,30,store_text,6);
}

/********************************************
| Changing 100% matches to new type      |
********************************************
void recodeO
{
    node_j[recode_num].value = T_Value;
    new_value = T_Value;
    old_value = T_Value;
}

/********************************************
| ADIOS!                                |
********************************************
void call_exitO
{
    XFreeGC(p_disp, theGC);
    XFreeGC(p_disp,exitGC);
    XFreeGC(p_disp,inputWGC);
    XFreeGC(p_disp,ZGC);
    XFreeGC(p_disp,OGC);
    XDestroyWindow(p_disp,Main);
    XCloseDisplay(p_disp);
    exit(0);
}
Appendix C
Programmed by: Alex Tay Leng Phuan
Program name: CONTROL.C

This program is an implementation of FLANN II for a control problem. It obtains the neural network control states from a paper published by J. Zhang and P.D. Roberts. The article was published in the Transactions of the Institute of Measurement and control. Vol 14, no. 4 (1992) pp 179 - 188. This is the first problem on the mixing process.

The program reads in the training data and the test data from a given file. The difference between the training data and the test data is that the training data have a definition byte > 0 and the test data has a 0 for the last byte. The program then learns the training set and classifies the test data according to its training set. The results are then output on screen.

The entire program works on a linked list basis and grows when new training states are encountered. But this given test set does not provide, so you won't see it work.

Program is dated: 9 Nov 1993
** The author and programmer does not guarantee this program is bug free. Use it if you must, learn it if you want, but **
** do remember the credit that it due, to non other than ****
**** YOURS TRUELY ****

#include <stdio.h>
#include <math.h>

#define div_value 100

struct output {
    int bit[6];
    int problem_number;
    struct output *next;
};

#define OUTPUT struct output

/* Beginning of main program which starts with the initialization of the entire neural network system. Note that rou and */
l tol can be changed by specifying the
l values at the command line.
\***************************************************/

main(argc, argv)
int argc;
char *argv[];

{ FILE *infile; /* input file name specified by the command line */
OUTPUT *createO; /* creates dynamically, an output node */
float MOD(); /* modulus function to get positive values only */
OUTPUT *head, *node, *temp; /* linked list nodes used in program */
OUTPUT *min, *tail; /* more linked list nodes used for operations */
float rou = 0.6; /* rou can be defined by the user at the command line */
int tol = 0; /* tol or delta, can be defined by user as above */
int t[7]; /* temporary storage of 101 dimensional signal vector */
int value; /* actual difference used for minimum function */
int min_value; /* most current minimum value */
int v_sum; /* integer value of descending sum */
int number = 6; /* total number of elements in vector */
int begin=1, end_flag; /* begin is used to determine head of list */
/* end_flag determines when tail is reached */
int i, found; /* found determines if a match has been located */

if (argc < 2){
printf("Usage: function <filename> <[rou] [tolerance]>");
exit(1);
}

if((infile=fopen(argv[1],"r")==(FILE*)NULL){
printf("Error trying to open %s\n",argv[1]);
exit(1);
}

if (argc == 4){
rou = (float) (atoi(argv[2])/(float)div_value);
tol = atoi(argv[3])/div_value;

}/** *************/

while ((fscanf(infile,"%d",&t[0])!=EOF){
for(i=1;i<7;i++)fscanf(infile,"%d",&t[i]);

while (fscanf(infile,"%d",&t[0])!=EOF){
for(i=1;i<7;i++)fscanf(infile,"%d",&t[i]);

}****
printf("%d %d %d %d %d %d %d\n",t[0],t[1],t[2],t[3],t[4],t[5],t[6]);

(rows not shown)

if (begin==1)
    begin = 0;
    head=create();
    for(i=0;i<6;i++){
        head->bit[i]=t[i];
    }
    head->problem_number = t[6];
tail = head;
}

(rows not shown)

else if(t[6] != 0){
    node = create();
    tail->next = node;
    tail = node;
    for(i=0;i<6;i++){
        node->bit[i]=t[i];
    }
    node->problem_number=t[6];
}

