1994

On-line fast learning with variable thresholds prototype neural classifier.

Mahmoud A. Abu-Nasr

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
ON-LINE FAST LEARNING WITH VARIABLE THRESHOLDS PROTOTYPE NEURAL CLASSIFIER

by

Mahmoud A. Abu-Nasr

A Dissertation submitted to the Faculty of Graduate Studies and Research through the Department of Electrical Engineering in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy at the University of Windsor

Windsor, Ontario, Canada

1994
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-93251-1
ABSTRACT

Pattern classification is one of the most successful applications of neural networks. Most of the previous research focused around training the multi-layer perceptrons (MLP's) with the back propagation algorithm. Although the MLP's are capable of resolving pattern classes separated by non-linear class boundaries, they have some drawbacks that limit their acceptance for classifying problems of the real-world. Namely they require very lengthy training times, they cannot learn incrementally, and their convergence is not guaranteed.

In this thesis, we develop two prototype based classifiers that require training times that are orders of magnitude less than the MLP's, and can be trained in increments. One of the classifiers uses prototypes of fixed thresholds to represent all classes. The other classifier forms prototypes of different firing thresholds, which it chooses and adjusts according to an algorithm.

The classifiers were tested on standard and real world problems like hand-written numeral recognition, in which they demonstrated superior performance in terms of speed of learning and memory efficiency.
ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, Dr. M. A. Sidd-Ahmed for his guidance, constructive criticism, and constant encouragement throughout the progress of this thesis. I also would like to extend my thanks to Dr. J. J. Soltis, Prof. P. H. Alexander and Dr. H. Toews for their valuable advice as members of my Ph.D. committee.

Thanks are also due to my parents and to my brother without their encouragement, inspiration and love, this work would not have started.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER I</td>
<td>i</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Neural Network Paradigms:</td>
<td>3</td>
</tr>
<tr>
<td>1.1.1 The Perceptron:</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 The Multi-Layer Perceptron:</td>
<td>4</td>
</tr>
<tr>
<td>1.1.3 The Hopfield Network:</td>
<td>7</td>
</tr>
<tr>
<td>1.1.4 The ART Network</td>
<td>10</td>
</tr>
<tr>
<td>1.1.5 Kohonen's Self Organizing Feature Maps</td>
<td>13</td>
</tr>
<tr>
<td>1.2 Motivation of this Research</td>
<td>15</td>
</tr>
<tr>
<td>1.3 Thesis Organization</td>
<td>16</td>
</tr>
<tr>
<td>CHAPTER II</td>
<td>20</td>
</tr>
<tr>
<td>NEURAL NETWORK CLASSIFIERS</td>
<td>20</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>20</td>
</tr>
<tr>
<td>2.2 The Multi-Layer Perceptron Classifiers:</td>
<td>21</td>
</tr>
<tr>
<td>2.3 The LBAQ Neural Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>2.3.1 Software Simulation of LBAQ</td>
<td>28</td>
</tr>
<tr>
<td>2.3.2 Teaching the XOR Pattern to LBAQ</td>
<td>33</td>
</tr>
<tr>
<td>2.4 Comparison of the LBAQ and Multi-layer Perceptron Training Times</td>
<td>35</td>
</tr>
<tr>
<td>2.5 Discussion and Conclusions</td>
<td>36</td>
</tr>
<tr>
<td>CHAPTER III</td>
<td>43</td>
</tr>
<tr>
<td>A VARIABLE THRESHOLDS PROTOTYPE NEURAL CLASSIFIER</td>
<td>43</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>43</td>
</tr>
<tr>
<td>3.2 The NNC Architecture</td>
<td>45</td>
</tr>
<tr>
<td>3.3 NNC TRAINING ALGORITHM</td>
<td>49</td>
</tr>
<tr>
<td>3.4 Classification Examples</td>
<td>61</td>
</tr>
<tr>
<td>3.4.1 Four Concentric regions</td>
<td>62</td>
</tr>
<tr>
<td>3.4.2 Numeral Recognition:</td>
<td>67</td>
</tr>
<tr>
<td>3.5 Summary and Conclusions</td>
<td>73</td>
</tr>
<tr>
<td>CHAPTER IV</td>
<td>74</td>
</tr>
<tr>
<td>APPLICATION OF THE VARIABLE THRESHOLDS NEURAL CLASSIFIER TO REAL-WORLD DATA</td>
<td>74</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>74</td>
</tr>
<tr>
<td>4.2 Recognition of Hand-Written Numerals</td>
<td>74</td>
</tr>
</tbody>
</table>
4.2 Experiments on Hand-Written Numeral Recognition using NNC . 76

CHAPTER V ................................................................. 83
SUMMARY AND CONCLUSIONS ........................................... 84

APPENDIX ................................................................. 86
THE NNC SIMULATION PROGRAM ..................................... 86

REFERENCES ............................................................... 110
LIST OF TABLES

Table

2.1 Parity-2 Data. ......................................................... 37
2.2 Parity-3 Data ......................................................... 39
2.3 Parity-4 Data ......................................................... 40
3.1 Three Disjoint Class Data. ......................................... 61
3.2 The NNC Numeral Identification Results. ....................... 72
4.1 Available Samples of Hand-Written Numerals. ................... 77
4.2 Hand-Written Numeral Recognition Test Results. ............... 78
# LIST OF FIGURES

### Figure

1.1 The Perceptron                              .................................................. 5  
1.2 The Perceptron Learning Algorithm.           .................................................. 6  
1.3 The Sigmoidal Activation Function.           .................................................. 8  
1.4 The Back-Propagation Algorithm.             .................................................. 9  
1.5 The Hopfield Network.                       .................................................. 11  
1.6 The ART Network Architecture.               .................................................. 12  
1.7 The ART Algorithm.                         .................................................. 14  
1.8 Kohonen's Feature Map Neural Network.       .................................................. 18  
1.9 Kohonen's Algorithm.                       .................................................. 19  
2.1 The Multi-Layer Perceptron                  .................................................. 22  
2.2 The LBAQ Architecture.                     .................................................. 29  
2.3 The LBAQ Algorithm                          .................................................. 30  
2.4 Comparison of the Training Times Of the LBAQ and the Multi-Layer Perceptron on Parity-2, Parity-3, Parity-4 and Parity-5 Problems. .................................................. 41  
2.5 Comparison of the Training Time of the LBAQ and the Multi-layer Perceptron on Parity-6 Problem. .................................................. 42  
3.1 The NNC Architecture.                      .................................................. 46  
3.2 The NNC Training Algorithm.                .................................................. 58  
3.3 Prototype Formation with NNC.              .................................................. 60  
3.4 Bullseye Pattern.                          .................................................. 63  
3.5 Scatter Plots of Data Classified By NNC AS Regions 1 and Region 2. .................................................. 64  
3.6 Scatter Plots of Test Data Classified By NNC as Regions 3 and 4. .................................................. 65  
3.7 Distribution of the NNC Firing Thresholds After Learning the Bullseye Pattern. .................................................. 66  
3.8 True Type Numeral Training Set.             .................................................. 68  
3.9 Digitized Zero and Two in Bangkok Font.    .................................................. 70  
3.10 Distribution of NNC Firing Thresholds after Learning the True Type Numeral Fonts. .................................................. 71  
4.1 Samples of Hand-written Numeral Zero.       .................................................. 80  
4.2 Samples of Hand-Written Numeral Two.        .................................................. 81  
4.3 Samples of Hand-Written Numeral Two.        .................................................. 82  
4.4 A 64 x 80 pixel representation of Numeral Zero. .................................................. 83  

CHAPTER 1

INTRODUCTION

Ideas leading to the development of computational neural modeling as a field of scientific research can be traced back to the 1940's. In 1943 McCulloch and Pitts [1] proposed a simple model of a neuron as a binary threshold unit. This model neuron computes a weighted sum of its inputs from other units and outputs a one or zero according to whether this sum is above or below a certain threshold. For almost fifteen years after that, pioneer researchers established the basic models of the neural networks experimentally [2], [3], [4], [5] and theoretically [6], [7], [8], [9], [10]. Around 1960, Rosenblatt and his group were actively researching some networks they called perceptrons. These networks consisted of units organized into layers with feed-forward connections between one layer and the next. At the same time, Widrow and Hoff were investigating very similar networks they called adalines. Rosenblatt has shown that perceptrons can be trained to classify patterns using a training set of input-output pairs, and an iterative algorithm for changing the perceptron weights.

In 1969, Minsky and Papert published the book Perceptrons [11], in which they pointed out that the perceptrons cannot classify non-linearly separable patterns. They used as an example the EXCLUSIVE OR (XOR) problem. After the publication of the book Perceptron, the funding agencies like DARPA cut off the funds for neural network research, and funneled the funds towards what was thought to be more useful approaches of artificial
intelligence. As a result, research in the area of neural networks has diminished with only a small number of researchers remaining in this field of research [12], [13], [14], [15]. This situation lasted for about sixteen years. A major research theme during this period was the associative content addressable memory [16], [17], [18], [19], [20], [21], [22]. Grossberg [23] reformulated the general problem of learning in neural networks. Marr [19], [20], developed network theories of the cerebellum, cerebral neocortex, and hippocampus assigning specific functions to each type of neuron. The visual system was studied by a number of people including Marr [24] and von der Malsburg [25]. In 1982, Hopfield [26] introduced the use of the energy function to the neural networks research and proposed the notion of memories as dynamically stable attractors. Revival of interest in neural network research was due to a great extent, to the development of the multi-Layer learning algorithm, known as the back-propagation algorithm. The algorithm enabled multi-layer feed-forward networks to handle the kind of problems shown to be unsolvable by a single layer perceptron. Credits to the development of this algorithm go to several people: Werbos [27], Rumelhart, Hinton and Williams [28], but certainly the credit of publicizing the back-propagation algorithm goes to David Rumelhart and JC McClelland with the publication of the "PDP Books" (Parallel Distributed Processing, Volumes I and II). Since then the interest in neural research has exploded. In 1987, the first open conference on neural networks, the IEEE International Conference on Neural Networks was held in San Diego and the International Neural Network Society (INNS) was formed. In 1988 the INNS journal: Neural Networks was founded, followed by Neural Computation in 1989 and the IEEE Transactions on Neural Networks in 1990.
1.1 Neural Network Paradigms:

In this section we will examine some of the neural networks that are considered landmarks in neural research. They also represent the variety of learning techniques, learning algorithms, neural architectures, and various applications. In general neural networks learn in one of two ways: supervised or unsupervised. In supervised learning the network is provided with examples of the correct input/output association, and in unsupervised learning the network is only presented with input patterns and is asked to group them according to some criteria. The architectures of neural networks vary between single layer networks and multi-layer networks. A layer consists of one or several processing nodes. The pattern of connection between the nodes in different layers, or the nodes in the same layer, and the node activation functions vary between the different neural networks.

