OBJECT ORIENTED ARTIFICIAL NEURAL NETWORK SIMULATOR

IN TEXT AND SYMBOL RECOGNITION

A Master's Thesis

Presented to

Computer and Information Science Division

In Partial Fulfillment

of the Requirements for the

Master of Science Degree

SUNY Institute of Technology

at Utica/Rome

by

Alan T. Piszcz

September 1993

SUNY COLLEGE OF TECHNOLOGY
AT UTICA/ROME

DEPARTMENT OF COMPUTER SCIENCE

CERTIFICATE OF APPROVAL

Approved and recommended for acceptance as
a thesis in partial fulfillment of the requirements
for the degree of Master of Science in
computer and information science

Sept. 1993

____
DATE

Naseem Ishaq
____
Dr. Naseem Ishaq
Advisor

____
Dr. Jorge E. Novillo

Dr. Sam Sengupta

ABSTRACT

Objected oriented languages and artificial neural networks are new areas of research and development. This thesis investigates the application of artificial neural networks using an object oriented C++ backpropagation simulator. The application domain investigated is hand printed text and engineering symbol recognition.

An object oriented approach to the simulator allows other simulator paradigms to reuse a large body of the object classes developed for this particular application. The review and implementation of image feature extraction methodologies is another area researched in this paper. Four feature techniques are researched, developed, applied and tested, using digits, upper case alphabet characters and engineering symbol images. Final implementation and testing of the feature extraction methods with a baseline technique is analyzed for applicability in the domain of hand printed text and engineering symbols.

TABLE OF CONTENTS

			Page
LIST OF TABLES			vii
LIST OF FIGURES			viii
CHAPTER 1	INTRODUCTION		
	1.1	Introduction to OCR and Artificial Neural Networks	1
	1.2	Neural Network History	2
	1.3	Optical Character Recognition (OCR) History	5
	1.4	Thesis Overview and Motivation	9
		1.4.1 Thesis Layout	11
CHAPTER 2	LITERATURE SEARCH		12
	2.1	Review of Related Literature and Research	12
CHAPTER 3	BIOLOGICAL NEURAL NETWORKS		24
	3.1	Neural Network Biological Metaphor	24
	3.2	Combining Processing Elements into a network	31
CHAPTER 4	ARTIFICIAL NEURAL NETWORK SIMULATOR		35
	4.1	Artificial Neural Network Simulator Description	35
	4.2	Backpropagation Network Mathematical Model	38
	4.3	ANNS Computational Complexity Models	43
		4.3.1 Network Connections (NC)	45
		4.3.2 Network Complexity Index (NCI)	46
		4.3.3 Connection Updates Per Second (CUPS)	46
	4.4	Backpropagation Network Configuration	47
		4.4.1 Exclusive OR Network Configuration	48
	4.5	Backpropagation Network Training I/O Pairs	50
	4.6	Backpropagation Network Test Set Data	51
	4.7	Object-Oriented Programming Overview	53
		4.7.1 Class Description	53

4.7.2 Object Description 54
4.7.3 Information Hiding 54
4.7.4 Inheritance 55
4.7.5 Virtual Functions 55
4.7.6 Polymorphism 56
4.7.7 Dynamic Binding 56
4.8 Backpropagation Class Hierarchy 58
4.9 Software Components Of The Simulator 59
4.10 Simulator Porting Issues 70
4.11 Simulator Sample Operation 72
4.11.1 Operation With XOR Problem 73

CHAPTER 5 FEATURE METHODS AND APPLICATIONS 78
5.1 Introduction 78
5.1.1 Image Acquisition 78
5.1.2 Image Database 79
5.1.3 Sample of Digits and Uppercase Test Images 83
5.1.4 File Formats 85
5.1.5 Image Processing 87
5.2 Feature Vector Creation 88
5.2.1 Feature Vector Interpretation 88
5.3 Raw Bitmap Feature 90
5.3.1 Creation of the Raw Bitmap Feature Vector 92
5.4 Side Contour Profile Feature 102
5.5 Quadrant Method Feature 107
5.6 Hough Transform Feature 116

CHAPTER 6 FINAL ANALYSIS AND CONCLUSION 131
6.1 Introduction 131
6.2 Accuracy 132
6.3 Complexity 135
6.4 Performance 136
6.5 Object Oriented Artificial Neural Networks 137
6.6 Feature Methods 138
6.7 Conclusion 142

APPENDIXES 145

A. Artificial Neural Network Supporting Source Code 146
B. Raw Bit Map Supporting Source Code 183
C. Side Contour Profile Supporting Source Code.......................... 191
D. Quadrant Method Supporting Source Code.............................. 199
E. Hough Transform Method Supporting Source Code 205

REFERENCES 216

LIST OF TABLES

Table 2-1 Generalization comparison of different networks....................................18
Table 3-1 Biological versus artificial neural networks analogies...............................30
Table 3-2 Well Known Neural Networks...34
Table 4-1 Domain knowledge spectrum..36
Table 4-2 Exclusive OR Problem. 48
Table 5-1 Raw Bit Map Training Summary...94
Table 5-2 RBM Performance Matrix Digit Results..95
Table 5-3 RBM Uppercase Results Summary...97
Table 5-4 RBM Performance Matrix Uppercase Results.......................................98
Table 5-5 RBM Performance Matrix Symbol Results..100
Table 5-6 Side Contour Profile Training Summary. 105
Table 5-7 SCP Performance Matrix Digit Results. 105
Table 5-8 Quadrant Method Subsample Sizes..107
Table 5-9 Quadrant Method Training Summary. 110
Table 5-10 Quadrant Method Digit Results..110
Table 5-11 Quadrant Method Uppercase Results Summary..................................112
Table 5-12 QM Performance Matrix Uppercase Results.......................................113
Table 5-13 QM Performance Matrix Symbol Results..115
Table 5-14 Hough Method Training Summary. 125
Table 5-15 Hough Method Digit Performance Matrix. 126
Table 5-16 Hough Method Uppercase Results Summary. 127
Table 5-17 Hough Performance Matrix Uppercase Results...................................128
Table 5-18 Hough Performance Matrix Symbol Results..130

LIST OF FIGURES

Figure 1-1 Two dimension to one dimension reduction examples. 7
Figure 1-2 Peephole Method...8
Figure 3-1 Simple Neuron. 25
Figure 3-2 Example of a nerve structure..26
Figure 3-3 Sample transfer functions. 29
Figure 3-4 Processing element and associated parameters. 30
Figure 3-5 Example neural network architecture...31
Figure 3-6 Single element with feedback...32
Figure 3-7 Local Minimum Diagram. 33
Figure 4-1 Neural Network System. 37
Figure 4-2 Neural Network System Simulator. 38
Figure 4-3 Backpropagation Topology. 39
Figure 4-4 Programming Model. 41
Figure 4-5 CASE #1 NETWORK. 44
Figure 4-6 CASE #2 NETWORK. 44
Figure 4-7 Neural Network Configuration. 47
Figure 4-8 Neural Network Training Data I/O Pairs.....................................50
Figure 4-9 Neural Network Test Set Data. 51
Figure 4-10 Backpropagation Class Hierarchy...58
Figure 4-11 Backpropagation Class Hierarchy Overview. 61
Figure 4-12 Backpropagation Network Class Methods. 61
Figure 4-13 Class 'net' Methods. 62
Figure 4-14 'testbp' Interface Code. 62
Figure 4-15 'vecmat' Vector Code Method Overview. 62
Figure 4-16 Class 'vector' Vector Methods. 63
Figure 4-17 Class 'matrix' and Methods..64
Figure 4-18 Class 'vecpair' and Methods. 65
Figure 4-19 Simulator Modes Overview. 66

Figure 4-20 Flow Diagram For 'cycle' and 'encode'. 67
Figure 4-21 Flow Diagram For 'test.'...68
Figure 4-22 Flow Diagram For 'run.' 69
Figure 4-23 OR.DEF Network Definition File. 73
Figure 4-24 OR.FCT Network Input Output Pairs With Limits. 74
Figure 4-25 OR.IN Test Input Vectors..74
Figure 4-26 OR.IN Test Input Vectors..74
Figure 4-27 Convergence of XOR Test Case. 76
Figure 4-28 Convergence of XOR Test Case. 77
Figure 5-1 Digit Image Database Sample..79
Figure 5-2 Uppercase Alphabet Image Database Sample............................80
Figure 5-3 Utility Drawing used for Symbols..81
Figure 5-4 Symbol Set From Utility Drawing. 82
Figure 5-5 Sample of Digit Test Set 83
Figure 5-6 Sample of Uppercase Test Set. 84
Figure 5-7 Raw Bit Map to Feature Vector Mapping.................................91
Figure 5-8 RBM Performance Matrix Contour Plot For Digits. 96
Figure 5-9 RBM Performance Matrix Contour Plot For Uppercase. 99
Figure 5-10 Performance Matrix Symbol Contour Plot. 101
Figure 5-11 SCP Feature Overview. 103
Figure 5-12 SCP Feature Plot...104
Figure 5-13 SCP Performance Matrix Digit Contour Plot............................106
Figure 5-14 QM Feature Diagram. 108
Figure 5-15 Quadrant Method Digit Contour Plot......................................111
Figure 5-16 Quadrant Method Uppercase Contour Plot. 114
Figure 5-17 QM Performance Matrix Symbol Contour Plot........................115
Figure 5-18 Hough Transform Mapping XY Plane to Parameter Space........117
Figure 5-19 Hough Quantization Of The Parameter Plane. 118
Figure 5-20 Hough Image to Hough Parameter Space Feature Diagram. 121
Figure 5-21 Process Flow of Hough Feature Diagram.................................122
Figure 5-22 Data Flow of Hough Feature. 123
Figure 5-23 Hough Transform Examples..124
Figure 5-24 Hough Method Digit Performance Matrix Contour Plot............126
Figure 5-25 Hough Performance Matrix Contour Plot................................129
Figure 5-26 Hough Performance Matrix Symbol Contour Plot. 130
Figure 6-1 RBM - HT - QM Overall Accuracy. 132

Figure 6-2 RBM - HT - QM Digits Accuracy..133
Figure 6-3 RBM - HT - QM Uppercase Accuracy...................................134
Figure 6-4 RBM - HT - QM Symbol Accuracy..135
Figure 6-5 RBM - HT - QM Network Complexity Index Comparison.......136
Figure 6-6 RBM - HT - QM Connection Updates Per Second..................137
Figure 6-7 Pattern Recognition Overview. 139

CHAPTER 1

INTRODUCTION

1.1 Introduction to OCR and Artificial Neural Networks

Since the creation of the first computers, complex problems have been analyzed and modeled into programs which could be understood by a computer. Many of the early and successful problem areas were solved because of their corresponding digital representation, either by mathematical operations or by a strict numerical analysis approach. Analog computers were also used for similar computations; they have the advantages of very high speed but lack the flexible modification tools that digital systems employ. As time progressed, more difficult problems with greater computational complexity were applied to the computer. Computer speed and performance quickly limited these computationally complex problems. This exponential search space is one of the key motivations behind Artificial Intelligence (AI), which attempts to use intelligence rather than a 'brute force' approach to solving the problem. If the search space can be adequately 'pruned' by AI techniques, then extremely complex and previously intractable problems become possible applications with solutions.

Determination of what data in the data set is critical to solving the problem and discarding features which are not relevant are standard operations in AI and neural network solutions. The problems in this domain are just starting to be solved successfully; many others are still well funded research programs in industry, academia and the Department of Defense. The problem domain for this thesis is hand printed character recognition. A brief history of

neural networks and optical character recognition will represent some of the research milestones in these both of these important areas.

1.2 Neural Network History

Understanding the human brain and its operating mechanisms has been a strong motivation of scientists and researchers for centuries. If one could capture the knowledge of how the brain works, perhaps it could be reproduced with mechanical or electrical systems to emulate it. The earliest documented information was produced around 3000 B.C., it detailed the sensor and motor control functions in the brain[1]. In the late 1800's, more studies were performed in an attempt to determine how the brain functioned. Most of these were speculative due to the lack of understanding of the brain physiology. In 1936 Alan Turing was the one of the first to use the human brain processes, as a computing paradigm. He produced computing machine models which were extendible and could solve problems previously thought to be impossible for solutions using machines.

After the neurophysiologist, Warren McCulloch and an eighteen year old mathematician, Walter Pitts wrote a paper on how the neurons might work, the real model of the neuron was brought to the research world. In 1943, they were working on the understanding of neurons. Later, other researchers like John von Neumann used it to model intelligence into machines he was researching and developing. Donald Hebb's The Organization of Behavior (1949) pointed out that the neural pathway is reinforced each time it is used. This was one of the early learning rules used to describe neural systems. It is known as Hebb's Learning Rule and is still referenced today. Many of the years 1890-1950 were spent on research of understanding the physiology of the biological neural networks of nature.

The 1950's brought technological possibilities to simulate the previously researched neural networks in software. Research performed by Nathaniel Rochester and others at the IBM Research Laboratory created software models of Donald Hebb's research. Problems with the simulations and models resulted in no great success stories from the IBM laboratories.

An Artificial Intelligence research project at Dartmouth in 1956 brought together high level AI researchers and low level (neural network) researchers which resulted in a fresh new inspiration for work in the area neural networks. The late 1950's had two more respected researchers join in the neural network community. John von Neumann (Yale University) and Frank Rosenblatt (Cornell) developed hardware models to simulate a single neuron. The first practical and widely used neural network implementation was the ADALINE and MADALINE (Multiple ADApative LINear Elements). The work performed by Bernard Widrow and Marcian Hoff (Stanford) 1959 performed echo cancellation on phone circuits.

Shortly after the success of the MADALINE Stephen Grossberg (Boston University) performed extensive physiological research to develop accurate neural models. His work led to a network named Avalanche in 1967, which was applied to continuous speech recognition and robot arm control. As with most fields of limited understanding, the expectations of future possibilities started to grow out of proportion with reality. This caused a upsurge in interest which swept the neural network community. However the systems did not produce results to meet the unrealistic expectations.

Enthusiam dampened in 1969 when Marvin Minsky and Seymour Papert published *Perceptrons*[2]. The information published by these highly respected AI researchers led the community to believe that neural networks could not solve any problems of significance. Corresponding cuts in neural network research in United States occurred, followed by increased funding in AI research. Many of the true believers in the neural network research community continued their work at a much lower profile. The shock wave created by "*Perceptrons* " severely stunted the growth of neural networks until 1981.

What happened next to influence a revival in the research funding was a convergence of a number of key events. A highly respected physicist, John Hopfield, presented a paper which contained a very clear mathematical analysis of neural networks and their limitations. This paper, presented in 1982, is considered by many the biggest spark to renewed interest in neural networks. The following events occured in quick succession and interest exploded:

- 1982 - Japan announced development projects in the Sixth Generation Effort, creating thinking machines for robotic applications.
- 1985 - American Institute of Physics started the Neural Networks for Computing meetings.
- 1987 - Institute of Electrical and Electronics Engineers' (IEEE) First International Conference on Neural Networks had 1,800 attendees and 19 vendors.
- 1987 - The International Neural Network Society (INNS) was formed and within two years had 3,000 members.
- 1989 - INNS and IEEE joined together for conference sponsorship. This first meeting produced 430 papers, 63 were in application development.

All the effort expended in the last five years has produced useful applications which are being sold, and are producing revenues for the customers. More rigorous mathematical models are being developed to help describe and match the best neural network model to a particular domain. Application areas include medical, chemical, computer information theory, adaptive control, adaptive systems, and AI research for machine learning systems. Development of hardware and software systems in this field are currently being targeted at what is being predicted as a $6 billion industry by 1998. An understanding of this field will soon be a necessity and considered part of the computer scientists tool kit for solving problems in a wide spectrum of domains.

The quest for building systems that can function automatically has attracted a lot of attention over the centuries and created continuous research activity. The current trend is toward building autonomous systems that can adapt to changes in their environment, but there is much to be done before we reach this point. The desire to achieve "intelligent engineering systems" is here to stay. Artificial neural network technology is an integral part of designing these intelligent systems which can enable companies to respond rapidly to changes global markets and thus stay competitive[3].

In 1992, the Department of Engineering Management at the University of Missouri-Rolla organized the second conference of Artificial Neural Networks in Engineering (ANNIE '92), co-sponsored by International Neural Network Society, McDonnell Douglas Corporation, Lockheed Corporation-Corporate Headquarters, and Washington University in St. Louis. The

conference attracted over 150 papers from around the world, which, after being peer reviewed and revised were included in *INTELLIGENT ENGINEERING SYSTEMS THROUGH ARTIFICIAL NEURAL NETWORKS, VOLUME 2*[4].

1.3 Optical Character Recognition (OCR) History

Optical Character Recognition (OCR) represents a class of image recognition problems which require the separation of the input image space into many different output classes. OCR research was one of the first areas of study in pattern recognition[13]. In part, this was due to the prevalent attitudes which thought it could be solved easily. What grew from these optimistic expectations were not solutions for OCR, as many new problems were discovered, but new areas in image understanding which created specialized niche areas of study. Image understanding itself represents many different problems; to most researchers in pattern recognition these problems have common threads for which discoveries and research in one area usually can be applied to other areas. Research funding is usually proportional to the amount of application products it can help sell. In OCR this is a very large dollar amount. Technological development of OCR research has three major categories: 1) Template matching, 2) Structural analysis, 3) Neural networks. Recently all three techniques have begun to fuse, using the advantages of each individual methods inherent ability to solve the problem.

The first patents applied for with OCR solutions were from Dr. Tausheck obtained in Germany in 1929[5], and Handel[6] from the U.S. did the same in 1933. These conceptual solutions of reading characters and numerals which have been hand printed would not realize practical hardware until two decades later. Tausheck's patent was a technique which utilized template matching by using mechanical masks illuminated by a light source and detected by photodetectors.

Template matching is still one of the most heavily used techniques for machine printed characters. Kelner and Glauberman[7] implemented a template matching technique which projected the two dimensional information from an image into one dimension. The character is scanned vertically top to bottom by a slit through which reflected light is transmitted to the

photodetector. Outputs of the photodetectors are summed to produce the total for that specific time. In recognition operation this technique is used to measure the differences of the source character against the target character, the cases where the error is smallest represents a match. Because of the distortion introduced by 2-dimensional to 1-dimensional transformation many errors occurred, no machines were ever produced or commercialized for use.

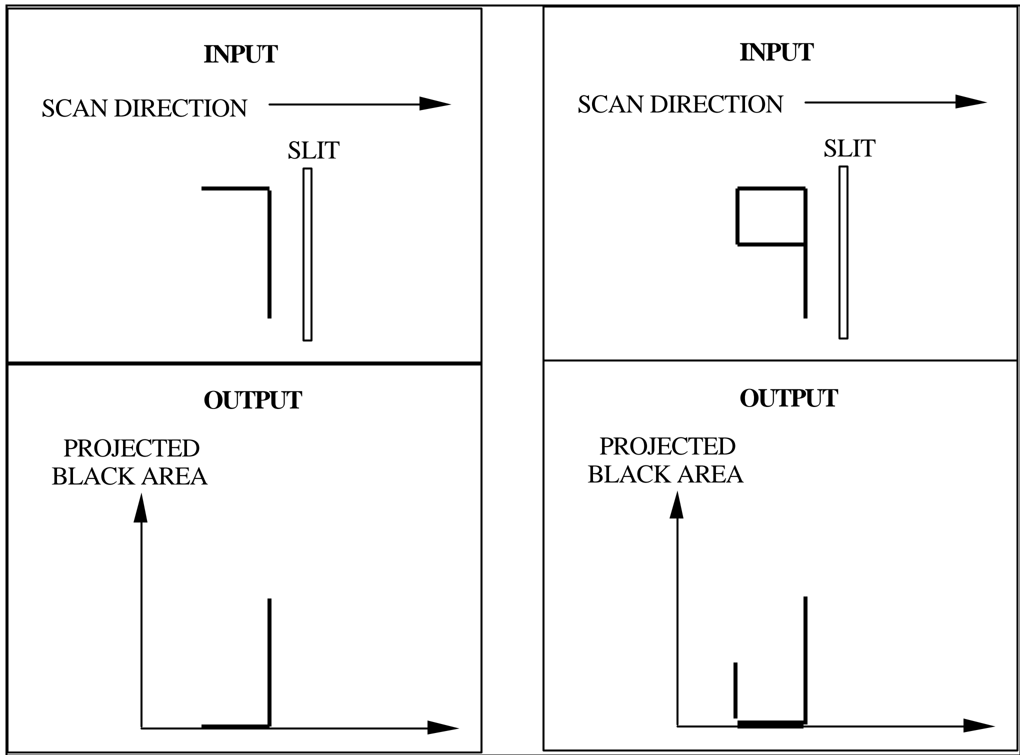


Figure 1-1 Two dimension to one dimension reduction examples.

Figure 1-1 shows that this technique has flaws, and the primary problem is the loss of axis symmetry information. In the second case example a 9 or 6 will have the same output.

RCA applied optical template matching to OCR in 1962[8] through use of electronics and optical techniques. Hannan concluded that this device proved that the RCA optical masking-matching technique can be used to recognize English and Russian fonts. At least that was his claim, no commercial RCA OCR technology was produced. This is an example of why research continues in this domain, many claims are made but to date there are no successful commercial hand printed OCR products.

Solatron Electronics Group Ltd.[9] introduced the Electric Reading Automaton (ERA) in 1957. It read machine printed numerals printed by a cash register. The technique used for recognition was the peephole method. By assigning groups of white and black pixel coordinates to each character class detected by a detector, input character images can be distinguished from

characters belonging to other classes. A grid was made up of a 10x12 mesh from which 44 specific peepholes were used. Figure 1-2 illustrates the peephole method.

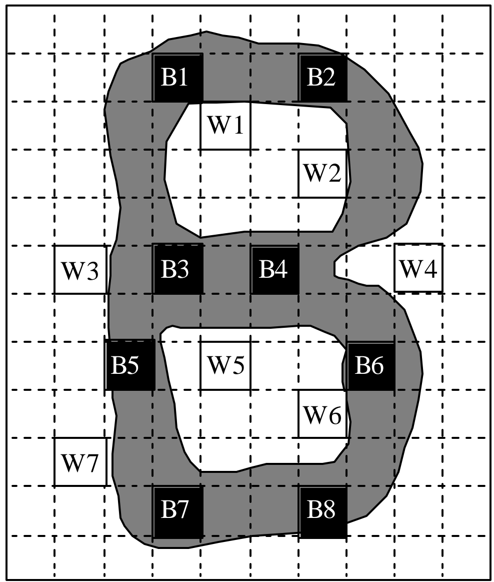


Figure 1-2 Peephole Method.

Template matching techniques have some inherent problems which are still being studied. Automatic registration or correlation of the character with an alignment grid is a key element in all template matching methods. Two initial methods that were applied to obtain the exact orientation of the character were autocorrelation and moments. Ideally the technique will be rotation, scale and shift invariant. Substantial research is still underway in this very important area. Autocorrelation problems have been researched by IBM in 1961 by Horowitz and Shelton[10] and the Japanese Government's Radio Wave Research Laboratory, Sato[11]. Results from Sato's work were disappointing. Differences between "R" and "B" were 0.4% when normalized, problems with differences of less than 1% also occurred in the following character pairs: "K" and "R", "A" and "V", "U" and "D".

Template matching has been successful with machine printed text. Hand printed text has such a large variation of style it is difficult to create templates. Templates have been created for certain writers using constrained text.

In the case of the structural analysis method, there is no mathematical principle. It involves using shapes to represents parts of the image, then matching these shapes with expected values to reconstruct the components of a character[12]. Rather it is still an open problem and there is no indication that it will be solved in the near future. Hence, our intuition has been the most reliable weapon attacking this problem. However, there appears to be certain informal strategies that can be used in structure analysis. Since a structure can be broken into parts, it can be described by the features of these parts and the relationships between them. Problems exist in choosing features and relationships between them so that the description gives each character clear identification. Feature extraction, therefore, has become the key in pattern recognition research[13].

OCR research continues to be researched at many different sites across the United States and the world. Full explanation and detailed analysis of the four methods using structure features from characters are examined in Chapter 5.

1.4 Thesis Overview and Motivation

The problem of character recognition is a class of image recognition problems which requires the separation of input regions into many different output classes. The resultant output for recognition is a classification of the input class relative to the available output class patterns. Some techniques in the past focused on statistical and syntactical methods to model the features as strokes and curves of the character image. Many of these techniques were very sensitive to rotation, scale and writing instruments.

Out of the mature signal processing domain have come many of the image processing algorithms adapted and converted from there signal processing functions. Statistical classifiers and basic voting schemes have been used for classifying this information created from the image. Many of these methods are still valid in constrained domains and applications. Current research is now researching methods of producing features from images and quickly classifying the results using artificial neural networks. An overview of the research is in Chapter 2. Artificial neural

networks paradigms allow problems to be modeled for complex non-linear functions in a relatively short amount of time when compared to other techniques. Many different neural network models exist; every year brings new architectures which may apply to a specific domain problem. The backpropagation paradigm has been applied successfully in classification problems and will be used for this problem. Discussion of the backpropagation technique is described in Chapter 4.

The research developed in this thesis will focus in following areas:

- Implementation of a C++ based artificial neural network simulator. The simulator chosen is a port and modification of a work initiated by Adam Blum[14]
- Research and investigation of feature extraction techniques for text recognition.
- Development, testing and analysis of feature extraction techniques.
- Training, testing and architecture investigation of the neural network.
- Analysis of results from testing the feature extraction techniques, and neural network for classification accuracy.

Motivating this effort is the understanding of artificial neural networks, feature extraction methods, data structures and an end-to-end system that will perform classification of isolated character images. Of particular concern is the complexity of the feature methods used and the training time required when using artificial neural networks. Training times totaling hundreds of hours are common in backpropagation networks, an understanding of the issues involved are investigated in an attempt to reduce this time. Reduction of time must not forfeit classification accuracy, therefore a balance of efficiency and accuracy must be maintained.

The main emphasis in this work will be on the feature extraction methods. Creating generalized descriptors of the input image patterns in the most effective way possible for use by the neural network is the main goal of this thesis. A preliminary discussion of neural networks will be served for background information on the artificial neural network backpropagation

simulator chosen. The artificial neural network simulator is used to classify the feature data produced by the feature extraction methods. Architecture modifications of the network configuration are tested to determine an optimum solution for text recognition. A literature review of hand printed text features, and current OCR research is presented in Chapter 2.

1.4.1 Thesis Layout

Chapter 1, which is this chapter, contains the introduction and a brief history of neural networks and optical character recognition.

Chapter 2, is a review of related literature and research. This chapter is a survey of the research in the field of OCR. Full understanding of each topic synopsis is not necessary during reading. Many of the points brought out in this section are fully described in the remaining chapters of the thesis.

Chapter 3, is about neural network analogies and their description.

Chapter 4, comprises a description of an object oriented backpropagation simulator.

Chapter 5, comprises a description of feature extraction and applications.

Chapter 6, comprises a description of the final analysis and conclusion.

CHAPTER 2

LITERATURE SEARCH

2.1 Review of Related Literature and Research

Research in OCR and related systems is an intensely studied research area. Hundreds of researchers are working on producing solutions to solve problems related to this large domain. The following synopsis will explore the current research in OCR research. A focus on more recent work, covering the last 3 years is reviewed in this section. Some of the following works reviewed may not be specifically focused on OCR, but instead some other two dimensional object recognition techniques. If the techniques are relevant and could be applied as a feature extraction method it will be included. Artificial neural network research will also be explored.

The first collection of references are papers closely related to this thesis and have a few paragraphs summarizing the work. Following these descriptions are papers that have relevance in certain areas of this thesis such as feature extraction or image processing, and these papers are treated with one or two sentence descriptions.

Complementary Algorithms for the Recognition for Totally Unconstrained Handwritten Numerals[15]

This paper presents some relevant issues, in recognition of handwritten numerals. Research on unconstrained handwriting applies to handwritten mail sorting, and handwritten check processing. Two feature extraction techniques are combined to investigate how they complement each other. Method #1 uses a skeleton of a character pattern by decomposing it into parts called branches. Then features are then extracted from the branches and classified. Steps in this procedure are skeletonization, decomposition, vectorization, reduction, feature extraction and classification. Skeletonization has been a problem due to noise spurs and skeletonizing induced errors. The authors here have introduced a new method for skeletonization that minimizes the normal problems. An unpublished paper "A Dynamic Shape Preserving Thinning Algorithm"[16] will describe this.

Decomposition determines the endpoints of each branch by examining neighboring pixels elements with 1 or more neighbors. From the branches a vectorization process is implemented assigning a direction code to each pixel in a branch. Feature extraction consists of creating a set of features to describe the branch: shape, length, angular change, degree of curvature, vertical and horizontal general directions, the nature of start and end points, coordinates and the distance between them.

Classification is performed by 11 modules, one of which is dedicated to single branch patterns. Each other module is a specialized recognizer for examining one of the ten digits.

The second method presented in this paper uses structural features extracted from the contours of the digits. Key aspects of this technique are: combines edge smoothing, contour extraction, and simple measurements on holes (area and minimum bounding box) in a single pass over the binary picture. This technique extracts structural features from contours. First edge extraction is performed using a sequential tracking algorithm, with this information contours are produced which are output as a list of (row, column) coordinate pairs. Collection of features from the outside contour are put into the following types, cavity, bend, and endpoint. These types are determined based on the radius bend and length of the contours. Results from the holes and outer contour features are then sent to a classifier.

Experiments were run and measured for the individual techniques and then combining both techniques. Results show a 3-6% accuracy increase by using a combination of the two techniques. In the authors' conclusions he mentions looking into voting schemes and other integration techniques. The individual methods here produced good results and are fast. Neural networks were not used for classification.

An Interactive System for the Selection of Handwritten Numeral Classes[17]

Fourier descriptors are presented in this paper as a feature type for handwritten numerals. Classification is based on a similarity definition introduced by Banach algebra of continuous and bounded plane curves. It is mentioned that the ability for humans to recognize isolated characters is 97%. Elsewhere 96% is used; in either case this has been accepted as the upper limit for machine interpretation or recognition of hand printed characters. The system

proposed here allows the user to define the classes for the different ways that humans represent digits, for example a '7' and a '0' can be represented with or without slashes through them. Variations of the digit '4' and '6' are also wide and distinct for images that represent the same value. To classify the handwritten element four processes are applied: C • DFT • F • IDFT.

C is a sampling function
DFT is the Discrete Fourier Transform including Fourier descriptors computation
F is a five point rectangular window low-pass digital filter
IDFT is the Inverse Discrete Fourier Transform

The conclusion states that method is currently under investigation and experiments are in progress. The DFT's seem to be somewhat limited by unique feature data. This is a method worth further consideration.

Off-Line Recognition of Handwritten Cursive Script for the Automatic Reading of City Names on Real Mail[18]

This work focused on using the easiest recognizable parts of an address, the zip code and state, in a contextual assistance mode. By using the easily recognized zip code to help validate the state and then using that information to help determine the city. Of interest to this thesis is the method used for feature extraction and classification of the segmented data.

First all the segmented objects are stored in a 30x30 matrix. The matrix is the output of the segmenter. After the matrix is filled the first measurement is to check to see if a character can be reached; this is performed a concavity measurement at each white pixel and testing in the eight possible directions for a pattern. The higher the number of directions leading to the character, the more closed is the concavity observed for the white point. These vectors are then used to perform recognition of the character. Each character contains one or more concave areas around it. This technique uses the number, location and size of these shapes as the features. Using a statistical method to classify the features results in 60-70% accuracy of the zip code and this includes context information about the values. This is a unique feature type that is useful for shapes with arcs such as cursive script as studied by these researchers.

Handwritten Zip Code Recognition with Multilayer Networks[19]

Accuracy of these experiments were in the 92% range. One of the major interests in this paper is that no feature extraction is used. Raw bit map images are sent into multiple layer and multiple type of artificial neural networks. The complexity of these networks is significant and probably is the reason so many individuals were involved in this project. Using raw bit maps as a non-feature input type to a neural network is definitely worth using as a baseline when comparing feature driven simulators with each other.

Error Correlation in Contemporary OCR Systems [20]

Errors and their cause in OCR are studied to produce a better understanding of where improvements should be made. Recognition problems identified by this paper:

• features due to shape of characters employed: ligatures, stylized fonts, serifs.

• factors affecting horizontal and vertical spacing: subscripts, superscripts, proportional spacing, kerning.

• deviations introduced by the production and reproduction processes: broken characters, smudged characters, speckle, skew, projection distortions.

• other marks added to the page subsequent to printing: underlining, highlighting, annotations.

The work focused on using a Calera RS9000 (considered the best text conversion system available for English text) and simulating the sources of error and measure the systems effectiveness with errors introduced. Introduction of this work is for understanding the types of error that can occur in OCR systems.

The use of a trie structured dictionary as a contextual aid to recognition of handwritten British postal addresses[21]

The main focus of this paper is determining British postcodes by use of combining syntactic and contextual post-processing with a statistical character recognition algorithm. For feature extraction of the images the Enhanced Loci Algorithm is used. It has three stages:

1) Feature extraction, each white pixel is characterized by the black pixel regions that surround it in each of the 4 directions. Black points are labeled as being a part of a horizontal, vertical or slanting edge, or being completely enclosed by other black pixels.

2) Feature unification, incorrectly labeled pixels are corrected, typical problems are usually due to 'noise.' For example a single isolated black pixel may either be removed or assigned part of a larger neighboring object. In the opposite case a black object may have a white noise speck in it that should be changed to black.

3) Feature concentration, this stage combines feature codes of each pixel with the feature codes of the surrounding regions.

Problems with the technique are that extremely large number of classes of required. In the example given there are 65,000 possible feature codes. It seems to me that this method is too complex to resolve all possible problems in OCR. Other points that caused problems in the British postal code are that alphabet and numeric characters are both used in the ZIP code. Consequently the following characters were confused during the classification process: '8' and 'B', 'D' and '0', '5' and 'S', 'U' and 'V', '2' and 'Z', and 'Q' and 'O' with a second subclass of confused characters being 'M', 'N', 'H' and 'W.' Without context information many of these classes are nearly or are identical in feature space. Humans also would have problems recognizing them without context. Basic recognition rates were 63% for the entire alphanumeric character set.

A Structural Character Recognition Method Using Neural Networks[22]

While this paper is aimed at machine printed fonts it presents some useful insight on hidden nodes and how errors are mapped to them as a function. The feature extraction method is as follows: a) bit map; b) thinning by medial axis transformation (MAT); c) polygonal approximation of the MAT; d) representative polygonals when distortion is corrected; e) decomposition in terms of circles and arcs; f) creation of an Attribute Relational Graph (ARG) and h) complete the grammar.

ARG attributes will be I) the arc's relative size compared to character overall size; II) the measurement of the angle subtended by this arc; and III) the direction of the arc's normal on the concave side.

Recognition rates varied from 93.5 to 95%, depending on the number of hidden nodes used in the neural network which varied from 10 to 70. Using more hidden nodes decreased the misclassified results dramatically dropping to 1% with 70 nodes. The expense of training time increases linearly with the number of hidden nodes.

Neural Net OCR Using Geometrical And Zonal-pattern Features[23]

Three feature types are used in this paper, Feature based on Smoothed image (FS), Feature based on contour Direction (FD), and a geometrical feature, Feature based on Bending point (FB).

FS: The FS method is a typical zonal-pattern feature which is sensitive to the position of black pixels in a two dimensional space and can tolerate local distortions but is weak when discriminating between characters which have similar local features.

FD: Features based on contour directions, each detected direction is counted four times dependent upon its four projections: vertical, horizontal, and to diagonal angles. This feature is sensitive to the local direction of strokes as well as the position of strokes.

FB: Feature based on bending points uses the image trace along the curvatures determining either concave or convex bending points.

This evaluation into three types of feature extraction methods introduced here produced recognition rates between 80-95% depending upon the number of errors accepted. Each of these feature spaces may be useful and combined by a higher level system for increased contextual analysis.

Design of Supervised Classifiers using Boolean Neural Networks[24]

In this paper a method of using a boolean neural network is introduced. Classifier methods are Nearest-to-an-Exemplar (NT), and a Boolean K-Nearest Neighbor (BKNN). Advantages of the simulators presented are that they learn or are trained in one shot, and

recurrent instability is not an issue. Required mathematical operations are extremely simple, needing only binary multiplication, integer addition and comparison to be performed. The American Bankers Association (ABA) E-13B font was used for testing. Results ranged from 92-100% for this fixed font size machine printed typeface.

Supervised Competitive Learning Part I: SCL with Backpropagation Networks[25]

The goal of this work is to create a system which will both learn new prototypes continuously, using Supervised Competitive Learning. This technique is based on using Adaptive Resonance Theory (ART) and backpropagation. Pen based computing is the target computing and the system is in the learn mode continuously. The author's example of how this would work is if certain prototypes already exist for the digit '7' and suddenly a new type of '7' is written, then the user would be asked to verify the character they just wrote by typing in the correct value. By building a subclass prototype of the new '7' it could then be selected by a selector when testing for classifications of shapes similar to '7.'

Extremely brief treatment is given to the features used, there are two types: 30 energy-based features, and a combination of graphical and energy type features (60). No explanation is given for these either of the feature types. In the conclusion success rates are 99% for digits and 96% for letters. Applications in pen-computers are being pursued by this system.