(rows not shown)

else {
    end_flag = 1;
    found = 0;
    node = head;

    while(end_flag==1){

v_sum = 0;
for (i=0;i<6;i++) { /* Vigilance testing mode */
    v_sum += Decern(tol-(MOD(node->bit[i]-t[i])));
} 
if(((float)v_sum/(float)number)>rou) { /* min calculating mode */
    value = 0;
    for (i=0;i<6;i++) {
        value += (node->bit[i]-t[i])*(node->bit[i]-t[i]);
    }
    if (found == 0) { /* indication of first find */
        min_value = value;
        min = node;
        found = 1;
    }
    else{
     if (min_value > value) { /* found another, but not the first */
         min_value = value;
         min = node;
     } 
    }
    if (node->next != NULL) { /* getting next node */
        temp = node->next;
        node = temp;
    } else end_flag = 0;
} 
if (found == 1) {
    printf("problem_number is %d\n",min->problem_number);
} else if (found == 0) { /* if no match, create a new node and */
    node = create(); /* store the novel vector in. Append */
    tail->next = node; /* the new node to the tail of the */
    tail = node; /* linked list */
    for(i=0;i<6;i++) { /* don't forget to define the new state */
        node->bit[i] = t[i];
    }
    printf("Please define new problem number\n");
    scanf("%d",node->problem_number);
}

****************************
| Dynamic allocation function for |
| new output nodes.               |
|********************************|
OUTPUT *create()
{
OUTPUT *node;
node = (OUTPUT*)malloc(sizeof(OUTPUT));
node->next=NULL;
return(node);
}

/* Decern function is just an ON/OFF function that helps in counting the number of correct vector element matches within the vector */
int Decern(value)
int value;
{
if (value < 0) return(0);
return(1);
}

/* Modulus function returns positive float numbers */
int MOD(value)
int value;
{
if (value < 0) return(-value);
return(value);
}

/* Complexity is really dependent on whether you are a Mathematician or an Engineer. */

This program is an implementation of FLANN II for a control problem. It obtains the neural network control states from a paper published by J. Zhang and P.D. Roberts. The article was published in the Transactions of the Institute of Measurement and control. Vol 14, no. 4 (1992) pp 179 - 188. This is the second problem on the CSTR system.

The program reads in the training data and the test data from a given file. The difference between the training data and the test data is that the training data have a definition byte > 0 and the test data have a 0 for the last byte. The program then learns the training set and classifies the test data according to its training set. The results are then output on screen.

The entire program works on a linked list basis and grows when new training states are encountered. But this given test set does not provide, so you won't see it work.

Program is dated: 9 Nov 1993
** The author and programmer does not guarantee this program **
** is bug free. Use it if you must, learn it if you want, but **
** do remember the credit that it due, to non other than **
**** YOURS TRUELY ****

#include <stdio.h>
#include <math.h>

#define div_value 100

struct output{
    int bit[14];
    int problem_number;
    struct output *next;
}

#define OUTPUT struct output
Appendix C  FLANN II Programs

/*****************************/
| Beginning of main program which starts |
| with the initialization of the entire |
| neural network system. Note that rou and |
| tol can be changed by specifying the |
| values at the command line. |
\******************************/

main(argc, argv)
int argc;
char *argv[];

{  
FILE *infile; /* input file name specified by the command line */
OUTPUT *createO; /* creates dynamically, an output node */
float MODO; /* modulus function to get positive values only */
OUTPUT *head, *node, *temp; /* linked list nodes used in program */
OUTPUT *min, *tail; /* more linked list nodes used for operations */
float rou = 0.6; /* rou can be defined by the user at the command line */
int tol = 0; /* tol or delta, can be defined by user as above */
int t[15]; /* temporary storage of 15 dimensional signal vector */
int value; /* actual difference used for minimum function */
int min_value; /* most current minimum value */
int v_sum; /* integer value of deceming sum */
int number = 14; /* total number of elements in vector */
int begin=1, end_flag; /* begin is used to determine head of list */
    /* end_flag determines when tail is reached */
int i, found; /* found determines if a match has been located */

if (argc < 2){
    printf("Usage: function <filename> <[rou] [tolerance]>");
    exit(1); }

if((infile=fopen(argv[1],"r"))==(FILE*)NULL){
    printf("Error trying to open %s
",argv[1]);
    exit(1);  }

if (argc == 4){
    rou = (float) (atoi(argv[2])/(float)div_value);
    tol = atoi(argv[3])/div_value;
}