1.1.1 The Perceptron:

Rosenblatt[8] used the name perceptron to describe a type of networks in which a set of N inputs \([p_1, p_2, \ldots, p_N]\) are connected to an output layer of nodes as in Fig. 1.1. The input/output relation can be described by:
\[ y_i = \text{sgn}(\sum_{k=1}^{N} w_{ik} p_k - \theta_i) \]

Where
\text{sgn(}) is the activation function of the output nodes.
\text{sgn(}x) = 1 \quad x \geq 0;
\text{sgn(}x) = 0 \quad x < 0; \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
Fig 1.1 The Perceptron
Perceptron Weight Updating Rule

\[ w_{ik}^{new} = w_{ik}^{old} + \Delta w_{ik} \]

\[ \Delta w_{ik} = \begin{cases} 2 \eta y_{\text{desired}} p_k & \text{if } y_{\text{desired}} \neq y_{\text{actual}}; \\ 0 & \text{otherwise}; \end{cases} \]

Fig 1.2 The Perceptron Learning Algorithm.
The output of any node \( j \) in layer \( C \) assuming that the node has a sigmoidal activation function:

\[
o_j = \text{sgm}(h_j)
\]  \hspace{1cm} (1.3)

Where

\[
\text{sgm}(x) = \frac{1}{1 + e^{-\left(\frac{x - \theta}{\theta_o}\right)}}
\]  \hspace{1cm} (1.4)

The effects of \( \theta \) and \( \theta_o \) are illustrated in Fig[1.3].

The outputs of the nodes in Layer \( C \) will be different in general from the desired outputs. By using the back propagation algorithm, the weight values are adjusted iteratively to minimize the mean square error between the actual outputs of the network and the desired outputs. The back propagation algorithm is shown in Fig. 1.4.

1.1.3 The Hopfield Network:

The Hopfield network is used as content addressable memory. When partial inputs or inputs corrupted by noise are presented to the network, it produces at its output the stored pattern that most closely resembles the input pattern. The architecture of this net is shown in Fig. 1.5. It has \( n \) nodes containing hard limiting non-linearities. The inputs and outputs
$$y = \frac{1}{1 + e^{-(x + \theta) / \phi}}$$

\[\theta = 10, \quad \phi = 10\]

\[\theta = 10, \quad \phi = 100\]

\[\theta = 10, \quad \phi = 200\]

Fig 1.3 The Sigmoidal Activation Function.
Fig 1.4 The Back-Propagation Algorithm.
have the binary values [1, -1]. The output of each node is fed back to the rest of the nodes
modified by weight factors. Hopfield [29] has proven that patterns can be stored in the
network by setting the N x N weight matrix as follows:

\[
W = \begin{bmatrix}
0 & w_{12} & \ldots & w_{1,N-1} \\
w_{21} & 0 & \ldots & w_{2,N-1} \\
& & \cdots & \cdots \\
w_{N-11} & w_{N-12} & \ldots & 0
\end{bmatrix}
\]  

(1.5)

Where

\[
w_{ij} = \sum_{s=1}^{M} x_i^s x_j^s \quad i \neq j
\]

= 0 \quad i = j \quad 0 \leq i, j \leq N-1

M is the number of patterns.
\(x_i^s\) is component i of the exemplar for class s.

The number of patterns that can be stored and accurately recalled by a Hopfield net is very
limited. If too many patterns are stored, the network may converge to a pattern different
from all the stored exemplar patterns. Hopfield has shown that this will happen infrequently
if the number of stored patterns M is less than 0.15 x N.

1.1.4 The ART Network:

ART stands for adaptive resonance theory developed by Carpenter and Grossberg [30],
[31]. The input to the network and a stored prototype are said to resonate when they are
sufficiently similar. There are two types of ART networks: ART1 deals with binary inputs
INPUTS (Applied At Time Zero)

Fig 1.5 The Hopfield Network.
Fig 1.6 The ART Network Architecture.
and ART2 deals with analog inputs. Carpenter and Grossberg used previously developed building blocks that were based on biologically reasonable assumptions. The architecture of the ART network is shown in Fig. 1.6. It has two successive stages, $F_1$ and $F_2$ which encode the patterns in the short term memory (STM). Bottom-up and top-down pathways between $F_1$ and $F_2$ contain adaptive long term memory traces which multiply the signals in these pathways. The ART network basically implements a clustering algorithm that is very similar to the leader algorithm [32] [33]. The first input to the network is selected as the first cluster prototype. The next input is compared to the stored prototype. It would be clustered with the first prototype if it is sufficiently similar to it; otherwise it is used by the network to start a new cluster prototype. The process is repeated for all the rest of the input patterns. The number of clusters will grow with time and will depend on the similarity measure and the definition of sufficiently similar. Carpenter and Grossberg have defined a network parameter they called vigilance $v$, with $v$ greater than 0, and less than or equal to 1. If $v$ is large the similarity condition becomes very stringent, so many prototypes are formed. A small $v$ on the other hand will reduce the number of prototypes. The ART algorithm is shown in Fig. 1.7.

1.1.5 Kohonen’s Self Organizing Feature Maps:

This network uses an unsupervised learning algorithm to arrange the output units geometrically in a line or a plane preserving the neighborhood relations from the input space into the output space. Thus geometrically nearby output nodes will correspond to similar input patterns. The network architecture is shown in Fig. 1.8. Inputs to the network are $N$ dimensional continuous patterns. Each input node is connected to all the output nodes
Fig 1.7 The ART Algorithm.
via weights $w$. As each input is presented to the network, a competitive learning rule is used to choose the output node that is closest to the input. Then the weights from this node to the inputs are updated according to the algorithm of Fig. 1.9. The weights of the neighbor nodes are also updated. A typical choice of the neighborhood function would be:

$$\Lambda(i_1, i_2) = \exp\left(-\frac{|D_{i_1i_2}|^2}{2\sigma^2}\right) \quad (1.6)$$

where

$D_{i_1i_2}$ is the Euclidean distance between node $i_1$ and node $i_2$ in the output array, $\sigma$ is a width parameter which is gradually decreased with time.

1.2 Motivation of this Research:

In spite of the voluminous neural network research, acceptance of neural networks in the real-world has been slow. Neural networks have been most successfully applied as classifiers in pattern recognition problems. They were also applied in function estimation, data compression, feature extraction, and clustering. In this dissertation we focus on neural classification. Most of the previous research centered around the back-propagation algorithm for training multi-layer perceptron classifiers [28]. The research showed that these classifiers are capable of resolving pattern classes separated by non-linear class boundaries, after they have been trained with examples of associations of the input and the output. Research also showed that the multi-layer perceptron classifiers trained by back propagation have the following drawbacks:

1. They require very lengthy training sessions.
2. They cannot be trained incrementally. Training the classifier on a new class
involves retraining the network with a data set containing the new class and the previously learned classes.

3. There are no techniques for determining the number of layers, or the number of nodes per layer that are needed for a particular classification problem.

4. The convergence of the back propagation algorithm is not guaranteed. After a long training session, one may find that the algorithm has found a local minimum of the error function.

In this research we were looking for neural classifiers that overcome the drawbacks of the multi-layer perceptron and the back-propagation algorithm. We investigated the prototype based neural classifiers. We proposed two algorithms for training the prototype based classifiers that require training times that are orders of magnitude less than multi-layer perceptrons trained by back-propagation. They also can be used in incremental training sessions. One of the algorithms uses prototypes of fixed thresholds, and the other uses prototypes of variable thresholds. The variable thresholds algorithm is also efficient in its prototype memory usage.

1.3 Thesis Organization:

The thesis is organized into five chapters. The first chapter is the introduction, which presents some of the landmark research in neural networks, as well as the motivation of this research.

Chapter two describes the back-propagation neural classification and introduces the prototype based neural algorithm: the LBAQ. The chapter ends by contrasting the
performance of the LBAQ neural network with the multi-layer perceptron trained by the back-propagation algorithm.

Chapter three introduces the variable thresholds prototype neural classifier as a classifier that exhibits fast learning and is conservative in its memory usage. The learning algorithm is presented. The formation of prototypes is illustrated through a simple pattern classification example. The use of this classifier in continuous and in binary pattern classification is demonstrated with two examples. The first one teaches the classifier four concentric regions of a bullseye pattern, and the second one teaches the classifier the numerals zero to nine in seventeen different true type fonts.

Chapter four demonstrates the use of the variable thresholds neural classifier on real-world data: a database of hand-written numerals maintained by the United States Postal Service.

Chapter five is a conclusion of the thesis and a summary of its results.
Fig 1.8 Kohonen’s Feature Map Neural Network.
Fig 1.9 Kohonen's Algorithm.
CHAPTER II

NEURAL NETWORK CLASSIFIERS

2.1 Introduction:

Pattern classification with multi-layer perceptrons (MLP's) trained by the back-propagation algorithm has been extensively studied in the past few years [28], [32]. The studies show that the MLP's are capable of separating classes with non-linear boundaries, but require excessively long training sessions with a comprehensive training set representing all the pattern classes. This limits the application of these classifiers in real-world problems where it is not unusual to encounter new pattern classes after training the classifier with a particular training data set. Training the MLP with the back-propagation algorithm can continue for several days, with either no convergence or convergence on a local minimum of the error surface. McInerney, Haines, Biafore and Hecht-Nielson reported that after a run of twelve hours on a CRAY-2 super computer, their network converged on a local minimum of the error surface. Prototype based classifiers on the other hand have many features that make them attractive candidates for problems like optical character recognition, speech recognition, analysis of seismic data, inspection of manufactured parts and quality control. They can resolve pattern classes separated by non-linear class boundaries with much less training time compared to the MLP's. They are also non-parametric, i.e. they do not assume any probability distribution in the training data set.

The organization of this chapter is as follows. In section two we present the back-
propagation algorithm for training multi-layer perceptron classifiers. In section three we present the LBAQ prototype based neural classification algorithm. In section four we compare the training times of the LBAQ and the multi-layer perceptron in standard classification problems. Conclusions are presented in section five.

2.2 The Multi-Layer Perceptron Classifiers:

The system architecture of such a classifier is shown in Fig 2.1. It is made up of nodes arranged in layers. The output of each node in any one layer is transmitted to the nodes in the following layer modified by weight factors which can be either positive or negative. The input to each node in any layer of the network except the input layer is the sum of the weighted outputs of the nodes in the previous layer. The input to a node \( j \) in layer \( L_n \) is:

\[
    \text{input}_j = \sum_{i=0}^{n} w_{ji} \cdot o_i
\]  

(2.1)

where \( w_{ji} \) is the weight between node \( j \) in layer \( L_n \) and node \( i \) in layer \( L_A \). \( o_i \) is the output of node \( i \). \( n \) is the number of nodes in layer \( L_A \). The output of node \( j \) is:

\[
    o_j = f(\text{input}_j)
\]  

(2.2)

where \( f \) is the activation function.