Analog Electronic Neural Networks For Pattern Recognition Applications[26]

This work develops microelectronic neural networks for pattern matching. In particular this device was created for reading ZIP codes for the U.S. Postal service. Electronic neural networks have been experimented with since the 1960's. Recently the reduction of size and the understanding of neural networks have led to a new interest in developing hardware to perform the simulation of neural networks. In 1988 at a conference 50 designs of hardware were presented to perform simulation. Most of the paper describes the electronic implementation of neurons and networks. Applications of the device is covered at the end of the paper, it is of particular interest since it deals with handwritten characters from ZIP codes. Steps of interest which are used in this implementation are binarization, deskewing, skeletonization, and feature extraction. Skeletonization is performed by scanning 5x5 pixel windows across the image, 20 25 bit templates are compared to the image, optimizing for noise the best selection which leaves one pixel width of the stroke is used. The templates were created in a systematic 'ad hoc' fashion.

Feature extraction is accomplished by applying 49 different templates which are created to detect features in a 7x7 pixel window of the original image. Templates are used to test for orientation of the line, line end points, and arcs. If the conditions of the template are met a 1 is

set in the map for the corresponding feature. For classification a backpropagation neural network is used.

Unconstrained Handwritten Character Classification Using Modified Backpropagation Model[27]

This paper provides a very good insight to performing Fourier Descriptors and Hole features. An introduction of the backpropagation artificial neural network is given with no architectural examples with the described features. However this paper closely represents the entire picture of recognition and classification in one instance. The authors point out the problem characters associated with Fourier Descriptors. Outline energy information of the following character pairs are not significantly different for classification: '1' and '0', '7' and '9', '8' and '1.' Though not described in detail it is mentioned that additional attributes are created for each characters Fourier Descriptor feature vector, in particular a hole topology is described as being a valid discriminator to be used with this problem.

Constrained Neural Networks For Pattern Recognition[28]

The ensemble of all possible network configurations compatible with a fixed architecture is explored to define a probability distribution over the space of input-output maps. Such distribution fully describes the functional capabilities of the chosen architecture. Its entropy measures the intrinsic functional diversity of the network ensemble[29].

Material is presented as follows: brief description of single neuron processing, layered neural network architecture, supervised learning, optimization problem, emergence of generalization ability, receptive fields with local connectivity, shift invariant feature detectors for a digit recognition problem and use of architectural constraints to achieve generalization

Section 7 presents three combinations of networks for study. The input pattern is a 16x16 binary pattern of digits. The authors develop a metric for measure of the generalization ability '%G.' For example one, three layers are defined N_0, N_2 and N_3. N0 and N2 are the input and output layers. Network configuration format is (INPUT Rows x Columns-Hidden layer 1- Hidden layer 'n'-OUTPUT). The first configuration is 16x16-12-10. To calculate the generalization ability the following steps are developed. Through the authors calculations this networks generalizes to 87%. Three sample architectures are examined. A summary is given in Table 2-1.

Table 2-1 Generalization comparison of different networks

Network	Generalization
1) 16x16-12-10	87%

2) 16x16-64-16-10	89.5%
3) 16x16-128-16-10	94%

This technique makes sense for digit recognition tasks and would really benefit from an example or architecture test. There are no quantitative tests of actual networks indicated in this paper.

A Structural Approach to Automatic Primitive Extraction in Hand-Printed Character Recognition[30]

A procedure for automatic primitive extraction in character recognition is presented. A 2 column by 4 row, 8 cell grid is laid over the original image. Each cell in the grid is examined for a specific primitive pattern which can be classified. Examples are: vertical stroke, horizontal stroke, left and right arcs and others. Once these primitives are classified a syntactic context free parse tree is constructed. Each primitive leads down a specific path in the search tree until a match is located.

This researcher describes the data set used for testing but no results of accuracy or success for the ZIP code data tested. Digits 0-9 were recognized properly is the closing comment.

Primitives implemented here are very similar to template matching as mentioned by other researchers. Benefits of primitives are they provide a unique classification system and allow a great deal of raw data compression, thus reducing the size of the syntactic parse tree.

One View of On-Going Problems in Handwriting Character Recognition[31]

This author is providing a devils advocate viewpoint. Rather than exploring the research issues, he pins down the engineering problems related to this topic. For completeness and an alternate viewpoint this paper is worth reading. The following points are factors he presents as an impediment to solving the problem:

- Local variations in writing style for the same character.
- Social and generational differences in writing styles taught for the same character.
- Definitions and acceptance in the representation of characters varies.
- Psycho-motoric factors, such as fatigue affect writing styles.
- Constraints during the writing event, short notes which are quickly written versus well printed applications for a drivers license.
- Humans can not even recognize many cases of hand printed characters out of context.
- Digitization techniques introduce errors which are beyond control of interpretation techniques.

For the most part these points are correct and must be addressed, still useful systems are now being implemented which are overcoming many of the above issues and research is

ongoing to resolve others. His viewpoint is as if he was designated to design a perfect, low cost product for general purpose hand printed text recognition. Many of his arguments point out flaws in commercial attempts to market the technology.

Reading Unconstrained Handwriting with Bounded Context[32]

Even though there has been a great deal of research in low level recognition techniques not enough has been done using the available context for the given problem. In this paper a number of points about context are highlighted:

- Recognition and Comprehension are separable with constrained writing.
- Unconstrained writing requires contextual (linguistic and spatial) knowledge.
- Many practical problems involve unconstrained writing and bounded context.

Other points are made with specific reference to the research that Dr. Srihari is performing for the United States Postal Service. This introduction to contextual knowledge is very important as Dr. Srihari describes it. Still the basic recognition modules must be mature enough to have confidence in them for employment in higher level decision systems.

Multiple Algorithms for Handwritten Character Recognition[33]

ZIP codes are the domain of this paper, in respect to unconstrained text recognition. Three methods for features are discussed, holistic, contour, and structural. The justification for the first technique which is template matching is that a large number of classifications will fall into this class, each image is normalized to a 16x16 grid and compared with prototypes. Claims are of 90-94% accuracy.

Structural analysis algorithm has been implemented and consists of a 5x5 grid which is used to determine the presence or absence of the following: horizontal, vertical strokes, hole, crosspoint, endpoint, and small and large concavity of the character. The 180 element output vector is then input to a Bayesian classifier which determines the best fit matches. Accuracy of 90-97% is claimed depending on the number of guesses allowed.

Contour analysis uses a 4x4 grid and determines the following primitives in each grid location: spur, stub, wedge, curl, arc, null, inlet and ray. Each of these primitives are curve or arc descriptions. This feature list is then input to a rule based system for analysis and best fit matching with existing prototypes.

These techniques are unique and interesting, since these methods were not classified using a neural network this may be a starting point for a feature type. It would be interesting to see how well they can generalize each class and be invariant to rotation, scale and translation.

A Thinning Method Based on Cell Structure [34]

Due to the inherent problems with typical thinning techniques; expensive computationally, difficult to determine global window size, spurious aberrations created during thinning and other documented problems, this paper explores a new cell based technique.

An Off-Line Writer Identification System Based on a Syntactic Approach[35]

This paper uses barycentres and superior order moments as discriminant features. After acquiring the digits by a 300 dots per inch (dpi), noise reduction is performed to remove noise, holes and other problems introduced during digitization.

Skeletons from chain-coded contours [36]

Using chain-coded contours and a simple connectivity test it can be determined if a specific pair of chain codes and be replaced by another set of chain-codes resulting in reduction of the image.

Knowledge-Based Understanding of Road Maps and other Line Images[37]

This research deals with high level understanding of primitives which are extracted from the road maps or line images.

Pattern Classification Using Teurons [38]

The methods researched here are Gödel encoding to "remember" the exemplar for each class.

Online Recognizer For Runon Handprinted Characters [39]

The main goal of this runon character recognizer which runs in real time is to recognize and then segment based on the best recognition hypotheses.

Recognition and Verification of Postcodes In Handwritten and Handprinted Addresses[40]

This paper presents a part of system being researched for the Post Office Research organization in the United Kingdom.

A Complexity Measure based Algorithm for Multifont Chinese Character Recognition[41]

This is an examination of multifont Chinese characters. Feature extraction is mentioned and is the only area chosen for review.

Detecting Parametric Curves Using the Straight Line HOUGH Transfrom[42]

Using the Hough transform as a method to detect and extract lines is well documented.

A General Approach for Parametrizing the HOUGH Transform[43]

This is a mathematical study only which develops new methods for mapping an image into a space of functions.

Performance of Parametric and Reference Pattern Based Features in Static Signature Verification: A Comparitive Study[44]

A feature technique; Reference Pattern Based Features (RPBFs) is presented; and an enhanced method is proposed to improve the original method.

Shape-Contour Recognition Using Moment Invariants[45]

Description of an object using moments has been successful many different objects in images. Specifically Zernike and pseudo-Zernike moment invariants are used for handwritten numerals.

Quantifying the Unimportance of Prior Probabilities in a Computer Vision Problem[46]

Examining entropy, Bayesian statistical approach and probability distributions for classification experiments with ZIP code recognition are studied in the paper.

Multi-Phase Recognition of Multi-Font Photoscript Arabic Text[47]

This system uses geometric features of the image with reference to a base line passing through the image. Image object width, height, height above baseline and a histogram are the features used.

Reading Newspaper Text[48]

This work which is funded by the National Science Foundation was performed at SUNY Buffalo. Dr. Srihari is one of the more popular researchers in text recognition.

Dot Image Matching Using Local Affine Transformation[49]

This paper describes an iterative technique for gradually deforming a mask dot image with successive local affine transformation (LAT) operations so as to yield the best match to an input dot image.

Generating Skeletons and Centerlines from the Medial Axis Transform[50]

Description of image objects by skeletons is popular because it removes a great deal of information which is redundant or confusing to classify.

Robust Description of a Line Image[51]

Determination of lines in a busy drawing.

Geometric Modeling of Digitized Curves[52]

Creating geometric representations of digitized or scanned images is discussed in this paper. Each contour or curve is vectorized and then fitted with a quadratic spline curve.

An Approach to Automatic Construction of Structural Models for Character Recognition[53]

The method created in this paper is one of learning. Creation of shape dictionaries automatically by quasi-topological features and singular points are used to generate classes and subclasses which can be merged and generalized. Features are created for this system by first thinning or contour tracing the image.

Performance Evaluation of Skeletonization Algorithms for Document Image Processing[54]

A very complete presentation of 19 different skeletonization algorithms are studied and explained.

A Hybrid Approach for Document Image Segmentation and Encoding[55]

A global approach to entire document recognition is presented.

Constrained Neural Network For Unconstrained Handwritten Recognition[56]

This is an adaptation of the paper mentioned earlier in this literature review. It provides similar information in a different format.

Accuracy as a Measurement Metric

One of the main areas of problems in researching all the work of others is that no one is testing against a standard database of characters. In 1992 National Institute of Standards and Testing developed such a test database. A competition was held against blind data and approximately 15 OCR companies performed the test and attended the conference. Through a standardized test much of the previous accuracy information can be benchmarked, the database will be available to researchers and companies for this purpose.

CHAPTER 3

BIOLOGICAL NEURAL NETWORKS

3.1 Neural Network Biological Metaphor

Most models of neural networks were derived from studying biological networks. Human biology is used to translate the concepts of the biological networks to the artificial neural networks. This metaphor is sometimes referred to as an Anthropomorphism:[57] *"2: ascribing human characteristics to nonhuman things"*. Though this metaphor can help to explain neural networks some researchers feel that this term is coupling biological research efforts too closely with the artificial implementation. "Connectionist" is the term referring to the artificial neural network research models, though neural networks and similar biological terminology will likely be used by both disciplines for the foreseeable future. This section will provide a description of the biological neuron.

The entity of a simple neuron is a nerve cell with all of its associated processes which make it function. Neurons in our bodies range from microscopic to as long as three meters. Depending upon the attributes used to classify them, a human has between seven and a hundred different types of neurons.

Figure 3-1 depicts a neuron from the retina. This type of neuron is bipolar, meaning it has two processes. Parts of the neuron include: Dendrites, which conduct impulses toward the cell body. Nucleus, this has one or more dendrites leading into it. Axon, conducts impulses away from cell body.

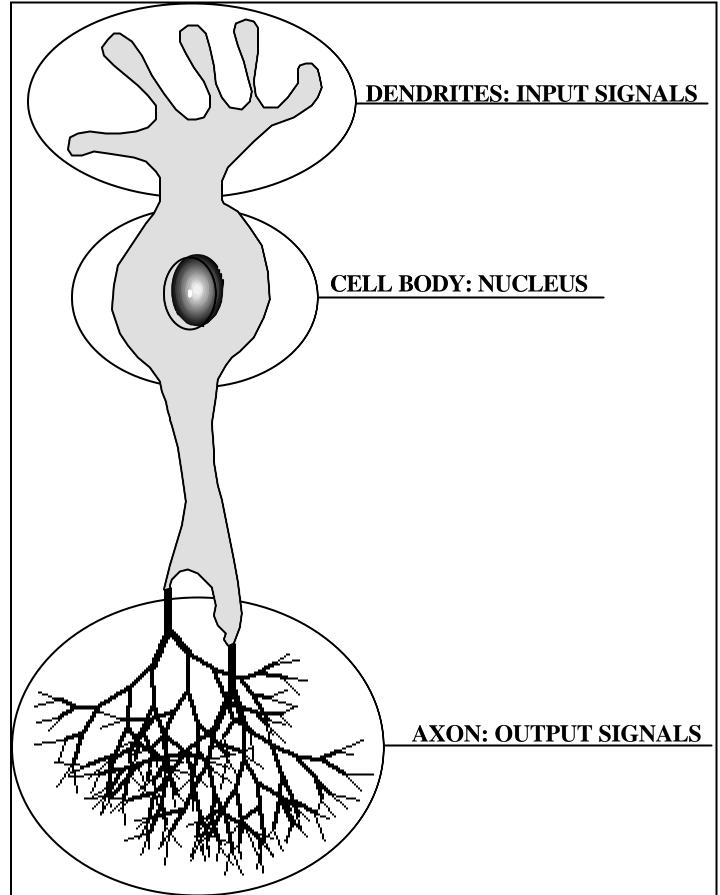


Figure 3-1 Simple Neuron.

Large collections of neurons formed in bundles make up what is called a nerve structure. Receptor organs such as the eyes, ears, nose, and throat create signals which are received by the dendrites. The impulses are compared against a threshold in the cell body; if this is met, then an impulse is generated and sent down the axon to the next cell of the bundle. This may be eventually tied to an effector organ such as a muscle or gland. When the impulse is transmitted between neurons it must cross a region named the synapse. The distance separating the two neurons will determine how much of the impulse is translated from neuron A to neuron B, see Figure 3-2. The transfer of the impulses occur only in one direction, and travel 390 to 4,700 inches per second. A typical sequence follows: dendrites are stimulated, the cell body

containing the nucleus electrical threshold is met, it then fires impulses down the axon to the next cell.

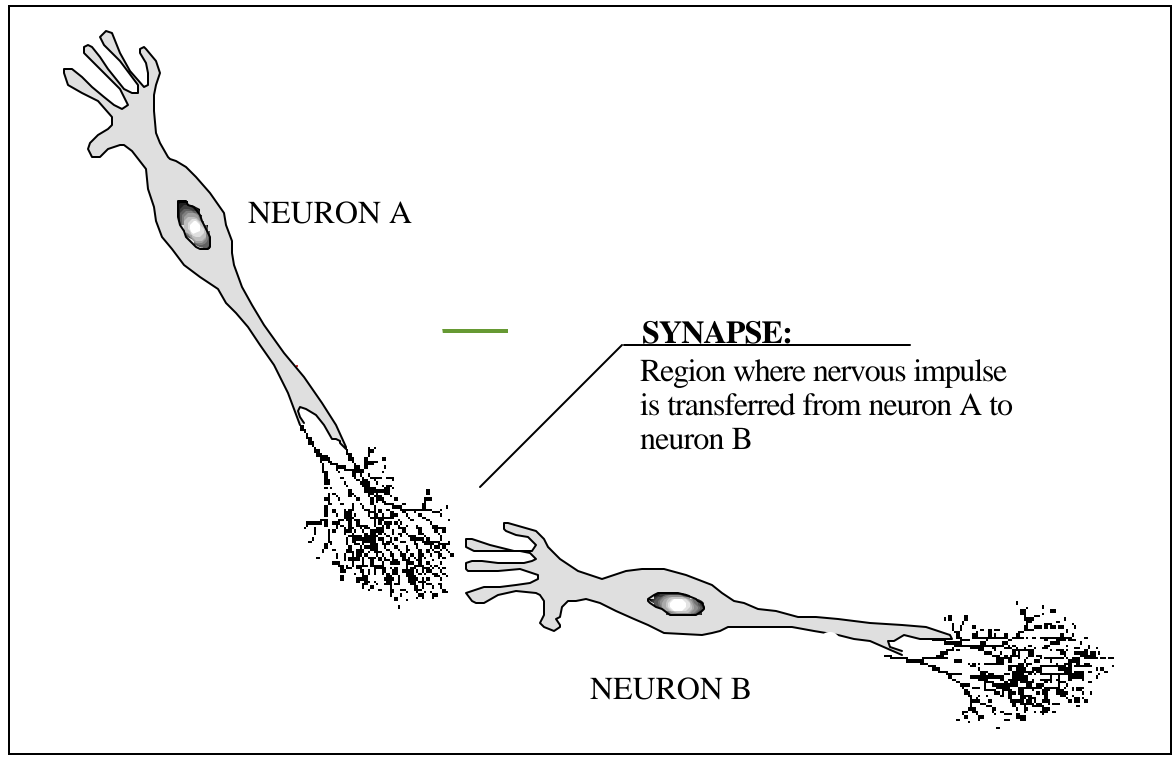


Figure 3-2 Example of a nerve structure.

The following points explain the analogy between biological neural networks and artificial neural networks. Signals that feed the synapse (neuron A output) are *weighted* by the number of nearby outputs from other neurons. This weighted value can be positive by having many other neurons contribute more energy, or negative by inhibiting the energy at the synapse. The net effect at the synapse is a sum of all other neuron outputs feeding the synapse. A specific threshold value is stored at neuron B which is equal to or greater than the sum of the input impulses. If this conditions exists, it fires or sends impulses down the axon to the next neuron or possibly to an effector organ. When the threshold is not met nothing happens. A simple analogy of this is when a person has their hand on a pan which is being heated on a stove. At time zero (t_0) nothing is sensed at some time $>t_0$ the heat energy is transferred to enough cells to sense the warmth. When time $>>t_0$ the energy exceeds the pain threshold for

the hand and an automatic response is generated to quickly jerk the hand from the heat source. When the transition occurs from room temperature to hot extremely fast, < 1 second by the time all the neurons fire and a person reacts, their skin is damaged.

Conditions which vary in the body such as oxygen, fatigue, and chemical deficiencies will alter the strength and impulse firings of the neurons. The synapse threshold and response time will vary based on the environmental conditions. Net effect of this is an integration over space and time of all the inputs with respect to their outputs of the system. Some of the features of the biological neural network are:

- Approximately 10 billion neurons.
- Maximum fire rate is approximately 1,000 pulses per second.
- Accepts inputs and generates responses to them.
- Self organizes unknown input data into new classes of information.

Neural networks do not operate as conventional computer algorithms, instead they have attributes such as, behavior, reaction time, self-organization, learning, generalization, forgetting and memorization.

The connectionist model will define the artificial neural network elements which are implemented in computer programs and hardware implementations. The neuron is translated as a node defined as a processing element. This processing element performs the following functions, evaluation of inputs, determine strength of each input, calculate a sum of all inputs and determine if they meet the nodes threshold value, if it does meet the conditions then an output signal must be generated.

Inputs and outputs: As in the biological model there are many inputs and one output for each processing element. Some inputs may be specialized providing a *bias term* or a *forcing term* in the connectionist model of a neuron.

Weighting factors: The energy transmitted from each previous axon into the current neuron has different strengths depending on distance, chemistry and function, the connectionist term is weight of the input. An input is scaled by the weight value for the specific input, this emulates a connection strength between neurons or in the connectionists model processing elements. Each input value is multiplied by each input weight, this is equivalent to a dot product. $Input_i = Input_i * Weight_i$.

Neuron functions: The result of the summing of all inputs multiplied by their weights creates a value which is tested against a threshold, if it meets the threshold an output value is produced.

Activation function: The previously summed result may be passed to an activation function which would determine the output of the summation based on other parameters such as time.

Transfer functions: The threshold value or transfer function is usually determined by a non-linear function. Problems with linear transfer functions were introduced in *Perceptrons*[58] where Minsky used the Exclusive-OR problem to demonstrate the concept. Many transfer functions exist and more are under research. A few common ones are shown in Figure 3-3.

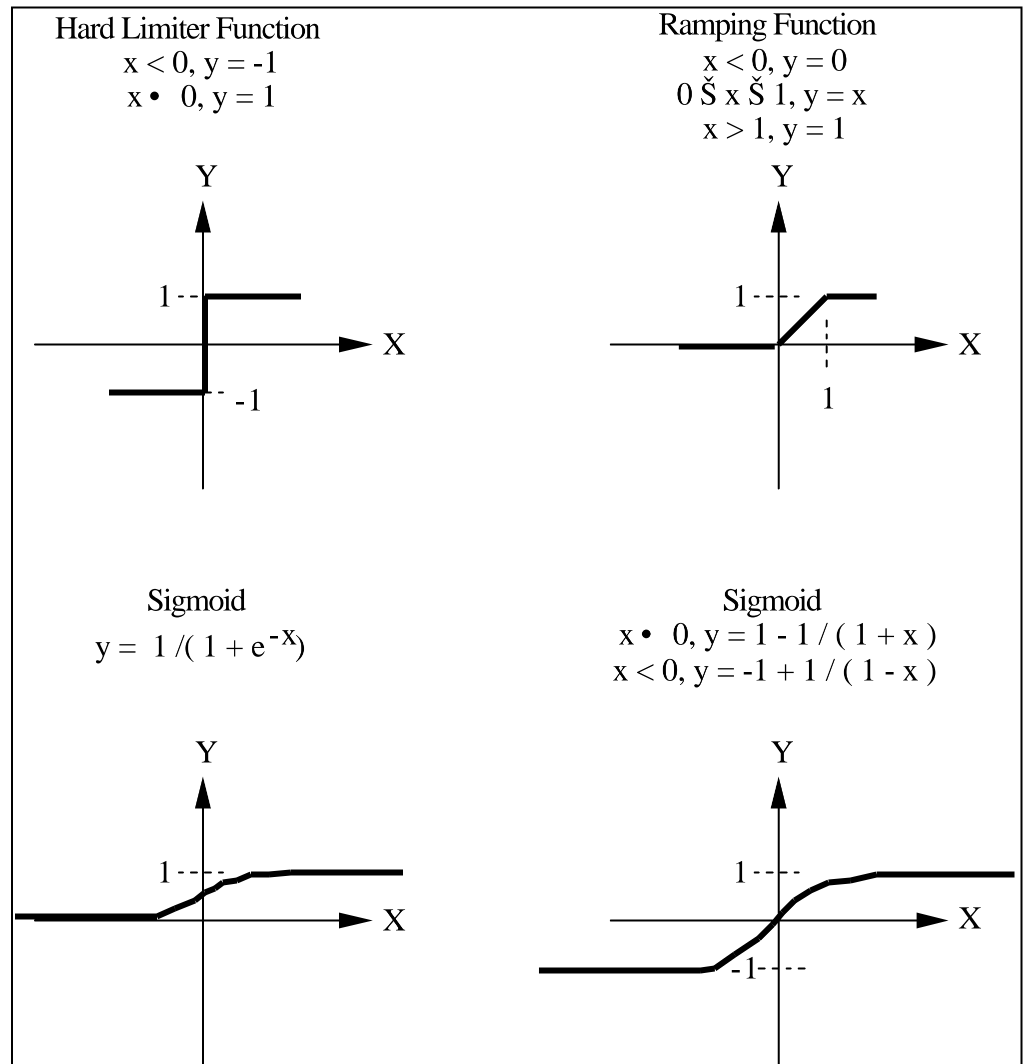


Figure 3-3 Sample transfer functions.

The above elements are the components for a single processing element. Table 3-1 contrasts the biological versus the artificial neural network terminology.

Table 3-1 Biological versus artificial neural networks analogies.

BIOLOGICAL NEURAL NETWORK	ARTIFICIAL NEURAL NETWORK
Stimulation levels	Input values
Synaptic strengths	Weights
Neuron impulse	Output
Electrical potential	Numerical value

Figure 3-4 is a composite processing element illustrating all the fundamental components. Note the output value is a single element, and can be connected to many other processing elements input nodes.

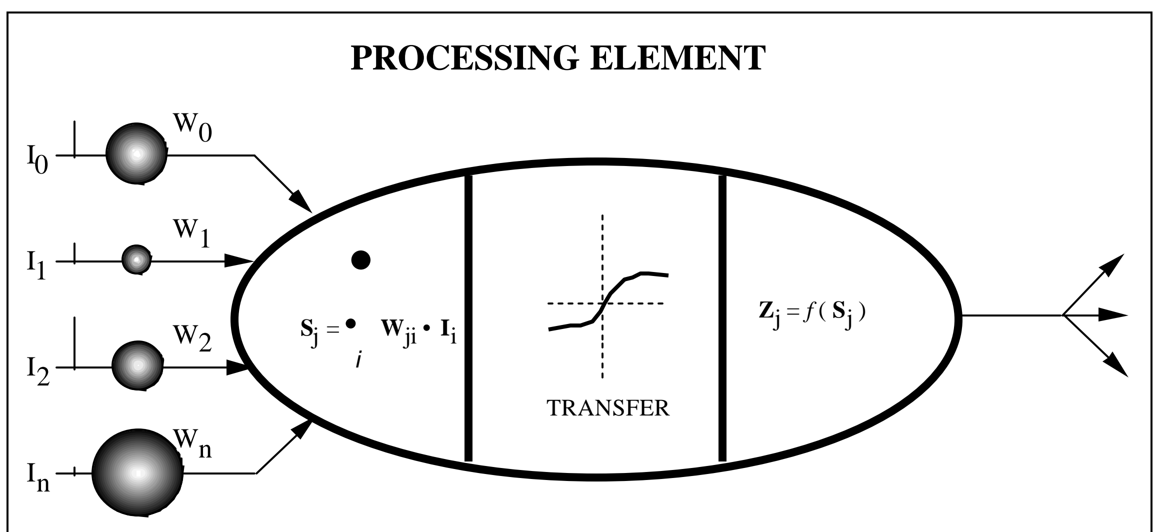


Figure 3-4 Processing element and associated parameters.

This defines the processing element, the next level of definition is combining the elements into a network.

3.2 Combining Processing Elements Into A Network.

A network is an arrangement of processing elements. All networks will contain an input and output layer. Some different types of networks may contain one or more hidden layers. Backpropagation networks as selected for this thesis may exhibit typical feedback system problems. Artificial neural networks are normally trained to measure the error from the input and expected output pair. This error is fed back into the network to provide a correction signal or value to update the network.

A collection of processing elements is formed to create a layer. The layers and processing elements between layers are connected by weights. Using the anthropomorphism, the connections which are biological axons and synapses are implemented by weights. Usually a network will contain at least one hidden layer. Neural network topology varies widely depending upon the application domain, Figure 3–5 illustrates a generic network architecture.

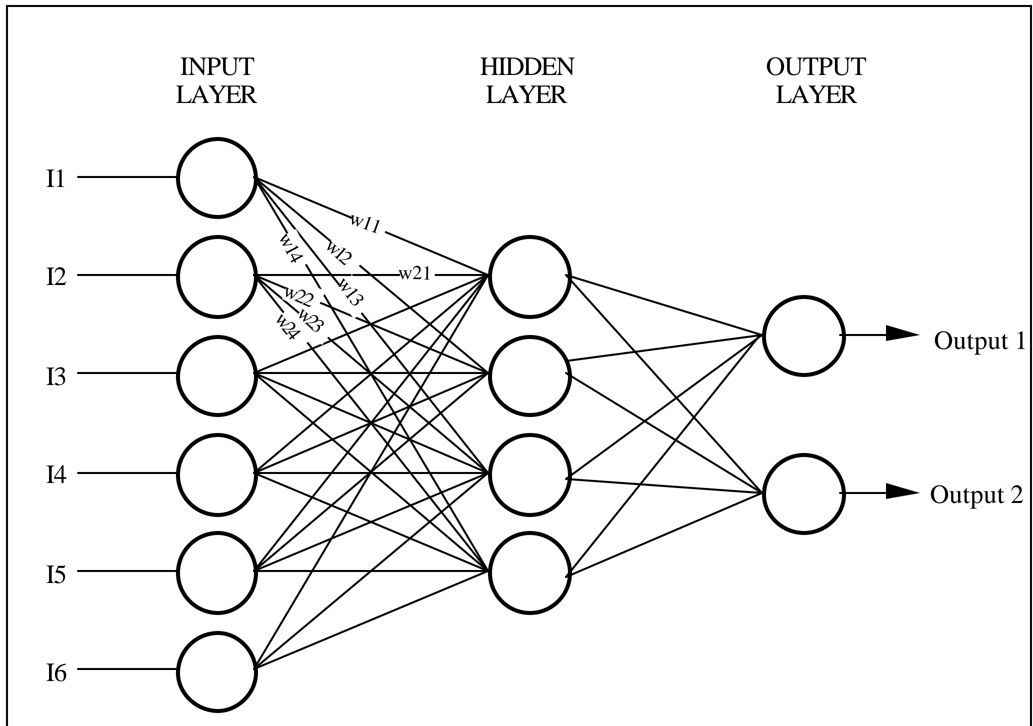


Figure 3-5 Example neural network architecture.

With this network the input layer receives the raw data values from the outside world. These values represent the external events to the system. The hidden layer is connected by weights which are connected from the input layer to the hidden layer. Values stored in these weights determine the strength of the connection. In this network since all nodes are interconnected with all other nodes it is said to be fully connected. In the output layer all connected hidden nodes are summed to create the value available at the output node.

Networks can be inter-connected in different methods to provide alternate learning patterns. Two categories are feedforward and feedback networks. Feedback networks that have completely closed loops are recurrent systems. The information fed back is the error difference between the expected output and actual output. Figure 3.6 shows a single node with feedback to itself which is the same level. Feedback can also be designed to go back to previous layers in the network. The network implemented in this thesis is of the multiple layer feed forward type. Historical reasons have labeled this type of network as a multi-layer perceptron.

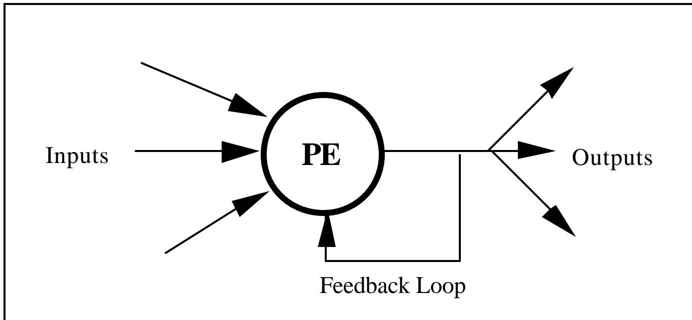


Figure 3-6 Single element with feedback.

Complexity of the neural networks is dependent upon many factors. Input nodes can range from 1 to 4,000, hidden layer nodes 1 to 1,1000 and output nodes 1 to 1,000 are some typical numbers for these layers. This chapter has reviewed the analogies between biological and connectionist models of neural networks. Since there are many types of networks and

hundreds of configurations for them, for purposes of this thesis the only model and network described in detail will be the backpropagation as implemented.

Training by backpropagation of errors can have flaws. The function used for minimizing the error steps in the direction required to reduce the error to a minimum is called a gradient descent function. Such a function attempts to minimize the distance or error between the actual and expected output vectors. Since we are always stepping in the best direction for reducing error we should converge to a minimum rapidly. The problems are in the parameters which control this learning; the learning rate and momentum may be to small. In this case it would take an extremely long time to converge, hundreds to thousands of hours in some cases. The opposite and equally serious problem is when the step size is too large and the function jumps over and back and forth about the local minimum. One last problem is that a local minimum of the function may be in a mathematical function valley and the real minimum may be just over the next hill. This hill climbing problem will cause the convergence to stop at a false or less then optimal function. Figure 3-7 is a diagram of the local minimum problem.

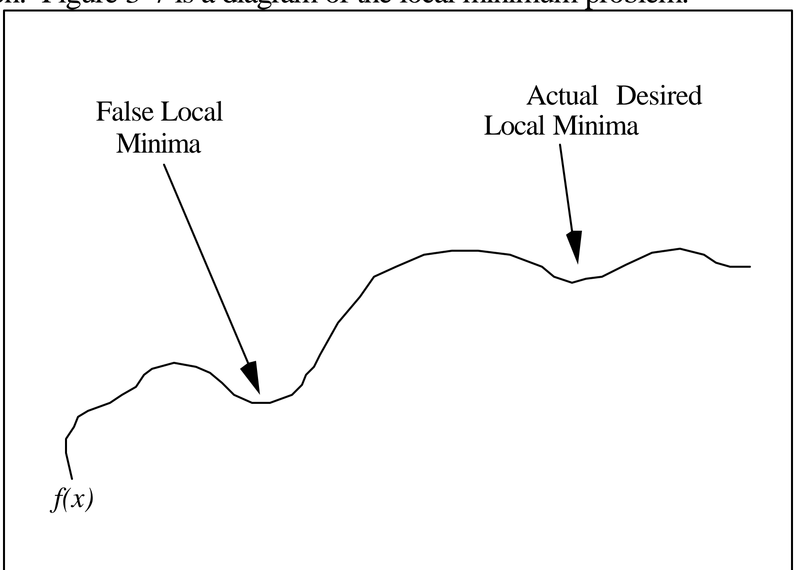


Figure 3-7 Local Minimum Diagram.

Table 3-2 provides a summary of well known neural network models.

Table 3-2 Well Known Neural Networks.

Type	Developer	Year	Applications	Notes	Disadvantages
Adaptive resonance theory	Gail Carpenter, Stephen Grossberg	1978-1986	Pattern recognition, (radar/sonar voiceprints)	Sophisticated; very few applications	Sensitive to translation, distortion and scale
Avalanche	Stephen Grossberg	1967	Continuous-speech recognition, robot commands	Class of networks, no single network can do this	Difficult to alter speed, or interpolate movement
Back Propagation	Paul Werbos, David Parker, David Rumelhart	1974-1985	Speech synthesis from text; robot arms; bank loans	Powerful, predictable	Supervised training only, need lots of input and output samples
Bi-directional associate memory	Bart Kosko	1985	Content-addressable associative memory	Easy to learn, associates pieces of data with complete data	Low storage density; data must be coded
Boltzmann & Cauch machines	Jeffrey Hinton, Terry Sejnowsky, Harold Szu	1985-1986	Pattern recognition for images, sonar and radar	Simple networks, noise function used to global minimum	Long training time
Brain State in a Box	James Anderson	1977	Extraction of knowledge from data bases	Similar to bi-directional in completing fragmented inputs	One-shot decisions, no iteration
Cerebellatron	David Marr, James Albus, Andres Pellionez	1969-1982	Control motor action of robotic arms	Similar to avalanche	Requires complicated control input
Counter-propagation	Robert Hecht-Nielsen	1986	Image compression, statistical analysis	Self-programming look-up table,	Many PEs and connections required for high accuracy
Hopfield	John Hopfield	1982	Retrieval of complete data from fragments	Can be implemented on a large scale	Does not learn, weights must be set in advance
MADALINE	Bernard Widrow	1960-1962	Nulling of radar jammers; modems; phone equalizers	In commercial use for > 20 years	Assumes linear relationship between input and output
Neocognitron	Kunihiko Fukushima	1978-1984	Handprinted character recognition	Most complicated; insensitive to scale, translation and rotation	Requires many PEs and connections
Perceptron	Frank Rosenblatt	1957	Typed character recognition	Oldest network, built in hardware	Sensitive to scale and distortion

Self-organizing map	Teuvo Kohonen	1980	Maps 1 geometric region (grid) to another (aircraft)	More effective than many algorithms for aerodynamics	Requires long training

This table is an adaptation of a table in "Neurocomputing: picking the human brain[59]."

CHAPTER 4

ARTIFICIAL NEURAL NETWORK SIMULATOR

4.1 Artificial Neural Network Simulator Description

Many types of technology have been applied to pattern recognition, in both hardware and software implementations. If the power of the human brain could be built into machines which would be able to perform the same pattern-information processing capabilities that we possess, making machines to handle real world tasks would be much easier. Solving problems in various domains usually requires the task to be scoped based on available knowledge which is available to represent the problem.

Issues of particular concern with this thesis is a metric which will allow measurement and analysis of the complexity of the artificial neural network and its training and accuracy performance. These methods are developed in this section and used later during the performance and testing phase of the various features in Chapter 5. Use of the network and measuring the ability of the configuration to classify the data with minimum training time is another goal pursued in this section. To that end another metric is defined and used in Chapter 5 for proving the measures effectiveness.

Artificial Neural Network Simulators (ANNS) have been proven among the best techniques of pattern classification. Table 4-1 shows a spectrum of issues involving the domain knowledge and issues related to it.

Table 4-1 Domain knowledge spectrum.