/*****************************/
| End of initialization of the system. Now is the |
| time to read the file so that FLANN II can begin |
| forming itself. |
\******************************/
while (fscanf(infile,"%d",&t[0])!=EOF) {
    for(i=1;i<15;i++)fscanf(infile,"%d",&t[i]);
    printf("%d %d %d %d %d %d %d %d %d %d %d %d %d %d\n", t[0],t[1],t[2],t[3],t[4],t[5],t[6],t[7],t[8],t[9],t[10],
          t[11],t[12],t[13],t[14]);
}

/*****************************************
| Creation of the head of the linked list |
*******************************************/
if (begin==1) {
    begin = 0;
    head=创造O;
    for(i=0;i<14;i++) {
        head->bit[i]=t[i];
    }
    head->problem_number = t[14];
    tail = head;
}

/*****************************************
| If the head already exists, then process |
| the input data and check if the last bit |
| is a non zero.                           |
*******************************************/
else if(t[14] != 0) {
    node = createO;
    tail->next = node;
    tail = node;
    for(i=0;i<14;i++) {
        node->bit[i]=t[i];
    }
    node->problem_number=t[14];
}

/*****************************************
| Will execute classification if a 0 is |
| encountered at the end of the state    |
*******************************************/
else {
    end_flag = 1;
    found = 0;
    node = head;
}
while (end_flag==1) {
  v_sum = 0;
  for (i=0;i<14;i++){ /* Vigilance testing mode */
    v_sum+=Decern(tol-(MOD(node->bit[i]-t[i])));
  }
  if(((float)v_sum/(float)number)>rou){ /* min calculating mode */
    value = 0;
    for (i=0;i<14;i++)
      value+=(node->bit[i]-t[i])*(node->bit[i]-t[i]);
  }
  if (found == 0){ /* indication of first find */
    min_value = value;
    min = node;
    found = 1; }
  else{
    if (min_value > value){ /* found another, but not the first*/
      min_value = value;
      min = node; }
  }
  if (node->next != NULL){ /* getting next node */
    temp = node->next;
    node = temp; }
  else end_flag = 0;
}
if (found==1){
  printf("problem_number is \%d\n",min->problem_number);
}
else if (found == 0){ /* if no match, create a new node and */
  node = create(); /* store the novel vector in. Append */
  tail->next = node; /* the new node to the tail of the */
  tail = node; /* linked list */
  for(i=0;i<14;i++)
    node->bit[i]=t[i];
  printf("Please define new problem number\n");
  scanf("\%d",node->problem_number);
}
void createO(OUTPUT *node)
{
    node = (OUTPUT*)malloc(sizeof(OUTPUT));
    node->next=NULL;
    return(node);
}

int Decern(int value)
{
    if (value < 0) return(0);
    return(1);
}

int MOD(int value)
{
    if (value < 0) return(-value);
    return(value);
}
Programmed by: Alex Tay Leng Phuan
Program name: WAVENET.C

This program is an implementation of FLANN II for recognition of waveforms. In this scenario, 4 waveforms are presented to the network. They are the functions \(\sin(A)\), \(\sin(2A)\), \(\cos(A)\) and \(\cos(2A)\). FLANN II is first of all initialized with \(\text{rou}\) and \(\text{delta}\) values. (NB: \(\text{delta}\) is \(\text{tol}\)). The waveform digitized signals are then read into the neural network. The system, based on the given \(\text{rou}\) and \(\text{delta}\) values, begins to form its own decisions of whether the signal is novel or old. If it is novel, it will ask for a new definition, \(\text{wave_number}\). If it is an old waveform, it will establish the \(\text{wave_number}\) and print it out.

Program is dated: 8 Nov 1993

** The author and programmer does not guarantee this program is bug free. Use it if you must, learn it if you want, but **
** do remember the credit that it due, to non other than **
**** YOURS TRUELY ****

```
#include <stdio.h>
#include <math.h>

#define div_value 100.0

struct output{
    float bit[101];
    int wave_number; /*10-sine(A) 11-sine(2A) 20-cos(A) 21-cos(2A)*/
    struct output *next;
}

#define OUTPUT struct output

main(argc, argv)
int argc;
char *argv[];
```
Appendix C