For a sigmoidal activation function, we have:
Figure 2.1 The Multi-Layer Perceptron
\[ o_j = \frac{1}{1 + e^{-\text{input}_j \cdot o_j}} \] (2.3)

Similarly the input for node \( k \) in Layer \( L_C \) is:

\[ \text{input}_k = \sum_{j=1}^{N} w_{kj} o_j \] (2.4)

where \( N \) is the number of nodes in layer \( L_B \). The output of node \( k \) is:

\[ o_k = f(\text{input}_k) \] (2.5)

During the training of this network, it is presented with an ordered pair \( \{p, T_p\} \) as an input, where \( p \) is an input pattern, and \( T_p \) is the desired or target output. \( T_p \) is a vector defined as follows:

\[ T_p = \{ t_{p1}, t_{p2}, \ldots, t_{pm} \} \] (2.6)

The network adjusts its weights such that the desired outputs are obtained at the output nodes. Then the network is presented with another ordered pair. The objective of training is to find a single set of weights that satisfies all the ordered pairs presented to it. For pattern \( p \) we define the error \( E_p \) as the sum of the square of the difference between the actual output and the target output:
\[ E_p = \sum_{k=1}^{m} (t_{pk} - o_{pk})^2 \] (2.7)

In a gradient descent search, we should obtain an expression for the error over all the ordered pairs, and then we find the set of weights that will minimize this error. The back-propagation algorithm adjusts the weights sequentially minimizing the error in equation (2.7) for each input pattern. For convenience we will omit the p subscript from the error expression, and we will rewrite equation (2.7) as:

\[ E = \frac{1}{2} \sum_{k=1}^{m} (t_k - o_k)^2 \] (2.8)

The factor of one-half was introduced to simplify the results later on. The error \( E \) in equation (2.8) can be minimized by adjusting the weights connected to the output nodes in layer \( L_C \) by increments as in the following equation:

\[ \Delta w_{kj} = -\eta \frac{\partial E}{\partial w_{kj}} \] (2.9)

where \( \eta \) is a proportionality constant.

We can evaluate the partial derivative in equation (2.9) using the chain rule:

\[ \frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial \text{input}_k} \frac{\partial \text{input}_k}{\partial w_{kj}} \] (2.10)

Using the expression for \( \text{input}_k \) from equation (2.4) we obtain:
\[ \frac{\delta_{\text{input}_k}}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}} \sum_{j} w_{kj} o_j = o_j \]  \hspace{1cm} (2.11)

By defining

\[ \delta_k = -\frac{\partial E}{\partial \text{input}_k} \]  \hspace{1cm} (2.12)

we can rewrite equation (2.9) as:

\[ \Delta w_{kj} = \eta \delta_k o_j \]  \hspace{1cm} (2.13)

To compute the partial derivative in equation (2.12), we again use the chain rule:

\[ \delta_k = -\frac{\partial E}{\partial \text{input}_k} = -\frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial \text{input}_k} \]  \hspace{1cm} (2.14)

\[ = (t_k - o_k) o_k (1 - o_k) \]

Substituting equation (2.14) in equation (2.13) we have:

\[ \Delta w_{kj} = \eta (t_k - o_k) o_k (1 - o_k) o_j \]  \hspace{1cm} (2.15)

for any node k in output layer L_C. Weights not connected to L_C are modified as follows:

\[ \Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} \]

\[ = -\eta \frac{\partial E}{\partial \text{input}_j} \frac{\partial \text{input}_j}{\partial w_{ji}} \]

\[ = -\eta \frac{\partial E}{\partial \text{input}_j} \frac{\partial o_j}{\partial \text{input}_j} \]

\[ = -\eta (\frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \text{input}_j}) o_i \]  \hspace{1cm} (2.16)
Using the multi-dimensional chain rule:

$$\frac{\partial E}{\partial o_j} = \sum_{i=1}^{m} \frac{\partial E}{\partial \text{input}_k} \frac{\partial \text{input}_k}{\partial o_j} \quad (2.17)$$

Substituting Equation (2.4) into equation (2.17) we have:

$$\frac{\partial E}{\partial o_j} = \sum_{i=1}^{m} \left( \frac{\partial E}{\partial \text{input}_k} \right) \frac{\partial}{\partial o_j} \sum_{z=1}^{N} w_{kj} o_z$$

$$= \sum_{i=1}^{m} \left( \frac{\partial E}{\partial \text{input}_k} \right) w_{ki} \quad (2.18)$$

Substituting equation (2.12) into equation (2.18) we have:

$$\frac{\partial E}{\partial o_j} = -\sum_{k=1}^{m} \delta_k w_{kj} \quad (2.19)$$

Substituting equation (2.19) into equation (2.16) we obtain the following expression for changes that should be made to the weights not connected directly to the output in order to minimize the error $E$:

$$\Delta w_{ji} = \eta o_j (1 - o_j) \sum_{k=1}^{m} \delta_k w_{kj} \quad (2.20)$$
2.3 The LBAQ Neural Algorithm:

LBAQ is an acronym for Learning By Asking Questions, stressing the interactive nature of an algorithm for constructing neural classifiers. Teaching LBAQ a new feature is orders of magnitude faster than teaching a multi-layer network the same feature, with the backpropagation algorithm. As shown in Fig. 2.2, LBAQ is conceptually composed of two successive stages. The first stage determines whether the input pattern is new, i.e. was not previously learned, or old, i.e. previously learned by LBAQ. It then relays this information to the second stage which performs one of two actions. Either it produces a label at its output indicating to the teacher that a pattern exists at its input and that it already knows its label, or it produces a question at its output indicating to the teacher that an unknown pattern exists at its input. It then prompts the teacher if he wants to label the unknown pattern. LBAQ will keep asking the teacher about a label to this pattern each time it encounters it, if it had not previously obtained a label for it.

The first stage determines whether an input pattern is new or old by performing the algorithm shown in Fig. 2.3. It stores any new input pattern as a center of a cluster, represented by neural weights from the input to the center of the cluster. Any input pattern which falls within a hypersphere in the Euclidean space of radius R, and centered around an existing neural weight, i.e. a previously stored pattern, is considered similar to this previously stored pattern, and will not be separately stored. Instead the center of the hypersphere will be slightly modified, and the new pattern will be considered a member of a group, centered around the modified center. A pattern which does not fall within any existing hypersphere, will be a center of a new cluster. Effectively, this stage could be
considered as an intelligent memory that analyzes the input patterns and groups them according to a similarity criterion. The Second Stage acts as an interface to the teacher. It informs the teacher about events inside stage one, like the existence of a pattern at LBAQ's input, whether LBAQ knows the Label of this pattern, or whether it does not. It also solicits the teacher answers on how he/she wants to label a particular pattern. Basically, the algorithm of stage one causes activity in one and only one cluster with each exposure to an input pattern. By identifying this cluster, the second stage can provide all the stored attributes of this pattern to the teacher. Some of these attributes are collected by LBAQ without the assistance of the teacher, like the number of members of a particular cluster. Other attributes require the input from the teacher, like group labels. A missing label for a particular group will trigger the following question to the teacher:

How should I label this group? The teacher can provide a label, or he can decline and answer at a subsequent presentation of this pattern.

2.3.1 Software Simulation of LBAQ:

LBAQ was implemented in software using C on an 80386 IBM PC. The following basic steps are executed by the program:

1. Initialize the network:

In the case of software simulation, this means checking the existence of an input parameter file representing previous learning sessions. The input parameter file will contain the following parameters:

a. Number of prototypes (cluster nodes) already learned by LBAQ.
Figure 2.2 The LBAQ Architecture
Figure 2.3 The LBAQ Algorithm
b. Radius of cluster nodes.
c. Number of features in each pattern.
d. The prototypes weights, or the coordinates of the cluster centers.
e. A label for each prototype, if it was previously labeled.
f. Number of members in each cluster node. If the input file does not exist, LBAQ will prompt for these parameters, and they will be provided interactively.

2. Get input patterns and cluster them:

If the input pattern was the first to be presented to LBAQ, then it will basically create the first cluster node centered around the coordinates of the input pattern. If the input pattern was not the first to be presented to the LBAQ, then it will check whether the input pattern is similar to any of the already formed prototypes. This is done using a similarity measure, like the Euclidean distance for continuous patterns or the Hamming distance for binary patterns. If the distance between the input pattern and any of the prototypes is less than the radius of the cluster, then:

a. Include this pattern in this cluster and modify the coordinates of its prototype center according to the following formula:

\[
\text{wt}_{\text{new}}[\text{cluster-no}][i] = \frac{(\text{wt}_{\text{old}}[\text{cluster-no}][i] \times n + \text{pattern}[i])}{m} \tag{2.21}
\]

where \( n \) is the number of cluster members, and
\[ m = n + 1 \quad (2.22) \]

b. Increment the number of cluster members. If the new pattern is not similar to any of the previously stored prototypes, then create a new cluster node centered around this new pattern.

3. Label the input pattern:

At this stage, the internal data structure representing the cluster node is checked to find out whether it has a label for this cluster (prototype). If it has a label, then LBAQ will output this label. If it does not have a label then it will ask the teacher if he wants to provide a label at this time, and if the answer is YES, LBAQ will take the input label and assign it to the label field in the data structure representing the node. If the teacher did not provide a label, then the label field in the data structure representing the node will remain empty.

During future references to this particular cluster node, LBAQ will always ask the teacher about a label for this node, indicating that it recognizes the input pattern, but it does not know its label. If LBAQ provides a wrong label to the pattern, all the internal representations are destroyed and a smaller cluster radius is chosen by the LBAQ. The LBAQ has to be retrained with all the members of the training set.

4. Repeat step number two and three until there are no more patterns.

5. Save the neural network:

Before exiting, save the state of the neural network, for future usage. The state of the neural network is completely defined by the following items:
a. Number of internal prototypes (clusters).
b. The radius of the cluster node.
c. Number of pattern attributes (features) that LBAQ was dealing with.
d. The weights from every cluster node to the input.
e. The label and number of members in each cluster node.

2.3.2 Teaching the XOR Pattern to LBAQ:

In this example we will teach LBAQ the XOR pattern. Several cases might arise, depending on whether this network had encountered any of the XOR patterns before, and whether they were given any labels. We will take them case by case.

A. LBAQ has no Internal Representations:

Here we have an LBAQ which was not presented with any patterns before. We will present one of the XOR patterns, say the (1,0) pattern to it. LBAQ will check its internal representation space, and will find out that it does not have any internal representations yet. So it will form the first internal representation, in the form of a Hyper-Sphere with (1,0) as the center, and a predetermined radius (that is the only innate knowledge that LBAQ has). Now since this is the first internal representation element, LBAQ is sure that it does not have a label for it, so it will ask the teacher about the name of this pattern. If the teacher provides a name, LBAQ will associate this name with the internal representation. Otherwise the internal representation will remain unlabeled. In this case let us assume that the teacher will give it the label of ONE.
B. **LBAQ has at Least One Labeled Internal Representation:**

B.1 A new pattern is presented to LBAQ:

Let us assume that LBAQ has an internal representation from a previous session, say for (1,0). The teacher now takes another pattern, say the (0,0) pattern and presents it to LBAQ. LBAQ will search its internal representation space for a representation similar to (0,0). In this case, the only internal representation it has is (1,0). Therefore LBAQ will form a new internal representation for the (0,0), and will ask the teacher if he or she wants to provide a label. If a label is provided, it will be associated with the internal representation of the (0,0) pattern, otherwise its internal representation will remain unlabeled.

B.2 One of the previously encountered patterns is presented to LBAQ:

Again, let us assume that LBAQ has an internal representation for the (1,0) pattern from a previous session, and is labeled as ONE. If (1,0) is presented again to LBAQ, it will first search its internal representation space, and will find that it already has an internal representation for this pattern. It will also find out that this internal representation has the label ONE. So it will output the label of this pattern indicating that it recognized a pattern that it has learnt previously.