DOMAIN KNOWLEDGE

HIGH ⟷ LOW

	NUMERICAL ANALYSIS	RULE-BASED SYSTEMS	CASE-BASED REASONING	NEURAL NETWORKS
	ALGORITHMS	FUZZY LOGIC	CASE BASED REASONING INDUCTION	PATTERN RECOGNITION
TARGET DOMAINS	FINITE RELATIONSHIPS CLOSED WORLD	COMPELX RELATIONSHIPS SOMEWHAT BOUNDED WORLD		RAW, SIMPLE DATA LOTS OF DATA POINTS
TRAINING	KNOWLEDGE STATIC NO LEARNING	MINIMAL EXPLANANTION LEARNING DIFFICULT	RICH EXPLANATION LEARNING AUTOMATIC	NO EXPLANATION
PROPERTIES	PRECISE ANSWERS LIMITED INPUT	DIFFICULT MAINTENANCE	NEED EXPERT TO CREATE CASES	COMPLEX ARCHITECTURE, PROLONGED TRAINING

Considerable research has been invested in studying various different type of artificial neural network architectures. As described earlier in the neural network history chapter, network architectures are being created to solve specific domain problems. Problem domains can be mapped into specific architectures of neural networks. Different networks are created to solve a variety of problems such as: prediction, classification, data association, data conceptualization, data filtering, and optimization.

For purposes of this thesis the multilayer perceptron backpropagation architecture will be implemented. It has been shown to be very effective in classification problems. Its ability is to classify input patterns by training the network to learn all possible classes from a training set. Part of the implementation is using C++ and object oriented techniques to create a set of class libraries which can be reused and expanded by different neural network paradigms. This chapter will cover all aspects of the simulator as implemented for this pattern recognition effort. First the theory and mathematical operations will be explained and developed for

backpropagation paradigm of a neural network simulator. Secondly, the implementation of the simulator in an Object Oriented Programming (OOP) language will be discussed. Finally the simulator source code will be explained in sufficient detail to understand the framework and how other simulators could reuse classes developed here for other neural network paradigm implementations.

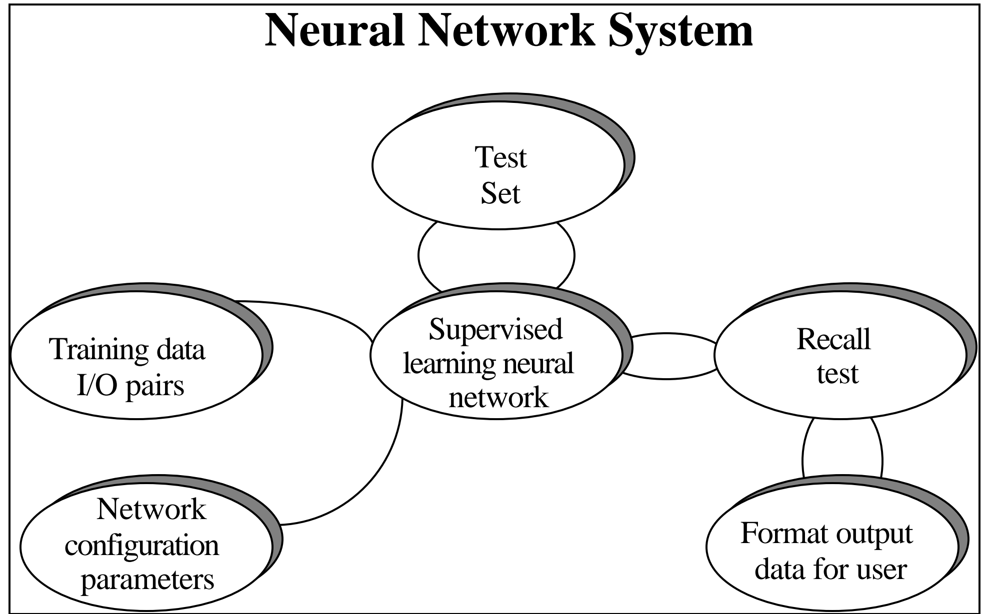


Figure 4-1 Neural Network System.

Figure 4-1 will be presented as an index to the area being studied; as each specific area is discussed it will be highlighted. The first element is the neural network simulator.

4.2 Backpropagation Network Mathematical Model

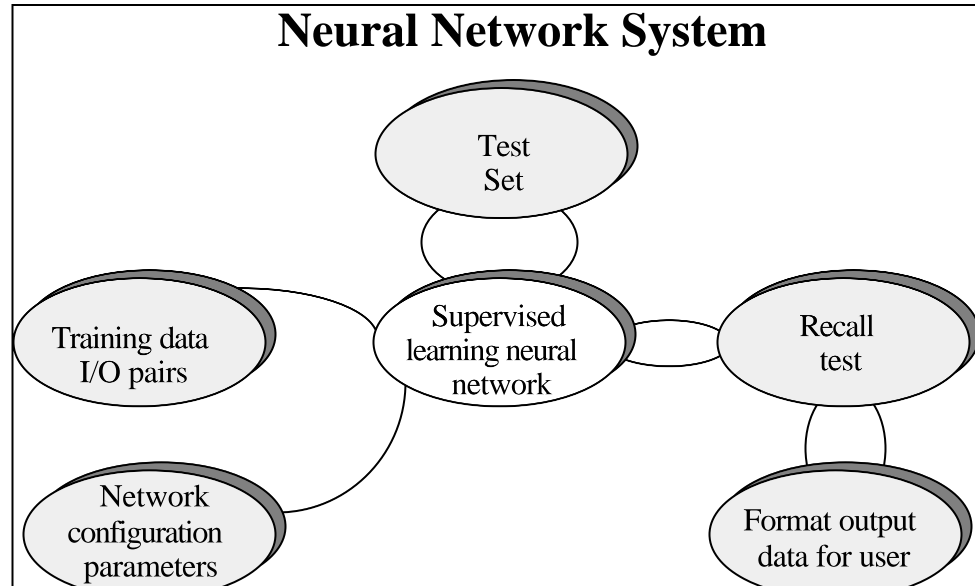


Figure 4-2 Neural Network System Simulator.

David Rumelhart and Paul Werbos are associated with the invention of the backpropagation simulator. Around 1985, David Parker introduced algorithms which have allowed this simulator paradigm to be implemented on computers. Backpropagation is a method which uses errors generated by misclassifications as error signals which are sent back into the system to correct it. The Backpropagation simulator may be abbreviated to BP. The BP network contains three layers of nodes, input, output, and one or more hidden layers. Each of the layers is completely or fully connected to the neighboring layers. Weights connect each element to all other in the succeeding layer. During learning, as input patterns are presented to the input nodes errors generated at the outputs are calculated and the weights are updated with the changes calculated for the current cycle.

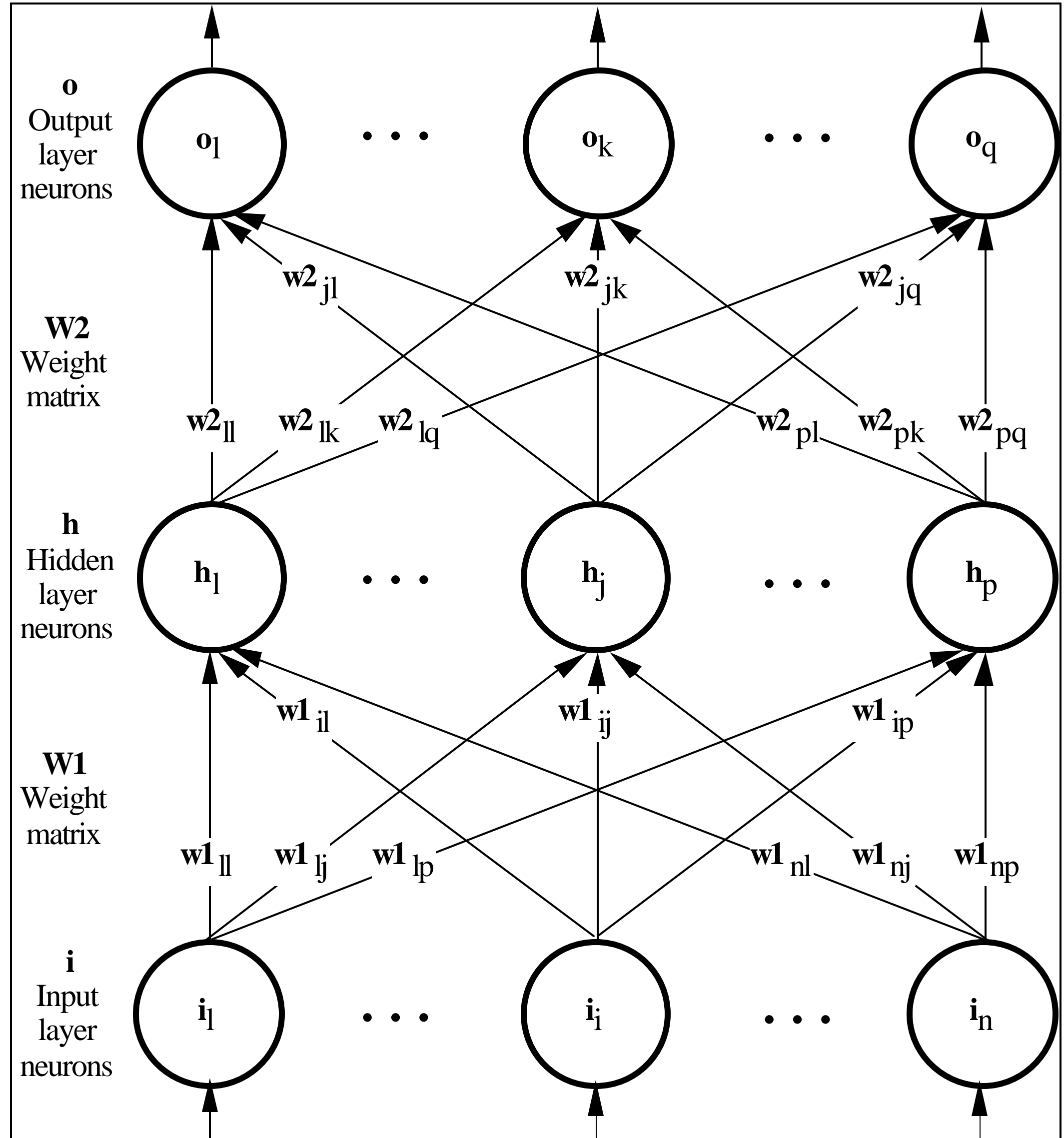


Figure 4-3 Backpropagation Topology.

Figure 4-3 depicts the backpropagation system described in this section. Each symbol in the figure which is in **boldface** type represents a vector in the actual implementation. A vector for these operations is a set of floating point values where each element in the vector represents a single node. The two weight matrix symbols **W1** and **W2** represent a two dimensional matrix. Many presentations of this material use the summation **?** symbol and define limits as being 1 and the maximum number of nodes in the layer. Instead of this method

vector notation is used, it may be a little confusing at first but once understood actually makes it easier to implement and maintain.

The following definitions are required:

α learning rate

Θ ... momentum factor, allows the previous weight change to influence the weight change in this cycle

$f()$ activation function, for this case the sigmoid function

$$f = \frac{1}{1 + e^{-x}}$$

c indicates current cycle of the network calculation

d vector of errors for output neurons

e vector of errors for each hidden layer neuron

i input vector for input layer neurons

h vector of hidden layer neurons

o output vector for output layer neurons

t target output vector, supplied in training mode

W1 weight matrix between the input and hidden layers

W2 weight matrix between the hidden and output layers

There are many descriptions and mathematically rigorous derivations of every backpropagation step. As presented the explanation has been formatted into a vector notation which is slightly different from the conventional description. Propagating the error generated at the output back through the layers prior to the output is the method in which this algorithm operates.

Figure 4-4 represents the programming model for the neural network simulator. The two matrices 'dW1' and 'dW2' at the top of Figure 4-4, are presented in the equations which follow as ΔW1 and ΔW2. The relationship of the vector data structures can be understood and related source code is presented in the Appendix A.

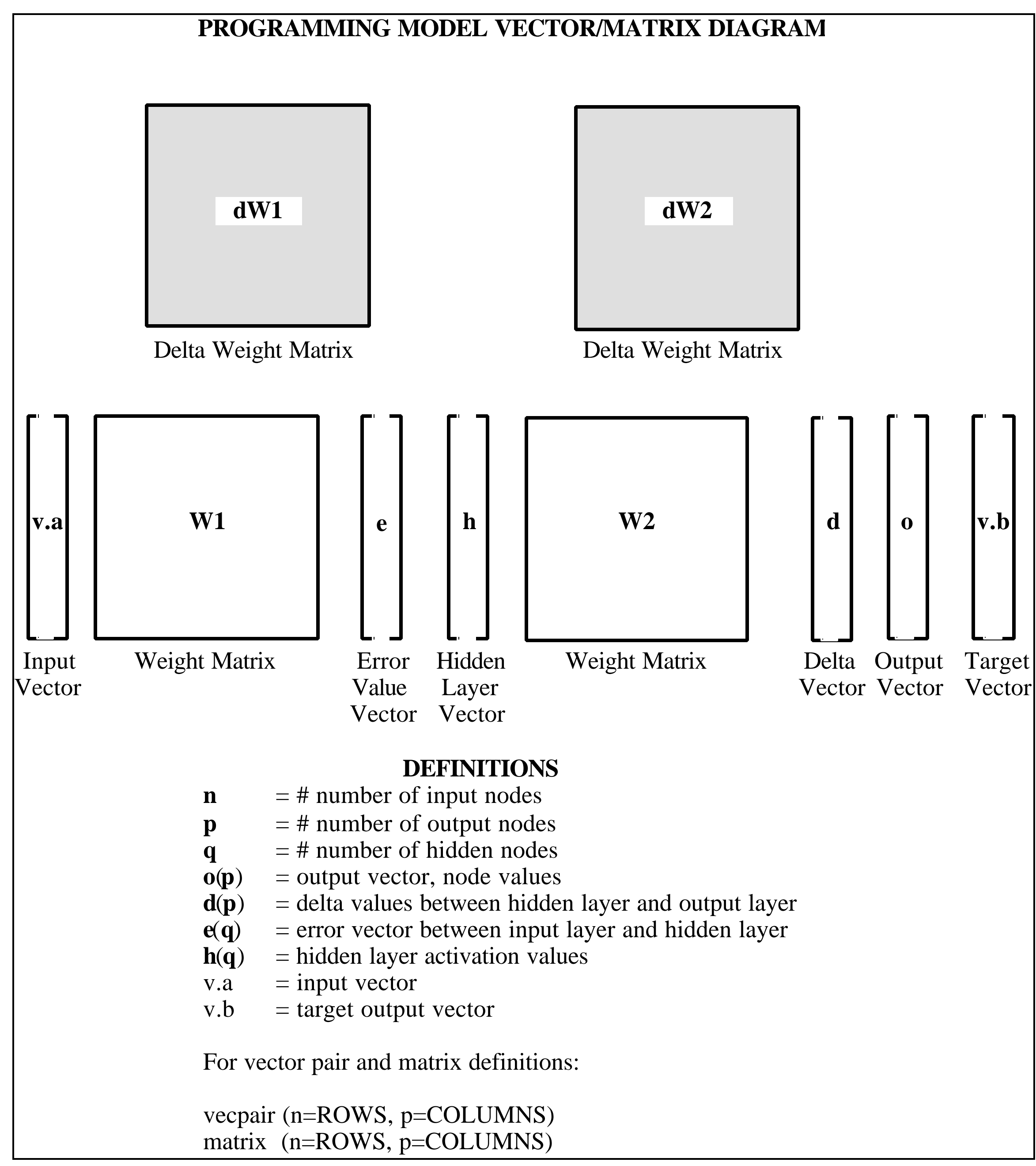


Figure 4-4 Programming Model.

The following explanation of the backpropagation algorithm is adapted from *Neural Networks in C++*[60], *NEURAL COMPUTING*[61] and *PARALLEL DISTRIBUTED PROCESSING*[62].

ENCODING OR TRAINING

Steps in a standard backpropagation algorithm for encoding or training on input, output pairs.

INITIAL CONDITIONS

The input vector **i** is presented to the network input layer with a corresponding or desired target output vector **t** (target) at the output layer.

FORWARD PASS

1 - Compute the hidden layer neuron activation values. This is a dot product between the input vector **i** and the **W1** matrix.

$$\mathbf{h} = f\ (\ \mathbf{i} \bullet \mathbf{W1}\ )$$

2 - Compute the output-layer neuron activations:

$$\mathbf{o} = f\ (\ \mathbf{h} \bullet \mathbf{W2}\ )$$

BACKWARD PASS

3 - Compute the output layer error, this is the difference between the target output and the observed output. **d** is the vector of errors for each output neuron.

'I' is a vector of the same length as the output vector containing all ones as elements.

$$\mathbf{I} = [1.0,\ 1.0,\ 1.0,\ .\ .\ .,\ \ 1.0]$$

$$\mathbf{d} = \mathbf{o}\ (\mathbf{I} - \mathbf{o})\ (\mathbf{o} - \mathbf{t})$$

4 - Compute the hidden layer error, this is the first derivative of the hidden layer error relative to the W2 weight matrix.

$$\mathbf{e} = \mathbf{h}\ (1 - \mathbf{h})\ \mathbf{W2} \bullet \mathbf{d}$$

5 - Calculate the weight change for the second layer of weights.

$$\mathbf{W2} = \mathbf{W2} + ?\mathbf{W2}$$

?**W2** is a temporary matrix where the delta is calculated by the following:

$$?\mathbf{W2_c} = \alpha\ \mathbf{h} \bullet \mathbf{d} + \Theta\ \Delta\mathbf{W2}_{c-1}$$

Symbol 'c' represents the cycle, the current delta change is based upon the previous cycle set of weights.

6 - Calculate the weight change for the second layer of weights.

$$\mathbf{W1} = \mathbf{W1} + \Delta\mathbf{W1}_C$$

$\mathbf{W1}_C$ is a temporary matrix where the delta is calculated by the following:

$$\mathbf{W1_c} = \alpha\ \mathbf{i} \bullet \mathbf{e} \bullet \mathbf{d} + \Theta\ \Delta\mathbf{W1}_{c-1}$$

Steps 1 through 6 are applied on all pattern pairs presented to the network until a specified tolerance value is reached. If the tolerance is specified as .1 or 10% then, the network would be fully trained when 90% of the pairs match.

RECALL OR RUN MODE

1 - Compute the hidden-layer activation:

$$\mathbf{h} = f\ (\ \mathbf{i} \bullet \mathbf{W1}\ )$$

2 - Compute the output-layer activation:

$$\mathbf{o} = f\ (\ \mathbf{h} \bullet \mathbf{W2}\ )$$

The vector '**o**' is the output.

4.3 Artificial Neural Network Simulator Computational Complexity Models

Development of the artificial neural network simulators involves testing different configurations and understanding performance tradeoffs for training and operation. Very limited information is presented in the papers reviewed in Chapter 2 relating to complexity or performance of the networks used for classification. Methods for analyzing the total number of network connections, a network complexity index and a performance measure of the number of connections that are updated each second will be developed and explained here. For

introduction to this analysis CASE 1, an Exclusive OR network, and CASE 2, a second arbitrary network will be analyzed, see Figure 4-5, and Figure 4-6.

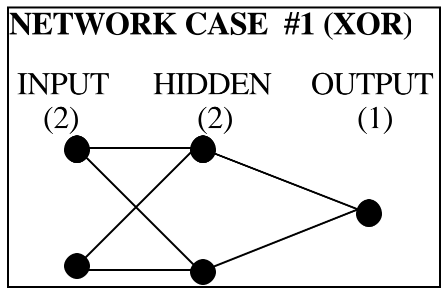


Figure 4-5 CASE #1 NETWORK.

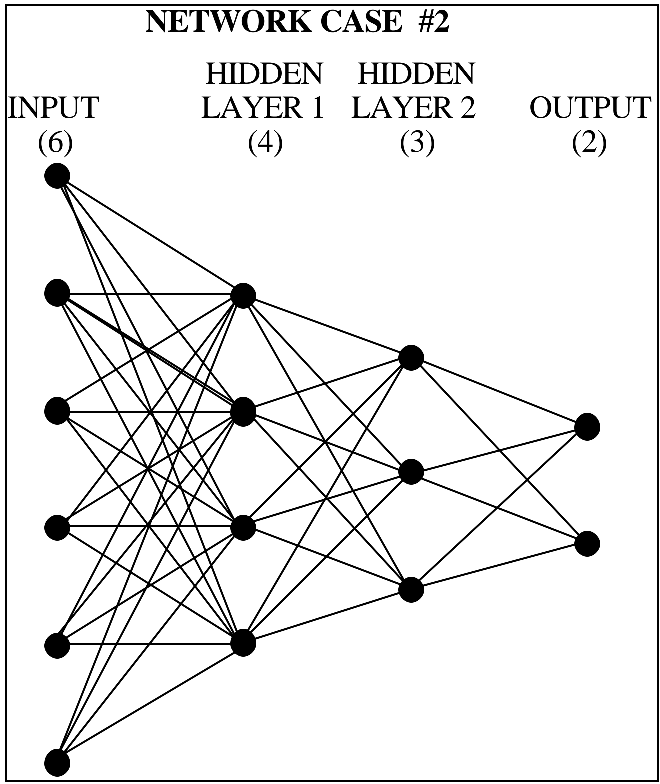


Figure 4-6 CASE #2 NETWORK.

The following metrics are defined:

NC Network Connections, total number of interconnecting weights between all nodes.

NCI Network Complexity Index, a value which represents the complexity of the network in relation to the number of outputs that can be classified.

CUPS Connection Updates Per Second, this measurement is based on the NC value and time. Performance based calculations are subjective to: the type of platform being used, the number of users on the platform and the loading at any given time of the platform. A value of CUPS is still one method of evaluating other systems running the same neural network topology. It can assist in the assessment of the performance of the neural network simulator simulation.

4.3.1 Network Connections (NC)

Network Connections for a one or more layer backpropagation neural network consist of the sum of all of the interconnecting weights between nodes. If only one hidden layer exists the middle bracketed term drops out.

I	Input Layer
H	Hidden Layer
L	Last Hidden Layer, layer number
O	Output Layer
n	Number of nodes at the given layer
l	Hidden Layer, layer

$$NC = (I_n \bullet H_n) + \sum_{l=1}^{\#\ hidden\ layers} \left[(H_{\ln} \bullet H_{\ln+1}) \right] + (H_{Ln} \bullet O_n)$$

CASE 1:

$$NC = (2 \bullet 2) + (0) + (2 \bullet 1) = 6$$

CASE 2:

$$NC = (6 \bullet 4) + [(4 \bullet 3)] + (3 \bullet 2) = 42$$

4.3.2 Network Complexity Index (NCI)

Indicates complexity of network relative to the number of unique classifications possible. This comparison between CASE 1 and CASE 2 shows that CASE 2 has an overall complexity of a factor of 7 greater than CASE 1.

$$NCI = \frac{NC}{O_n}$$

CASE 1:

$$NCI = \frac{6}{2} = 3$$

CASE 2:

$$NCI = \frac{42}{2} = 21$$

4.3.3 Connection Updates Per Second (CUPS)

CUPS is a figure of merit for performance for inter-platform or intra-platform comparisons. This value is also useful as a measure between different implementations of a similar network or comparisons between vendors of neural network simulator products.

$$CUPS = \frac{NC \bullet (\#\text{Patterns presented})}{\text{Runtime seconds per cycle}}$$

This concludes the development of metrics required to analyze the artificial neural network architectures used in this thesis. These values are be calculated and reported in the training summary of each network type used in Chapter 5.

4.4 Backpropagation Configuration Description

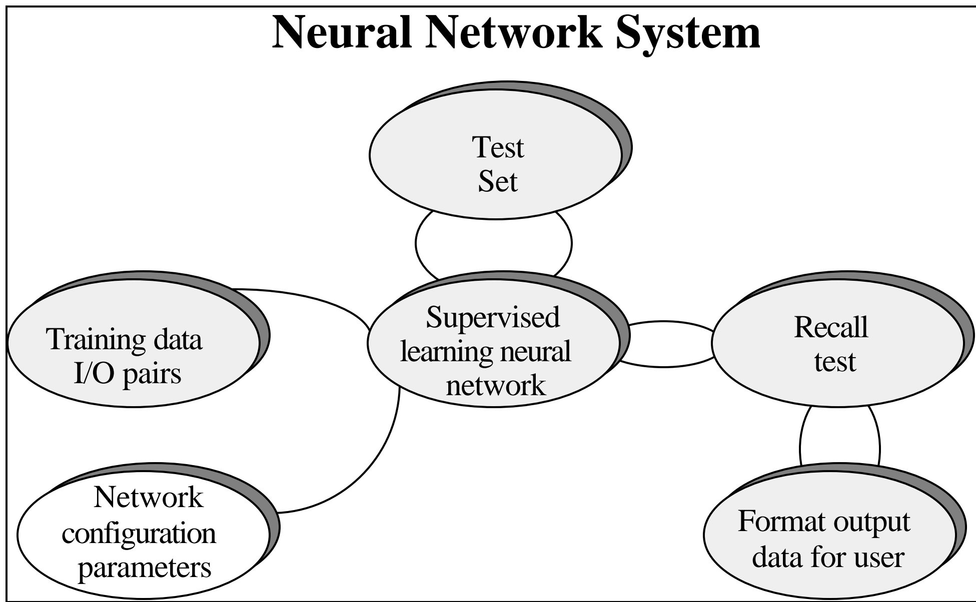


Figure 4-7 Neural Network Configuration.

In order to use a neural network simulator the necessary configuration parameters must be defined. These parameters are typically modified and tested a number of times to determine the optimum training parameters. Choosing some of the values is a matter of experience and understanding of the type of problem attempting to be classified. For an example scenario the Exclusive OR problem (also referred to as XOR) will be used to setup a sample architecture. The Exclusive OR problem represents a problem of non-linearity which is widely used to prove concepts of different neural network paradigms.

Figure 4-7 will be used to highlight each area of the artificial neural network description. In this case the Network configuration parameters object is white, the remaining items are a discussed in future sections.

4.4.1 Exclusive OR Network Configuration

Table 4-2 Exclusive OR Problem.

EXCLUSIVE OR		
INPUT		OUTPUT
0	0	0
0	1	1
1	0	1
1	1	0

The file format of the following section will describe the components of the configuration file used with the simulator implemented for this thesis. The first line of the title represents the topic followed by an actual file entry. All file entries include a keyword and a value.

INPUTS :**INPUTS 2**

The number of inputs must be determined, usually this is based on the number raw data features being processed. For the XOR problem two input values exist.

OUTPUTS :**OUTPUTS 1**

The number of outputs which are required. This case we have a single output node.

HIDDEN :**HIDDEN 2**

This determines how many hidden nodes are contained in the hidden layer. For the XOR problem in this example, two will be used.

RATE :**RATE 0.5**

This is also referred to as the learning rate, and is the value that is multiplied with the hidden layer and the error of the output vector, it is also multiplied with the input vector and error of the hidden layer. It controls the magnitude of change during each network cycle.

MOMENTUM: **MOMENTUM 0.2**

This value will control how much influence the previous weight change which occurred will affect the current weight change. It is meant to help accelerate training if it is converging in the right direction, and if it is moving in the opposite direction it will retard the change.

TOLERANCE: **TOLERANCE 0.1**

This value is the measurement criteria to determine if the value output by the output vector satisfies the requirements. For example when set to 0.1 this would mean that an output value of 0 desired may be represented by values from 0.00 to 0.1. At the other extreme it means 0.9 to 1.0 are also accepted as an output value for 1.

EPOCH :**EPOCH 1**

EPOCH training is a method in which the entire set of training patterns are presented and the error values are accumulated for the entire set of input patterns. At the end of the cycle when all patterns have been presented, the weight change calculations are made. This has the effect of averaging errors over the entire training set. For the XOR problem with a training set of four patterns a single epoch would present all four patterns to the network and then update the weights. Without this keyword in the configuration file, training by pattern is the default. This means as each pattern is presented to the network immediate error propagation and weight change calculation takes place.

The contents of the XOR problem configuration file is:

XOR.DEF
INPUTS 2
OUTPUTS 11
HIDDEN 2
RATE 0.2
MOMENTUM 0.0

4.5 Backpropagation Training I/O Pairs

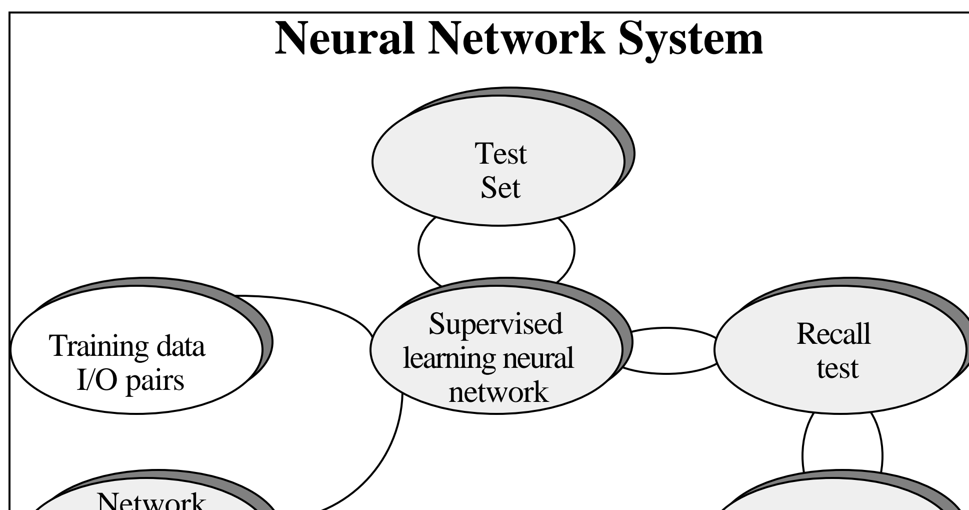


Figure 4-8 Neural Network Training Data I/O Pairs.

In training mode the simulator requires both inputs and outputs available to perform the backpropagation calculations at the nodes of the simulator. Implementations of the input values vary depending upon the best model for the problem being studied. In this case the training data I/O pairs are contained in a single file: It has a feature of autoscaling the input vector data. Most neural network simulators have a limited range of floating point values they accept, usually between 0 and 1, or -1 and 1. The inputs and outputs are preprocessed and scaled to meet the range criteria established by the simulator.

In this implementation the file format is as follows:

LINE 1: Minimum vector values for input, minimum vector values for output ,
LINE 2: Maximum vector values for input, maximum vector values for output ,
LINE 3: ':' Comments following the colon, usually used to describe the input and output vector.
LINE 4: Input vector , output vector expected ,
.
.
.
LINE n: Input vector , output vector expected ,

For the example presented by the XOR problem:

XOR.FCT
LINE1: 0.0 0.0 , 0.0
LINE2: 1.0 1.0 , 1.0
LINE3: :INPUT A INPUT B , OUTPUT #1
LINE4: 0.0 0.0 , 0.0
LINE5: 0.0 1.0 , 1.0
LINE6: 1.0 0.0 , 1.0
LINE7: 1.0 1.0 , 0.0

For other cases the input and output vectors values could be ranges such as -345 to 199, or 0.4 to 0.5, as long as the limits are defined properly they will be linearly scaled between 0 and 1.0. If any inputs exceed the limits defined, they will be clipped to 0 or 1.0 depending upon which range limits are exceeded. Chapter 5 will provide explanation of how feature data information is formatted for input, and output vector pairs.

4.6 Backpropagation Network Test Set Data

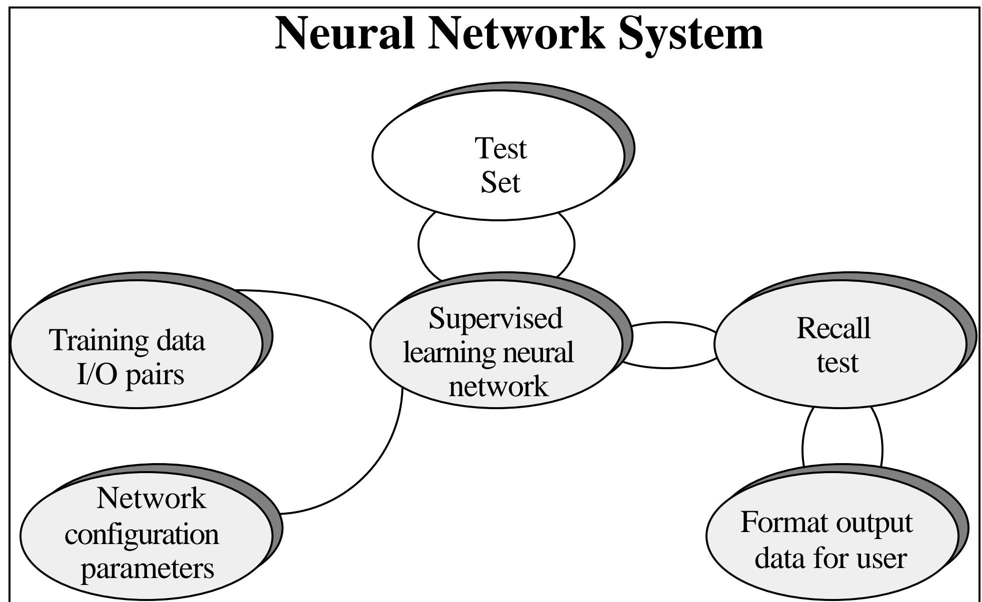


Figure 4-9 Neural Network Test Set Data.

The test set data is similar to the training data with the exception that there is no output vector included. The file has a '.IN' suffix. Instead the output vector is created and written out

to a file ".OUT". As each input test vector is presented a corresponding output vector is generated, this data is comma separated for each output vector.

For the example presented by the XOR problem:

XOR.IN
LINE1: 0.0 0.0 ,
LINE2: 1.0 1.0 ,
LINE3: :INPUT A INPUT B ,
LINE4: 0.0 0.0 ,
LINE5: 0.0 1.0 ,
LINE6: 1.0 0.0 ,
LINE7: 1.0 1.0 ,

XOR.OUT
0.002 , 0.93, 0.96 , 0.001

Another mode which this simulator supports is test mode, which uses a test file suffixed with '.TST'. The format is identical to the FCT or training file format presented above. Instead of training in this mode it only reads and performs a recall on the input vectors, at run time the percentage recognized is output to the user when testing is complete. This mode may be used for a quick check of the simulator. The output file is much more useful since it provides the actual node activation values which can be evaluated to determine convergence of the output pattern.

This is a summary list of the files the simulator uses:

1 - *netname*.DEF Definition and description of the simulator architecture.

2 - *netname*.FCT Training file contains I/O pairs for training.

3 - *netname*.TST Testing file only, the simulator reports the accuracy of the recognition performed on this file.

4 - *netname*.IN Contains input patterns only.

5 - *netname*.OUT Contains output vectors resulting from the *netname*.IN patterns.

4.7 Object-Oriented Programming Overview

Object-Oriented Programming (OOP) allows an advanced level of abstraction for implementation of artificial neural network simulators. The conceptual advantages of designing and using objects to represent parts of the simulation program allow easier understanding, improved data hiding, and better control of complex data types. Initially a procedural program is designed with a flow diagram implementing the way the procedures will handle the data, sometimes this is referred to as a procedure driven system.

With C++ or any OOP language, the data of the system is what drives the procedures or in C++ the methods. Data encapsulation provides tighter control of how data is used and accessed. Due to some of the new terms used in OOP an explanation will be given to help understand the major components in OOPs languages. Specifically C++ will be discussed. C++ was initially developed at Bell Labs by B. Stroustrup, accompanying the announcement of the language is the book he authored, *The C++ Programming Language*[63], considered by many the reference manual of the language. The following is a brief introduction to the terminology of C++/OOP and some of the advantages to each concept.

4.7.1 Class Description

A class is an abstract data type. It is a structure which contains all the elements necessary for existence and operations on the data types it manages. The operations or 'methods' that operate on the data are defined with the class. Normally, for the strict implementation of OOP no other methods should be allowed to access the data it contains, all operations should be carried out by the class methods defined within the class.

4.7.2 Object Description

An object is an instance of a class. Objects are created by instantiating a class. A similar analogy in C or PASCAL would be the dynamic allocation of a structure or record. In the OOP environment messages are sent to objects to perform the specified operations on the data contained in the object. A message is a control mechanism for activating certain methods within the class.

4.7.3 Information Hiding

Information hiding is the concept which specifies access to data only by methods contained within the same class as the data. There are attributes which allow control of the class defined data, and how well it is hidden. Levels of hiding are 'Public', 'Private', and 'Protected.' Public methods or data are available to any other class within the OOP environment. In well designed OOP applications very little 'Public' data exists. 'Private' specifies that only methods within the same class can use the data and methods of that class. 'Protected' is slightly more advanced, in OOP once a class is created other classes, usually different can inherit the property of different classes. For example we have two classes: Human and Employee.

The structure for classes is an inverted tree similar to the UNIX file system. So Human is considered a superclass, it is near the top of the hierarchy. Employee could inherit Humans attributes and methods if Employee has specified them as 'Protected.' Employee is considered a sub-class of Human. Advantages of this design methodology are, once certain objects are developed and debugged, other users can 'inherit' them and use all the previously designed and tested functions for their new class with confidence that they will operate consistently. A very loose analogy to this is the math libraries in 'C' for example, a programmer may wish to write their own routines to perform a logarithm and take the risk that something was not tested. A much safer way to implement the logarithm is to use an existing 'math library', written by experienced programmers and which is fully tested. Typically this 'library' code will be very efficient and use as little memory as possible.