FLANN II Programs

{ FILE *infile; /* input file name specified by the command line */ OUTPUT *createO; /* creates dynamically, an output node */ float MOD(); /* modulus function to get positive values only */ OUTPUT *head, *node, *temp; /* linked list nodes used in program */ OUTPUT *min, *tail; /* more linked list nodes used for operations */ float rou = 0.9; /* rou can be defined by the user at the command line */ float tol = 0.05; /* tol or delta, can be defined by user as above */ float t[101]; /* temporary storage of 101 dimensional signal vector */ float value; /* actual difference used for minimum function */ float min_value; /* most current minimum value */ int v_sum; /* integer value of discerning sum */ float number = 101.0; /* total number of elements in vector */ int begin = 1, end_flag; /* begin is used to determine head of list */ /* end_flag determines when tail is reached */ int i, found; /* found determines if a match has been located */

if (argc < 2){
    printf("Usage: function <filename> <[rou] [tolerance]>
");
    exit(1);
}

if((infile=fopen(argv[1],"r")==(FILE*)NULL){
    printf("Error trying to open %s
",argv[1]);
    exit(1);
}

if (argc == 4){
    rou = (float) (atoi(argv[2])/div_value);
    tol = (float) (atoi(argv[3])/div_value);
}

/****************************
  End of initialization of the system. Now is the
  time to read the file so that FLANN II can begin
  forming itself.

 ****************************/

while (fscanf(infile,"%f",&t[0])!=EOF){
    for(i=1;i<101;i++)fscanf(infile,"%f",&t[i]);
    printf("%f %f %f %f %f %f %f %f %f %f \n",t[0],t[1],t[2],t[3],t[4],t[5],
           t[6],t[7],t[8],t[9],t[10]);

/****************************
  Creation of the head of the linked list

  ****************************/

if (begin==1){

begin = 0;
head=create();
for(i=0;i<101;i++){
    head->bit[i]=t[i];
}
tail = head;
printf("Enter waveform class number 10-sine(A) 11-sine(2A) 20-cos(A) 21-
cos(2A)\n");
scanf("%d",&head->wave_number);

if the head already exists, then process
the input data and check if FLANN II has
a similar waveform.

else {

Init indicators for FLANN II
end_flag = 1;
found = 0;
nod = head;

Continue doing as long as there
are more FLANN II output nodes

while (end_flag==1){
    \ v_sum = 0;
    for (i=0;i<101;i++){ /* Vigilance testing mode */
        \ v_sum+=Discern(tol-(MOD(node->bit[i]-t[i]));
    }
    if(((float)v_sum/number)>rou){ /* min calculating mode */
        value = 0;
        for (i=0;i<101;i++){
            value+=(node->bit[i]-t[i])*(node->bit[i]-t[i]);
        }
        if (found == 0){ /* indication of first find */
            min_value = value;
            min = node;
            found = 1;
        }
        else{
            if (min_value > value){ /* found another, but not the first*/
                min_value = value;
            }
        }
    }
}
Appendix C  FLANN II Programs

min = node;
}
}

if (node->next != NULL) { /* getting next node */
    temp = node->next;
    node = temp;
} else end_flag = 0;

if (found==1) {
    printf("min->wave_number is %d", min->wave_number);
} else if (found == 0) { /* if no match, create a new node and */
    node = create(); /* store the novel vector in. Append */
    tail->next = node; /* the new node to the tail of the */
    tail = node; /* linked list */
    for(i=0;i<101;i++) { /* don't forget to define the new waveform */
        node->bit[i]=t[i];
    }
    printf("Enter waveform class number 10-sine(A) 11-sine(2A) 20-cos(A)
    21-cos(2A)\n");
    scanf("%d",&node->wave_number);
}
}
}

/************************************
| Dynamic allocation function for |
| new output nodes.              |
\************************************/

OUTPUT *create()
{
    OUTPUT *node;
    node = (OUTPUT*)malloc(sizeof(OUTPUT));
    node->next=NULL;
    return(node);
}