C. **LBAQ Has at Least One Unlabeled Internal Representation:**

Let us assume that the LBAQ has internal representations for (1,0), (0,0), (0,1). The (1,0) has the label ONE, (0,0) has the label (ZERO), and (0,1) is unlabeled.
C.1 New pattern is presented to LBAQ:

If we present the (1,1) pattern to LBAQ, it will search its internal representation space and it will find out that there are no internal representations similar to the new pattern. Therefore it will create a new internal representation centered around the new pattern (1,1) and it will ask the teacher about its label. As in the previous cases, if the teacher provides a label, it will be associated with the internal representation, otherwise, it will remain unlabeled.

C.2 A previously encountered pattern is presented to LBAQ:

If we present one of the previously labeled patterns again to LBAQ, it will behave exactly as in case B.2 above. But if we present the pattern (0,1) to LBAQ, which according to our assumption has an unlabeled internal representation, it will behave differently. After it searches its internal representation space, it will find out that it already has a similar representation for this pattern although unlabeled. So it will ask the teacher if he or she would like to give it a label at this time. If a label is provided, it will be associated with the internal representation that was already there, otherwise it will remain unlabeled.

2.4 Comparison of the LBAQ and Multi-layer Perceptron Training Times:

To compare the training times of the LBAQ to the multi-layer perceptron, we conducted the following training sessions to teach the neural networks the parity problem. In the general parity-N problem, there are $2^N$ input patterns and one output. The pattern attributes are binary [0,1]. The output is one when the input pattern contains an even
number of 1's and is zero otherwise. We used a software simulation of a back propagation network with three layers, and a software simulation of the LBAQ network. The software ran on a PC-486 25 MHz computer. In Table 2.1 and in Table 2.2 we show the data for parity-2 and parity-3 problems respectively. The results of the training time comparisons for parity-2, parity-3, parity-4, parity-5, and parity-6 are shown below.

2.5 Discussion and Conclusions:

In this chapter, we introduced the LBAQ neural classifier. The LBAQ neural classifier learns by creating internal representation prototypes of the input patterns. One prototype may represent as many patterns that are similar. The LBAQ uses the Euclidean distance for continuous patterns or the Hamming distance for binary patterns as a measure of similarity between an input pattern and a prototype, the closer they are in the feature space the similar they are considered by LBAQ. This technique is contrasted to the traditional multi-layer perceptron classifiers trained by back-propagation. The speed of LBAQ classifier was demonstrated by using the parity problem. Both the LBAQ and the multi-layer perceptron were simulated by software on a PC-486 25 MHz computer. They were trained to classify the individual patterns of the respective parity problems. The LBAQ learned to classify these patterns orders of magnitude faster than the multi-layer perceptron.

The LBAQ creates internal representation prototypes of similar fixed thresholds. Once it makes a classification mistake, the network destroys all its internal representations, chooses a smaller radius for its prototypes and it has to be retrained on all the patterns of
Table 2.1 Parity-2 Data.

<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>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
the training set. This is not necessarily bad, because the LBAQ learns very fast and also it learns incrementally, i.e. it can be trained in several sessions with subsets of the original training set. Nonetheless, to overcome this drawback of the LBAQ, another classifier with variable thresholds for its prototypes was developed and is presented in the next chapter.
Table 2.2 Parity-3 Data

<table>
<thead>
<tr>
<th>INPUT 1</th>
<th>INPUT 2</th>
<th>INPUT 3</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Table 2.3 Parity-4 Data

<table>
<thead>
<tr>
<th>INPUT 1</th>
<th>INPUT 2</th>
<th>INPUT 3</th>
<th>INPUT 4</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Figure 2.4  Comparison of the Training Times Of the LBAQ and the Multi-Layer Perceptron on Parity-2, Parity-3, Parity-4 and Parity-5 Problems.
Figure 2.5  Comparison of the Training Time of the LBAQ and the Multi-layer Perceptron on Parity-6 Problem.
CHAPTER III

A VARIABLE THRESHOLDS PROTOTYPE NEURAL CLASSIFIER

3.1 Introduction:

Pattern classification plays an important role in many engineering applications, including recognition of speech and characters, analysis of seismic, sonar and radar data, and automatic inspection systems in manufacturing [34]-[43]. Because of the wide applicability of classifiers, and with the resurgence of interest in neural networks, neural techniques are being researched for implementing classifiers that reduce classification error rates, training and classification time, and are adaptive and memory efficient. In the previous chapter we presented the LBAQ neural classifier, and we demonstrated how fast it is in learning pattern-class associations compared to the multi-layer perceptron trained by the back-propagation algorithm. We also saw that the LBAQ classifier has some drawbacks related to its use of fixed size hyperspheres or fixed firing threshold conditions for its prototypes. This chapter is about a neural network classifier (NNC) that learns different classes by setting neurons as prototypes of the different pattern classes, adjusting internal connection
weights and firing thresholds of its neuron prototypes. Studies on prototype based
classifiers show that despite their learning speed, they have the following drawbacks [41],
[44], [47]:

- There is no technique for choosing one of the important parameters of the
network, that is the initial firing threshold conditions of its neurons. Usually
it is a matter of experimenting with different ones, and choosing the threshold
that yields the best results.

- Their memory requirements could be excessive if narrow threshold conditions
are chosen for the firing of its neurons.

In this chapter we present a technique for designing prototype based classifiers, that
has the following advantages:

- It guides the choice of the firing threshold condition of every prototype neuron
as it is created.

- It is efficient in its usage of memory for prototypes.

- It allows the classifier to learn new classes, and modify old ones without
destroying old class information.

- It needs shorter training time compared to the other techniques where
complete retraining with old and new information is necessary to add new
classes. In this NNC only new information is used for retraining.

- It can resolve nonlinear class boundaries.

- It does not assume a priori knowledge about the underlying probability density
functions of each class.

The structure of this chapter is as follows. Section two describes the NNC architecture. Section three describes NNC training algorithm, and geometrically illustrates the changes in NNC as it learns three disjoint classes. Section four provides examples of the application of NNC in continuous and binary classification problems. Section five is a summary and conclusion for this chapter.

3.2 The NNC Architecture:

The neural network that implements the NNC is shown in Fig 3.1. The input layer $L_A = (a_1, a_2, a_3, \ldots a_n)$ consists of $n$ neurons, one for each component of the input pattern $P$. Each input neuron is connected to all the $N$ prototype neurons in layer $L_B = (b_1, b_2, b_3, \ldots b_N)$. The connection weights from neurons in layer $L_A$ to neurons in layer $L_B$ are represented by:

\[ W_1 = (w_1, w_2, \ldots, w_n)^T \]  \hspace{1cm} (3.1)

where $w_i = (w_{b_{i1}}, w_{b_{i2}}, \ldots, w_{b_{in}})$ is the weight vector of the $i^{th}$ neuron in layer $L_B$.

Each neuron in layer $L_B$ is connected to all the neurons in the third layer $L_C = (c_1, c_2, c_3, \ldots c_m)$ representing pattern classes. Connections from $L_B$ to $L_C$ are represented by:

\[ W_2 = (\bar{w}_1, \bar{w}_2, \ldots, \bar{w}_m)^T \]  \hspace{1cm} (3.2)
Figure 3.1 The NNC Architecture.
Connection weights $W_z$ are initialized to zero, and are assigned the binary values of one or zero as each prototype neuron is added during learning, satisfying the following condition:

\[
\text{if } \quad w_{b_j c_k} = 1 \\
\text{then } \quad w_{b_j c_i} = 0 \\
\forall i = 1, 2, \ldots, m, \text{ and } i \neq k
\]  

(3.3)

If an input pattern $P$ causes neuron $c_k$ to fire, it is classified as belonging to class $c_k$.

A neuron $c_k$ will fire if its inputs satisfy the following condition:

\[
\sum_j b_j w_{c_k b_j} \geq 1
\]  

(3.4)

where $b_j$ is the output of neuron $b_j$.

In response to a pattern $P^k$ the output of the NNC can be represented by the following equation:

\[
Y = [c_1, c_2, \ldots, c_k, c_m]^T
\]  

(3.5)

where

$Y$ is the output of the NNC,
$c_k$ is the neuron in layer $L_C$ of the NNC corresponding to class $k$.

$m$ is the number of neurons in layer $L_C$ of the NNC corresponding to the number of classes.

During training, the output of NNC as described in (3.5) can be in one of 4 possible types:

- Only one neuron corresponding to the correct class $k$ is firing.
  \[ c_k = 1, c_r = 0 \quad r = 1, \ldots, m ; \quad r \neq k \quad (3.6) \]

- No neurons are firing, i.e. the NNC did not identify the input pattern.
  \[ c_r = 0 \quad r = 1, \ldots, m \quad (3.7) \]

- One or more neurons corresponding to classes other than $c_k$ are firing, and the neuron corresponding to class $c_k$ is not firing, i.e. the NNC misclassified the input pattern.
  \[ c_k = 0, c_r \in [0, 1] \quad r = 1, 2, \ldots, m ; \quad r \neq k \quad (3.8) \]

- The neuron representing the correct class $c_k$ is firing, as well as neurons
representing other classes, i.e. the NNC is uncertain.

\[ c_k = 1 \ , \ c_r \in [0, 1] \quad r = 1, 2, \ldots, m : \ r \neq k \quad (3.9) \]

In the context of this work, training the NNC means ensuring that its output could always be described as in equation (3.6), for any input pattern \( P \). We achieve this by doing the following:

- Creating prototype neurons in layer \( L_B \).
- Adjusting the connection weights between neurons of layers \( L_A \) and \( L_B \).
- Adjusting the firing threshold conditions of prototype neurons in layer \( L_B \).
- Assigning one or zero values to the connection weights between neurons in \( L_B \) and neurons in \( L_C \).

An algorithm for training NNC is described in the following section.

### 3.3 NNC Training Algorithm:

Training NNC consists of selecting a training set \( T \) that has a number \( Q \) of ordered pairs:

\[
\{ P^k, c_i \} , \ k = 1, 2, \ldots, Q \\
i = 1, 2, \ldots, m 
\quad (3.10)
\]
where; \( m \) is the number of classes, \( P^k \) is an input pattern and \( c_i \) is its class. Input patterns have \( n \) dimensions and if they are continuous they can be described by:

\[
P^k = \{ p^k_1, p^k_2, \ldots, p^k_n \}
\]
\[
p^k_1, p^k_2, \ldots, p^k_n \in \mathbb{R}^n
\]  

(3.11)

Binary patterns can be described by:

\[
P^k = \{ p^k_1, p^k_2, \ldots, p^k_n \}
\]
\[
p^k_1, p^k_2, \ldots, p^k_n \in \{ 0, 1 \}
\]  

(3.11)

Training proceeds by successively presenting ordered pairs from \( T \) to the NNC. An evaluation of the response is made in each case, and accordingly the NNC learns using an algorithm described in detail in the following subsections. We will begin by following the case where an ordered pair \( \{ P^k, c_i \} \) from the training set \( T \) is presented for the first time to an NNC with no previous training. Then we will describe the algorithm for any other case.

a. First Pattern Presentation

Since there are no prototype neurons in \( L_B \) that are yet committed to labels (connection weights between \( L_a \) and \( L_C \) are all zero), there will be no response from the NNC to this pattern. The following steps will immediately occur:

- A neuron \( b_j \) will be selected from \( L_B \) to be a prototype of class \( c_i \).
Connections from \( b_j \) to all the neurons in \( L_A \) will be set according to

\[
w_{b_j a_s} = p_s^k, \quad s = 1, 2, \ldots, n
\]  \hspace{1cm} (3.12)

where \( b_j \) is a prototype neuron in \( L_B \), \( a_s \) is an input neuron in \( L_A \),
\( w_{b_j a_s} \) is the connection weight between neuron \( b_j \) in \( L_B \) and neuron \( a_s \) in \( L_A \),
\( p_s^k \) is component \( s \) of input pattern \( P^k \).