4.7.4 Inheritance

This is the ability to reuse previously defined classes, and modify them for a new subclass. For the class circle we may want to inherit the properties of shape. The class 'shape' may contain functions for computing the area, circumference and volume already built into the class. Most the Graphical User Interfaces (GUI) created recently used this concept, including X-Windows. By having a well defined superclass of widgets, buttons and other user interface components the application developer merely 'inherits' a class named 'gui.' Once this is done he may then access and respond to all the events which are controlled and detected by methods defined in the superclass 'gui'. One of the advantage in this situation is that developers of applications can feel free to work on the application problem rather than worrying about interface details. The largest impact of this is observed when hardware or operating system upgrades must be performed. By using the supplied class libraries the programmer either relinks the code or the system dynamically links to the new portions as needed, as many X-Windows applications do.

4.7.5 Virtual Functions

This capability allows classes which inherit other classes to invoke methods from the original class definition. In a case where shapes are the superclass, and the rectangle class has a rectangle draw method in a subclass, another subclass of rectangle may be square. Squares draw function may be virtual, in the sense that it would point to or use the rectangle draw method.

4.7.6 Polymorphism

Polymorphism is the ability and concept of sending different messages to different objects and having each object respond appropriately. This is a capability which includes inheritance and the ability to determine at run time which set of parameters are required for the message input and output. This operation during run-time is referred to as dynamic binding. It offers a new degree of flexibility. In this simulator an example is:

If we need to do a matrix to matrix addition, its class may be an 'overloaded operator' meaning the same operator performs similar operations on different data types. For the example (Matrix +) inherits (Vector +) which inherits (Scalar +). So the operator for all three types of addition is '+' but dependent on the data type used a different method will be used.

4.7.7 Dynamic Binding

This technique requires a "method table" which is maintained during run-time. During the invocation of the message the function to be executed is determined by looking up the method name and class attributes in the method table. While this adds greater flexibility to OOP implementations it must be considered against the cost of the lookup for very repetitive functions. Depending upon the number of method and classes in the system this technique may place an overhead on the program which is not acceptable.

Since the introduction of C++ in 1985 a large number of applications have been ported and translated to use the advantages of C++. C++ offers an unparalleled efficiency in OOP languages while introducing some of the power of OOP. True or purist OOP is really implemented completely or as much as possible by languages like LISP and SmallTalk.

Dr. Stroustrup has written that he only intended to implement pieces of OOP which could be done without hindering the inherent power and efficiency of the C language. In fact C++ gains a few new OOP paradigms and is updated to implement existing techniques at intervals of about one year.

AT&T implements a CFRONT technology which essentially converts the C++ source into C code by means of the CFRONT preprocessor. This is typically the way C++ is

implemented at this time. The port of this simulator is from an IBM Personal Computer environment known as BORLAND C++ and another as ZORTECH C++, to C++ Version 2.0 from AT&T. Initial compilation problems and compiler access (Single User) led me to search elsewhere for porting this software to make it operational. The more powerful UNIX workstations CPUs provide the processing power required when training with neural network simulators. Approximately 99% of the time the simulator is performing floating point math operations.

4.8 Backpropagation Simulator Class Hierarchy

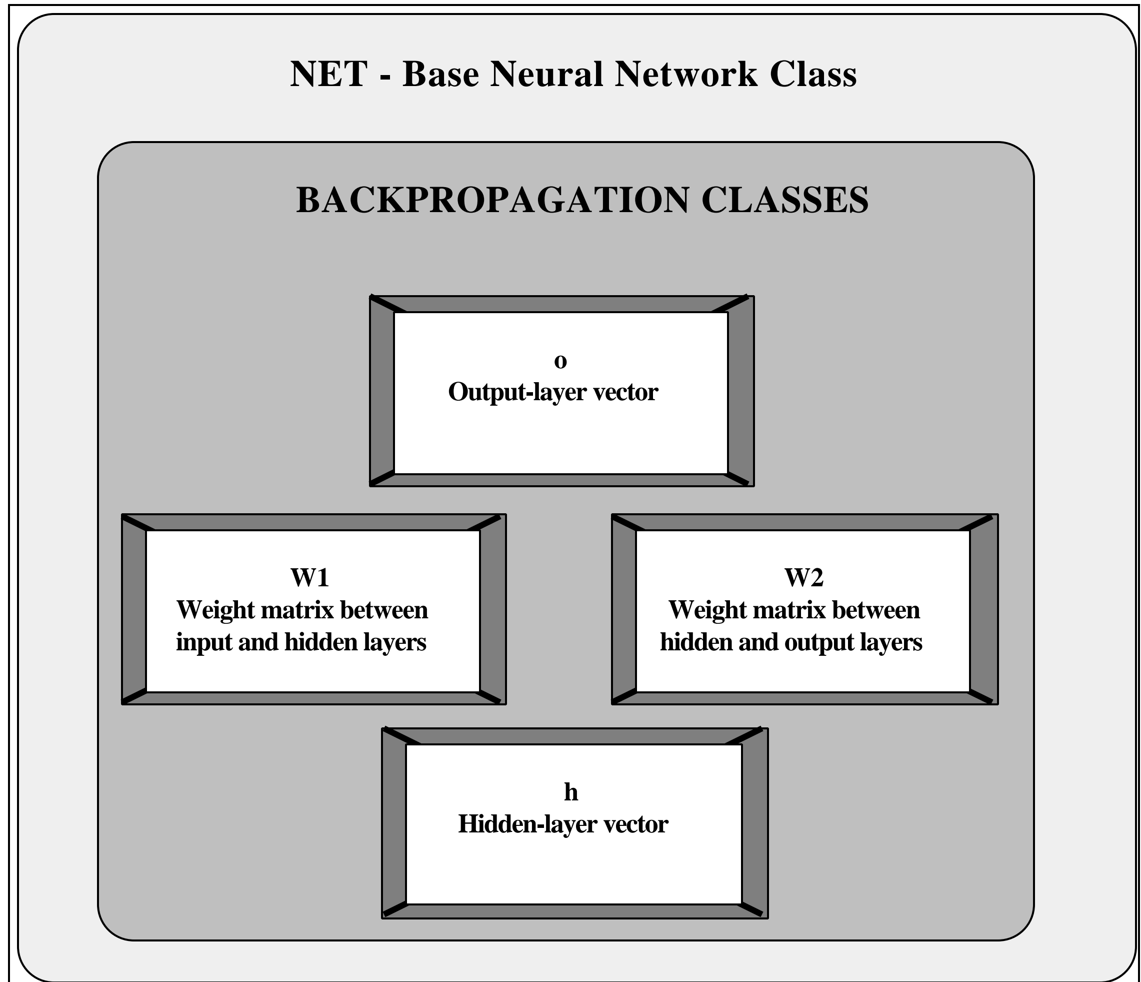


Figure 4-10 Backpropagation Class Hierarchy.

The class hierarchy depicts what classes are inherited by specific root or super classes. This breakdown of each block in the diagram follows with a brief description of every method and class within this simulator. Other artificial neural network simulators would be implemented as a subclass to the NET class. The author of the simulator has also implemented a Counter Propagation, Bi-directional associative memory and Hopfield neural network paradigms. For the class of problem studied in this thesis the backpropagation paradigm was chosen.

4.9 Software Components Of The Simulator

A brief description of each file will occur in alphabetical order with the header file or class description appearing first. A detailed method breakdown will follow this section.

bp.h — Contains definitions for class variables public and private along with public methods for the backpropagation class. Four private methods (initvals, saveweights, loadweights, cycle) and 4 public methods (bp - constructor, bp - destructor, encode, recall) are defined here. These eight methods are the capabilities implemented for this simulator.

bp.cc — Contains the implementation of the eight methods described in the class definition of the bp.h file.

net.h — Contains the definition for the parameter data structure, this is the information read from the file.DEF file. *Protected* members of this class are virtual methods saveweights and loadweights. Public methods defined here are: (net - constructor), (net - destructor), with virtual methods of encode, recall, cycle, train, test, and run.

net.cc — This is the implementation of the above methods. Virtual methods, as described in the net.h file, are modified for increased functionality in this module.

testbp.cc — Implementation of the user interface code is in this module. It is the 'main' of the simulator taking different command line arguments depending on the mode desired and invoking that mode by sending the appropriate message to the class. Command line arguments are: **L** - Learn mode, in this mode the *netname*.FCT file is read and the network is trained until convergence is met; **T**

- Test mode, the *netname*.TST is read and a count of good versus bad matches are calculated to determine the overall accuracy of recognition; **R** - Run mode, the *netname*.IN is read and it produces an output file named *netname*.OUT. The contents and or requirements of these files are described in the previous section.

vecmat.h Defines the classes matrix, vector, and vector pair, and all possible methods which can be applied to them.

vecmat.cc Implementation code for the methods described in vecmat.h.

The power and abstraction of the classes can be understood after reviewing this software. For more detailed information on the actual methods and or classes see the software listing in APPENDIX A. Vector and matrix classes allow a definition of all the necessary vector and matrix methods to be implemented in a very compact yet understandable format. For vector operations two items are necessary the data type of the vector and the length of the vector. For matrix operations the width and height must be known along with the data type. A graphic view of the methods and the classes they are defined in, follow:

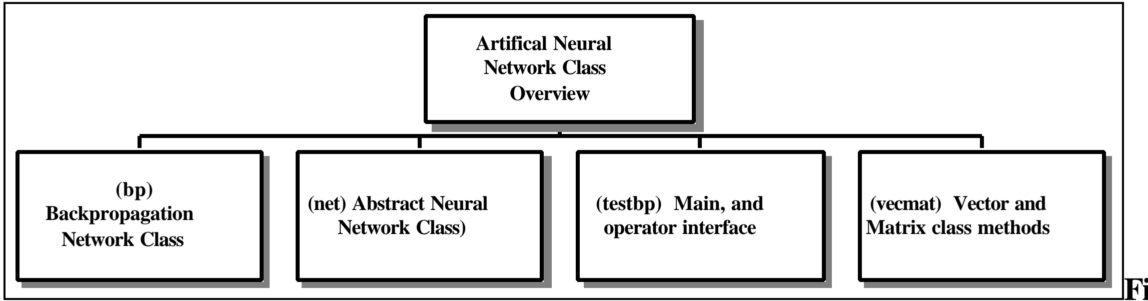


Figure 4-11 Backpropagation Class Hierarchy Overview.

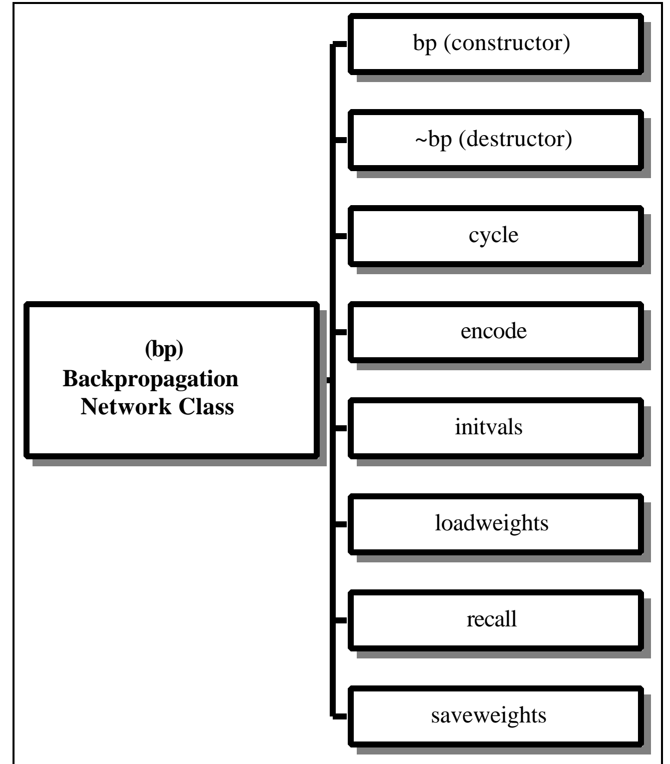


Figure 4-12 Backpropagation Network Class Methods.

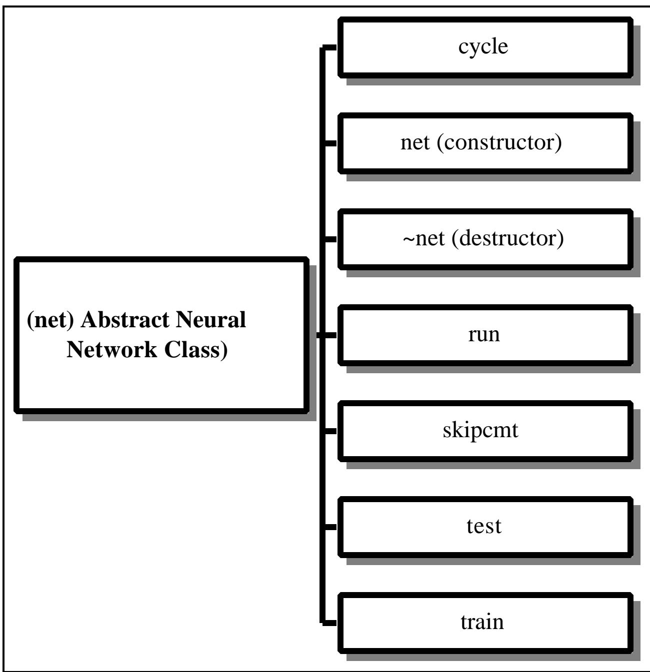


Figure 4-13 Class 'net' Methods.

(testbp) Main, and operator interface

Figure 4-14 'testbp' Interface Code.

The following class is broken down in two levels, first the overview and then each of the methods defined for operating on vector, matrix and vector pair object types.

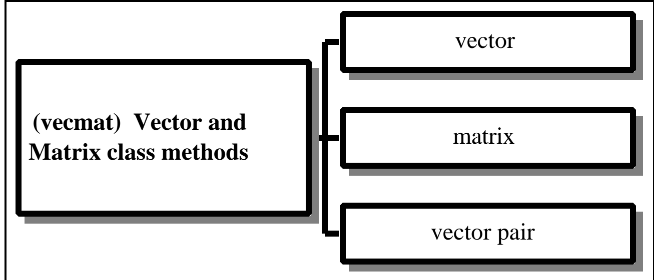


Figure 4-15 'vecmat' Vector Code Method Overview.

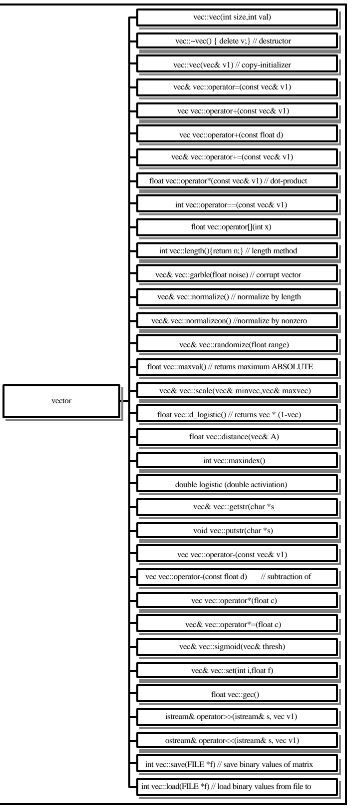


Figure 4-16 Class 'vector' Vector Methods.

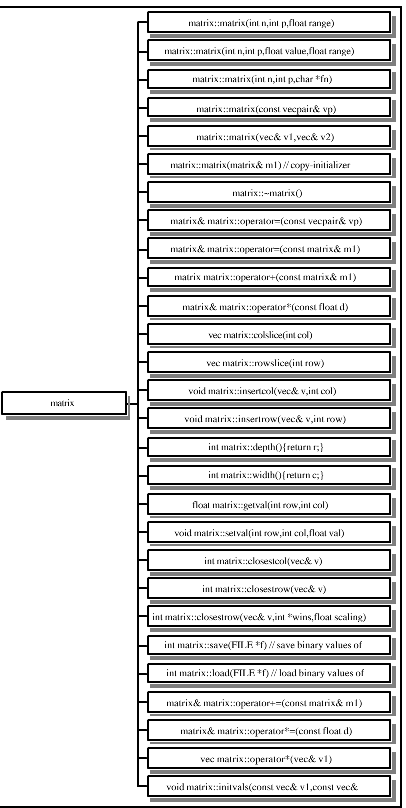


Figure 4-17 Class 'matrix' and Methods.

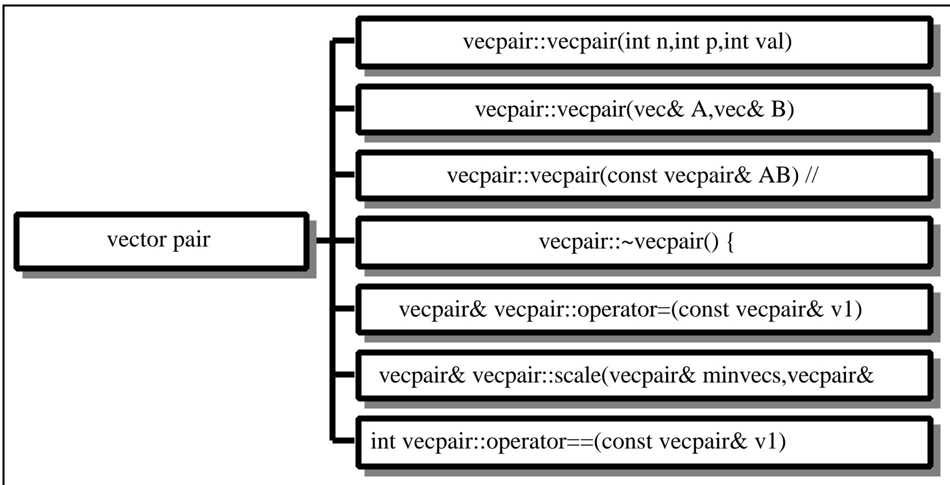


Figure 4-18 Class 'vecpair' and Methods.

The above overview of methods and classes are all components of the system. Problems with translating the code and porting to a different architecture will be discussed in the

conclusion of this section. Next an outline and flow chart of the artificial neural network simulator will document how the program operates. In particular the 'encode' cycle which is the most heavily used part of the system during training will be examined. Figure 4-19 illustrates the three major operating modes of the simulator:

- **L**earn Training mode, reads the *netname*.FCT file, outputs to a net.WTS file.
- **T**est Perform recognition of a *netname*.TST file, output to standard output.
- **R**un Perform recognition of a *netname*.IN and output to a netname.OUT file.

The cycle of the learn mode will be broken down further for an understanding of the training procedure. Test and run mode are nearly identical utilizing the 'recall' method to present the unknown input vector to a trained network.

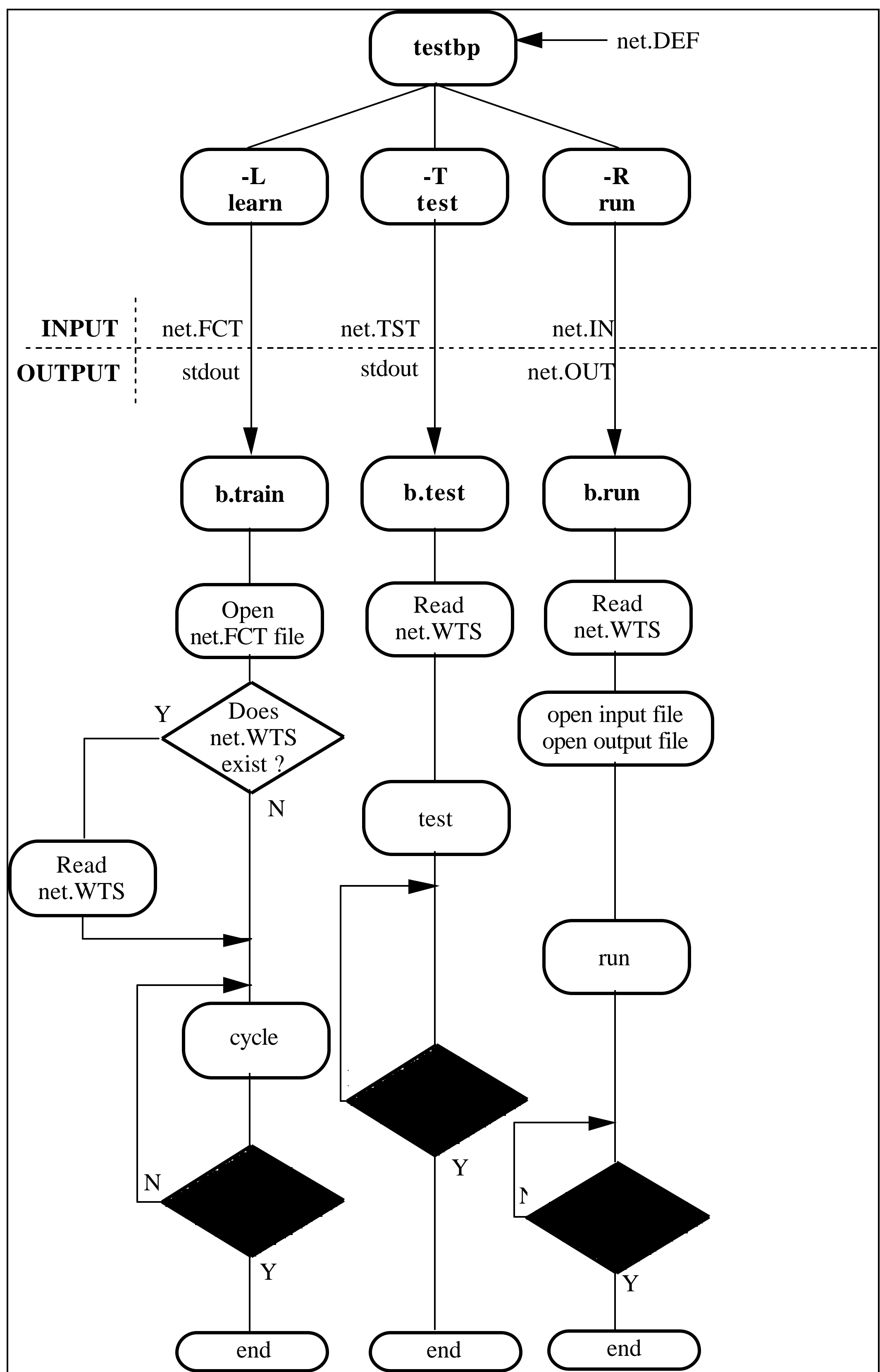


Figure 4-19 Simulator Modes Overview.

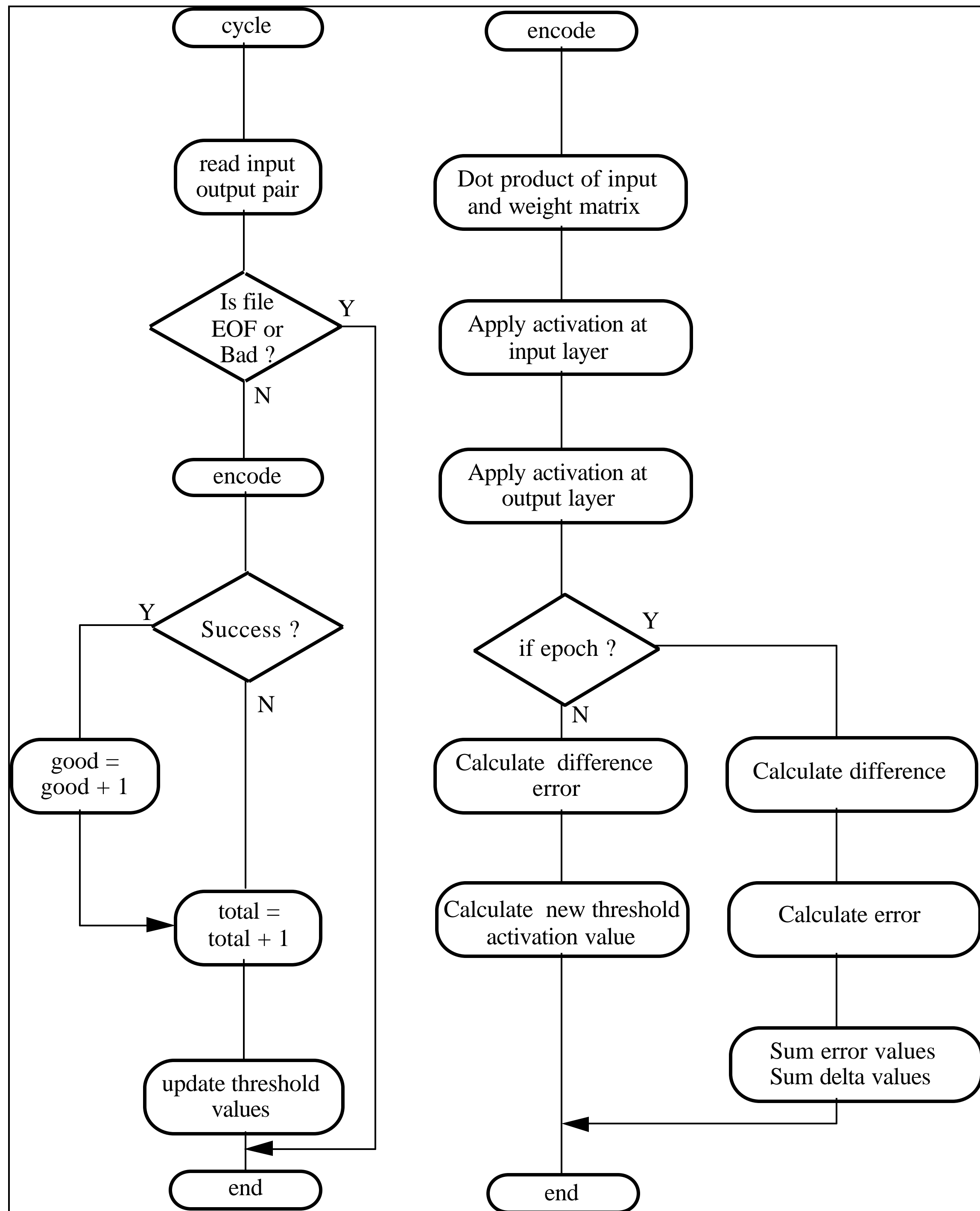


Figure 4-20 Flow Diagram For 'cycle' and 'encode'.

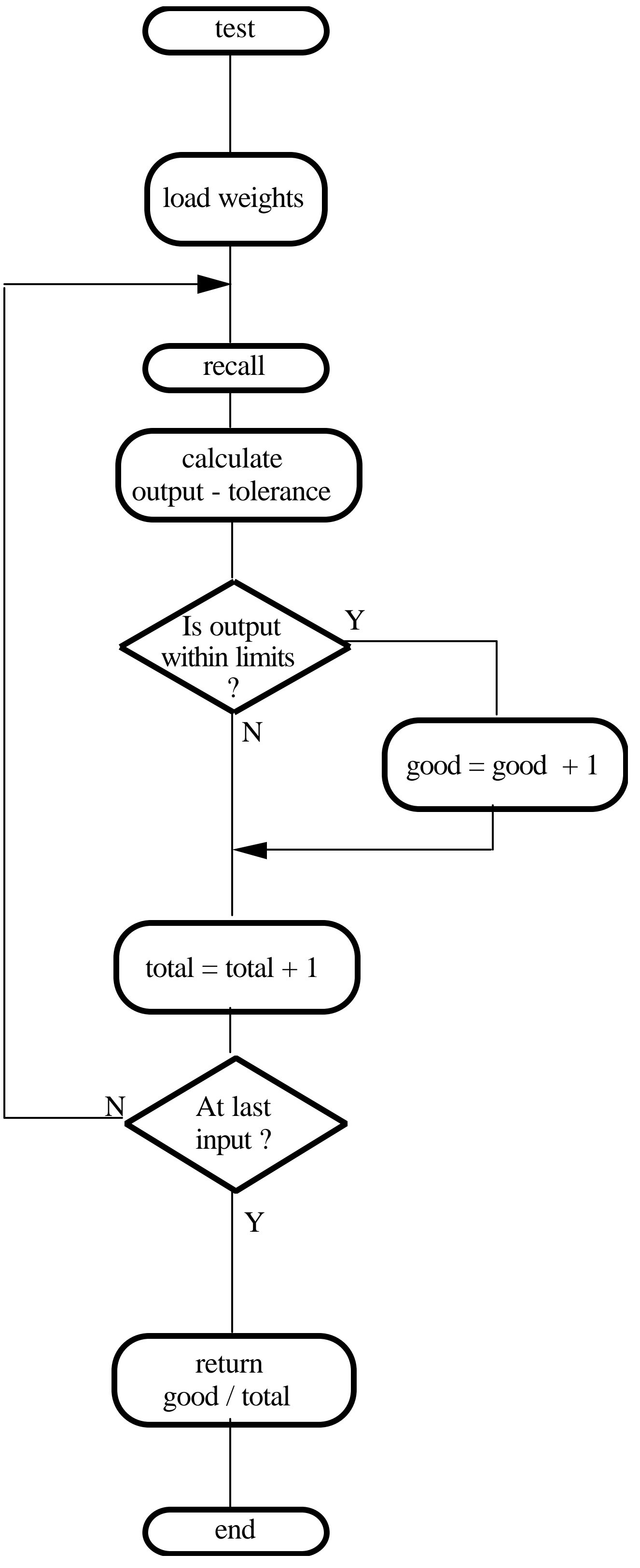


Figure 4-21 Flow Diagram For 'test.'

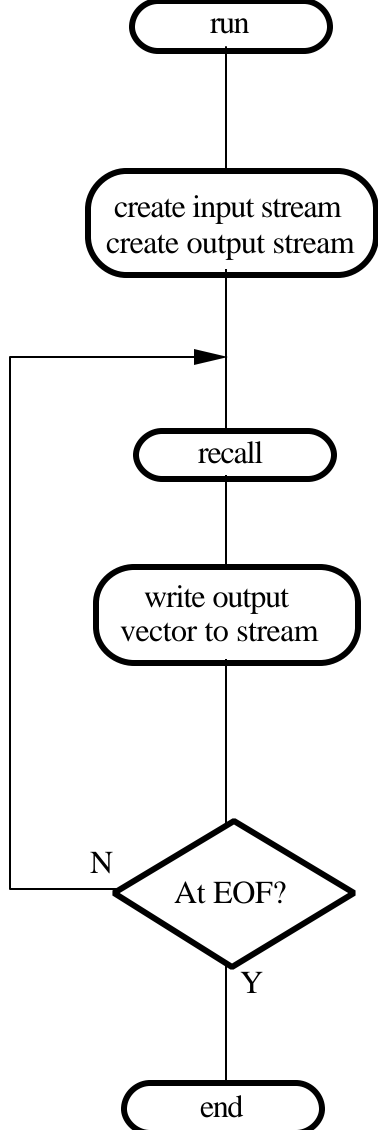


Figure 4-22 Flow Diagram For 'run.'

In addition to the existing source a new and enhanced method of tracking the current simulation status has been implemented. This modification of code has been made to the train method of the net.cc file. Simulator enhancement items include:

- Programming model diagram see Figure 4-4.
- Current cycle count displayed at intervals relevant to the size of the training set.
- Elapsed hours simulator has been training on current training set.

- Real time clock output for monitoring the length of time required plus a history of the number of restarts that occurred for parameter changes.
- Accuracy output to display current number of correct pattern matches. With large data sets this is very useful for monitoring convergence of the simulator.

4.10 Simulator Porting Issues

Translating the simulator to the UNIX environment caused a number of compile and programming errors to surface. Originally this software was developed in an IBM PC environment, using a compiler that was not available for UNIX systems. Errors that were detected during the port were corrected if the simulator operated incorrectly. Other errors and possible coding flaws that did not impair operation were noted and left for future work. Implementation notes or hints were non-existent in the source description. No control or data flow diagrams were provided for description of the software. This is the motiviation for the previous diagrams, as it made understanding the simulator software less tedious.

- Automatic declaration of storage for a structure name PARMS did not occur with the AT&T C++ Cfront compiler. It was necessary to use a static declaration for this structure.

- Implementation difference: originally the code had incremented C float variables by using: good++;, this does not work in the AT&T version of C++ compiler. Code change implemented to add a float value of 1.0 to good: good = good + 1.0;

- Epoch mode of the simulator is flawed in the programming methods, this mode of operation is supposed to sum all error and delta vectors and at the end of the entire training set update the weights with the new weight delta calculations. After 3 weeks of investigation of this problem it was placed on hold and the other training method of pattern-by-pattern was used. In the case of the data sets required for this thesis either method provides the exact same end

results. For radically varying input vectors an EPOCH mode may help smooth or average the learning.

- No method was provided for displaying results at a predetermined interval. Modifications were made to add a DISPLAY parameter to the setup file. This modifies the interval (indicated by the number of cycles) that simulator status is displayed.

- A measure for global error was needed to help analyze training and convergence of the simulator. After several attempts of creating a simple method for providing this feature a formal approach was developed. Three vector methods were added to the vecmat.h class. Most systems generate specific error measures which are application dependent. The methods implemented here are flexible and extendible so other calculations could be added. Methods for this function are 'maxerror', 'maxerrorreset', and 'getglobalerror.' As defined in vecmat.h:

```
float maxerror();                            // calculate maximum error
float maxerrorreset();                       // reset the error accumulator
float getglobalerror();                      // get the global error value
```

To determine the error value for this system an absolute sum of the difference of output node delta and target value is calculated by 'maxerror.' Resetting the global error value to 0.0 is performed by 'maxerrorreset.' Acquiring the global error value is performed by the 'getglobalerror' method, it returns the type of float error value. Using the reset function the error can be measured between any two points or for any given cycle interval. The definition of the error calculation follows:

e = vector element

t = target vector

o = output vector

n = number of output nodes

d = delta vector (difference of output and target vector)

$$global \quad error = \sum_{e=1}^{e=n} abs(\mathbf{t}[e] - \mathbf{o}[e])$$

Since the delta vector in the implementation represents the delta between target and output values the formula as implemented in the maxerror method is:

$$global \quad error = \sum_{e=1}^{e=n} abs(\mathbf{d}[e])$$

• There was no method to determine the best node output produced at the output vector based on a given input, a C program was created to determine this and will be explained in the feature extraction section.

4.11 Simulator Sample Operation

Hardware Environment

To debug and test the simulator operation the Exclusive OR problem was used. This problem is a relatively simple function and represents a non-linear mapping function for its inputs versus the output. The environment used for operating the simulator includes:

• SUN SPARCStation 10 Model 41 computer

• 5 GB of attached disk, and 128 MB of main memory.

• SUN OS 4.1.3, OpenWindows 3, C compiler, SUN AT&T C++ V2.0 compiler.

Hardware of this class is considered state of the art in products being shipped in 1993. Operation, testing, and running the simulator varied due to the demands placed by other processes and users on this workstation throughout the period that this simulator was executing.

4.11.1 Operation With XOR Problem

First, a definition file must be created. The files shown in this text are the actual files used for operation, and will be enclosed in a box to eliminate confusion with any surrounding text. The Exclusive OR function has already been defined in Table 4-2.

```
INPUTS 2
OUTPUTS 1
HIDDEN 2
RATE 0.2
MOMENTUM 0.1
TOLERANCE 0.1
INITRANGE 0.1
DISPLAY 2000
```

Figure 4-23 OR.DEF Network Definition File.

Once the definition file is created it is not rigid, these parameters are the run time options the user controls during simulation. Stopping and restarting will not affect anything and the weights will continue to be valid from the last cycle executed. Changing the structure of input, output or hidden nodes however will affect everything. When doing architecture studies it is best to create different instances of the entire network. The OR.FCT file follows describes the minimum, maximum, comments and input, output vector pairs. For the Exclusive OR problem a 7 line file is required.

```
 0.0 0.0 , 0.0 ,
 1.0 1.0 , 1.0 ,
 : A  B , OUTPUT
 0.0 0.0 , 0.0 ,
```

```
0.0 1.0 , 1.0 ,
1.0 0.0 , 1.0 ,
1.0 1.0 , 0.0 ,
```

Figure 4-24 OR.FCT Network Input Output Pairs With Limits.

For test purposes the OR.IN file contains the entire cover of possible inputs to test. This file is also used by an additional monitoring shell program which I have created to analyze training along with the simulator output. Normally the simulator output is written into a file named OR.OUT. For better insight into the convergence of the system it is desirable to be able to monitor and create a file tracing the history of the training. To perform this function the shell script program actually runs the simulator with the current value of weights in run mode. Along with the output vector a time stamp and network architecture file is concatenated to the same line. This is an excellent reference for monitoring changes in the network parameters and observing the affects on the network training. By using this scenario any changes made to the simulator are tested or checked out by examining the XOR sample problem.

```
0.0 0.0 ,
0.0 1.0 ,
1.0 0.0 ,
1.0 1.0 ,
```

Figure 4-25 OR.IN Test Input Vectors.

Each comma separated value is the resulting vector of an input vector. Here we have the four output values created by the input vectors shown in Figure 4-24. Since the tolerance is .1, the first and fourth value represent an output of 0, and the second and third output represent the output value 1.

```
0.051 ,0.93 ,0.92 ,0.081 ,
```

Figure 4-26 OR.IN Test Input Vectors.