/************************************
| Discern function is just an ON/OFF |
| function that helps in counting    |
| the number of correct vector      |
| element matches within the vector  |
\************************************/

int Discern(value)
float value;
{
    if (value < 0) return(0);
    return(1);
}

/***********************************
* Modulus function returns positive *
* float numbers
***********************************/

float MOD(value)
float value;
{
    if (value < 0) return(-value);
    return(value);
}
Appendix D
#include <stdio.h>
#include <dos.h>
#include <thread.h>
#include <sema.h>
#include <par.h>
#include <net.h>
#include <time.h>
#include "spira1.h"

static SEMA para_ready;
static SEMA para_ready2;
POINT pt[192];
done = 0;
tally2 = 169;
struct {int x, y, x1, y1;} work_size;

main()
{
FILE *infile;
int tally = 0;
clock_t start, stop;
int i, j, ready, len;
int count = 0;
void broadcast();
void send();
struct {POINT p;} result;
struct {POINT p[64];} f_result;

if ((infile=fopen("spiral.dat","r"))==(FILE*)NULL){
    printf("Error opening spiral.dat\n");
    exit(1);
}

for (i=0;i<192;i++) {
    fscanf(infile,"\%f \%f \%d\n", &pt[i].x, &pt[i].y, &pt[i].class);
}
/**Useful feedback for debugging**

/*printf("Completed reading file\n");
printf("size of pt is %d\n",sizeof(pt));
printf("size of pt[0] is %d\n",sizeof(pt[0]));
printf("sizeof float is %d\n",sizeof(pt[0].x));
printf("sizeof int is %d\n",sizeof(pt[0].ciass));*/

/*Creating a thread for broadcasting to all slaves*/
if (thread_create(broadcast,200000,sizeof(pt),pt)==NULL) {
printf("Insufficient Heap Size, Change Configuration\n");
exit(1);
}

sem_signal(&para_ready);
while (tally < processors) {
len = net_receive(&result,&ready);
tally++;
}

/*More useful debuggin information*/
/*sema_wait(&par_sema);
printf("%f %f %d\n",result.p.x,result.p.y,result.p.ciass);
sema_signal(&par_sema);*/

if (thread_create(send,200000,0,NULL)==NULL) {
printf("Insufficient Heap Size, Change Configuration\n");
exit(1);
}
count = 0;
sema_signal(&para_ready);
while (done != 1 || tally2 > count) {
len = net_receive(&f_result,&ready);
/*sema_wait(&par_sema);
for (i=0;i<64;i++)
printf("%f %f %d\n",f_result.p[i].x,f_result.p[i].y,f_result.p[i].ciass);
sema_signal(&par_sema);*/
}
count ++;
/*sema_wait(&par_sema);
printf("count is \%d\n",count);
sema_signal(&par_sema);*/
}  

stop = clock();
sema_wait(&par_sema);
printf("time elapse is \%d secs\n",stop-start);
sema_signal(&par_sema);

void broadcast(pt)
POINT pt[192];
{  
struct {POINT p[192];} work;
int i;

//re-assignment for work package transmission

sema_wait(&para_ready);  
for (i=0;i<192;i++)  
{  
work.p[i].x=pt[i].x;
work.p[i].y=pt[i].y;
work.p[i].class=pt[i].class;
        
        /*sema_wait(&par_sema);
printf("size of work is \%d\n",sizeof(work));
printf("size of work.pt[0] is \%d\n",sizeof(work.p[0]));
sema_signal(&par_sema);*/

/*semawait(&par_sema);
printf("size of work is \%d\n",sizeof(work));
printf("size of work.pt[0] is \%d\n",sizeof(work.p[0]));
sema_signal(&par_sema);*/

//BEWARE: net_broadcast assumes you have nice
//round-off data sizes so it can broadcast and
//truncate at the 1024 byte point without chopping
//the data. As a result, the POINT structure had
//to use a 4 bit char pad to fill the gap. Not a
//very good library function, but it works.

net_broadcast(sizeof(work),&work);
sema_signal(&para_ready);
}

void send(void)
{
int i, j;
int start, end;
}
sema_wait(&para_ready);
start = -size/2;
end = size/2;
/*for(;;){*/
    for (i=start;i<end;i+=8){ /*8X8 block size*/
        work_size.x = i;
        work_size.x1 = (i+8);
        for (j=start;j<end;j+=8){
            work_size.y = j;
            work_size.y1 = (j+8);
            /*sema_wait(&par_sema);
            printf("x is %d x1 is %d y is %d y1 is %d\n", 
                work_size.x,work_size.x1,work_size.y,work_size.y1);
            sema_signal(&par_sema);*/
            net_send(sizeof(work_size),&work_size,1);
        }
    }
    done = 1;
    /* */
}/* */

Configuration file for processor farming. Use with fconfig.