The connection weight from neuron \( b_j \) to class \( c_i \) in \( L_C \) is set to one:

\[
w_{b_j c_i} = 1
\]  \hspace{1cm} (3.13)

where \( w_{b_j c_i} \) is the connection weight between \( b_j \) and \( c_i \).

The firing threshold condition \( b_{j\tau} \) for neuron \( b_j \) determines when \( b_j \) will fire
according to the following equation for continuous patterns:

\[
b_{j\sigma} = 1, \quad \sqrt{\sum_{s=1}^{n} (w_{b_j a_s} - p_s^k)^2} \leq b_{j\tau}
\]

\[
= 0, \quad \text{otherwise}
\]  \hspace{1cm} (3.14)
Neuron $b_j$ will fire if it is similar to an input pattern $P$. The similarity measure is taken as the Euclidean distance between $P$ and $b_j$. The NNC will determine $b_{jT}$ according to the following equation:

$$b_{jT} = \min \left( \frac{p_{s \max}}{p_s} \right) \quad \forall \ s = \{1, 2, \ldots, n\}$$

(3.15)

where $p_{s \max}$ is the maximum value that dimension $p_s$ of the input pattern can assume.

For a binary pattern, we use the Hamming distance as the similarity measure between the input pattern and a prototype neuron in $L_g$. Equations (3.14) and (3.15) become the following respectively:

$$b_{jo} = 1, \quad \sum_{s \in 1}^n [ w_{b_{ps}} \cdot (1 - p_s) + (1 - w_{b_{ps}}) \cdot p_s ] \leq b_{jT}$$

$$= 0, \quad \text{otherwise.}$$

(3.14)

$$b_{jT} = n$$

(3.15)

The way that NNC sets its prototype firing thresholds differs from other designs in several ways:
1. The firing threshold is particular to every prototype neuron in $L_\text{in}$.

2. The firing threshold is automatically set by the NNC, initially and during training according to a problem independent guiding principle given by equation (3.15).

3. According to equation (3.15) for continuous patterns and (3.15) for binary patterns, the NNC initially sets the firing threshold of a neuron prototype to the maximum value it can assume, without violating any known constraints. This allows the NNC to minimize the number of prototype neurons in $L_\text{in}$ and consequently the memory required for implementation.

b. Subsequent Pattern Presentations:

In subsequent pattern presentations, the response of the NNC will be one of the four types described in section 3.2. A description of the training algorithm in each case follows:

i. Classification is Correct:

An ordered pair $\{P^b, c_t\}$ from training set $T$ is presented to the NNC. For convenience, we will drop the superscript from the reference to the input pattern hereafter. Neuron $c_f$ in $L_c$ is the only firing neuron. From equation (3.4), at least one prototype neuron
say \( b_j \) in \( L_B \) is firing. In the case of a continuous pattern the NNC modifies the connections between \( b_j \) and the neurons of \( L_A \) according to the following formula:

\[
w_{b_j a_{\text{new}}}^{a_{\text{old}}} = \frac{w_{b_j a_{\text{old}}} + p_i}{M + 1}, \quad \forall \quad i = 1, 2, \ldots, n
\]  

(3.16)

Where \( w_{b_j a_{\text{new}}}^{a_{\text{old}}} \) is the new value of the connection weight between neuron \( j \) in \( L_B \) and neuron \( i \) in \( L_A \). \( w_{b_j a_{\text{old}}}^{a_{\text{old}}} \) is the old value of the same connection weight, \( p_i \) is dimension \( i \) of the input pattern \( P \).

\( M \) is a popularity measure corresponding to the number of patterns which were determined to be similar to this prototype in previous presentations. This formula insures that connection weights to popular prototypes stabilize faster than the less popular ones. If more than one prototype neurons in \( L_B \) are firing, the NNC only modifies the connections to the most popular one using (3.16).

In the case of a binary pattern, no modifications are made to the connections between \( b_j \) and neurons of \( L_A \).

ii. Pattern not Identified:

No neurons in \( L_C \) are firing in response to an ordered pair \( \{ P, c_x \} \). In this case, the NNC will select a neuron in layer \( L_B \) say \( b_j \) as a prototype for the input pattern class, set the connection weights from this neuron to neurons in \( L_A \), and set to one the connection weight to \( c_x \) in \( L_C \) as in the case of first presentation. The NNC will determine the firing threshold \( b_{IT} \) in the following way:
\[ b_{jT} = \min(\ min ( \ p_{i \max} ), \ b_{jT \max}) \] \quad \forall \quad i = 1, 2, \ldots, n \tag{3.17} \]

Where \( p_{i \max} \) is as defined before in (3.15). \( b_{jT \max} \) is the maximum value that satisfies the following relation:

\[ \sqrt{\sum_{i=1}^{n} (p_i - w_{b_j a_i})^2} \leq b_{jT \max} \]

if

then

\[ \sqrt{\sum_{i=1}^{n} (p_i - w_{b_x a_i})^2} > b_{xT} \quad \forall \ \text{neuron } b_x \ \text{in } L_B ; b_x \neq b_j ; w_{b_x c_q} \neq 1 \quad \forall \ c_q \neq c_e \ \text{in } L_C \tag{3.18} \]

For a binary pattern, the equations are:

\[ b_{jT} = b_{jT \max} \tag{3.17} \]

\( b_{jT \max} \) is the maximum value that satisfies the following relation:
if

\[ H(P, w_j) \leq b_{j_{\text{max}}} \]

then

\[ H(P, w_x) > b_{x_T} \forall b_x \text{ in } L_B ; b_x \neq b_j ; w_{b_x q} = 1 \forall q \neq c \text{ in } L_C \]

(3.18)

Where \( H(P, w) \) is the Hamming distance between \( P \) and \( w \) as defined in (3.14). By using (3.17)/(3.17) to set the firing threshold of \( b_p \), NNC ensures that prototype neurons representing other classes will not fire while \( b_j \) is firing.

iii. Classification is Incorrect:

One or more neurons in \( L_C \) are firing, in response to the presentation of \( \{P, c_g\} \) such that:

\[ c_i = 1, \quad i \neq g \quad \forall \text{ firing neurons in } L_C \]

(3.19)

In this case the NNC will:

- locate the firing neurons;

- reset their firing threshold conditions to satisfy the following relations for continuous and binary input patterns:
\[ b_{j,\text{new}} < \frac{\sum_{i=1}^{n} (p_i - w_{b_{j,ai}})^2}{n} \quad \text{for all firing neurons} \ b_j \ \text{in} \ L_B \]

(3.20)

\[ b_{j,\text{new}} < H(P, w_j) \quad \text{for all firing neurons} \ b_j \ \text{in} \ L_{\text{II}} \]

(3.20)

- select new neuron from layer \( L_B \) to represent class \( c_k \). The connections between this new prototype neuron and the neurons of \( L_B \) and \( L_C \) are set as in case ii above.

iv. NNC is Uncertain:

One neuron representing class \( c_k \) is firing, as well as other neurons representing other classes in response to the presentation of \( \{P, c_k\} \). In this case the NNC will:

- locate the firing neurons in \( L_B \) that contribute to the firing of the wrong class neurons in \( L_C \);

- reset their firing threshold conditions as in (3.20)/(3.20);

- locate the firing neurons in \( L_B \) that contribute to the firing of the correct class neuron in \( L_C \);

- modify the connections to the most popular firing neuron as in (3.16) for continuous
Figure 3.2 The NNC Training Algorithm.
patterns and leave them as they are for binary patterns.

The NNC training algorithm is shown in Fig 3.2 and is summarized below:

- present a labeled pattern from a training set to NNC;

- let the NNC classify the pattern;

- evaluate the response of NNC as one of four types:
  - correct;
  - incorrect;
  - unidentified;
  - uncertain;

- modify connections and/or firing thresholds and/or create new neuron prototypes in layer $L_a$ according to the criteria mentioned above.

Fig 3.3 illustrates the process of prototype formation in the NNC as it learns three disjoint two dimensional classes. The only constraints given to the NNC are:

$$\min x = 0;$$

$$\min y = 0;$$
Figure 3.3 Prototype Formation with NNC.
Panel (a) shows the three classes. Class data is shown in Table 3.1. Panel (b) shows the NNC with one prototype of radius four after learning class 1. The radius corresponds to the firing threshold condition of the prototype neuron representing class 1 in $L_{\beta}$. Panel (c) shows that the NNC has formed another prototype after learning class 2. Panel (d) shows that the NNC has reduced the radius of class 1 prototype and created a set of overlapping prototypes to represent class 3. The NNC does not allow prototypes representing different classes to overlap, and allows overlapping of prototypes representing the same class.

Table 3.1 Three Disjoint Class Data.

<table>
<thead>
<tr>
<th>Class 1</th>
<th>Class 2</th>
<th>Class 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4,5)</td>
<td>(12,2)</td>
<td>(3,3)</td>
</tr>
<tr>
<td>(5,5)</td>
<td>(12,3)</td>
<td>(3,4)</td>
</tr>
<tr>
<td>(5,6)</td>
<td>(12,1)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>(6,5)</td>
<td>(11,2)</td>
<td>(2,3)</td>
</tr>
<tr>
<td>(5,4)</td>
<td>(13,2)</td>
<td>(4,3)</td>
</tr>
</tbody>
</table>

3.4 Classification Examples:

Two examples have been selected to illustrate the operation and performance of the NNC. The first one is a data set representing four concentric regions in a bullseye pattern. It illustrates the performance of the NNC on nonlinearly separable continuous patterns. The second one is a numeral recognition problem, where numerals from zero to nine of different
type fonts are digitized and taught to the NNC. It illustrates the performance of the NNC on binary patterns. Classification results for the two examples are shown in the following subsections.