During the run the simulator produces the output which follows:

```
TESTBP - Interactive Backpropagation Network Tester
       HIDDEN: 2
     MOMENTUM: 0.1
   INIT RANGE: 0.1
        EPOCH: 0
```

```
    TOLERANCE: 0.1
Training from:OR.FCT
       INPUTS: 2
      OUTPUTS: 1
   LEARN RATE: 0.2
   DECAY RATE: 0
        ITERS: 0
      DISPLAY: 2000

------- ---------  ------------  --------  ---------
 CYCLE   ELAPSED    TIME STAMP   ACCURACY    GLOBAL
  (#)    (HOURS)   yymmddhhmmss               ERROR
------- ---------  ------------  --------  ---------
      0    0.000   930722055719      0.00     2.1017
   2000    0.004   930722055733      0.00     1.6212
   4000    0.009   930722055750      0.00     1.4371
   6000    0.014   930722055808      0.00     1.3127
   8000    0.018   930722055825      0.00     1.2157
  10000    0.023   930722055842      0.00     1.1342
  12000    0.028   930722055900      0.00     1.0628
  14000    0.033   930722055917      0.00     0.9987
  16000    0.037   930722055934      0.00     0.9404
  18000    0.043   930722055952      0.00     0.8869
  20000    0.047   930722060009      0.00     0.8376
  22000    0.052   930722060026      0.00     0.7922
  24000    0.057   930722060044      0.00     0.7504
  26000    0.062   930722060101      0.00     0.7121
  28000    0.067   930722060119      0.00     0.6770
  30000    0.071   930722060136      0.00     0.6448
  32000    0.076   930722060153      0.25     0.6154
  34000    0.081   930722060210      0.25     0.5885
  36000    0.086   930722060228      0.25     0.5639
  38000    0.091   930722060245      0.25     0.5414
  40000    0.095   930722060302      0.25     0.5209
  42000    0.100   930722060320      0.25     0.5020
  44000    0.105   930722060337      0.25     0.4846
  46000    0.110   930722060355      0.25     0.4686
  48000    0.115   930722060412      0.25     0.4539
  50000    0.119   930722060429      0.25     0.4403
  52000    0.124   930722060447      0.25     0.4277
  54000    0.129   930722060504      0.25     0.4159
  56000    0.134   930722060521      0.25     0.4050
  58000    0.139   930722060539      0.50     0.3947
  60000    0.144   930722060556      0.75     0.3850
  62000    0.148   930722060613      0.75     0.3759
  64000    0.153   930722060630      0.75     0.3674
  66000    0.158   930722060647      0.75     0.3594
  68000    0.163   930722060705      0.75     0.3519
  70000    0.168   930722060722      0.75     0.3448
  72000    0.172   930722060739      0.75     0.3381
  74000    0.177   930722060757      0.75     0.3317
  76000    0.182   930722060814      0.75     0.3256
  78000    0.187   930722060831      0.75     0.3198
  80000    0.192   930722060849      0.75     0.3143
Training suspended at 81040 cycles.
```

Figures 4-27 and 4-28 display the convergence of the simulator with the XOR problem. Figure 4-27 illustrates the output of the simulator status. Figure 4-28 is a graph of the output values for each type of state in the XOR problem.

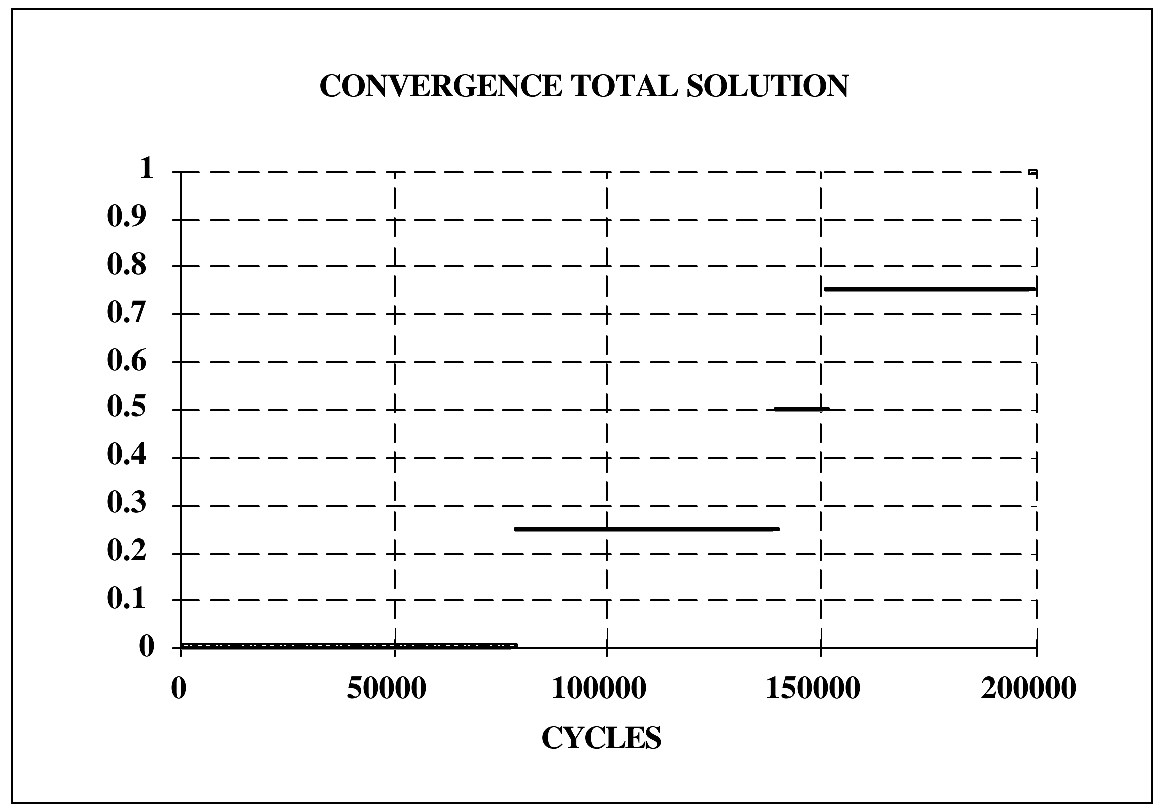


Figure 4-27 Convergence of XOR Test Case.

The check.sh shell program creates the following output. The ordering of the architecture parameters is the same as the OR.DEF entries. Order is not significant in the OR.DEF as keywords are used to locate the proper parameters. Following the tabular output of the data vectors is a graph of the same information, plotted against time.

(Four output values/vectors) (Date and Time) (Architecture parameters)

```
0.56 ,0.56 ,0.56 ,0.56 , 07/05/93 19:37:26 2 1 2 0.2 0.0 0.1 0.1
0.15 ,0.78 ,0.74 ,0.26 , 07/05/93 19:38:27 2 1 2 0.2 0.0 0.1 0.1
0.10 ,0.86 ,0.83 ,0.16 , 07/05/93 19:39:27 2 1 2 0.2 0.0 0.1 0.1
0.08 ,0.89 ,0.87 ,0.12 , 07/05/93 19:40:28 2 1 2 0.2 0.0 0.1 0.1
0.06 ,0.91 ,0.89 ,0.11 , 07/05/93 19:41:29 2 1 2 0.2 0.0 0.1 0.1
0.05 ,0.92 ,0.91 ,0.09 , 07/05/93 19:42:29 2 1 2 0.2 0.0 0.1 0.1
0.05 ,0.93 ,0.92 ,0.08 , 07/05/93 19:43:30 2 1 2 0.2 0.0 0.1 0.1
0.05 ,0.93 ,0.92 ,0.08 , 07/05/93 19:44:30 2 1 2 0.2 0.0 0.1 0.1
0.05 ,0.93 ,0.92 ,0.08 , 07/05/93 19:45:31 2 1 2 0.2 0.0 0.1 0.1
```

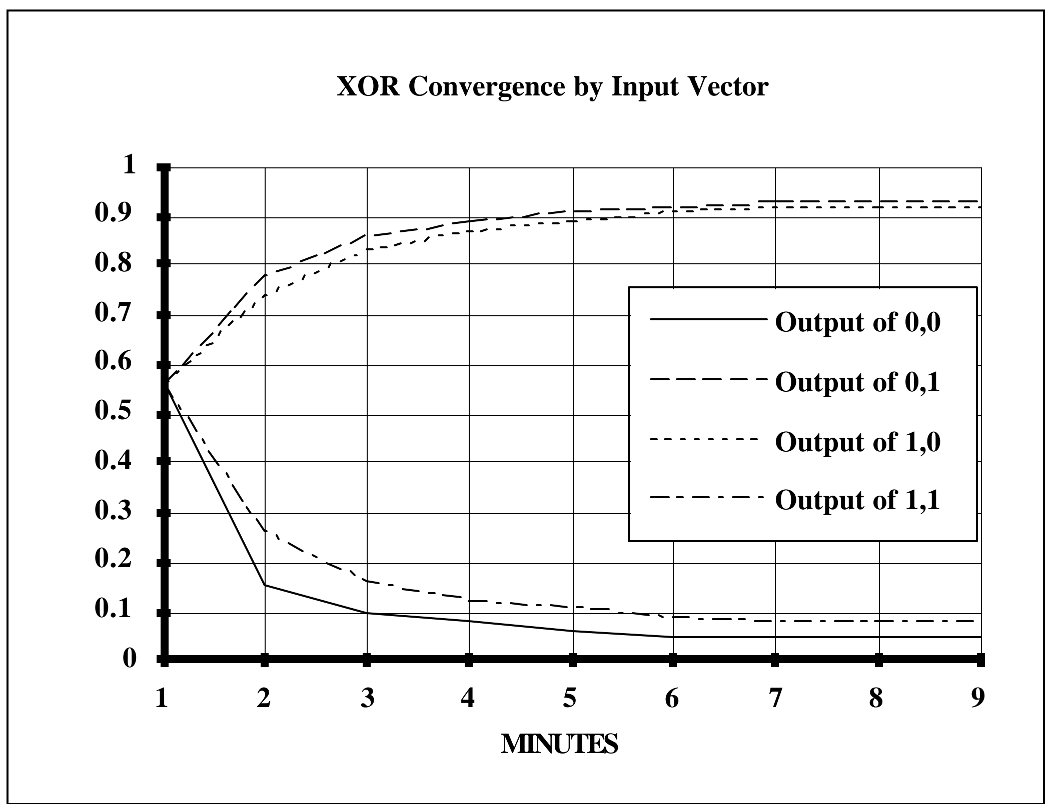


Figure 4-28 Convergence of XOR Test Case.

This concludes the introduction of a sample application using the artificial neural network backpropagation paradigm. Convergence, training speed, and accuracy are important aspects of using artificial neural networks with real world data.

In the next chapter, features will be described and extracted which will represent real world data. The feature set is separated into two categories, training and testing. After the simulator is trained it can then be run against the test features and scored for accuracy.

CHAPTER 5

FEATURE METHODS AND APPLICATIONS

5.1 Introduction

This chapter will develop and explain the feature extraction methods created for this thesis. Explanation of techniques for image acquisition, image database, image preprocessing, feature vector creation, and interpretation of output vectors will be described. Determination of which features to extract and at what resolution or granularity has been performed by ad hoc and formal approaches. Most techniques exhibit sensitivity to certain characteristics of a target image. The goal of finding a balance between sensitivity and generalization of the image is what will be examined in this chapter. A general discussion of each topic will introduce the fundamental concepts supporting them. After this explanation, each feature method developed and tested will be tabulated and analyzed.

5.1.1 Image Acquisition

Converting images into an electronic form can be accomplished by the following :

- Digitization from analog to digital values by a digitizing camera.
- Scanning the image on a digital scanner, scanners exist in flatbed and drum formats.
- Inputting characters manually using a digitizing tablet.
- Using a pen based computer which is similar to using a digitizing tablet.

5.1.2 Image Database

The database used for the digit and uppercase characters was converted from the National Institute of Standards and Technology (NIST) SD3 database, referred to as NIST SD3. This entire database represents constrained forms which were filled out by 2,100 different writers across the country. For purposes of this thesis and because of size constraints only a very small portion of this database will be used for testing and training, since the original database contains over 300,000 images. Also available for image input is an 11x17", 300 dpi, 24 bit, scanner which was used for acquiring the engineering drawing depicted in Figure 5-3. The vision lab has a digitizing camera system that can also be used for acquiring images. Creating a database such as NIST SD3 is a very costly and time consuming process. Figure 5-1 and 5-2 are samples of the images used for training from the database.

The engineering symbols used were cut from a power distribution schematic, (see Figure 5-3) which was hand drawn by utility company personnel. Symbols used for this thesis are shown in Figure 5-4.

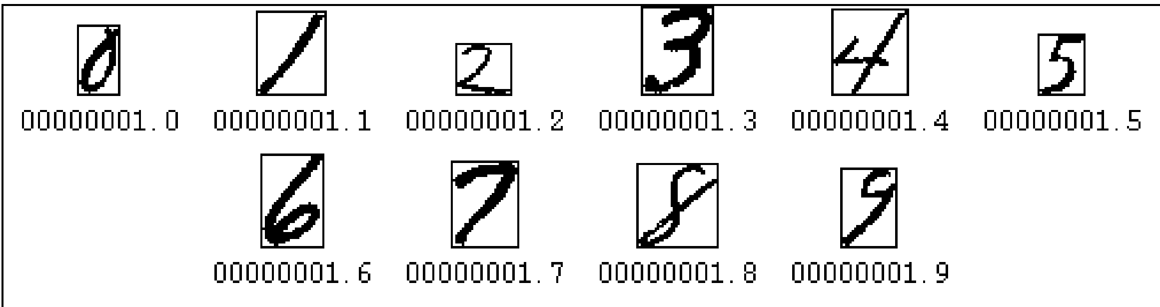


Figure 5-1 Digit Image Database Sample.

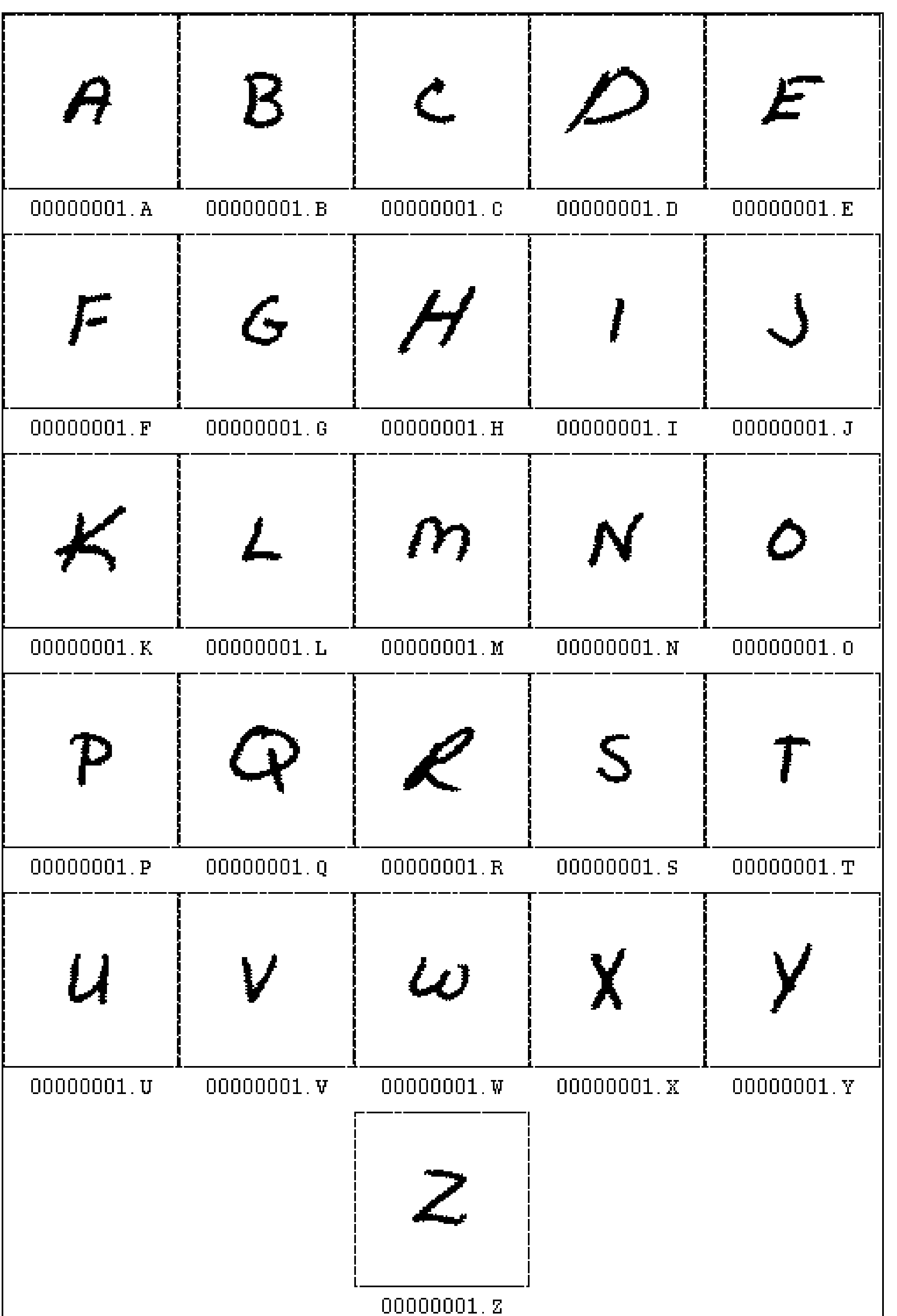


Figure 5-2 Uppercase Alphabet Image Database Sample.

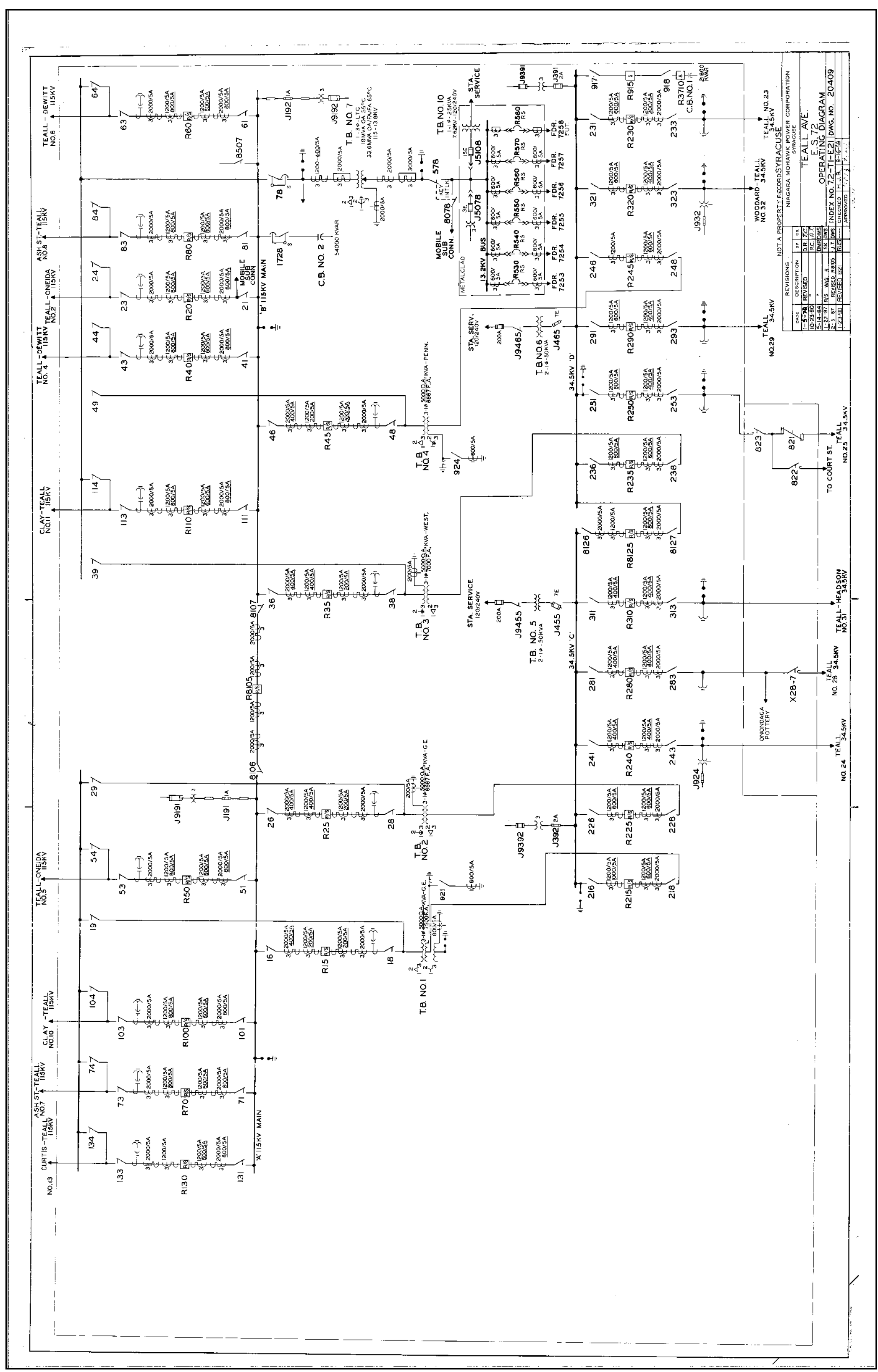


Figure 5-3 Utility Drawing used for Symbols.

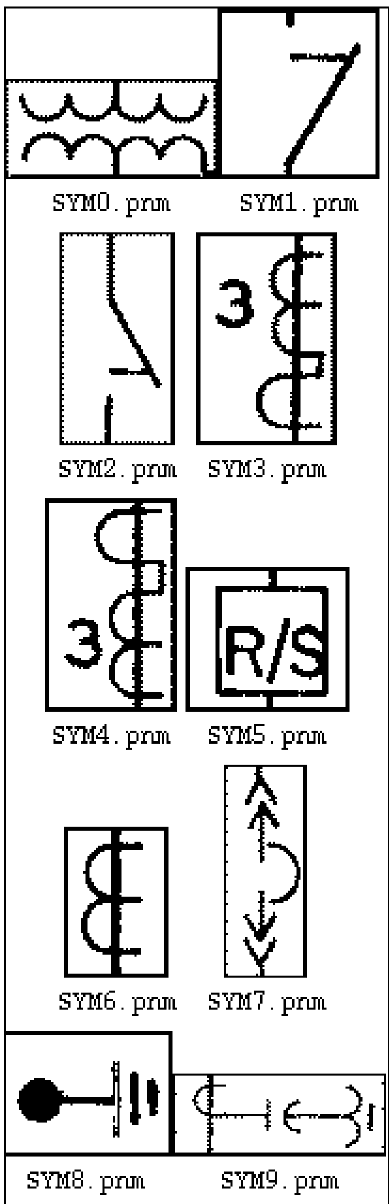


Figure 5-4 Symbol Set From Utility Drawing.

5.1.3 Sample of Digits and Uppercase Test Images

To understand what the images that are used for testing look like, a cross sampling of assembled images are displayed in Figure 5-5 and Figure 5-6. Every 10th set of samples is displayed here for digits and uppercase characters. As is readily apparent the data used for testing is difficult. Test data is used from another NIST database provided specifically for testing.

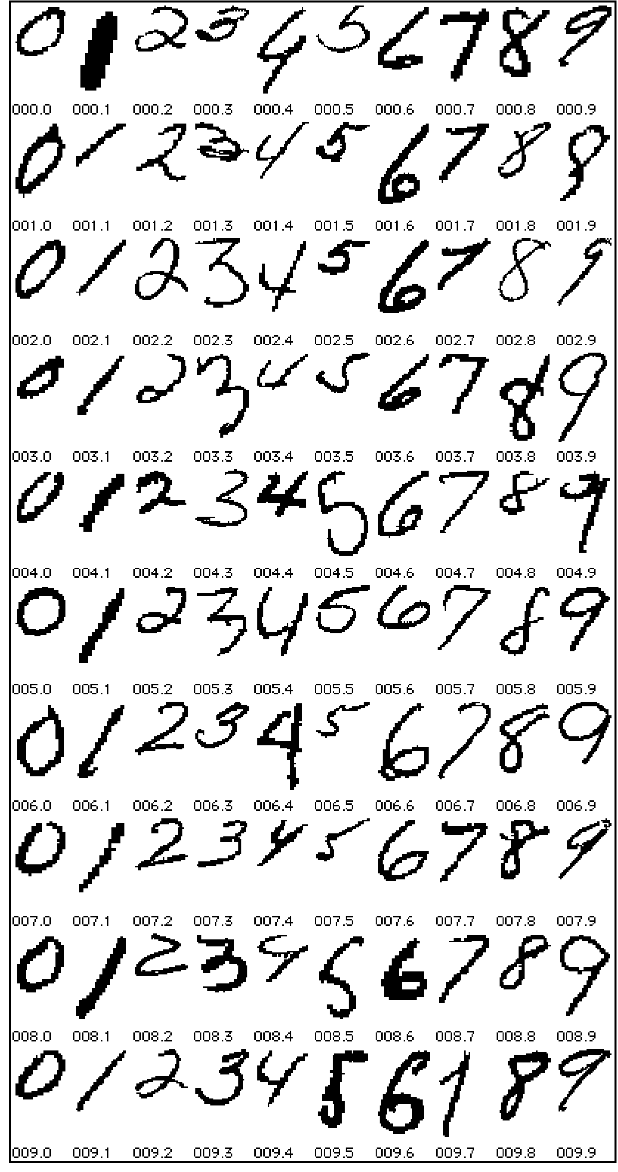


Figure 5-5 Sample of Digit Test Set .

Figure 5-6 Sample of Uppercase Test Set.

5.1.4 File Formats

Storage and conversion techniques for digital images are usually proprietary methods which represent a particular vendors implementation. A few formats are 'de facto' standards which are commonly used to translate images between different computer systems have been developed for the public domain. There are at least 50 different image formats with 5 to 10 being used frequently. A paper could easily be written on the techniques, the advantages and history behind each file format. For the purposes of this thesis a general description of what a file format normally entails for storing images is provided.

File formats for images were created to enable efficient methods of storing images, which are generally not handled efficiently in traditional computer operating systems. Many image models exist, for example: monochrome (1 bit per pixel), color or gray scale (2-24 bits per pixel), multiband images which consist of 2 to 'n' layers of color or gray scale images. These parameters define the depth of the image. Columns and rows define the geometric dimensions of the rectangle images are stored in. A typical definition for an image may be 100x200x1 which is an 100 columns by 200 rows with a pixel depth of 1, for 1000x2000x24, there are 1000 columns, 2000 rows of pixels each 24 bits deep. Another attribute which defines an image is the resolution or scale of dots per inch (dpi). This information is usually defined during digitization of the image. Typical scanner resolutions are 75, 100, 150, 200, 300, 400, 600, 1200 dpi. By knowing the original scanner resolution and the dimensions of the image the actual image dimensions in inches can be determined. At 100dpi the 1000x2000x24 bit image would be 10"x20" in size. Higher resolutions capture more detail about the image. Depending on the problem and how the image is being used, lower resolutions may be better to use than higher ones.

Compression algorithms are commonly employed for use with images to reduce their size for storage and transmission. For binary images common in OCR and document management systems, CCITT GROUP 3 and GROUP 4 are very efficient algorithms, resulting

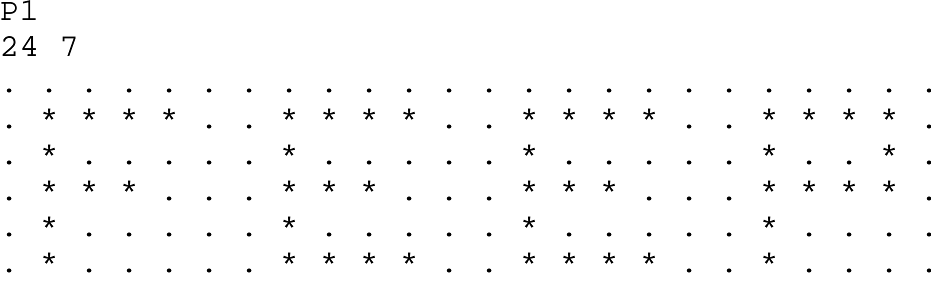
in up to 10:1 file compression. CCITT GROUP 3 is the method used in most FAX machines. Color images employ more sophisticated techniques which can be separated into two categories, lossless, and lossy. Examples of lossless are Huffman and Run Length Encoding. A very common lossy algorithm used on 24 bit deep images is JPEG. For purposes of this thesis two image formats will be processed.

The original file storage technique will be images in SUN Raster Format which uses run length encoding (RLE). SUN raster files will be converted into a more portable format known as Portable BitMap, (PBM). PBM is a portable bitmap format which is a lowest common denominator monochrome file format. It was originally designed to make it reasonable to electronically mail bitmaps between different types of machines using the network mailers currently in use, which handle 7 bit ASCII data only. Now it serves as the common definition of a large family of bitmap conversion filters. A file in this format contains three parameters in the header, type P1 or P4, P4 indicates binary, P1 indicates ASCII, the number columns, and rows. After the header, image data follows.

This is an example of the letters FEEP in the ASCII file format of PBM.

```
P1
24 7
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  1  1  1  1  0  0  1  1  1  1  0  0  1  1  1  1  0  0  1  1  1  1  0
0  1  0  0  0  0  0  1  0  0  0  0  0  1  0  0  0  0  0  1  0  0  1  0
0  1  1  1  0  0  0  1  1  1  0  0  0  1  1  1  0  0  0  1  1  1  1  0
0  1  0  0  0  0  0  1  0  0  0  0  0  1  0  0  0  0  0  1  0  0  0  0
0  1  0  0  0  0  0  1  1  1  1  0  0  1  1  1  1  0  0  1  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Same file with ones replaced with '*' and zeroes replaced by '.' for clarification when viewing.



```
P1
24 7
. . . . . . . . . . . . . . . . . . . . . . . .
. * * * * . . * * * * . . * * * * . . * * * * .
. * . . . . . * . . . . . * . . . . . * . . * .
. * * * . . . * * * . . . * * * . . . * * * * .
. * . . . . . * . . . . . * . . . . . * . . . .
. * . . . . . * * * * . . * * * * . . * . . . .
. . . . . . . . . . . . . . . . . . . . . . . .
```

Other formats have been created to support gray scale and color images. The difference between the binary and ASCII format, is that ASCII stores 1 bit per byte and binary stores 8 bits per byte. Binary offers an 8:1 storage size reduction. However, ASCII is easier to view with standard operating system tools.

5.1.5 Image Preprocessing

Before analysis of an image may begin, there are steps necessary to convert, enhance and translate images into a specific format for feature extraction. These processes can be combined to describe image preprocessing. Very few systems work with raw images. Typical preprocessing functions for OCR purposes are normalization, noise removal, smoothing, enlargement, reduction, rotation, and translation within the image frame. Image processing has a collection of very mature fundamental techniques as mentioned above. In order to concentrate on feature methods and the artificial neural network a public domain tool kit was utilized for the basic image preprocessing capabilities. The tool kit PBMPLUS allows transformations and conversions of images. An introduction to the PBMPLUS documentation follows:

PBMPLUS
Extended Portable Bitmap Toolkit
Distribution of 10dec91
Previous distribution 30oct91

Copyright (C) 1989, 1991 by Jef Poskanzer.

PBMPLUS is a toolkit for converting various image formats to and from portable formats, and therefore to and from each other. The idea is, if you want to convert among N image formats, you only need 2*N conversion filters, instead of the N^2 you would need if you wrote each one separately. In addition to the converters, the package includes some simple tools for manipulating the portable formats. The package is broken up into four parts. First is PBM, for bitmaps (1 bit per pixel). Then there is PGM, for grayscale images. Next is PPM, for full-color images. Last, there is PNM, which does content-independent manipulations on any of the three internal formats, and also handles external formats that have multiple types[64].

5.2 Feature Vector Creation

A method must be developed to describe the image space. A pattern may be used to describe an expected structure in the image or an object in the image. Patterns are developed by associations with geometric models, heuristic measures of the shape, templates and other techniques to form a descriptor. This descriptor is referred to as a *feature*[65].

The specific technique for creating "feature vectors" defined as a feature, is then applied to create a vector for an image containing a character. Once generated, they are the input part of the training or test patterns for the artificial neural network. For training however, a *truth* or expected output vector must also be part of the training or netname.FCT file entry. Truth or output vectors represent the expected output pattern for the given input. As defined earlier, a single entry in the netname.FCT file is (input vector, expected output vector).

5.2.1 Feature Vector Interpretation

After a network is trained on a set of training data the network is ready to be used for classification. To determine classification accuracy, a new set of images not related to the training set is used. In cases where training data and test data have common images then accuracy of testing is not valid. It is common to use 60% of the database for training data and 40% for testing data, or some combination similar to this. Output during testing will generate

values between 0 and 1 for each node in the output vector, which can be interpreted in different ways. For example on a digit classification system where the output vector has 10 elements, each element represents one of the classified digits.