Task Master File=SM Data = 500k
Task Worker File=SW Stack=10K Heap=400K Opt=Stack Opt=Code

Spiral.h header file
Some problems with net_broadcast is that if your
structure is not an even division of 1024 bits,
and you require greater than 1024 bits sent, it
kinda messes everything up. Don't they like to
leave these technical details from the manuals

typedef struct point {
    float x;
    float y;
    int class;
    int pad; /* For rounding each POINT struct to 16 bits */
    } POINT;

#define size 104 /* resolution of test points within area */
#define processors 1 /* number to tally with no. tputers */
/*****************************************************/
I Beginning of parallel C worker task for spiral problem. \
*****************************************************************************/

#include <net.h>
#include <dos.h>
#include <stdio.h>
#include "spiral.h"

main()
{
int len, ready=0;
int i = -1, j, k;
int l, min, flag;
float min_value, value, tx, ty;
******************************************************************************
I Declaration of structures for broadcast acceptance. IIIC
IIIC doesn't cater to odd bits during broadcasting, IIIC
IIIC so it is required that the structure POINT caters IIIC
IIIC to a rounded 1024 value when broadcasting. Otherw IIIC
IIIC the transimmted values will be garbage. IIIC
IIIC struct work is the broadcasted module, w is the IIIC
IIIC reconstruction after the broadcast. Therefore a IIIC
IIIC need to perform (+i*64) is loop till 3072 bits IIIC
IIIC have been completely sent. IIIC
\\******************************************************************************

struct {POINT p[64];}work;
struct {int x,y,x1,y1;}work_size;
struct {POINT p[192];}w;
struct {POINT pt;}result;

******************************************************************************
I Beginning of reconstruction of broadcasted message \\
\\******************************************************************************

while (ready!=1){
len = net_receive(&work,&ready);
i++;
for (j=0;j<64;j++){
w.p[j+(i*64)].x = work.p[j].x; /* depending on which block is */
w.p[j+(i*64)].y = work.p[j].y; /* received, + (i*64) to get */
w.p[j+(i*64)].class = work.p[j].class; /* correct array number */
}
}
Complete reconstruction of broadcasted message. Note that the POINT structure in spiral.h has extra int definition which is not used. It is used as a pad to round off the transmission so that nothing is truncated during transmission. Thanks to the lousy library presented in //C.

```
result.pt.x = w.p[190].x;
result.pt.y = w.p[190].y;
result.pt.class = w.p[190].class;
net_send(sizeof(result), &result, I);
```

Beginning of second section of processing in worker

```
k = -1;
for (;;) {
    len = net_receive(&work_size, &ready);
    for (i = work_size.x; i < work_size.x + 1; i++) {
        for (j = work_size.y; j < work_size.y + 1; j++) {
            k++;
            work.p[k].x = (float)((float)(7/52.0)*i);
            work.p[k].y = (float)((float)(7/52.0)*j);
        }
        work.p[k].class = w.p[min].class;
    }
    k = -1;
    net_send(sizeof(work), &work, I);
}
```

FLANN II nearest neighbour search with no constraints

```
flag = 0;
for (l = 0; l < 192; l++) {
    tx = (w.p[l].x - work.p[k].x);
    ty = (w.p[l].y - work.p[k].y);
    value = tx*tx + ty*ty;
    if (flag == 0) { /* first calculated node */
        min_value = value;
        min = l;
        flag = 1;
    } 
    if (min_value > value) {
        min_value = value;
        min = l;
    }
    work.p[k].class = w.p[min].class;
}
k = -1;
net_send(sizeof(work), &work, I);
```
Appendix D  Parallel Programs

/* *****************************************************/
/* Parallel waveform identification program based on */
/* Parallel Distributed Neural Networks concept (PDNN). */
/* *****************************************************/
#include <stdio.h>
#include <alt.h>
#include <chan.h>
#include <sema.h>
#include <par.h>
#include "FLANN.h"