3.4.1 Four Concentric regions:

Data for the bullseye pattern shown in Fig 3.4, were randomly generated using an equiprobable distribution. Sixteen thousand labeled points were generated for training and another four thousand unlabeled points were generated for testing. The NNC learned the pattern incrementally, first learning the inner most region or region one, then learning regions two, three and four respectively. After the NNC had been taught all the four regions, it was tested with the set of unlabeled patterns. Scatter plots of data classified by the NNC as region one and as region two are shown in Fig 3.5, region three and region four are shown in Fig 3.6. The total training time using software simulation of the NNC on a 486-25MHz PC was ~80 sec. At the end of the training session, three hundred and thirty three prototype neurons were used by the NNC to represent the bullseye pattern. The firing threshold conditions for these prototype neurons varied between [0.0, 4.5] as shown in Fig 3.7.
Figure 3.4 Bullseye Pattern.
Figure 3.5 Scatter Plots of Data Classified By NNC AS Regions 1 and Region 2.
Figure 3.6 Scatter Plots of Test Data Classified By NNC as Regions 3 and 4.
Figure 3.7 Distribution of the NNC Firing Thresholds After Learning the BullsEye Pattern.
3.4.2 Numeral Recognition:

In this example, we started by teaching the NNC numerals from zero to nine in six different type fonts: Bangkok, Switzerland, Zurich Calligraphic, Times New Roman, Toronto¹ and Avalon shown in Fig 3.8. Then we tested the NNC with numerals from zero to nine in other type fonts. Each numeral was first digitized using the following method:

- A 6 x 8 rectangle is placed around the character, with the right most and the bottom edges of the rectangle touching the right most and the bottom most pixels of the numeral (see for example Fig 3.9).

- The rectangle is divided into six equal- width columns, and eight equal- width rows, a total of forty eight equal area sub-rectangles.

- A 6 x 8 matrix representation of the numeral is obtained, by setting the elements in the matrix corresponding to black rectangles to one, and elements corresponding to white rectangles to zero.

After the matrix representation was obtained for each numeral in the above mentioned six type fonts, they were presented to an NNC with forty eight inputs with their corresponding labels. Before teaching the rest of the fonts in Fig 3.8 to the NNC, we used them as a test set, by presenting them unlabeled to the NNC.

¹ These are True Type fonts used under Microsoft windows and some are provided with the Corel Draw commercial package.
<table>
<thead>
<tr>
<th>Font</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bangkok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switzerland</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zurich Calligraphic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Times New Roman</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toronto</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avalon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Banff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Courier New</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dena Castle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erie</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>France</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fujiyama</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gatineau</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Newbrunswick</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nebraska</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ottawa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.8 True Type Numeral Training Set.
The outputs of the NNC in this experiment are shown in Table II, where R represents a right identification, U represents an unidentified and W represents a wrong identification. The table shows the following results:

- Correctly Identified: 38.1%
- Unidentified Numerals: 57.3%
- Wrongly Identified: 4.6%

The NNC was then retrained on the remaining fonts and 100% correct identification was obtained for all the 170 numerals in Fig 3.8 (17 true type fonts × 10 numerals in each). The total training time was ~7 seconds using software simulation of the NNC on a 486-25 MHz PC. The NNC used 109 prototype neurons which is ~2.3 times the number of inputs, and hence the NNC shows both memory efficiency and fast speed of training for this type of problem. The firing threshold conditions for these neurons ranged between [1.9] as shown in Fig 3.10.
Figure 3.9  Digitized Zero and Two in Bangkok Font.
Figure 3.10  Distribution of NNC Firing Thresholds after Learning the True Type Numeral Fonts.
Table 3.2 The NNC Numeral Identification Results.

<table>
<thead>
<tr>
<th></th>
<th>Banff</th>
<th>Arial</th>
<th>New</th>
<th>Dawn</th>
<th>Erie</th>
<th>France</th>
<th>Fujiyama</th>
<th>Gatineau</th>
<th>Newbrunswick</th>
<th>Nebraskas</th>
<th>Ottawa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R</td>
<td>R</td>
<td>U</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>U</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>U</td>
<td>R</td>
<td>R</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>R</td>
<td>U</td>
<td>U</td>
<td>R</td>
<td>U</td>
<td>R</td>
<td>U</td>
<td>R</td>
<td>U</td>
<td>R</td>
</tr>
<tr>
<td>5</td>
<td>U</td>
<td>R</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>R</td>
</tr>
<tr>
<td>6</td>
<td>U</td>
<td>W</td>
<td>U</td>
<td>U</td>
<td>W</td>
<td>U</td>
<td>U</td>
<td>R</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>7</td>
<td>R</td>
<td>R</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>U</td>
</tr>
<tr>
<td>8</td>
<td>U</td>
<td>R</td>
<td>W</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>R</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>9</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>R</td>
<td>U</td>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>U</td>
<td>R</td>
<td>R</td>
<td>U</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
3.5 **Summary and Conclusions:**

A variable thresholds prototype based neural network classifier (NNC) is introduced in this chapter. The classifier has the ability to learn on-line at a much faster rate than the gradient descent based neural classifiers, and it overcomes the drawbacks of the LBAQ classifier. Examples have been provided that demonstrate the ability of NNC in continuous classification problems with nonlinear class boundaries, and in binary classification problems. The NNC is different from other prototype based classifiers in that it tries to create prototype neurons with the maximum initial firing threshold condition, without overlapping prototypes representing other classes, and without violating any given constraints. The technique used by the NNC to set and modify the firing threshold conditions of its prototype neurons is described in this chapter.
CHAPTER IV

APPLICATION OF THE VARIABLE THRESHOLDS NEURAL CLASSIFIER TO REAL-WORLD DATA

4.1 Introduction:

In this chapter we apply the variable thresholds neural network classifier (NNC) to the problem of hand-written numeral recognition. In particular, we would like to show that the NNC can be useful in real-world recognition problems with large amounts of input data. The hand-written numeral recognition application was chosen because of its great practical value, and because it lends itself to simple representations. Numerals are represented by black pixels, well separated from a background of white pixels. The database used to train and test the network is a subset of the U.S. Postal Service database of handwritten numerals, digitized from handwritten zip-codes that appeared on U.S. mail. The digits were written by many different people, and represent a wide variety of writing styles, sizes and levels of care.

4.2 Recognition of Hand-Written Numerals:

The hand-written numerals pose enormous challenges to any recognition system. The wide range of styles used by different people, the different numeral sizes and the different levels of care that people use while writing are just a few of these challenges. In general a numeral recognition system will consist of two stages:
1. A preprocessing and segmentation stage.

2. A classification stage.

In the preprocessing stage, problems of acquisition, noisy inputs and segmentation of numerals are tackled, which are obviously very hard problems. For example, the segmentation would be a relatively easier task if we could assume that a character is contiguous and is disconnected from its neighbors, an assumption which rarely holds in practice. As a matter of fact, there are many ambiguous characters in the database as a result of mis-segmentations. The sizes of the numerals vary considerably too. Therefore it is necessary to normalize the size of the characters which is done in this stage using linear transformations to make the characters fit into a fixed size pixel grid. The transformation should preserve the aspect ratio of the character.

In the classification stage, many different algorithms and methodologies could be applied to classify the input image into one of the ten numeral categories. The input image could be used directly, or is processed to extract some features that are then used as an input to the classifier. Some of the features may come from moments, mathematical transforms, contours (such as loop, junction, concavities and convexities). The image could also be transformed into a binary representation; therefore the gray levels of each image are scaled and transformed to assume one of the binary image values.

In the experiments described below, we used a subset of the U.S. Postal Service database of hand-written numerals digitized from hand-written zip-codes that appeared on U.S. mail. They are already preprocessed and segmented. The database has 1641 samples of numerals, as shown in Table 4.1. Each numeral is represented by $64 \times 80$ pixels. The
numeral is represented by a pixel value of one, while the background is represented of pixel value of zero.

4.2 Experiments on Hand-Written Numeral Recognition using NNC:

Data from the database were used to train and test the NNC. For this purpose we used 20% of the numeral samples as a test set, and the rest of the data as a training set. The NNC was initially trained with the training set and then tested with the test set. The results of this testing are shown in Table 4.2. In the results tables we defined the reliability of recognition as follows:

\[
\text{Reliability} = \frac{R + U}{S} \times 100 \%
\]

Where,

R is the number of correctly identified numerals,

U is the number of unidentified numerals,

S is the number of samples tested.

All the numerals in the training and the test sets were transformed from 64 x 80 pixels to 8 x 10 pixels using the following procedure:

1. Each numeral image is divided into eight equal-width columns, and ten equal width rows, a total of eighty equal sub-areas.

2. If the number of pixels within the area with value one are \( \geq 32 \), the subarea is assigned the value of one, otherwise it is assigned the value of zero.
Table 4.1 Available Samples of Hand-Written Numerals.

<table>
<thead>
<tr>
<th>Numeral</th>
<th>Number of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>371</td>
</tr>
<tr>
<td>1</td>
<td>320</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>3</td>
<td>153</td>
</tr>
<tr>
<td>4</td>
<td>153</td>
</tr>
<tr>
<td>5</td>
<td>102</td>
</tr>
<tr>
<td>6</td>
<td>179</td>
</tr>
<tr>
<td>7</td>
<td>192</td>
</tr>
<tr>
<td>8</td>
<td>166</td>
</tr>
<tr>
<td>9</td>
<td>166</td>
</tr>
</tbody>
</table>
Table 4.2  Hand-Written Numeral Recognition Test Results.

<table>
<thead>
<tr>
<th>Numeral</th>
<th>Test Sample Size</th>
<th>Correct</th>
<th>Wrong</th>
<th>Unidentified</th>
<th>Recog %</th>
<th>Reliability %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>75</td>
<td>74</td>
<td>-</td>
<td>1</td>
<td>98.6</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>63</td>
<td>-</td>
<td>1</td>
<td>98.4</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>49</td>
<td>1</td>
<td>2</td>
<td>94.2</td>
<td>98</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td>29</td>
<td>-</td>
<td>2</td>
<td>93.5</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>30</td>
<td>-</td>
<td>1</td>
<td>96.8</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>18</td>
<td>2</td>
<td>1</td>
<td>85.7</td>
<td>90.4</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>34</td>
<td>-</td>
<td>2</td>
<td>94.4</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>39</td>
<td>38</td>
<td>-</td>
<td>1</td>
<td>97.4</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>34</td>
<td>33</td>
<td>1</td>
<td>-</td>
<td>97</td>
<td>97</td>
</tr>
<tr>
<td>9</td>
<td>34</td>
<td>33</td>
<td>-</td>
<td>1</td>
<td>97</td>
<td>100</td>
</tr>
</tbody>
</table>
The resultant representations are fed to an eighty input NNC with their corresponding labels during training, and without a label during testing. Some examples of the resultant representations for samples in the test set are shown in Fig. 4.1 - Fig. 4.3 for numerals zero, one and two. An original 64 x 80 pixel representation of numeral zero is shown in Fig. 4.4. In this figure, each of the eighty rows represents sixty four pixels, and every eight consecutive pixels in a row are represented by a byte value. The first row in Fig. 4.4 has eight bytes representing sixty four pixels of value zero. The eighty rows are shown in a two column format.