```
Actual image: 3
Output vector:0.01 0.04 0.00 0.63 0.03 0.09 0.04 0.05 0.56 0.03
Class:          0    1    2    3    4    5    6    7    8    9
```

From this example it can be seen that the class values for 3 and 8 are within .07 of each other. Due to this small variation in the output node the character may be either a 3 or an 8. There are different methods for interpretation of the output vector including voting, statistical and thresholding techniques. This simulator is implemented as a threshold detection method only, it uses the tolerance parameter specified in the netname.DEF file and looks for the first node meeting this criteria. In the above example it would produce no classification for this situation, since the tolerance is 0.1 an output activation value of at least 0.9 is required to be classified. Even though the correct classification is indicated by the peak activation in the output vector, it is lost due to the method used for determining classification. For this thesis a peak detection C program has been developed to determine the maximum value output node in the output vector.

The test images are labeled for identification purposes, each image filename contains the actual character class in the suffix. For example file name: 00000061.3 would indicate that the image in this file represents the digit character '3.' The score or accuracy is calculated by keeping a count of the image class pattern presented and the number of correct output classes produced by the neural network.

5.3 Raw Bit Map Feature

Most feature extraction methods require reduction of the original image size in order to create the feature vector. The Raw Bit Map (RBM) feature approach does perform this step, though only scaling is required to normalize the source image into a constrained target size. Each pixel of the input image is input directly into the input layer of the neural network, starting with position 0,0 to X maximum, Y maximum. This type of feature utilizes input vectors with sizes which are the product of the dimension of the target grid. For this case a 24x24 cell requires 576 input nodes. Figure 5-7 illustrates the mapping of binary image to the input vector. This feature will represent the baseline technique against which the others are compared, the absolute value is not as important as its position or rank when compared to the other methods. A cell size of 32x32 was used for the uppercase alphabet.

This method represents the lowest common denominator in providing preprocessing as very little preprocessing is performed.

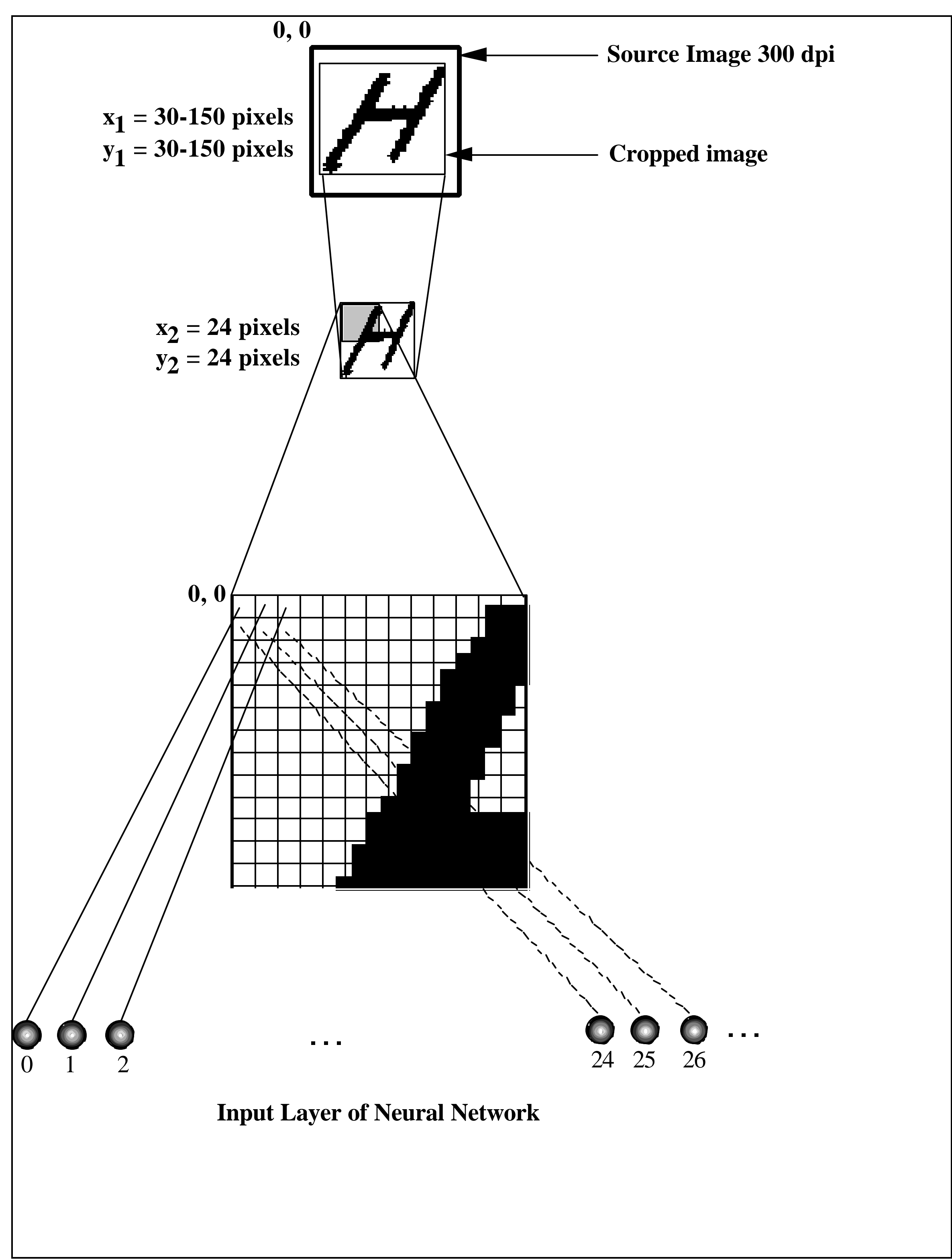


Figure 5-7 Raw Bit Map to Feature Vector Mapping.

5.3.1 Creation of the Raw Bit Map Feature Vector

Step 1

Convert the original SUN raster image to a PBM image.

Step 2:

Crop the PBM image by clipping off all white space which surrounds the smallest possible rectangle enclosing the image created by black pixels. The inner rectangle in the first or top image of Figure 5-7 shows the result of cropping.

Step 3:

Scale the PBM image from whatever the original source size to a 24x24 pixel image.

Step 4:

A C program operates on the PBM image output by STEP 3, reading each pixel value and creating a corresponding floating point value. When this process is complete it represents an RBM feature vector which is used directly by the neural network for training or testing.

Comments

Shift, rotation and scale invariance are desirable characteristics for feature extraction methods. These characteristics and generalization as described below are the ideal components of most feature extraction methods. In practice it is difficult to implement all of these characteristics in one feature method.

Shift invariance: Cropping of the image using the smallest surrounding rectangle coarsely centers the image in the target image area. Problems with this method occur when spurious branches protrude from the character due to stroke variation or not lifting the writing device from the media.

Rotation invariance: This method is sensitive to rotation of the character, characters rotated by more than $\pm 5°$ will cause classification errors. Methods of determining the rotation

or skew of characters can be used to determine and then rotate the image. Rotation methods typically cause distortion of the image and usually need smoothing to remove the pixel spikes created by the rotation.

Scale invariance: Scale invariance is addressed here by fitting the source image either larger or smaller into the target image. Problems arise if the character is printed extremely small such as a 'I', vertical bars may fill out the entire target image with black pixels.

Generalization: A diverse set of training characters will produce the best possible generalization of each class. Special characters such as '7's with slashes would not be classified properly if trained only on '7's without slashes.

Results for this technique are considered as the baseline for the artificial neural network simulator in this application. Since very little translation of the original information is performed it represents the baseline for all other methods developed for this thesis.

Table 5-1 Raw Bit Map Training Summary.

Image Data	Network Topology	Comments	Training Set Size	Test Set Size	NC NCI CUPS (train)	Acc-uracy
Digits 0-9	576.20.10		500, 50 per class	510, 51 per class	11,720 1,172 59,297	73.4%
Symbols 0-9	576.10.10		50, 5 per class		5,860 586 27,129	84.6%
Digits 0-9	576.20.10	Certain researchers show a quadruple in training set size halves the error rate. For time and space reasons this was not performed.	1000, 100 per class	510, 51 per class	11,720 1,172 49,326	74.6%
Upper A-Z	1024.30.26	37 hours for training.	1300, 50 per class	1300 total, 50 per class	31,500 1,211 94,791	63.8%

Table 5-2 is the performance matrix for the digits using RBM feature type. Ideally the diagonal values would be the same as with no off diagonal axis values. Each row represents the class being recognized, values not on the diagonal in the same row represent the misclassifications. For case '0' the row totals to 57 classifications, 57 - 51 = 6 are classifications from other classes. Ten occurrences were incorrectly classified as 5 in the case of 0, and one occurrence of misclassification occurred for the digits '1', '2', '4', '8', and '9.' This is the worst case misclassification of the entire matrix. Digit '5' was classified with the most errors, it was confused most with digit '0.'

Accompanying each performance matrix is a three dimensional contour plot representing the table. One difference between the table and plot exists; the row and columns are transposed. Ideally the contour plot would be a single dark diagonal line. Linear scaling has been applied to the plot in order to produce reasonable break points for each range of values. The Z axis of the table has a logarithm scale, this is represented by the legend. For each range the actual values are as follows:

Scale translation for Z-axis.

Actual Value	**Legend**
0 - 9	0.00 - 1.00
10 - 99	1.00 - 2.00
100 - 999	2.00 - 3.00

Through the use of these visualization plots, relationships of confusion between classes can be identified. Readily identifying problems and overlaps between methods is the main use of these plots.

The test data is representative of handwritten data from high school students who were less than motivated. With neatly printed test data accuracy would likely be in the 90-95% range, as most published papers typically use better test samples. Appendix B contains the supporting source code for this feature.

Table 5-2 RBM Performance Matrix Digit Results.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		
0	42	1	1	0	1	10	0	0	1	1		57
1	0	45	0	3	0	0	3	3	0	1		55
2	0	1	39	0	1	6	0	1	2	3		53
3	0	0	1	33	3	0	0	1	0	3		41
4	0	0	0	0	38	0	8	0	0	2		48
5	3	0	3	0	0	27	0	2	0	1		36
6	0	0	1	4	6	1	34	0	0	6		52
7	0	1	2	6	0	4	2	44	0	1		60
8	3	0	0	1	1	0	0	0	46	0		51

9	3	4	4	4	1	3	4	0	2	33		58
	51	52	51	51	51	51	51	51	51	51		

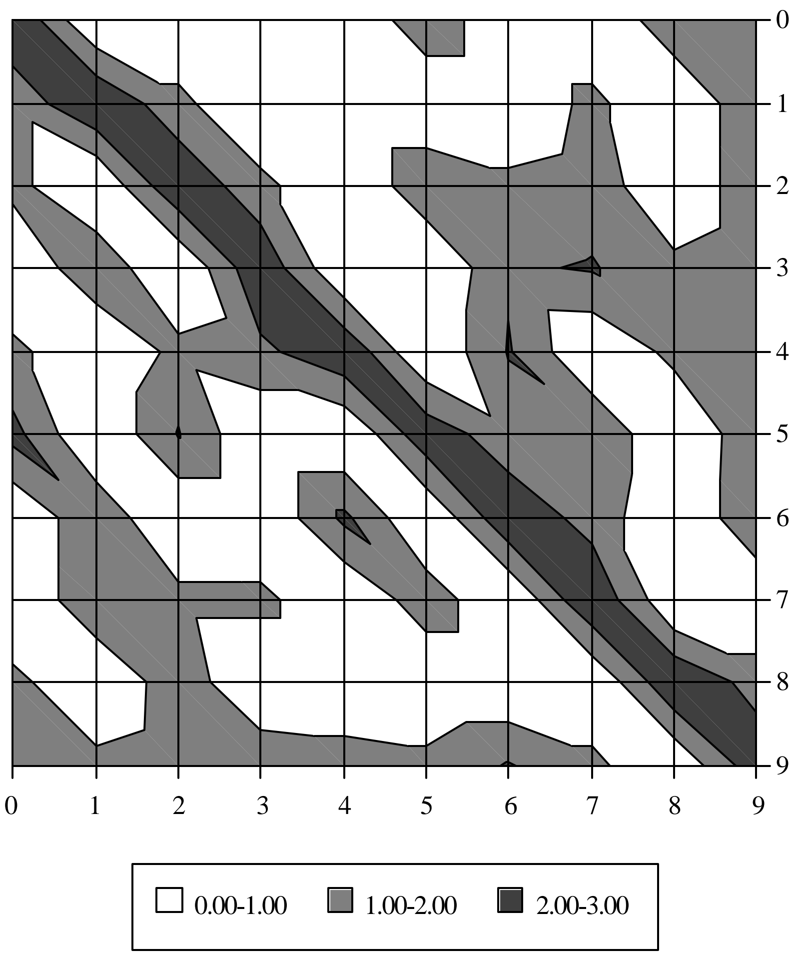


Figure 5-8 RBM Performance Matrix Contour Plot For Digits.

Table 5-4 is the performance matrix for the upper case characters using RBM feature type. Ideally the diagonal values would be the same as with no off axis values. Each row represents the class being recognized, values not on the diagonal in the same row represents the misclassifications. Each class will be characterized in table 5-3, the order of misclassifications appear from high to low in the comment field. In the comment field the first two characters listed are the misclassifications with the highest errors.

Table 5-3 RBM Uppercase Results Summary.

Class	Comment
A	E, D.
B	I, N, O. I is normalized and expanded to fill the entire cell and has a high correlation with B errors. See the I error when classifying B.
C	T, F, S, V, A.
D	M, R, T, H.
E	P, A, K, V.
F	C, E, M.
G	J, Q, U.
H	R, V, C.
I	B, C, N, O, S.
J	K, Q, S, V.
K	V, C, G, J, T.
L	B, N, S.
M	A.
N	I, O, P, Z.
O	X, W, Z, I, N.
P	E, H.
Q	G, B, S.
R	H, L, O, P, A, S, T.
S	L, B, C, I.
T	K, J, C.
U	L, K, Z.
V	W, O, I, X.
W	Z, L.
X	Y, S, H.
Y	L, Y, Z.

Z	W, G, L.

As with the case of the digits the test data is representative of handwritten data from high school students who were not highly motivated. With neatly printed test data accuracy would likely be in the 90-95% range, as most researchers publish results using better test samples. Again the absolute accuracy is only being used for comparison purposes with other techniques. Larger test and training sets could be used to increase the accuracy of the overall system.

Table 5-4 RBM Performance Matrix Uppercase Results.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**		
A	37	0	0	3	4	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	1	0	0	1	0	0		50
B	0	23	0	0	1	0	0	0	3	0	0	1	0	3	2	0	0	1	0	0	0	0	0	0	0	1		35
C	2	0	31	0	1	5	0	0	1	0	1	0	0	0	0	1	0	1	2	6	0	2	0	0	0	1		54
D	0	0	1	43	0	2	0	3	0	0	1	0	5	0	0	1	0	3	0	3	0	0	0	0	0	0		62
E	2	0	1	0	27	1	0	0	1	1	2	0	0	0	0	4	1	1	0	0	1	2	1	0	0	0		45
F	0	1	2	0	2	37	1	1	0	0	1	0	2	0	0	0	1	0	0	1	0	0	0	0	0	0		49
G	0	0	1	0	0	0	33	0	0	2	0	0	0	0	0	0	2	1	1	0	2	0	1	0	0	0		43
H	0	1	2	0	1	0	1	33	1	0	0	0	1	0	0	0	0	5	0	1	0	3	0	0	0	1		50
I	0	10	2	0	0	0	0	0	25	0	0	1	0	2	2	0	0	0	2	0	0	0	0	1	1	0		46
J	1	0	0	1	0	1	1	0	0	35	9	0	0	0	0	0	2	1	2	1	0	2	0	0	1	0		57
K	1	0	3	0	2	0	2	0	0	2	28	0	1	0	0	0	0	1	0	2	1	6	0	1	1	0		51
L	0	4	0	0	0	0	0	0	1	0	0	24	0	3	0	0	0	0	2	0	0	0	0	1	1	0		36
M	3	0	1	0	1	1	0	0	0	1	0	2	38	0	1	0	0	0	1	0	0	0	0	1	1	0		51
N	0	1	0	0	0	0	0	0	9	0	0	0	0	36	5	2	0	1	0	0	0	1	0	1	0	2		58
O	0	0	0	0	0	0	0	1	4	0	0	2	0	3	28	0	0	0	0	0	0	0	6	7	0	5		56
P	0	0	0	1	10	0	0	3	0	0	0	0	0	0	0	38	0	1	0	0	0	1	0	0	0	0		54
Q	0	5	0	1	0	0	6	0	0	0	0	0	0	1	0	0	40	0	3	0	0	0	0	0	0	1		57
R	2	1	1	1	1	0	0	4	0	0	0	3	0	0	3	3	1	29	2	2	0	1	0	0	1	0		55
S	0	3	2	0	0	0	1	0	2	0	0	4	1	1	0	0	1	1	31	0	1	0	0	0	0	1		49
T	1	0	2	0	0	1	1	2	0	3	5	0	0	0	0	0	0	1	0	32	0	0	0	0	0	0		48
U	0	0	0	0	0	0	0	0	0	1	3	4	0	1	0	0	0	0	0	0	43	0	0	0	0	2		54
V	1	0	0	0	0	0	0	0	3	1	0	0	0	0	7	1	0	1	0	0	0	31	10	3	1	1		60
W	0	0	1	0	0	1	1	0	0	0	0	3	0	0	0	0	0	1	0	0	0	0	28	1	1	5		42
X	0	0	0	0	0	0	1	2	0	0	0	0	1	0	1	0	1	1	3	1	0	1	1	33	4	0		50
Y	0	2	0	0	0	1	0	1	0	3	0	4	0	0	0	0	0	0	1	0	0	0	0	0	33	2		47
Z	0	0	0	0	0	0	2	0	0	0	0	2	0	0	0	0	1	0	0	0	1	0	3	0	5	28		42
	50	51	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50		

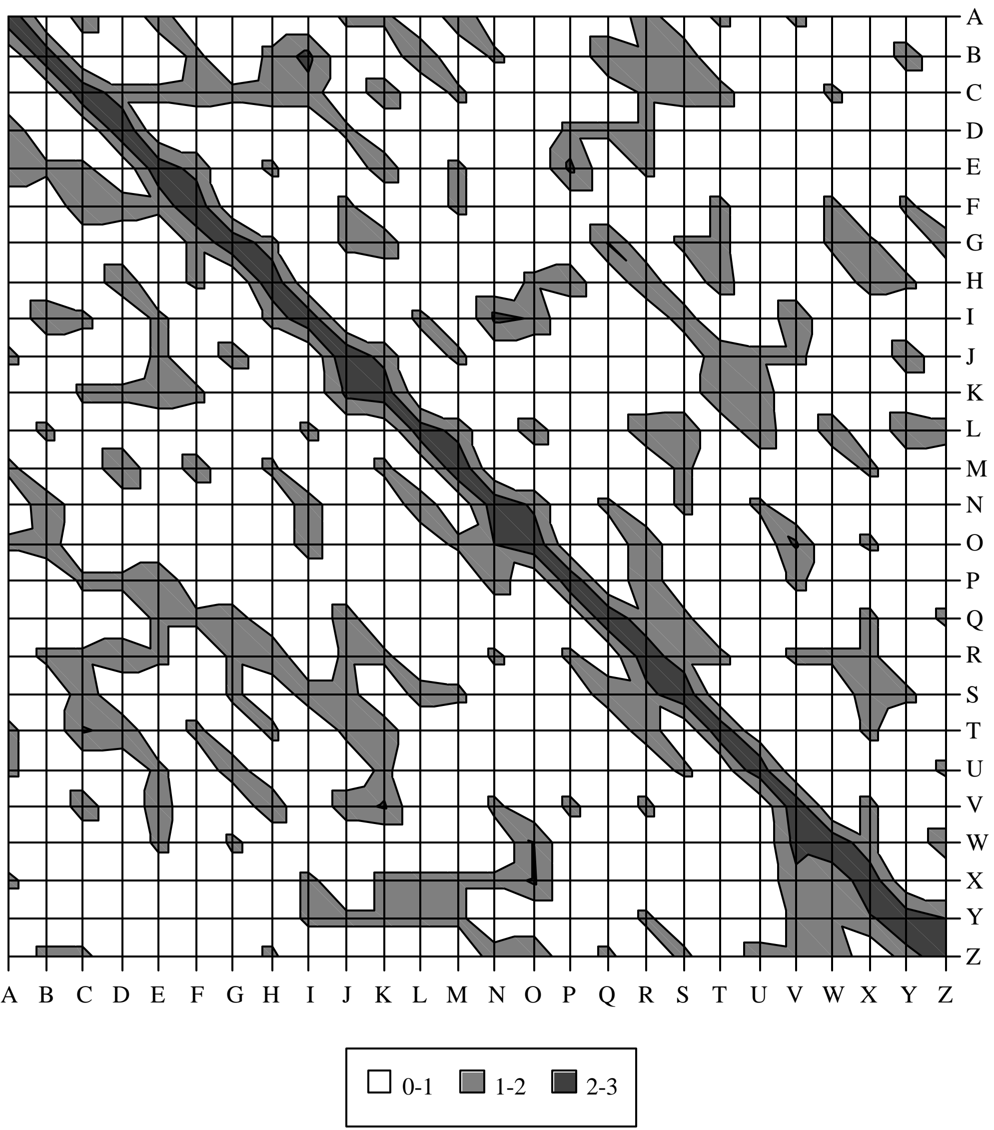


Figure 5-9 RBM Performance Matrix Contour Plot For Uppercase.

Table 5-5 is the symbol recognition performance matrix. Results > 85% are typical when the characteristics of the image being classified are well behaved. Though these symbols are hand drawn it was performed by a draftsperson and represents a motivated writer to present the best symbol possible. Results show very good classification of the limited data set used for testing.

Table 5-5 RBM Performance Matrix Symbol Results.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		
0	3	1	0	0	0	0	0	0	0	0		4
1	0	5	0	0	0	0	0	0	0	0		5
2	0	0	5	0	0	0	0	0	0	0		5
3	0	0	0	5	0	0	0	0	0	0		5
4	0	0	0	0	5	0	0	0	0	0		5
5	0	0	0	0	0	4	0	0	0	0		4
6	2	0	0	0	0	0	4	1	0	0		7
7	0	0	0	0	0	1	1	1	0	0		3
8	0	0	0	0	0	0	0	0	5	0		5
9	0	0	0	0	0	0	0	3	0	5		8
	5	6	5	5	5	5	5	5	5	5		

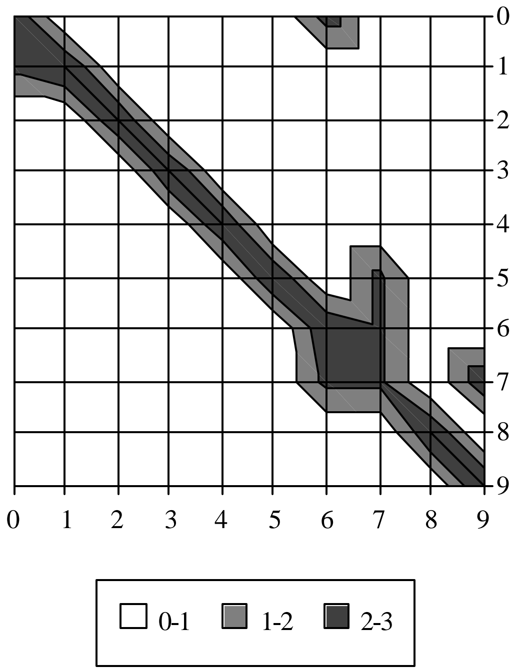


Figure 5-10 Performance Matrix Symbol Contour Plot.

5.4 Side Contour Profile Feature

Obtaining a Side Contour Profile (SCP) of each side surface by scanning an edge to the first black pixel in the image space and counting the pixels traversed allows the outline feature characteristics to be captured into a unique description which can be used as a feature. The size of the mesh will determine the level of detail retained by the feature. Experimentation with this method using 24x24 and 32x32 size bitmaps of characters were tested. These sizes were used after the character image was cropped, eliminating all excess white space around the character. Adding a margin border of white pixels around the character was also required so that a clean break point would be indicated by the side contour profile feature. Without the border it is difficult without further interpretation, to determine where edges of the image object end.

Figure 5-11 illustrates the process of creating the SCP feature. This method is similar to outline tracing but does not require using curve fitting functions. Curve fitting functions have been found to remove characterizing features of characters. Techniques using B-Splines for curve fitting are common, it is especially useful when converting large line drawings into vector data.

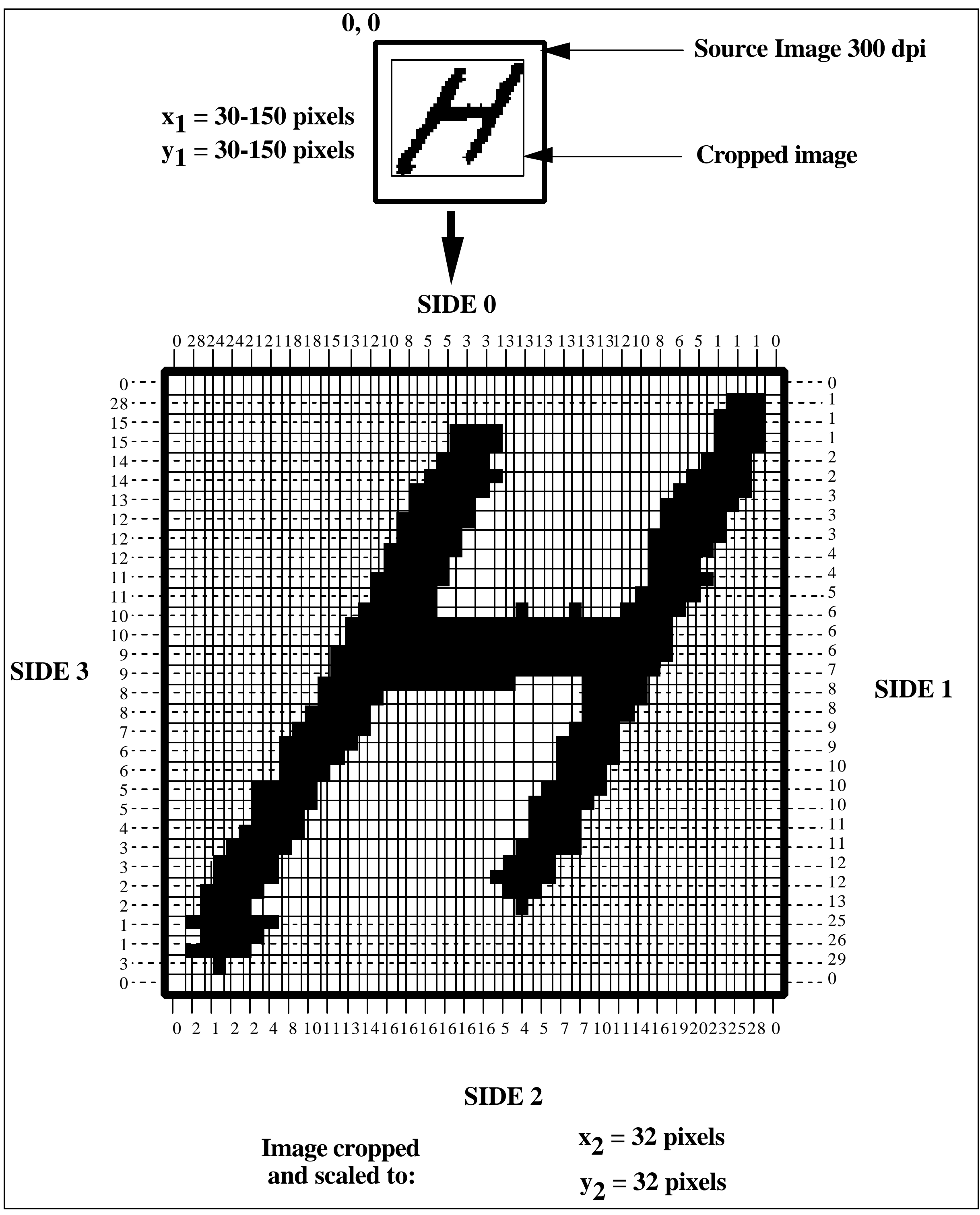


Figure 5-11 SCP Feature Overview.

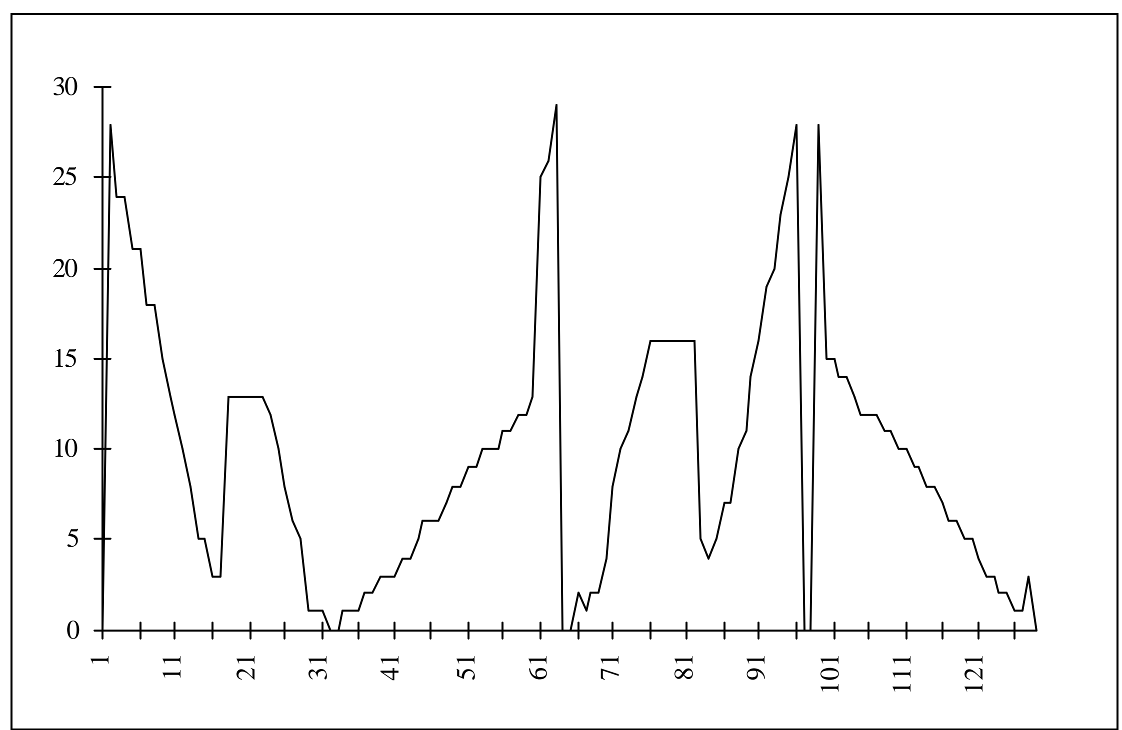


Figure 5-12 SCP Feature Plot.

Figure 5-12 is the plot of the side contour profile data obtained from the figure represented in Figure 5-11. Each surface extends 32 positions and starts at a Y location of 0 and returns to 0. Locations along the X-axis of 1-32 represent side 0, 33-64 side 1, 65-96 side 2 and 97-128 side 3. Table 5-6 summarizes the training and testing. Since the performance was so poor it was obvious that uppercase and symbol testing would not be worth spending any additional effort with this method. Table 5-7 shows that very few characters can be classified using this technique, most character contour side profiles appear as '6' or '8' instead of their true class. Appendix C contains the supporting source for this feature.

Table 5-6 Side Contour Profile Training Summary.

Image Data	Network Topology	Comments	Training Set Size	Test Set Size	NC NCI CUPS	Acc-uracy
Digits 0-9	128.40.10	Training time: 4 hours. Accuracy of 30% peaked after 4 hours of training.	500, 50 per class	510, 51 per class	5,520 520 40,350	13.9%
Digits 0-9	128.30.10	Training time: 2 hours, accuracy peaked at 15% during training.	500, 50 per class	510, 51 per class	4,140 414 35,937	N/A
Digits 0-9	128.20.10	Training time: 2 hours, accuracy peaked at 10%.	500, 50 per class	510, 51 per class	2,760 276 38,333	N/A

Table 5-7 SCP Performance Matrix Digit Results.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		
0	0	0	0	0	1	0	0	0	0	0		1
1	0	0	0	0	0	0	0	0	0	0		0
2	0	0	0	4	3	0	0	0	0	0		7
3	0	4	3	8	7	2	1	0	1	0		26
4	0	0	0	0	0	0	0	0	0	0		0
5	0	2	0	2	2	2	2	8	0	0		18
6	5	13	5	29	20	5	12	24	1	6		120
7	0	0	0	0	0	0	0	0	0	0		0
8	46	33	43	8	18	42	36	19	49	45		339
9	0	0	0	0	0	0	0	0	0	0		0
	51	52	51	51	51	51	51	51	51	51		

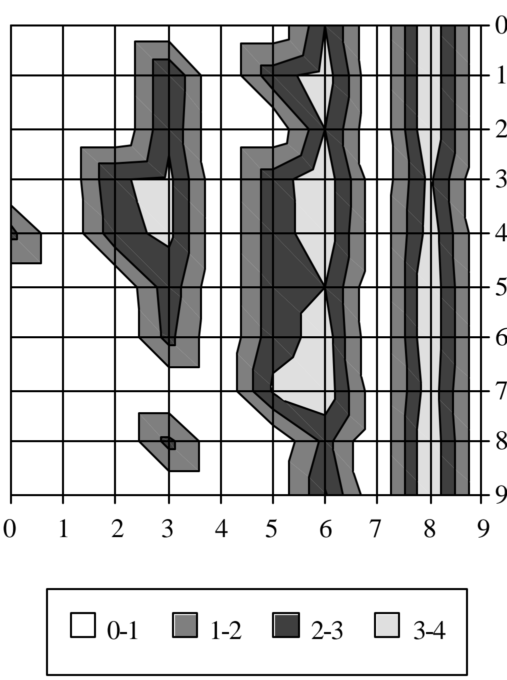


Figure 5-13 SCP Performance Matrix Digit Contour Plot.

5.5 Quadrant Method Feature

Subsampling the normalized image cell of 32x32 bits by using a sub-quadrant mapping is the focus of this feature. The idea here stems from our perception of images, and the minimum number of samples we really need to capture or classify the correct target class. Some of the first dot matrix printers and terminals used a coarse 5x7 dot matrix to form characters. From the 5x7 matrix 35 possible dots existed for forming a character, few people had difficulty recognizing these images. If the high resolution scanned image captured at 300 dpi or 200 dpi could be reduced in resolution, then much of the redundant pixel information could be removed without losing the uniqueness which creates the distinction of the image between classes. Choosing how much data is really required should be range of perhaps 5x5 to 8x8 cell of pixels to represent the original 32x32 pixel array. For testing and evaluation purposes the following sizes of subsampled arrays were evaluated:

Table 5-8 Quadrant Method Subsample Sizes.

Number of Quadrants	Quadrant Size
16	8x8 - 64 bits
64	4x4 - 16 bits
256	2x2 - 4 bits

After initial testing of the artificial neural network simulator models of the various sizes the 64 quadrant subsample size was chosen. For quadrants where noise is present or thin edges exist in a quadrant, a voting or threshold technique is implemented. Each pixel has a 1/16th weight per quadrant. After testing different types of data this was determined empirically. Figure 5-14 illustrates the mapping of the original image pixels 32x32 = 1,024 pixels into 64 (4x4) quadrants.

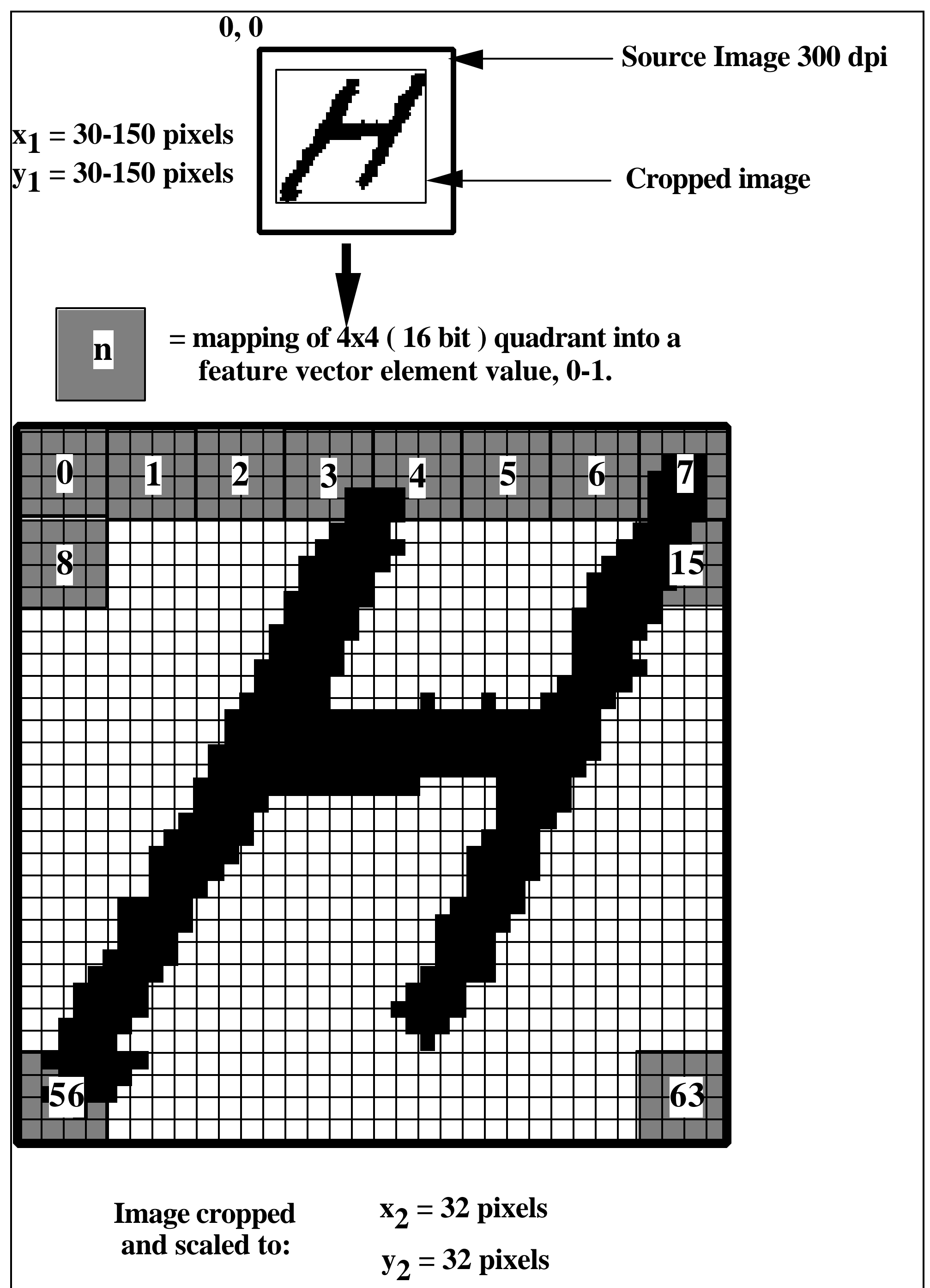


Figure 5-14 QM Feature Diagram.

This method produced impressive results. Not only was the accuracy better than RBM for all three test types, training time was decreased from 30-150 hours to 1 hour. Using only 64 elements for the input vector versus 1,024 for RBM is one difference.

Another difference is using the quadrants where an average, and data compression representation of the original 32x32 cell takes place. One image processing textbook presents a technique similar to this method but does not use an artificial neural network for classification, instead a binary tree is derived. A brief synopsis of a Quad-Tree coding method from *FUNDAMENTALS OF DIGITAL IMAGE PROCESSING*[66] is summarized here. It is used as a method of data compression of the raw feature. It is compared against run-length encoding. Unfortunately spatial x,y relationship information is lost using this technique. It has some similarity to the Quad-Method developed here in the sense of mapping groups of pixels to larger groups. The Quad-Method also votes for each quadrant utilization. No results of testing, or application explanation is given in the text.

Appendix D contains the supporting source code for this feature.

Table 5-9 summarizes results for training and testing this feature method. Following are the performance matrices for each type of data trained and tested, Table 5-10, Table 5-11, Table 5-12, Table 5-13.

Table 5-9 Quadrant Method Training Summary.

Image Data	Network Topology	Comments	Training Set Size	Test Set Size	NC NCI CUPS	Acc-uracy
Digits 0-9	64.20.10	Training time: 1.2 hours.	500, 50 per class	510, 51 per class	1,480 148 10,277	80.8%
Upper A-Z	64.30.26	Training time: 4 hours.	1300, 50 per class	1300, 50 per class	2,700 104 44,318	73.3%
Symbols 0-9	64.20.10	Training time: 0.1 hours	50, 5 per class	50, 5 per class	1,480 148 74,000	98.0%

Table 5-10 Quadrant Method Digit Results.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		
0	44	0	0	0	0	8	0	0	2	0		54
1	1	44	0	0	0	1	0	1	0	0		47
2	1	0	47	3	0	8	0	0	0	5		64
3	0	0	0	32	1	0	0	0	0	0		33
4	0	1	0	9	42	0	2	0	0	0		54
5	0	0	0	0	0	25	0	0	0	1		26
6	1	1	1	3	4	0	40	1	0	2		53
7	0	3	0	0	0	5	1	49	0	1		59
8	4	0	0	2	0	0	1	0	48	0		55
9	0	3	3	2	4	4	7	0	1	42		66
	51	52	51	51	51	51	51	51	51	51		

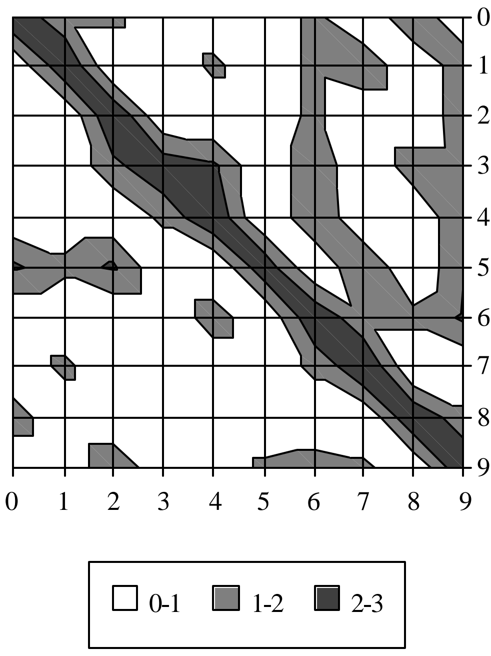


Figure 5-15 Quadrant Method Digit Contour Plot.

Table 5-11 Quadrant Method Uppercase Results Summary.

Class	Comment (Limited to highest 3 choices with a misclassification count greater than 1)
A	J, M, C.
B	I, S.
C	B, E, F, J, T.
D	P, E, F, H, M.
E	P.
F	C.
G	.
H	R, V.
I	B, O, V, Z.
J	Y, S, A, M.
K	J, E, T, V.
L	B, S, I, W, N.
M	L, W.
N	I, O, B.
O	W, L, V, Y.
P	E, R.
Q	G, D, S.
R	H, B, G, L, S.
S	B, O, Y, L.
T	K.
U	K.
V	W, I, K, O, P, X.
W	Z, Y.
X	O, H, I.
Y	J.
Z	.

Table 5-12 QM Performance Matrix Uppercase Results.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z		
A	44	0	2	0	0	0	0	0	0	9	0	0	3	0	0	0	0	2	1	1	0	0	0	0	1	0		63
B	0	26	0	0	0	0	0	0	3	0	0	1	0	1	0	1	0	1	2	0	0	0	0	0	0	0		35
C	1	3	38	0	2	2	0	0	1	2	0	0	0	1	1	1	0	0	1	2	0	1	0	0	0	1		57
D	1	0	0	46	3	3	0	3	0	0	0	0	2	0	0	4	0	0	1	0	0	0	0	0	0	0		63
E	0	0	0	0	30	0	0	0	0	0	0	0	0	0	1	3	1	0	0	0	0	1	1	0	0	0		37
F	0	0	4	0	1	43	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	1	1	0	0	0		54
G	0	1	0	0	0	1	35	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1		41
H	0	1	1	0	1	0	0	35	0	0	0	0	0	0	0	1	0	5	0	0	0	4	0	1	0	1		50
I	0	3	0	0	0	0	1	0	31	0	0	0	0	0	2	0	0	0	0	0	0	2	0	0	0	2		41
J	2	0	0	0	0	1	1	1	0	28	2	0	2	0	0	0	1	0	3	1	1	1	0	0	5	0		49
K	1	0	0	0	2	0	1	0	0	3	41	0	0	0	0	1	0	1	0	2	0	2	0	1	1	0		56
L	0	3	0	0	0	0	0	0	2	0	0	32	0	2	0	0	0	0	3	0	0	0	2	1	1	0		46
M	0	0	1	0	1	0	0	0	0	0	0	3	40	0	0	0	0	0	0	0	0	0	2	0	0	0		47
N	0	3	1	0	1	0	0	0	6	0	0	2	0	44	4	0	0	0	0	0	0	0	0	0	0	2		63
O	0	1	0	0	0	0	0	0	1	0	0	3	0	0	32	0	0	0	0	0	0	2	4	2	0	0		45
P	0	1	0	1	7	0	0	1	0	1	0	0	0	0	0	34	1	4	0	1	1	0	0	0	0	1		53
Q	0	0	0	2	1	0	5	0	0	0	0	0	0	1	0	1	47	1	2	0	0	0	0	0	0	1		61
R	1	4	0	0	1	0	3	6	1	0	1	3	0	0	0	1	0	35	3	0	2	0	0	0	0	0		61
S	0	3	1	1	0	0	0	0	1	1	0	2	0	0	3	0	0	1	33	2	0	1	0	0	3	0		52
T	0	0	1	0	0	0	0	1	0	0	2	0	0	0	0	1	0	0	0	39	0	0	0	0	1	0		45
U	0	0	0	0	0	0	1	0	0	0	2	0	0	0	0	0	0	0	0	0	45	0	0	0	0	0		48
V	0	0	0	0	0	0	1	0	2	0	2	1	0	0	2	2	0	0	1	0	0	34	7	2	1	1		56
W	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	31	0	3	4		40
X	0	1	1	0	0	0	0	2	2	0	0	1	1	0	5	0	0	0	0	1	0	0	0	43	0	0		57
Y	0	1	0	0	0	0	0	0	0	4	0	1	1	1	0	0	0	0	0	0	0	0	1	0	31	0		40
Z	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	2	36		41
	50	51	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50		

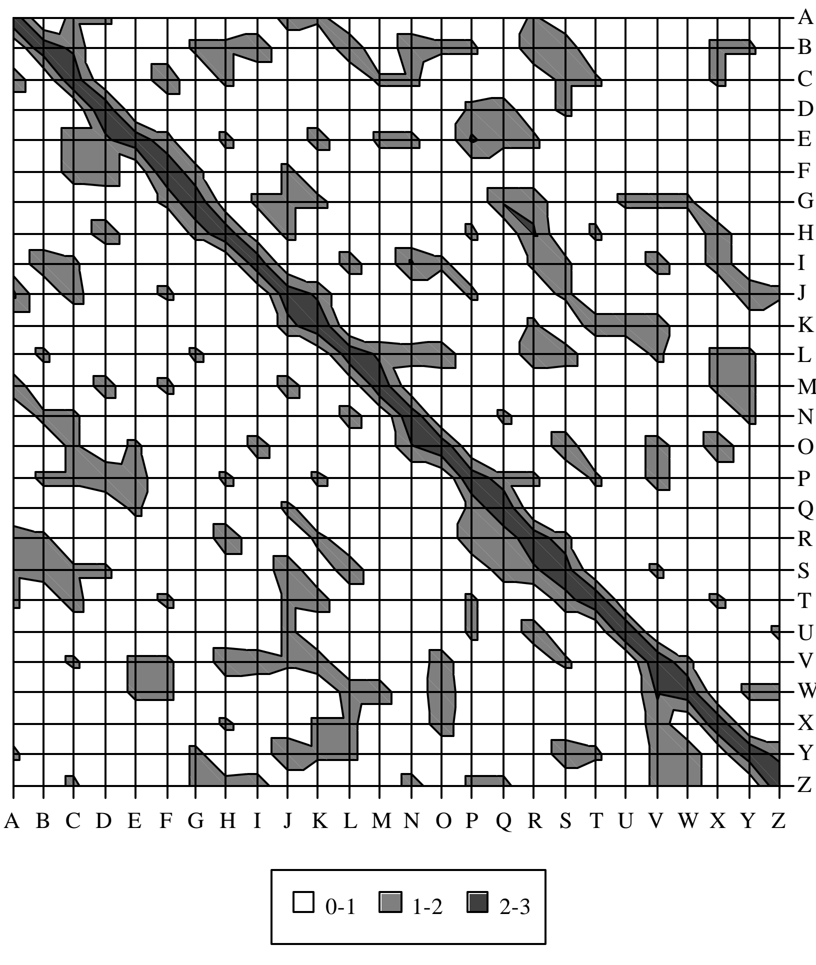


Figure 5-16 Quadrant Method Uppercase Contour Plot.

Table 5-13 QM Performance Matrix Symbol Results.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		
0	5	1	0	0	0	0	0	0	0	0		6
1	0	5	0	0	0	0	0	0	0	0		5
2	0	0	5	0	0	0	0	0	0	0		5
3	0	0	0	5	0	0	0	0	0	0		5
4	0	0	0	0	5	0	0	0	0	0		5
5	0	0	0	0	0	5	0	0	0	0		5
6	0	0	0	0	0	0	5	0	0	0		5
7	0	0	0	0	0	0	0	5	0	0		5
8	0	0	0	0	0	0	0	0	5	0		5
9	0	0	0	0	0	0	0	0	0	5		5
	5	6	5	5	5	5	5	5	5	5		

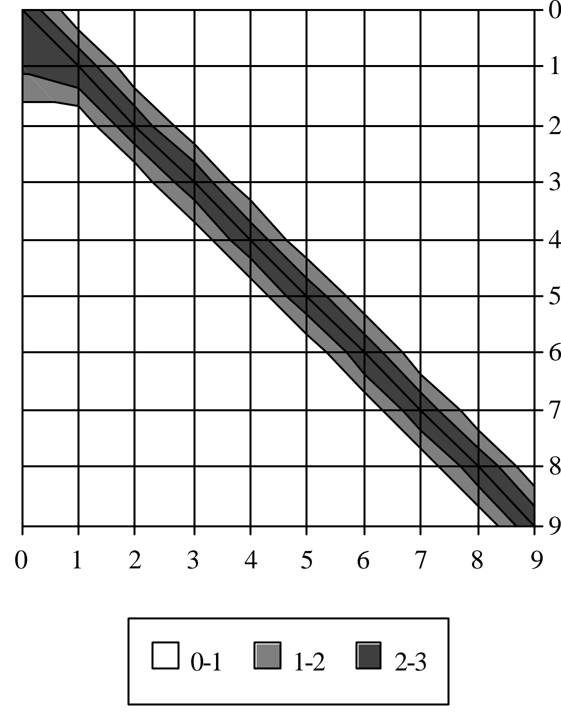


Figure 5-17 QM Performance Matrix Symbol Contour Plot.

5.6 Hough Transform Feature

The Hough[67] transform (HT) technique is a widely published and researched method for determining non uniform or analytic shapes in image space. Hough transform methods are computationally intensive and have had limited production success due to the computational complexity in performing the transform calculations.

The following description is from *Model-Based Image Matching Using Location*[68]. A feature in "image space" is mapped into a locus of possible registrations in the dual "parameter space." The parameter space is tiled and represented by an array of counters. Each feature point "votes" for registrations by incrementing counters in its dual locus. A global maximum count identifies a best registration.

Another description of the Hough transform comes from *Computer Vision*[69], "The classical Hough technique for curve detection is applicable if little is known about the location of a boundary, but its shape can be described as a parametric curve (e.g., a straight line or conic). Its main advantages are that it is relatively unaffected by gaps in curves and by noise."

This description describes the mathematical relationship between the xy space and parameter space[70]. Consider the point (x_i, y_i) and the general equation of a straight line in slope-intercept form, $y_i - ax_i + b$. Infinitely many lines pass through (x_i, y_i), but they all satisfy the equation $y_i = ax_i + b$ for varying values of a and b. However, writing this equation as $b = -x_i a + y_i$ and considering the ab plane (also called parameter space) yields the equation of a single line for a fixed pair (x_i, y_i). A second point (x_j, y_j) also has a line in parameter space associated with it, and this line intersects the line associated with (x_i, y_i) at (a', b'), where a' is the slope and b' the intercept of the line containing both (x_i, y_i) and (x_j, y_j) in the xy plane. All points contained on this line have lines in parameter space that intersect (a', b'). Figure 5-18 illustrates these concepts.

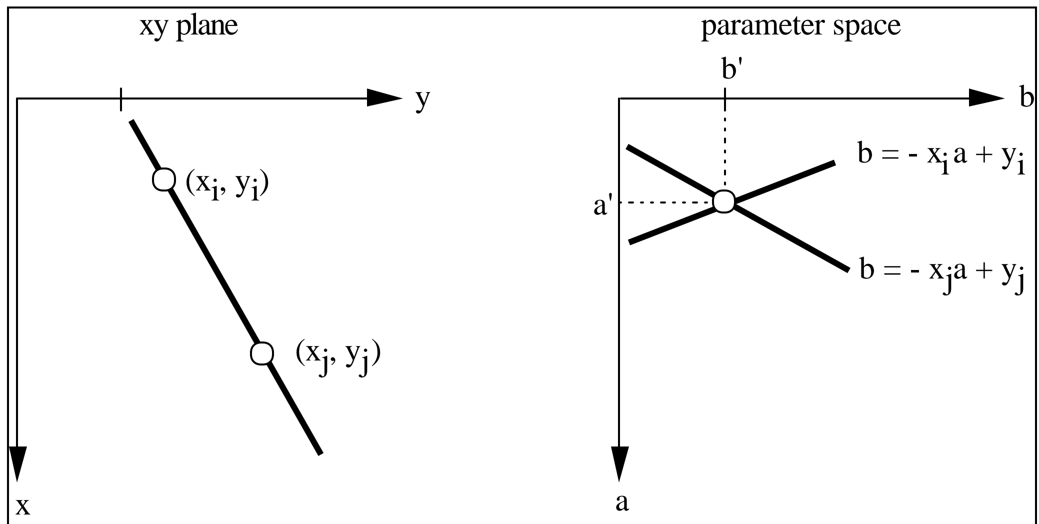


Figure 5-18 Hough Transform Mapping XY Plane to Parameter Space.

The computational attractiveness of the Hough transform arise from subdivision of the parameter space into so-called accumulator cells, as illustrated in Figure 5-19, where (a_{max}, a_{min}) and (b_{max}, b_{min}) are the expected ranges of slope and intercept values. The cell at coordinates (i, j), with accumulator value A(i, j), corresponds to the square associated with parameter space coordinates (a_i, b_j). Initially, these cells are set to zero. Then, for every point (x_k, y_k) in the image plane, we let the parameter a equal each of the allowed subdivision values on the a axis and solve for the corresponding b using the equation $b = -x_k a + y_k$. The resulting b's are then rounded off to the nearest allowed value in the b axis. If a choice of a_p results in solution b_q, we let A(p,q) = A(p,q) + 1. At the end of this procedure, a value of M in A(i, j) corresponds to M points in the xy plane lying on the line $y = a_i x + b_j$. The accuracy of the collinearity of these points is determined by the number of subdivisions in the ab plane.

Note that subdividing the a axis into K increments gives, for every point (x_k, y_k), K values of b corresponding to the K possible values of a. With n image points, this method involves nK computations. Thus the procedure just discussed is linear in n, and the product nK does not approach the number of computations discussed at the beginning of this section unless K approaches or exceeds n.

A problem with using the equation $y = ax + b$ to represent a line is that both the slope and intercept approach infinity as the line approaches the vertical. One way around this difficulty is to use the normal representation of a line:

$$x \cos \Theta + y \sin \Theta = p$$

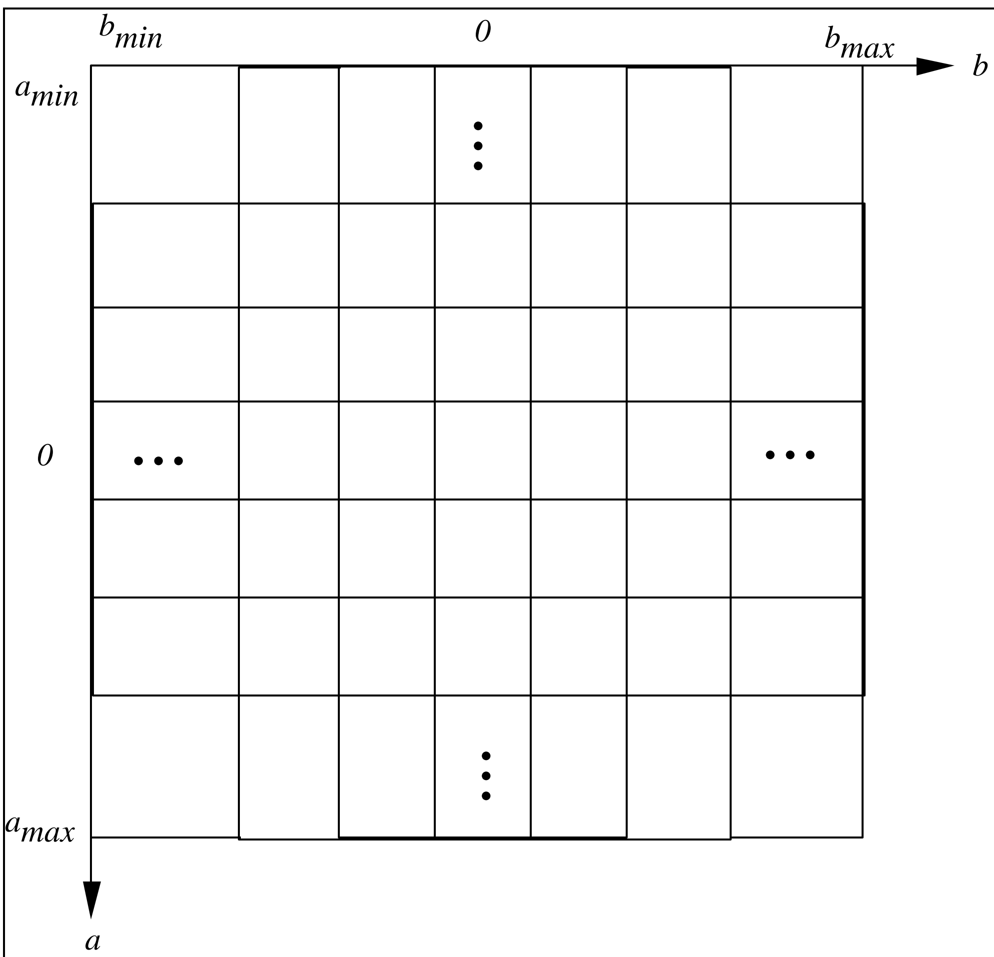


Figure 5-19 Hough Quantization Of The Parameter Plane.