/**********************************************************
This is the main control program of the 4 programs
Consists of Driver, S1, S2, S3. The S programs are
Slave programs. The driver is the master control
**********************************************************
main(argc,argv,envp,in_port,ins,out_port,outs)
int argc;
char *argv[], *envp[];
CHAN *in_port[], *out_port[];
int ins, outs;
{
/**********************************************************
Preparation of data for transmission
**********************************************************
FILE *infile;
int i, j, k;
int interclass[3];
int len, flag;
int node_number = 0;
int int_sum, int_win;
int sum;
int last_node, competition_flag;
INPUT sub1, sub2, sub3;
FINAL_VECTOR *create();
int MOD(), Discern();

if ((infile = fopen("data", "r")) == (FILE*) NULL){
    printf("Error opening data file\n");
    exit(1);
}

while (fscanf(infile, "%f", &sub1.wave[0]) != EOF){
    for (i=1;i<10;i++){
        fscanf(infile, "%f", &sub1.wave[i]);
    }
    for (i=0;i<10;i++){
        fscanf(infile, "%f", &sub2.wave[i]);
    }
}
for (i=0;i<10;i++)
{
    fscanf(infile,"%f ",&sub3.wave[i]);
}

/*****************************************
| Preparation of data completed |
 *****************************************/

/*****************************************
| Transmitting data to slaves through channels |
 *****************************************/

len = sizeof(VECTOR);

chan_out_word(len,out_port[2]);
chan_out_message(len,&sub1,out_port[2]);
chan_out_word(len,out_port[3]);
chan_out_message(len,&sub2,out_port[3]);
chan_out_word(len,out_port[4]);
chan_out_message(len,&sub3,out_port[4]);

/*****************************************
| Wait for processed data coming from slaves |
 *****************************************/

flag = 0;
while (flag < 3){
i=wait_vec(ins,in_port);
chan_in_word(&interclass[i-2],in_port[i]);
flag ++;
}

sema_wait(&par_sema);
printf("Got through\n");
printf("%d %d %d\n",interclass[0],interclass[1],
       interclass[2]);
sema_signal(&par_sema);

/*****************************************
| Second level FLANN to perform classification |
 of the first set of results obtained from |
 the first level of distributed FLANN |
 The second level FLANN may still employ the |
 estimation properties of ANNs by setting rou |
 and tolerance, but this example uses it as a |
 straight look-up table, because the classes |
 are fixed by the first level FLANN. The |
 selection process then uses the straight |
 nearest neighbour classification for results |
 *****************************************/

if (node_number == 0){

head = create();
for (i=0;i<3;i++){
    head->wave_class[i] = interclass[i];
}
head->final_class = node_number;
node_number++;
tail = head;
win_node = head;
}
else {
    node = head;
    last_node = 0;
    competition_flag = 0;
    while (last_node == 0){
        sum = 0;
        for (i=0;i<3;i++){
            sum+=Discern(m_tol-(MOD(node->wave_class[i]-interclass[i])));
        }
        if ((float)(sum/3.0) >= m_rou) {
            int_sum = 0;
            for (i=0;i<3;i++) {
                int_sum += (node->wave_class[i] - interclass[i]) *
                (node->wave_class[i] - interclass[i]);
            }
            if (competition_flag == 0) {
                int_win = int_sum;
                competition_flag = 1;
                win_node = node;
            }
            else if (int_win > int_sum) {
                win_node = node;
                int_win = int_sum;
            }
        }
    }
    if (node->next == NULL) last_node = 1;
    else {
        temp = node;
        node = temp->next;
    }
/*****************************************/
This section is after the contents of the
network are checked. If no patterns match
then competition_flag will still be 0.
*****************************************/

if (competition_flag == 0)
    new = createO;
    for (i=0;i<3;i++)
        new->wave_class[i] = interclass[i];
    new->final_class = node_number;
    node_number++;
    tail->next = new;
    win_node = new;
    tail = new;

/*****************************************/
If competition_flag is 1, then a match has
been found.
*****************************************/

sema_wait(&par_sema);
printf("final class is %d
",win_node->final_class);
sema_signal(&par_sema);

} /* Dynamic allocation function for
new output nodes. */

FINAL_VECTOR *createO
{
    FINAL_VECTOR *node;
    node = (VECTOR*)malloc(sizeof(VECTOR));
    node->next=NULL;
    return(node);
}