4.3. Conclusion:

The NNC was successfully applied to a real-world task. Most impressively it achieved an overall 95.3 % recognition rate and 98.5 % reliability with combined training session durations of ~ 40 seconds using software simulation of the NNC on a 486-25 MHz PC. The NNC used 840 prototypes to represent the handwritten numerals. The results demonstrate again the speed and memory efficiency of NNC.
Figure 4.1  Samples of Hand-written Numeral Zero.
<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<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>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<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>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>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<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>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>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>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>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 4.2** Samples of Hand-Written Numeral One.
Figure 4.3  Samples of Hand-Written Numeral Two.
Figure 4.4  A 64 x 80 Representation of Numeral Zero.
CHAPTER V

SUMMARY AND CONCLUSIONS

The main objective of the research described in this thesis is to develop neural classification algorithms that overcome the drawbacks of multi-layer perceptrons trained by the back-propagation algorithm. The drawbacks are outlined below:

- They require very lengthy training sessions.
- They cannot be trained incrementally. Teaching the classifier a new class involves providing a training set containing the new class as well as all the other classes that the network has learnt previously.
- There are no techniques for determining the number layers or the number of nodes per layer that are needed for a particular classification problem.
- The convergence of the back-propagation algorithm is not guaranteed.

Two neural classifiers were introduced and investigated in this work which overcome the above drawbacks. The classifiers are prototype based, i.e. they learn by forming prototypes of the patterns in the training set. The first classifier the LBAQ, uses hyperspherical prototypes that have the same firing threshold condition and it adjusts their centroids during training. One prototype can represent one or more input patterns which are similar to it. Similarity is based on a distance measure, which is the Euclidean distance for continuous patterns and the Hamming distance for binary patterns. The LBAQ learning is orders of magnitude faster than the multi-layer perceptron as demonstrated in the parity-N problem. The LBAQ is also capable of incremental on-line learning. A drawback of the
LBAQ is that it relies on the experience of its user for setting the firing thresholds of its prototypes. Once a classification mistake is made, the LBAQ has to be retrained using hyperspherical prototypes of a smaller radius. The second classifier, the NNC uses variable size hyperspherical prototypes in its internal representations. It still learns orders of magnitude faster than the multi-layer perceptron, yet it overcomes the problems of the LBAQ by automatically selecting and adjusting the size of its hyperspheres during the course of training. At any time it forms the maximum size hypersphere for a prototype, that will not overlap other hyperspheres representing different class patterns, but may overlap hyperspheres representing patterns of the same class. The NNC exhibited superior performance in continuous and binary classification examples. It was trained to classify hand-written numerals using a subset of the U.S. Postal Service database of hand-written numerals. It was also trained to classify printed numerals in seventeen different True Type fonts. The total training time was around seven seconds for the printed numerals and around forty seconds for the hand-written numerals, using software simulation of the NNC on a 486 - 25 MHz PC.

Some suggestions for future research extending the work in this thesis follow:

- Investigation of hardware implementation of the NNC.
- Training the NNC on features extracted from the training set, like moments or mathematical transforms, and comparing the training times and memory requirements to the training times and memory requirements obtained in this thesis using binary representation of the numerals.
APPENDIX

THE NNC SIMULATION PROGRAM

/*
 NNC SIMULATION
 /* This program simulates an artificial neural network, which finds */
 /* the decision boundaries between different pattern classes given */
 /* a set of patterns and their corresponding labels */
*/

#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include <alloc.h>
#include "classifyhc"
#include "classify.h"

/ ************************************************************************
 ** main **
 ************************************************************************/

main()
{
  int status;
  char file_name[20];
  float delta_center;
  float max_delta_radius;
  int iteration = 0;
  int firing_neurons;
  int answer;
  short train;

  printf("Enter the name of the input data file: ");
  scanf("%s", file_name);
  if ((in1 = fopen(file_name, "r")) == NULL)
  {
    printf("Cannot open input data file\n");
    exit(0);
  }
  initialize(&train);
  if (train == 0)
  {
    if ((out = fopen("results1.dat", "w")) == NULL)
{
    printf("\n Cannot open results data file\n");
    exit (0);
}
while (1)
{
    status = get_pattern();
    if (status == PROVIDE_ANSWER)
    {
        label_it();
    }
    else if (status == EOF )
    {
        break;
    }
    reset_neurons();
}
fclose(out);

} else
{
    do
    {
        iteration = iteration + 1;
        while (1)
        {
            label = 0;
            status = get_pattern();
            if (status == EOF)
            {
                break;
            }
            classify_it (&answer);
            if (answer == CORRECT)
            {
                modify_cluster_center();
            }
            else if (answer == DONTKNOW)
            {
                create_cluster();
            }
            else if (answer == WRONG)
{ 
    reduce_cluster_radii();
    create_cluster();
}
else /* mixed up */
{
    reduce_cluster_radii();
    modify_cluster_center();
}

find_delta_center(&delta_center);
find_max_delta_radius(&max_delta_radius);

reset_neurons();
}

fclose(in1);

in1 = fopen(file_input, "r");

} while ((delta_center >= 0.01) ||
         (max_delta_radius >= 0.01));

save_net(iteration);
}

/***************************************************************************/
/** GET_PATTERN */
*******************************************************************************/
get_pattern()
{
    int j;
    int row;

    if (pattern_attributes == 1) {
        if (fscanf(in1, "%f %d", &pattern[0],
                   &label) == EOF)
        {
            return (EOF);
        }
    }
}
if (label == 99)
{
    return (PROVIDE_ANSWER);
}

else if (pattern_attributes == 2)
{
    if((fscanf(in1, "%f %f %d", &pattern[0],
                &pattern[1], &label)) == EOF)
    {
        return (EOF);
    }
    if (label == 99)
    {
        return (PROVIDE_ANSWER);
    }
}
else if (pattern_attributes == 3)
{
    if((fscanf(in1, "%f %f %f %d", &pattern[0], &pattern[3],
           &pattern[2], label)) == EOF)
    {
        return (EOF);
    }
    if (label == 99)
    {
        return (PROVIDE_ANSWER);
    }
}

return (1);

/**********************************************************
** FIND FIRING NEURONS
***********************************************************/
find_firing_neurons()
{
    int labelq;
    CLUSTER *start;
    CLUSTER *end;

CLUSTER *p;
float sumsquare;
float sum;
int i;
float diff;
float min;
float temp;

for (labelq=0; labelq < 15; labelq++)
{
    determine_list (labelq, &start, &end);
    if (start != NULL)
    {
        p = start;
        do
        {
            sumsquare = 0;
            for (i=0; i<pattern_attributes; i++)
            {
                sumsquare = sumsquare +
                            ((pattern[i] - p->b[i]) * (pattern[i] - p->b[i]));
            }
            sum = (float) sqrt(sumsquare); /* Euclidean Distance */
            
            if (sum <= p->radius) //Rayleigh Distribution...
            {
                p->firing= 1;
            }

            p = p->next;
        }while (p != NULL);  
    }

} /*****************************************************************************/

/* CREATE A CLUSTER NODE */
****************************************************************************/
create_cluster()
{

    CLUSTER *ptr;
    float radius;
    CLUSTER *start, *end;
    int ilabel,
    int i;

    ilabel = label;
    determine_list (&ilabel, &start, &end);
    if (start == NULL)
    {
        end = start = malloc (sizeof(CLUSTER));
        if (start == NULL)
        {
            printf("Could not allocate memory\n");
            exit (1);
        }
    }
    else
    {
        ptr = end;
        end = malloc (sizeof(CLUSTER));
        if (end == NULL)
        {
            printf("Could not allocate memory\n");
            exit (1);
        }
        ptr->next = end;
    }

}

/* assign proper values to the cluster fields */

for (i = 0; i < pattern_attributes; i++)
{
    end->b[i] = pattern[i];
    end->delta_center[i] = 0;
}

end->num_members = 1;
end->next = NULL;
end->firing = 1;
determine_cluster_radius(&radius);
end->radius = radius;
end->label = label;
end->delta_radius = 0;

update_list(label, start, end);
}

/***************************************************/
/* DETERMINE THE BEGINNING AND END OF THE APPROPRIATE LINKED LIST */
/***************************************************/
determine_list(int ilabel, start, end)
int ilabel;
CLUSTER **start;
CLUSTER **end;
{
    if (ilabel == LABEL1)
    {
        *start = start_label1;
        *end = end_label1;
    }
else if (ilabel == LABEL2)
    {
        *start = start_label2;
        *end = end_label2;
    }
else if (ilabel == LABEL3)
    {
        *start = start_label3;
        *end = end_label3;
    }
else if (ilabel == LABEL4)
    {
        *start = start_label4;
        *end = end_label4;
    }
else if (ilabel == LABEL5)
    {
        *start = start_label5;
        *end = end_label5;
    }
else if (ilabel == LABEL6)
{
    *start = start_label6;
    *end   = end_label6;
}
else if (ilabel == LABEL7)
{
    *start = start_label7;
    *end   = end_label7;
}
else if (ilabel == LABEL8)
{
    *start = start_label8;
    *end   = end_label8;
}
else if (ilabel == LABEL9)
{
    *start = start_label9;
    *end   = end_label9;
}
else if (ilabel == LABEL10)
{
    *start = start_label10;
    *end   = end_label10;
}
else if (ilabel == LABEL11)
{
    *start = start_label11;
    *end   = end_label11;
}
else if (ilabel == LABEL12)
{
    *start = start_label12;
    *end   = end_label12;
}
else if (ilabel == LABEL13)
{
    *start = start_label13;
    *end   = end_label13;
}
else if (ilabel == LABEL14)
{
    *start = start_label14;
    *end   = end_label14;
}


```c
/**
 * UPDATE THE APPROPRIATE LIST
 */

update_list (ilabel, start, end)
int  ilabel;
CLUSTER  *start;
CLUSTER  *end;
{

    if (ilabel == LABEL1)
    {
        start_label1 = start;
        end_label1   = end;
    }
    else if (ilabel == LABEL2)
    {
        start_label2 = start;
        end_label2   = end;
    }
    else if (ilabel == LABEL3)
    {
        start_label3 = start;
        end_label3   = end;
    }
    else if (ilabel == LABEL4)
    {
        start_label4 = start;
        end_label4   = end;
    }
    else if (ilabel == LABEL5)
    {
        start_label5 = start;
        end_label5   = end;
    }
    else if (ilabel == LABEL6)
    {
```
start_label6 = start;
end_label6  = end;
}
else if (ilabel == LABEL7)
{
    start_label7 = start;
    end_label7  = end;
}
else if (ilabel == LABEL8)
{
    start_label8 = start;
    end_label8  = end;
}
else if (ilabel == LABEL9)
{
    start_label9 = start;
    end_label9  = end;
}
else if (ilabel == LABEL10)
{
    start_label10 = start;
    end_label10 = end;
}
else if (ilabel == LABEL11)
{
    start_label11 = start;
    end_label11 = end;
}
else if (ilabel == LABEL12)
{
    start_label12 = start;
    end_label12 = end;
}
else if (ilabel == LABEL13)
{
    start_label13 = start;
    end_label13 = end;
}
else if (ilabel == LABEL14)
{
    start_label14 = start;
    end_label14 = end;
}
else

{
    start_labelx = start;
    end_labelx = end;
}

/***************************
** DETERMINE CLUSTER RADIUS
***************************/

determine_cluster_radius (radius)
float *radius;
{
    int labelq;
    CLUSTER *start;
    CLUSTER *end;
    CLUSTER *p;
    float sumsquare;
    float sum;
    int i;
    float diff;
    float min;
    float temp;

    min = 1000000.0;
    for (i = 0; i < pattern_attributes; i++)
    {
        temp = (float) fabs((double) (pattern[i] - bmax[i]));
        if (temp < min)
        {
            min = temp;
        }

        temp = (float) fabs((double) (pattern[i] - bmin[i]));
        if (temp < min)
        {
            min = temp;
        }
    }

    for (labelq=0; labelq < 15; labelq++)
    {
        if (labelq != label)