The capability of being able to operate with noise and gaps in the objects makes it an attractive method for implementation. Many handwritten documents have noise and breaks in the character images due to the writing instrument and or scanning of the source document into digital form. This method may be used sparingly or when others fail because of the computations required to perform the transform.

To understand the algorithm utilized it was rewritten and modified using mathematically created lines for testing. All source code for support of the Hough transform appears in Appendix E. The Hough technique is a voting technique, based on some shape space used as a template for the vote. Voting occurs on all black pixels in the source image for any line of pixels passing through the current pixel under observation. Though this technique is using a line for this thesis other shapes such as ellipses, circles, boxes and triangles could be used to vote for features in Hough space.

Another description of the Hough transform is given from the viewpoint of being in the parameter space, *OBJECT RECOGNITION BY COMPUTER, The Role of Geometric Constraints*[71].

The generalized Hough transform finds possible solutions to the object pose problem by searching for large clusters of evidence in a discrete version of a parameter space. A parameter vector, p, represents a point in an n-dimensional space, P. Each point in P maps to a point in the n-dimensional discrete Hough space, H, that is specified by quantizing each of the n components of p. The Hough transform method is often referred to as parameter hashing, because each quantized parameter value is a hash key. Implementations of Hough generally use an n-dimensional table to represent H, and refer to table entries as buckets[72].

In the technique implemented a line in the source image is mapped into three dimensions of the Hough space, Θ (0 - 180°), -r to +r, vote count (VC). The Θ represents the slope of the current pixel under observation. For Figure 5-18 source image the angle is 45° for the majority of the pixels, in Hough space this line will translate to the 135° point on the Θ axis. The range 0° to 180° can be mapped to -90° to 90°, in which case 135° is the same point on the axis as 45°, which is the angle of the line in the source image. Symbol r has been mapped and normalized for a maximum source image of 256x256, it represents the Hough space position of the source image. For Figure 5-18, (r = 0). VC represents the number of pixels which were accumulated as votes for a particular line in the image space. Since the transform operates from the center of the image out to the edges and a 256x256 source image maximum is assumed the maximum number of contributing votes for any quadrant is 181. In implementation for the character images used for this thesis the images are much smaller. Vote

Counts for lines typically are in the 1-50 range, however to maximize the dynamic range of the 8 bits per pixel as available, a normalization is performed to scale the vote counts from 0 to 255.

Appendix E contains the supporting source code for this feature.

This is the algorithm description from the ALV toolkit:

ALV Public Domain Image Processing Toolkit for Sun Workstations[73]
Phillip G. Everson

Computer Science Dept.
Bristol University
United Kingdom

The ALV toolkit is a collection of programs supporting image processing research on Sun workstations. It uses the standard Sun rasterfile format allowing images of variable size and depth. Programs exist for general manipulation of images and to display them on various devices.

SYNOPSIS
hough [infile | -] [outfile | -]

DESCRIPTION
hough performs a Hough transform on a 1 bit deep raster to produce an 8 bit deep output of hough space.

In the output raster the X axis represents theta and the Y axis r. The origin for the hough transformation is taken to be the center of the image so as to keep the range of r, and hence the size of the final image, to a minimum.

Integer lookup tables for sin and cos are used. These record the actual value * 1000. This has the effect of working to three decimal places but speeds up the transformation tremendously; however, this will cause a loss of accuracy on values of r over 1000. As the maximum possible value of r for a normal (256 x 256) image is 181, this should not be a problem. The transformation is only performed over 180 degrees as the 'second half' is just a reflection of the first 180 degrees and is not necessary.

Figures 5-19 and 5-20 diagram the steps involved in processing the Hough transform. Extra steps are necessary due to the PBM format used. Figure 5-21 is a sampling of test images created to illustrate the transform on understandable geometric figures. The Hough maps represented in Figure 5-21 have had post processing performed on them:

1) The original source image is shown somewhere in the frame.

2) Inversion and an equalization stretch was performed to fully utilize the gray levels available in the printing device.

Figure 5-20 Hough Image to Hough Parameter Space Feature Diagram.

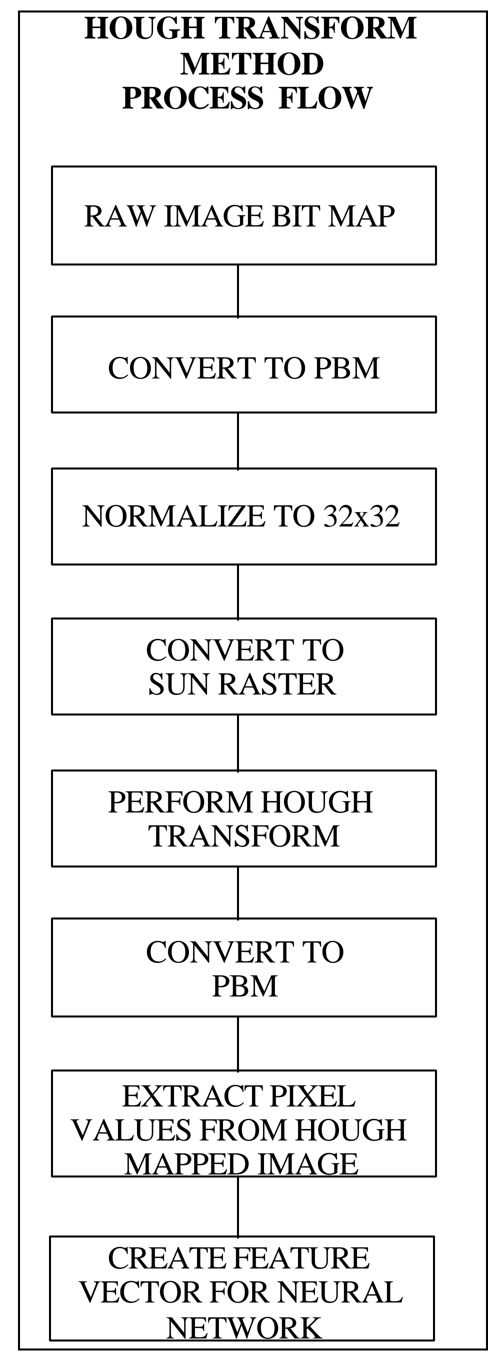


Figure 5-21 Process Flow of Hough Feature Diagram.

HOUGH TRANSFORM METHOD DATA FLOW

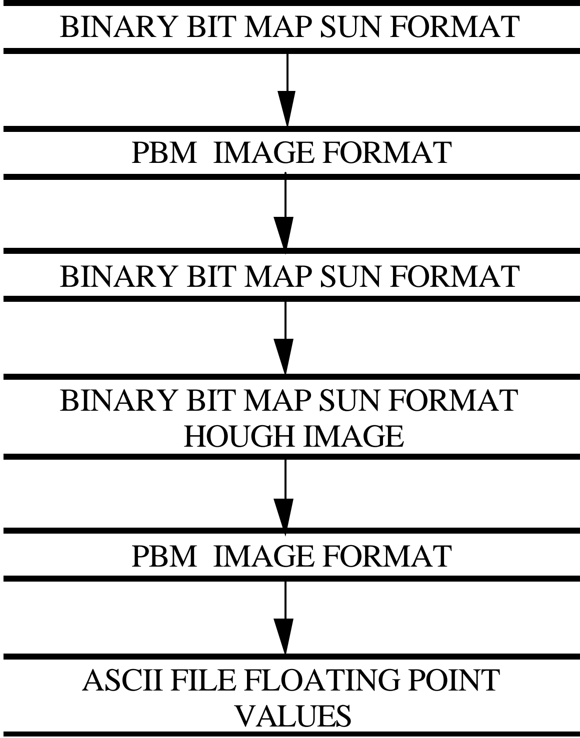


Figure 5-22 Data Flow of Hough Feature.

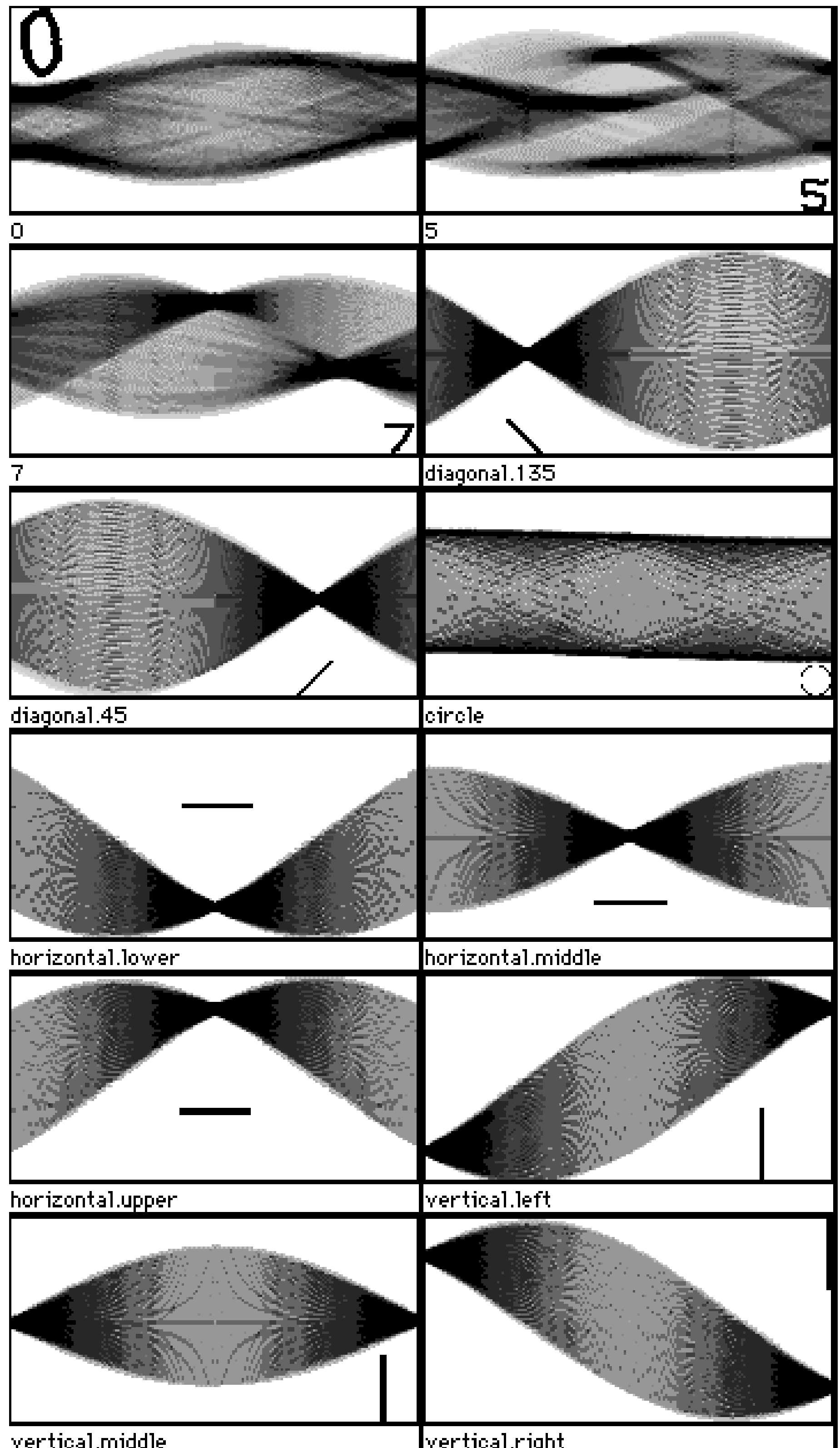


Figure 5-23 Hough Transform Examples.

Table 5-14 Hough Method Training Summary.

Image Data	Network Topology	Comments	Training Set Size	Test Set Size	NC NCI CUPS	Acc-uracy
Digits 0-9	8100.20.10		500, 50 per class	510, 51 per class	162,200 16,220 29,109	75.7%
Upper A-Z	8100.30.26		1300, 50 per class	1300, 50 per class	243,780 9,376 20,330	70.6%
Symbols 0-9	8100.20.10		50, 5 per class	50, 5 per class	162,200 16,220 14,610	98%

Table 5-15 Hough Method Digit Performance Matrix.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		
0	43	0	0	0	1	11	1	0	2	4		62
1	1	45	0	1	0	0	3	0	0	0		50
2	0	0	43	0	0	6	0	0	1	6		56
3	0	1	0	37	2	0	0	0	0	2		42
4	0	1	0	2	43	0	8	0	2	1		57
5	0	0	0	0	0	30	0	0	0	1		31
6	0	1	3	1	2	0	20	2	0	3		32
7	0	1	1	2	1	3	6	49	0	2		65
8	7	0	0	1	0	0	0	0	45	0		53
9	0	3	4	7	2	1	13	0	1	32		63
	51	52	51	51	51	51	51	51	51	51		

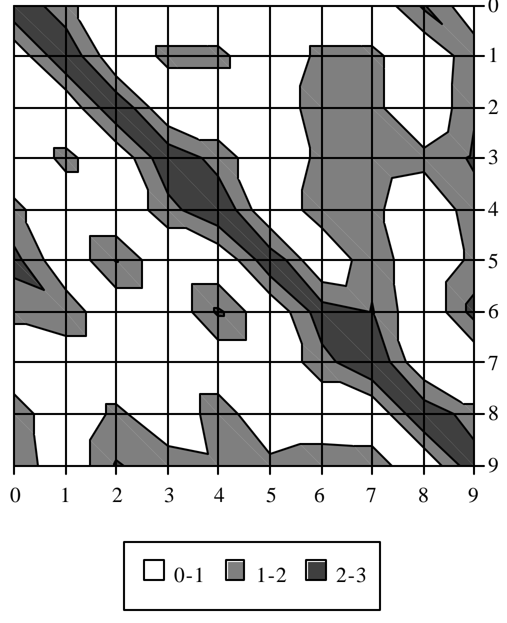


Figure 5-24 Hough Method Digit Performance Matrix Contour Plot.

Table 5-16 Hough Method Uppercase Results Summary.

Class	Comment (Limited to highest 3 choices with a misclassification count greater than 1)
A	J, E, F, K.
B	G.
C	T, F, J, B.
D	H.
E	P, K, C.
F	B, L, S.
G	Z.
H	X, R, T.
I	O, B.
J	K, Y.
K	J.
L	I, W, B.
M	.
N	B, I, Z.
O	X.
P	C.
Q	G.
R	P, H, N.
S	L, N.
T	K, Q, Y.
U	O, M, I.
V	W, O, I, M.
W	Y, Z.
X	O, Y, I.
Y	L, J.
Z	.

Table 5-17 Hough Performance Matrix Uppercase Results.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z		
A	45	1	1	2	3	3	0	1	0	13	3	0	1	0	0	0	0	1	2	1	0	0	0	0	1	0		78
B	0	31	1	0	1	0	2	1	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0		40
C	1	2	40	0	0	3	1	1	2	3	0	0	0	1	1	1	0	0	2	4	0	1	0	0	0	1		64
D	0	0	0	46	0	2	0	2	0	0	1	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0		54
E	1	0	2	0	38	0	0	1	0	0	3	0	0	0	0	6	0	0	1	0	0	1	0	1	0	0		54
F	0	5	0	0	0	39	1	0	0	0	0	3	0	0	0	0	0	0	3	2	0	0	0	0	1	0		54
G	0	0	0	0	0	0	32	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	3		38
H	0	0	1	0	3	1	0	33	0	2	0	1	0	0	1	0	0	8	0	4	0	1	0	9	1	3		68
I	0	3	0	0	0	0	0	0	30	0	0	2	0	0	6	0	0	0	0	0	0	0	0	0	0	0		41
J	3	0	0	0	0	0	1	0	1	22	4	0	1	0	0	0	2	1	2	1	0	0	0	0	3	0		41
K	0	0	0	0	0	0	0	0	0	3	30	0	0	0	1	1	0	0	0	1	1	0	0	0	0	0		37
L	0	2	0	0	0	0	0	0	3	1	0	29	0	1	1	0	0	0	0	0	0	0	3	0	1	0		41
M	0	0	0	0	1	0	1	0	0	0	0	1	44	0	1	0	0	0	0	0	0	0	0	0	0	0		48
N	0	5	0	0	1	0	0	0	5	0	0	0	0	39	2	0	2	0	0	0	1	0	0	0	1	3		59
O	0	0	0	0	0	0	0	0	1	0	0	0	0	0	24	0	0	0	0	0	0	1	0	3	0	0		29
P	0	0	3	0	1	0	0	2	0	0	1	0	0	0	0	32	0	2	0	0	0	0	0	0	0	0		41
Q	0	1	0	0	0	0	5	0	0	0	0	0	0	1	0	0	44	0	0	0	0	0	1	0	0	0		52
R	0	1	1	0	2	0	2	7	0	0	0	0	0	3	2	9	2	33	2	0	0	1	0	0	0	1		66
S	0	0	0	2	0	1	2	0	0	0	1	8	0	3	1	1	0	1	37	0	1	0	0	0	2	0		60
T	0	0	0	0	0	1	0	0	1	0	2	0	0	0	0	0	0	2	0	36	0	0	0	0	2	0		44
U	0	0	0	0	0	0	1	0	0	0	2	0	0	0	0	0	0	0	0	0	46	0	0	0	0	1		50
V	0	0	1	0	0	0	0	1	4	1	2	0	4	0	6	0	0	0	0	0	0	44	8	3	0	3		77
W	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	32	1	2	2		42
X	0	0	0	0	0	0	0	1	2	0	0	0	0	0	4	0	0	0	0	0	0	0	4	32	4	1		48
Y	0	0	0	0	0	0	0	0	0	2	0	3	0	1	0	0	0	0	0	0	0	0	1	0	30	1		38
Z	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	31		37
	50	51	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50		

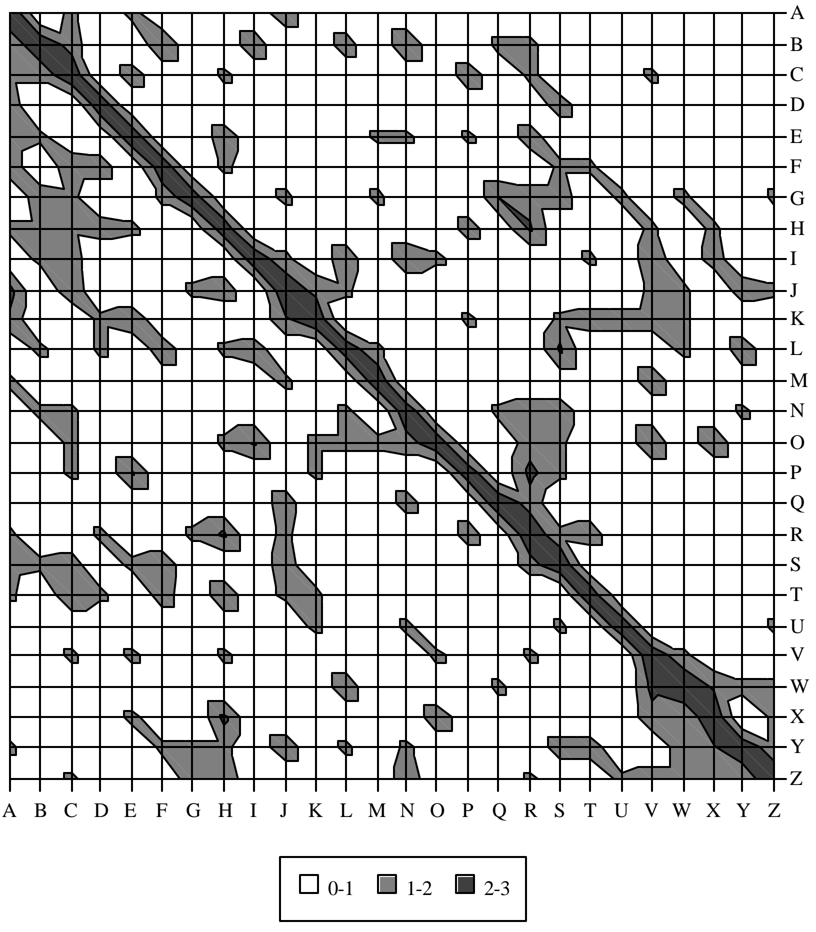


Figure 5-25 Hough Performance Matrix Contour Plot.

Table 5-18 Hough Performance Matrix Symbol Results.

	0	1	2	3	4	5	6	7	8	9		
0	5	1	0	0	0	0	0	0	0	0		6
1	0	5	0	0	0	0	0	0	0	0		5
2	0	0	5	0	0	0	0	0	0	0		5
3	0	0	0	5	0	0	0	0	0	0		5
4	0	0	0	0	5	0	0	0	0	0		5
5	0	0	0	0	0	5	0	0	0	0		5
6	0	0	0	0	0	0	5	0	0	0		5
7	0	0	0	0	0	0	0	5	0	0		5
8	0	0	0	0	0	0	0	0	5	0		5
9	0	0	0	0	0	0	0	0	0	5		5
	5	6	5	5	5	5	5	5	5	5		

HOUGH - Symbols

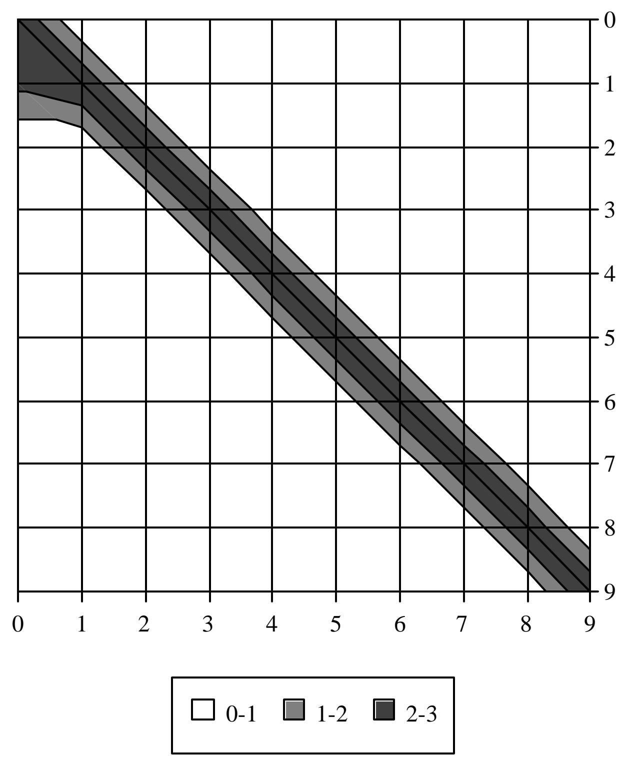


Figure 5-26 Hough Performance Matrix Symbol Contour Plot.

CHAPTER 6

FINAL ANALYSIS AND CONCLUSION

6.1 Introduction

This thesis represents approximately nine months of research and study in the areas of object oriented programming with C++, artificial neural network simulators, image feature extraction methods, file I/O, image processing and analysis. The research performed is still under investigation and will continue to be researched for the foreseeable future. It is unlikely that this problem will be completely solved in the near term. OCR intrigues many researchers and developers and as machine printed recognition becomes an essentially solved problem, hand printed text, and cursive will be the next level of effort for research and production. Though much research focuses classification problems of printed text images the domain that this system can span is not limited to text recognition. Fundamental techniques and integration of the entire system can be applied to areas such as remote sensing, Geographical Information Systems (GIS), military target recognition systems, physiological monitoring and many other domains where imaging or data acquisition is utilized.

Components of this system are extendible into different areas depending on the problem, making this system robust and flexible enough to be used in other domains.

The remainder of this chapter will examine accuracy, performance of the system as implemented and tested, complexity, artificial neural network paradigms, possible new areas of research and the conclusion.

6.2 Accuracy

Based on the rigorous and distorted test data samples the overall accuracy obtained in this experiment is very good. Figure 6-1 is a summary of the accuracy of each method against the respective data type.

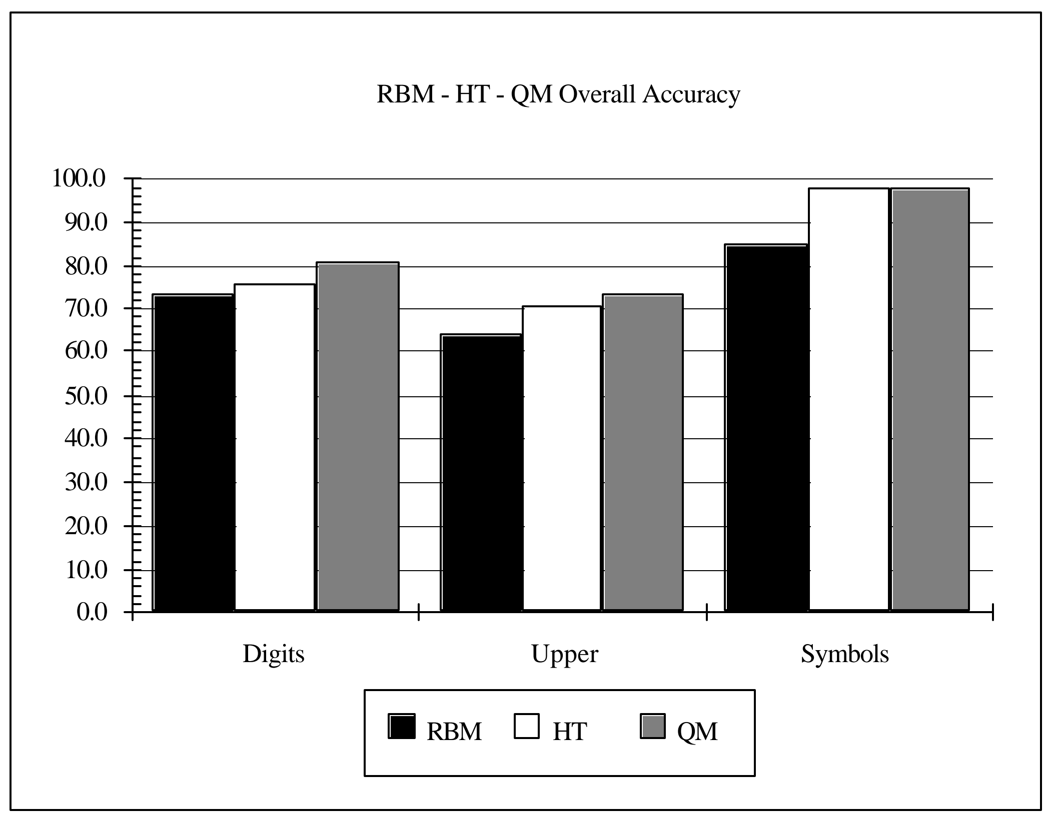


Figure 6-1 RBM - HT - QM Overall Accuracy.

Figures 6-2, 6-3 and 6-4 break down each methods accuracy for each unique class in the data set. Expectations of some large variances in accuracy among techniques were diminished by examining these plots. In the digit class *3*, *5*, and *6* proved to be problems for all methods. The class *6* proved very detrimental to the Hough technique, *6* by itself probably caused the overall performance of Hough to come in second place for digits. It is interesting that *9* which is a virtual reflection of *6* inverted horizontal did so much better by the Hough method.

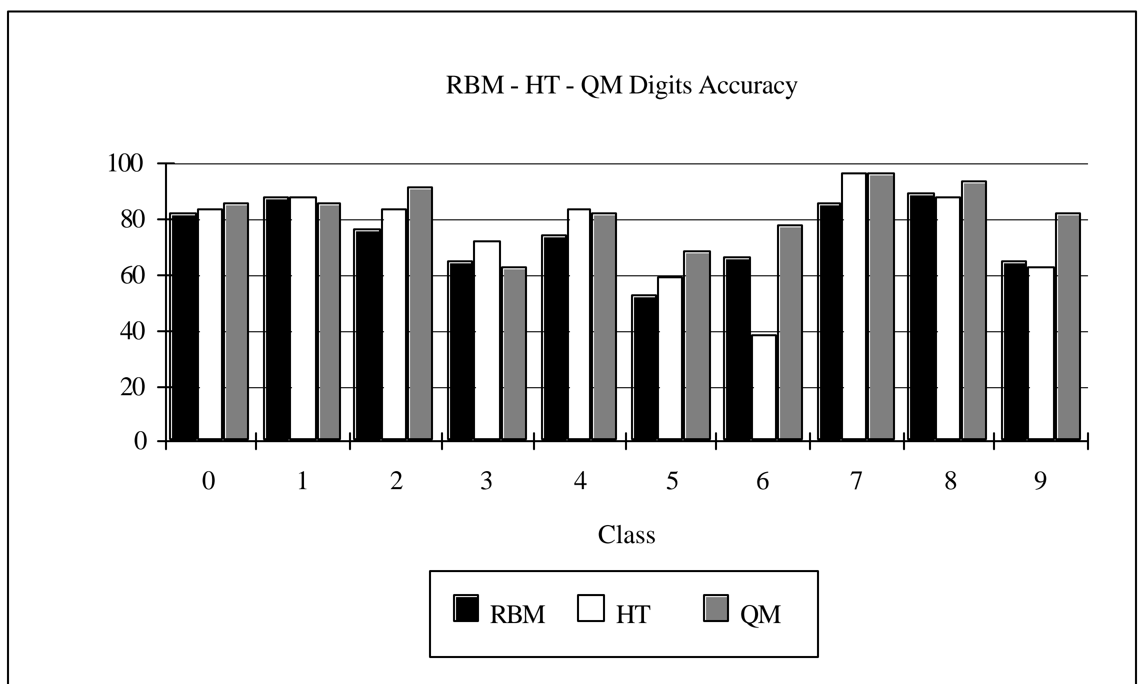


Figure 6-2 RBM - HT - QM Digits Accuracy.

Uppercase is the most complex set of patterns to classify of the three data sets. Extremely low performance is noted in the plot by RBM for B, I, and L characters. Hough had similar problems with J, and O coming in with the lowest accuracy. QM also suffered from poor recognition of the B, and J classes. Large differences in performance for the A, E, J, K, V and X classes indicate certain feature sensitivity for these characters is superior over the lower scoring methods.

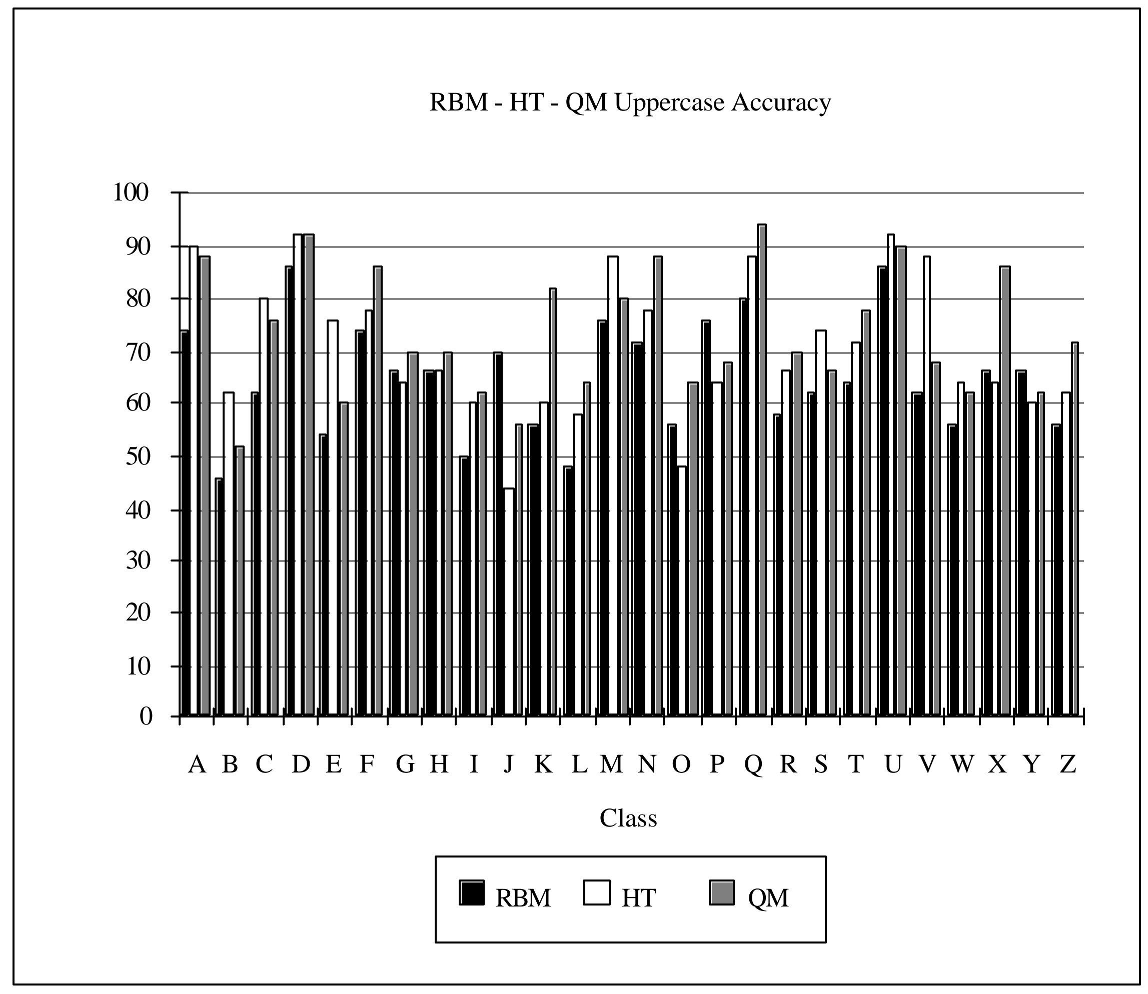


Figure 6-3 RBM - HT - QM Uppercase Accuracy.

Symbols proved to be extremely well behaved and easy to classify by two of the three techniques. Shape, scale and rotation are well controlled features when printed by the draftsperson. Global symbol recognition of engineering drawings appears to be another interesting area of research. To globally determine the locations and type of symbols on a number of related engineering drawings and then relate this information through global database for interpretation.

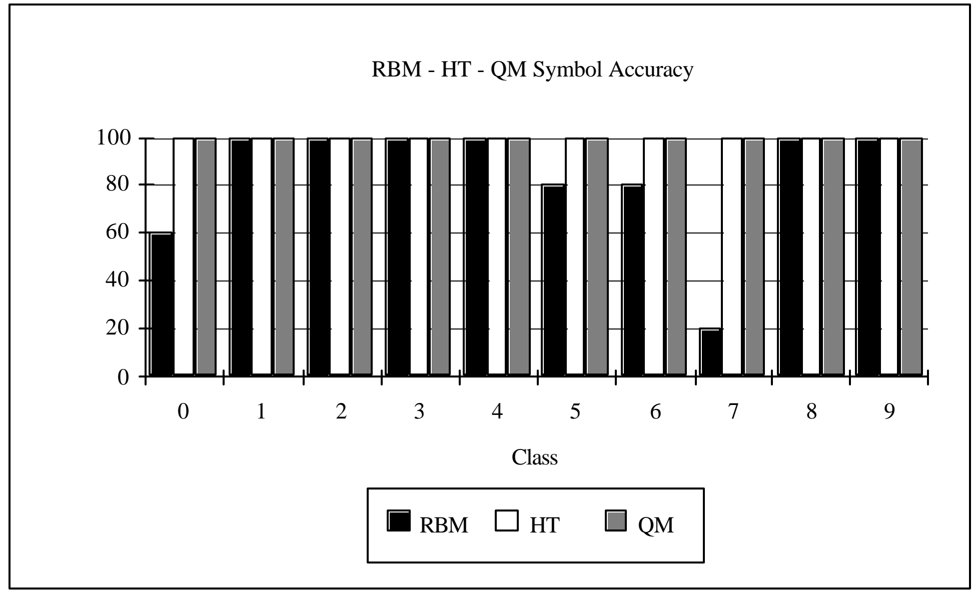


Figure 6-4 RBM - HT - QM Symbol Accuracy.

6.3 Complexity

Complexity as defined in Chapter 4 relates the number of interconnections between nodes and the number of outputs that can be classified by the network. This measure provides a metric for number of calculations required for the network and the memory requirements of the simulator. For this graph the lower the value the better, indicating less memory requirement and a lower number of calculations.

If we relate levels of complexity of this problem which for the human are relatively simple, with more confusing operations such as moving targets, obscured objects as in robotic applications; it clearly shows how well the human body is designed with its sensors for pattern recognition. The quadrant method column below has its values printed directly above the column for a clear indication of the values in this relatively simple network.

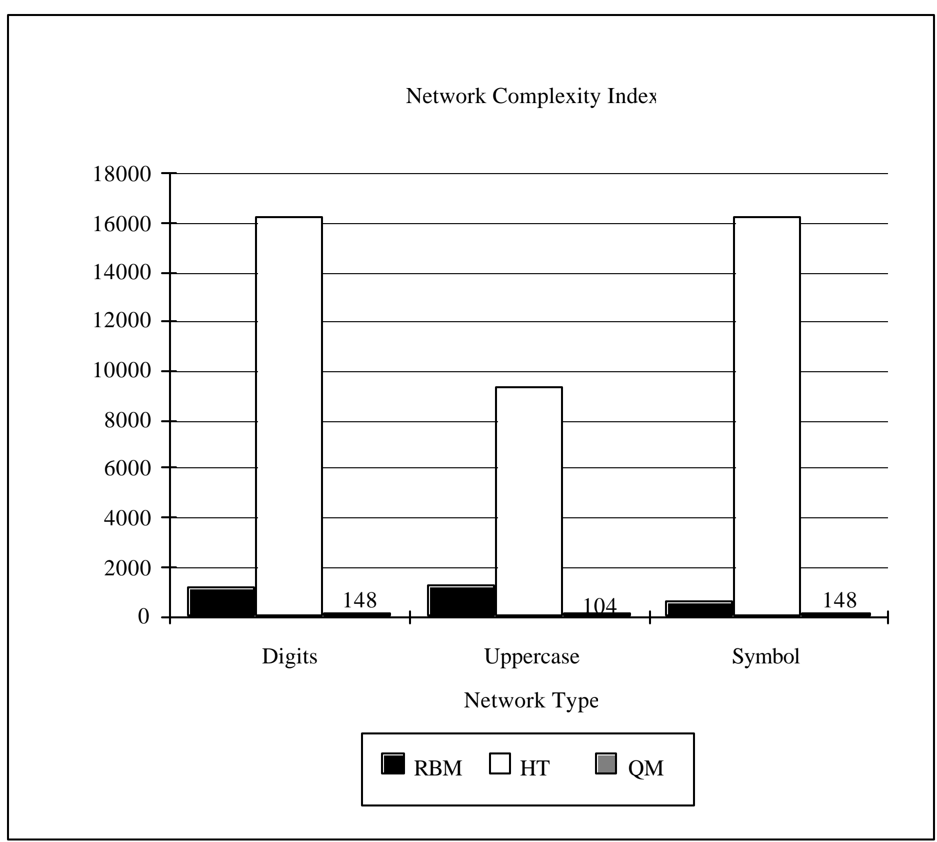


Figure 6-5 RBM - HT - QM Network Complexity Index Comparison.

6.4 Performance

Performance is a highly subjective measurement due to uncontrollable situations when using a multi-user computer system. As defined in Chapter 4, CUPS is a metric for at least obtaining a rough order of magnitude indicating what level of performance can be expected on different machine architectures with a given simulator. For a real performance evaluation a single test case could be run on a multi-user UNIX workstation by operating it in single-user mode. The higher the value here would indicate a 'faster' type of network.

Properties which can affect performance include the paradigm used for the neural network, the size of the raw data vector or feature, total number of outputs necessary for classification. and a conceptual model for mapping the problem into a neural network.

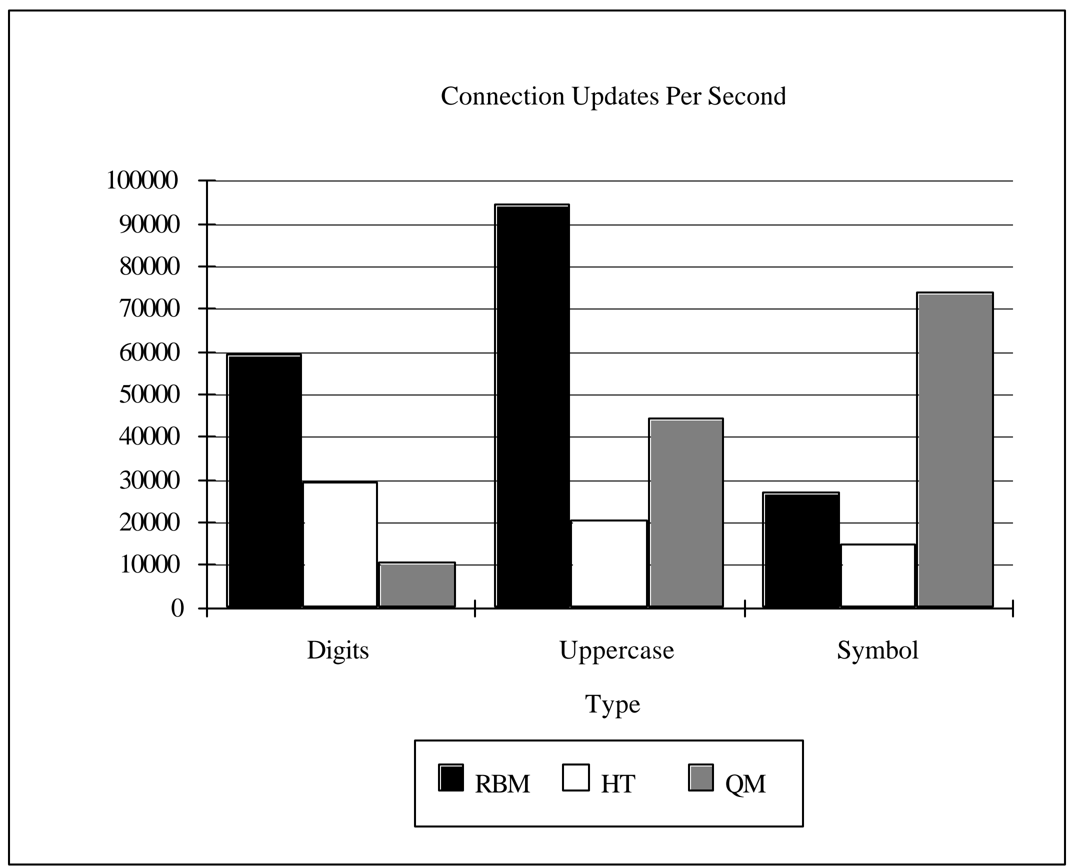


Figure 6-6 RBM - HT - QM Connection Updates Per Second.

6.5 Object Oriented Artificial Neural Networks

Concentration on the backpropagation paradigm for the simulator allowed the author to understand how a fully connected neural network is simulated. Many other paradigms exist; each has been developed with particular problem considerations. In particular, variations of the paradigms which are self organizing would be interesting to apply to cases where the backpropagation scheme has low confidence output values. Study and classification of the problem patterns for possible discriminator networks would also be of benefit in this area. For example if a 3 and 8 is always being confused or has similar output activation levels, a customized network could be created to only differentiate between each type of these two classes. When activation levels from the backpropagation network were very similar an

additional piece of information could be obtained by sending it into the custom two class network.

According to BLUM[74], other paradigms included in his work, (Bi-directional Associative Memories, Counter Propagation, and Hopfield Nets) would ideally be compiled and work with the existing vector and net classes. Based on my experience with the backpropagation code, there is probably further work required to get the other paradigms to operate satisfactorily.

6.6 Feature Methods

With many different ideas and representations of the raw image for a feature, this area of research seemingly could continue forever. Providing different methods to collapse or compress the raw image data into a unique representative feature vector is challenging, and with over a dozen methods reviewed no single method proved to be the ultimate mapping. Because of the variations in techniques, and the individual performance characteristics of each method a logical next step would be integration of techniques which perform well on different sets of the classes to be recognized. An example of this for digits would be a feature which can detect closed loops or circles, 0, 6, 8, and 9 would be optimized and be weighted more heavily than another network which may excel in straight line stroke detection. Combining the outputs of this example would allow the straight line feature technique (HOUGH) to contribute a larger proportion of its output to the straight stroke length characters, 1, 4, and 7 for example.

Many permutations of the above example can be tested and analyzed to determine which feature method should be believed the most for a particular class. This voting or proportional output scheme should raise the overall recognition accuracy to higher levels, than would be achievable with any of the individual techniques.

Figure 6-7 and following description provides a review of the entire pattern recognition effort along with a tree diagram showing the relationship of each technology area to pattern recognition.

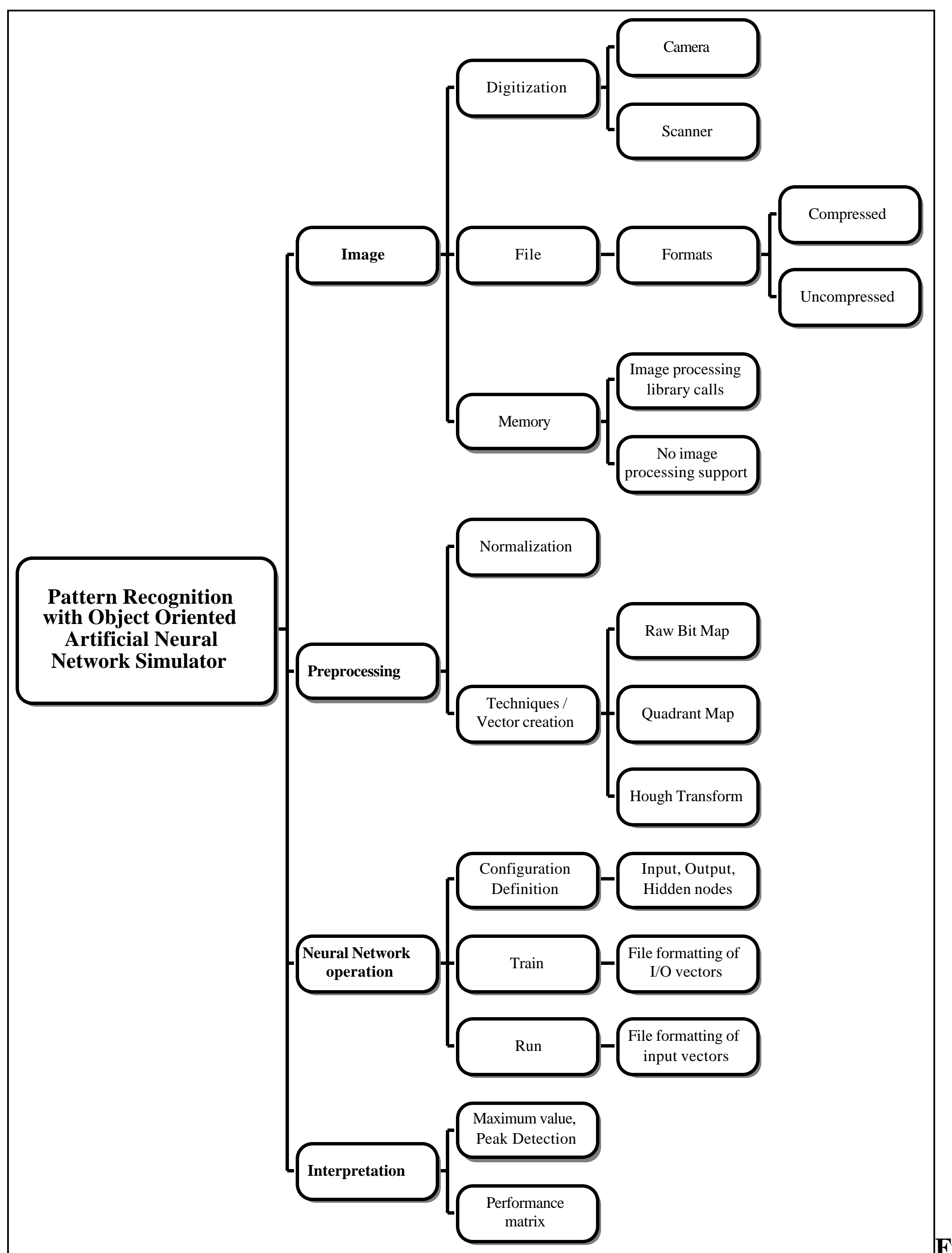


Figure 6-7 Pattern Recognition Overview.

Using Figure 6-7 as a guide, a review of the areas presented follows:

• Image

Images were acquired using a database which National Institute of Standards and Technology created specifically for hand printed recognition testing and analysis in preparation of use of such systems in a census bureau census poll. Symbol images were acquired using a flatbed scanner at a scanning resolution of 200 dpi. File formats were CCITT GROUP4 from NIST and SUN raster file from the scanner. All images were converted into a common PBM format as described in Chapter 5. Memory operations were supported for the SUN raster format, this 'Pixrect' library allowed simplified loading and interpretation of the image data loaded from compressed file format.

• Preprocessing

Images were normalized to a square grid. Other feature methods may leave the raw source image original in size. This has an advantage and a disadvantage: the advantage is if characters are well formed then more accurate representations would be provided. If however there is a lot of noise and the characters are distorted information which is useless and misleading may pollute the training data. Techniques for vector creation are covered in Chapter 5 which represent different levels success.

• NN Operation

Disadvantages: Understanding and interpretation of the network used was more difficult than first thought. The text describes general ideas and does not give a complete example. The description provided did not agree with the what was in the source code and implementation provided. The text description may even have been written by a different person than the author of the software. If a commercial neural network simulator package were used these problems would have been avoided. Advantages: Having access to the source code and a compiler allowed study and understanding of an object oriented design. Modifications as

described in Chapter 4 were possible because the source code was available. Extensions, enhancements and documentation can all benefit from having this available. Though a commercial package would have been easier and cleaner from a user viewpoint, learning how this simulator operates allows extendibility, cross platform development porting and studying of techniques which would not be available from a commercial package.

• Interpretation

Many methods of examining what the output activation values represent are being researched. Voting, statistical, and linear operations have been applied to determine the first guess, second guess and the confidence represented by the output activation value. Exploration using neural networks to interpret the outputs is also a very promising technique which through training can determine non-linear mappings of the output vector relative to the input presented.

6.7 Conclusion

The main goals of this thesis were implementation and understanding of artificial neural networks using an object-oriented approach, research and creation of unique and a traditional feature types, and metrics for measuring performance and complexity of the system. A foundation for image understanding with the use of artificial neural networks as a classification technique has proven to be useful in this domain and is applicable across other image problem domains. Understanding the backpropagation neural network architecture, implemented in an object oriented language, has allowed emerging research areas to be used together to accomplish pattern recognition.

A number of recent developments in computer systems have made this tehsis research possible. Lower cost and more powerful computer hardware and software systems allow research such as this to be carried out with meaningful data sets and test sets. In the past statistical, mathematical and rule based systems have been applied to classifying the image feature data. Most data sets were small and had localization properties as the number of writers producing the data was limited. Today organizations such as the National Institute of Standards and Technology, United States Postal Service and foreign researchers are making available large quantities of organized and classified data; giving new researchers a large jump into the field. Another resource which is critical is the worldwide electronic network INTERNET, allowing researchers from different geographic areas and time zones to communicate with related researchers around the planet. Along with complex tools are complex questions and issues which must be resolved in order to make a system come together for research or production. Much of this thesis focused on how the various subsystems are related and interfaced through data abstraction and representations at each level.

Abstraction of the artificial neural network in the object oriented language C++ allows other neural network paradigms to be integrated and reuse class libraries of software for another gain in the researchers toolkit. Optimization of class libraries and the network paradigm

under study can be explored individually and when improved can benefit other parts of the system collectively. Research needs to be expanded and developed to describe training issues and complexity issues so trial and error sessions of changing parameters can be reduced or eliminated. Methods of combining different neural network types in order for them to reinforce or augment each others decision is needed in order to correct or adjust the poor performance certain patterns cause in the network models. Generalization of the non-linear function developed by the neural network is another difficult area. Without hundreds of thousands or millions of characters and symbols to train with, a network will still produce low confidences for cases that are marginal but correct. The analogy that the neural network is modeling physiological systems may be an analogy for the connections. In reality, however, how humans match patterns is still unknown and we seem to be able to generalize shapes, patterns and objects with very few training samples. Heuristics about patterns and objects are obviously required in the next level of recognition.

For example knowing that the last field of a mail piece address is a ZIP code and either 5 or 9 digits with other dependencies based on the state and city make automatic address recognition possible. Context of the situation is as important as the individual symbols when attempting to recognize information with less then 100% confidence of the symbols.

It would be a very important breakthrough if an object or pattern description mathematical model could be developed. As in the case with Fractals a set of equations can represent complex and recursive shapes, eliminating the need for storing thousands of bits of raw data to represent an image. Today the state-of-the-art systems are employing artificial neural networks, implemented in software and hardware architectures to automatically create a non-linear mathematical model of multiple character classes. Systems such as these may be used with human operators in the loop. As characters are misclassified by errors in the recognition system, an operator designates the correct classification. In real time the system now learns or trains on that character and will classify it properly the next time. Though flawed for a variety of reasons it is now one of the most researched methods for classifying large

volumes of image data and feature data from the images. Test data used in this thesis is realistic, and in part the reason each feature method and recognition rate was scoring in the 75-85% accuracy range. If well printed patterns were consistently presented to the system, a 90-100% correct classification is achievable, as in the case with symbols.

In the age of automation and in performing this work it often comes to mind that we should learn a new symbol set, similar to the OCR font printed on checks for digits and driver licenses. With a standardized and controlled printed font errors would be much less likely in the sense of localization and generalization. Strong feelings have been generated about people who have been researching OCR for years in a community which is spread throughout the world. Typically this research is not noticed except for occasional media events by a new startup company promising 100% recognition of any document. Performing this thesis has expanded understanding of neural network technology, image processing and integration of these technologies to provide a base model for pattern classification with images. When encountering information in the future about any of the technologies described in this paper, a realistic assessment and future research will be possible because of the effort expended in this thesis.

APPENDIXES

APPENDIX A

Artificial Neural Network Supporting Source Code