/*****************************************/
Discern function is just an ON/OFF
function that helps in counting
the number of correct vector
element matches within the vector
*****************************************/

int Discern(value)
float value;
{
    if (value < 0) return(0);
    return(1);
}
/**********************************************
* Modulus function returns positive float numbers *
***********************************************/

int MOD(value)
int value;
{
if (value < 0) return(-value);
return(value);
}

/************************************************************
* Program configuration file for multiple sinusoidal program  *
*************************************************************/

processor host
processor root
wire jumper root[0] host[0]
task driver2 ins=5 outs=5 data=50k
task s1 ins=2 outs=2 data=50k
task s2 ins=2 outs=2 data=50k
task s3 ins=2 outs=2 data=50k
task filter ins=2 outs=2 data=50k
task afserver ins=1 outs=1

place afserver host
place filter root
place driver2 root
place s1 root
place s2 root
place s3 root

connect? filter[0] afserver[0]
connect? afserver[0] filter[0]
/**************************************************************************/ 
| Header file FLANN.h required for preset values                        |
**************************************************************************/

typedef struct vector 
    { 
    float wave[10]; 
    int class; 
    struct vector *next; 
    } VECTOR;

typedef struct input 
    { 
    float wave[10]; 
    } INPUT;

typedef struct final_vector 
    { 
    int wave_class[3]; 
    int final_class; 
    struct final_vector *next; 
    } FINAL_VECTOR;

#define s_tol 0.05
#define s_rou 0.9
#define m_tol 0
#define m_rou 0.9
/**Beginning of distributed neural networks. This program is loaded as S1, S2 and S3. Information is passed from driver2.c to neural nets for distributed classifications.** /

#include <stdio.h>
#include <chan.h>
#include "FLANN.h"

main(argc,argv,envp,in_port,ins,out_port,outs)
int argc, ins, outs;
char *argv[], *envp[];
CHAN *in_port[], *out_port[];
{
    INPUT input;
    VECTOR *create();
    float MOD();
    int competition_flag;
    int Discern();
    int int_sum;
    int last_node;
    float float_sum;
    float float_win;
    int len, i;
    int node_number=0;
    /***************
    | Wait perpetually for incoming data |
    
    for(;;){
        chan_in_word(&len,in_port[1]);
        chan_in_message(len,&input,in_port[1]);
    }
    /***************
    | Beginning of FLANN II architecture ANN |
    
    if (node_number == 0){
        head = create();
        for (i=0;i<10;i++){
            head->wave[i] = input.wave[i];
        }
        head->class = node_number;
        node_number++;
    }
tail = head;
win_node = head;
node = head;
}
else {
    node = head;
lascnode = 0;
    competition_flag = 0;
    while (last_node == 0){
        int_sum = 0;
        for (i=0;i<10;i++){
            int_sum+=Discern(s tol-(MOD(node->wave[i])-input.wave[i]));
        }
        if ((float)(int_sum/10) >= s_rou){
            float_sum = 0.0;
            for(i=0;i<10;i++){
                float_sum += (node->wave[i] - input.wave[i]) *
                                (node->wave[i] - input.wave[i]);
            }
            if (competition_flag == 0){
                float_win = float_sum;
                competition_flag = 1;
                win_node = node;    }
            else if (float_win > float_sum){
                win_node = node;
                float_win = float_sum;
            }
        }
    }
    if (node->next == NULL)lascnode = 1;
    else {
        temp = node;
        node = temp->next;
    }
}
if (competition_flag == 0){
    new = createO;
    for (i=0;i<10;i++){
new->wave[i] = input.wave[i];
  
new->class = node_number;
node_number++;
tail->next = new;
node = new;
tail = new;

/*******************************************************************************
  If competition_flag is 1, then a match has been found.
*******************************************************************************/
else {
  node = win_node;
}

/***********************************************************************
  Send data out of out port to master
***********************************************************************
chan_out_word(node->class,out_port[1]);
}

/*******************************************************************************
  Dynamic allocation function for new output nodes.
*******************************************************************************
VECTOR *create()
{
  VECTOR *node;
  node = (VECTOR*)malloc(sizeof(VECTOR));
  node->next=NULL;
  return(node);
}

/*******************************************************************************
  Discern function is just an ON/OFF function that helps in counting the number of correct vector element matches within the vector
*******************************************************************************
int Discern(value)
{
  float value;
  
  if (value < 0) return(0);
  return(1);
}