{
determine_list (labelq, &start, &end);
if (start != NULL)
{
    p = start;
do
    {
        sumsquare = 0;
        for (i=0; i<pattern_attributes; i++)
        {
            sumsquare = sumsquare +
                ((pattern[i] - p->b[i]) * (pattern[i] - p->b[i]));
        }
        sum = (float) sqrt (sumsquare);
diff = (float) fabs((double)(sum - p->radius));
if (diff < min )
    {
        min = diff;
    }
    p = p->next;
}while (p != NULL);
}
}

*radius = min;

 /***********************************************************************/
**    MODIFY CLUSTER CENTER  
***********************************************************************/
modify_cluster_center()
{
    int    temp_num;
    CLUSTER *p, *the_cluster, *start, *end;
    int    ilabel;
    int    i;
    float    temp;

    ilabel = label;

determine_list (ilabel, &start, &end);
if (start == NULL)
{
    return;
}

p = start;
temp_num = 0;
do
{
    if (p->tiring == 1)
    {
        if (p->num_members > temp_num)
        {
            temp_num = p->num_members;
            the_cluster = p;
        }
    }
    p = p->next;
}while (p != NULL);

for (i=0; i < pattern_attributes; i++)
{
    temp = the_cluster->b[i];

    the_cluster->b[i] =
    ((the_cluster->num_members * the_cluster->b[i]) /
    (the_cluster->num_members + 1))
    + (pattern[i]) / (the_cluster->num_members + 1);

    the_cluster->delta_center[i] =
    (float) fabs ((double)(the_cluster->b[i] - temp));

    the_cluster->num_members = the_cluster->num_members + 1;
}

/**
 * REDUCE CLUSTER RADII
 */
reduce_cluster_radii()
{
    int labelq;
CLUSTER *start;
CLUSTER *end;
CLUSTER *p;
float sumsquare;
float sum;
int i;
float temp;

for (labelq=0; labelq < 15; labelq++)
{
    if (labelq != label)
    {
        determine_list (labelq, &start, &end);
        if (start != NULL)
        {
            p = start;
            do
            {
                if (p->firing == 1)
                {
                    sumsquare = 0;
                    for (i=0; i<pattern_attributes; i++)
                    {
                        sumsquare = sumsquare +
                        ((pattern[i] - p->b[i]) * (pattern[i] - p->b[i]));
                    }
                    sum = (float) sqrt (sumsquare);
                    temp = p->radius;
                    p->radius = sum - sum /10000;
                    p->delta_radius = (float) fabs ((double)(temp - p->radius));
                }
            }
            p = p->next;
        } while (p != NULL);
    }
} /* end if */
} /* end for */

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
** CLASSIFY THE INPUT PATTERN */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
classify_it (answer)
int *answer:
{
    int labelq;
    CLUSTER *start;
    CLUSTER *end;
    CLUSTER *p;
    int i;
    int answer1 = 0;
    int answer2 = 0;

    find_firing_neurons();
    for (labelq=0; labelq < 15; labelq++)
    {
        determine_list (labelq, &start, &end);
        if (start != NULL)
        {
            p = start;
            do
            {
                if (p->firing == 1)
                {
                    if (labelq == label)
                    {
                        answer1 = 1;
                    }
                    else
                    {
                        answer2 = 1;
                    }
                    break;
                }
            }
            p = p->next;
        } while (p != NULL);

        if ((answer1 == 1) && (answer2 == 1))
        {
            break;
        }
    }

    if ((answer1 == 1) && (answer2 == 0))
    {

*answer = CORRECT;
}
else if ((answer2 == 1) && (answer1 == 0))
{
    *answer = WRONG;
}
else if ((answer1 == 0) && (answer2 == 0))
{
    *answer = DONTKNOW;
}
else
{
    *answer = MIXEDUP;
}
}

/*
** FIND CHANGE IN CLUSTER CENTERS: one cluster center would have changed
**...........................................................................

find_delta_center(delta_center)
float *delta_center;
{
    float max = 0;
    CLUSTER *start;
    CLUSTER *end;
    CLUSTER *p;
    int i;
    float sum;
    float sumsquare;

determine_list (label, &start, &end);
if (start != NULL)
{
    p = start;
    do
    {
        sumsquare = 0;
        for (i = 0; i < pattern_attributes; i++)
        {
            sumsquare = sumsquare +
                        ((p->delta_center[i]) * (p->delta_center[i]));
        }
        ...
    }
    ...
}
sum = (float) sqrt (sumsquare);
if (sum > max)
{
    max = sum;
}

p = p->next;
}while (p != NULL);
} /* end if */

*delta_center = max;

/**
 * FIND MAXIMUM RADIUS CHANGE
 */

find_max_delta_radius (max_delta_radius)
float  *max_delta_radius;
{
    int    labelq;
    CLUSTER  *start;
    CLUSTER  *end;
    CLUSTER  *p;
    float   sumsquare;
    float   sum;
    int     i;
    float   temp;
    float   max = 0;

    for (labelq=0; labelq < 15; labelq++)
    {
        if (labelq != label)
        {
            determine_list (labelq, &start, &end);
            if (start != NULL)
            {
                p = start;
                do
                {
                    if (p->firing == 1)
                    {
                        if (p->delta_radius > max)
                        {
                            max = p->delta_radius;
                        }
                    }
            }
```c
} 
```
/******************************************************************************
**    SAVE THE NEURAL NETWORK
*******************************************************************************/

save_net (iteration)
int  iteration;
{
    int  i, j;
    int  labelq;
    CLUSTER  *start;
    CLUSTER  *end;
    CLUSTER  *p;

    char  *file_name = "output1.dat";

    if  ((out = fopen(file_name, "w")) == NULL)
    {
        printf("\n Cannot open output data file\n");
        exit (0);
    }
    fprintf(out, "%d \n", pattern_attributes);
    for  (i=0; i < pattern_attributes; i++)
    {
        printf (out, "%.8f", bmax[i]);
        printf (out, "%.8f\n", bmin[i]);
    }
    fprintf (out, "%d \n", iteration);

    for  (labelq=0; labelq < 15; labelq++)
    {
        determine_list (labelq, &start, &end);
        if  (start != NULL)
        {
            p = start;
            do
            {
                for  (i=0; i<pattern_attributes; i++)
                {
                    printf (out, "%.8f ", p->b[i]);
                }
                printf (out, "%8d %8d\n", p->radius, p->label,
                         p->num_members);
            }
/* */
/* initialize */
/*******************************************************************/

initialize(train)
short *train;
{
    CLUSTER *p;
    CLUSTER *start, *ptr, *end;
    int i;
    int j;
    char response[80];
    int dummy;
    int status;

    *train = 0;
    for (i=0; i < NMXATTR; i++)
    {
        bmax[i] = 1000.0;
        bmin[i] = 0.0;
    }
    response[0] = '\0';
    printf("\n Is this a training session? Y/N ");
    scanf("%s", response);
    if ((response[0] == 'Y') || (response[0] == 'y'))
    {
        *train = 1;
    }
    response[0] = '\0';

    if ((in = fopen("output1.dat", "r")) == NULL)
    {
        /* Code to handle the error */
    }

    for (i=0; i < NMXATTR; i++)
    { /* Code to process each attribute */
    }

    if (*train)
    { /* Code to train the model */
    }
    else
    { /* Code to process the training dataset */
    }
}

fclose(out);
}
printf("\nNo initialization file\n");
if (*train == 0)
{
    printf("\nYou have to create the network before you can classify!!\n");
    exit(0);
}
printf("\nEnter the number of pattern attributes: ");
scanf("%d", &pattern_attributes);
printf("\nDo you want to enter constraints? Y/N ");
scanf("%s", response);
if ((response[0] == 'Y' || (response[0] == 'y'))
{
    for (i=0; i < pattern_attributes; i++)
    {
        printf("\nEnter the maximum value of attribute %d ", i+1);
        scanf(" %f", &bmax[i]);
        printf("\nEnter the minimum value of attribute %d ", i+1);
        scanf(" %f", &bmin[i]);
    }
}
else
{
    fscanf (in, "%d \n", &pattern_attributes);
    for (i=0; i < pattern_attributes; i++)
    {
        fscanf (in, "%f", &bmax[i]);
        fscanf (in, "%f \n", &bmin[i]);
    }
    fscanf (in, "%d \n", &dummy);
    while (1)
    {
        p = malloc (sizeof (CLUSTER));
        if (p == NULL)
        {
            printf("Could not allocate memory\n");
            exit (1);
        }
        p->firing = 0;
        p->delta_radius = 0;
        for (i=0; i < pattern_attributes; i++)
        {
            p->delta_center[i] = 0;
        }
for (i = 0; i < pattern_attributes; i++)
{
    status = fscanf (in, "%f", &p->b[i]);
}
if (status == EOF)
{
    break;
}
fscanf (in, "%f %d %d\n", &p->radius, &p->label,
          &p->num_members);
determine_list (p->label, &start, &end);
if (start == NULL)
{
    start = end = p;
    end->next = NULL;
}
else
{
    ptr = end;
    end = p;
    ptr->next = p;
    end->next = NULL;
}
update_list (p->label, start, end);

fclose (in);

}
CLUSTER *end;
CLUSTER *p:
int i:
short comment:
int answer1 :
int answer2 :

answer1 = 0;
answer2 = 0;
find_firing_neurons();
for (labelq=0; labelq < 15; labelq++)
{

determine_list (labelq, &start, &end);
if (start != NULL)
{
    p = start;
do
    {
        if (p->firing == 1)
        {
            if (answer1 == 0)
            {
                label = labelq;
                answer1 = 1;
                break;
            }
            else
            {
                if (labelq != label)
                {
                    answer2 = 1;
                    break;
                }
            }
        }
        p = p->next;
    } while (p != NULL);
}
if ((answer1 == 1) && (answer2 == 1))
{
    break;
}


```c
}

ii ((answer1 == 1) && (answer2 == 0))
{
    comment = CORRECT;
}

else if ((answer2 == 1) && (answer1 == 0))
{
    comment = WRONG;
}
else if ((answer1 == 0) && (answer2 == 0))
{
    comment = DONTKNOW;
}
else
{
    comment = MIXEDUP;
}

for (i=0; i<pattern_attributes; i++)
{
    fprintf(out, "%.4f ", pattern[i]);
}

fprintf (out, "%8d %8d\n", label, comment);
```
REFERENCES


D. L. Reilly, L. N. Cooper, and C. Elbaum, "A Neural Model for


VITA AUCTORIS

Mahmoud A. Abu-Nasr

1955 Born on August 24th, EGYPT.

1977 Received a B.Sc. degree in Electrical Engineering, from the University of Alexandria, Alexandria, Egypt.

1984 Received a M.A.Sc. degree in Electrical Engineering from the University of Windsor, Windsor, Ontario, Canada.

1994 Candidate for a Ph.D. degree in Electrical Engineering from the University of Windsor, Windsor, Ontario, Canada.