```
#####   #####            ####    ####
#    #  #    #          #    #  #    #
#####   #    #          #       #
#    #  #####    ###    #       #
#    #  #        ###    #    #  #    #
#####   #        ###     ####    ####

   1  // FILE: bp.cc
   2  // Implementation of backprop net
   3  // Copyright (c) 1990,91,  Adam Blum
   4  //
   5  // UNIX port by Al Piszcz, SUN C++
   6  // Revision in floating point usage of variables
   7  // Added parameter display
   8
   9  #include "bp.h"
  10
  11  extern int trace;
  12
  13
  14  bp::bp(char *s):net(s)                  // constructor
  15  {
  16      const NOPARMS = 5;
  17      static PARM parms[NOPARMS]={
  18          {"HIDDEN",    integer},
  19          {"MOMENTUM",  real},
  20          {"INITRANGE", real},
  21          {"EPOCH",     integer},
  22          {"TOLERANCE", real}
  23      };
  24      readparms(NOPARMS,parms,name);
  25      q  = parms[0].val.i;
  26      momentum = parms[1].val.f;
  27      initrange = parms[2].val.f;
  28      epoch     = parms[3].val.i;
  29      tolerance = parms[4].val.f;
  30
  31      cout << "       HIDDEN: " << q << "\n";
  32      cout << "     MOMENTUM: " << momentum << "\n";
  33      cout << "   INIT RANGE: " << initrange << "\n";
  34      cout << "        EPOCH: " << epoch << "\n";
  35      cout << "    TOLERANCE: " << tolerance << "\n";
  36
  37                                          // initialize both weight
  38                                          // matrices to random values
  39                                          // from -1 to +1
  40      W1=new matrix(n,q,-initrange);
  41      W2=new matrix(q,p,-initrange);
  42
  43      dW1=new matrix(n,q);
  44      dW2=new matrix(q,p);
  45
  46      h=new vec(q);
  47      o=new vec(p);
  48      d=new vec(p);
  49      e=new vec(q);
```

```
 50
 51     thresh1=new vec(q);
 52     thresh1->randomize(initrange);
 53     thresh2=new vec(p);
 54     thresh2->randomize(initrange);
 55
 56     if(epoch){
 57         totd=new vec(p);
 58         tote=new vec(q);
 59     }
 60
 61     minvecs=new vecpair(n,q);
 62     maxvecs=new vecpair(n,q);
 63
 64     cycleno=0;
 65
 66 }
 67
 68 bp::~bp()                               // Destructor
 69 {
 70     delete W1;
 71     delete W2;
 72
 73     delete dW1;
 74     delete dW2;
 75
 76     delete h;
 77     delete o;
 78     delete d;
 79     delete e;
 80
 81     if(epoch){
 82         delete totd;
 83         delete tote;
 84     }
 85
 86     delete minvecs;
 87     delete maxvecs;
 88 }
 89
 90 /////////////////////////////////////////////////////////////////
 91 //
 92 // BACKPROP ALGORITHM METHODS - ENCODE AND RECALL
 93 //
 94 /////////////////////////////////////////////////////////////////
 95
 96
 97 /////////////////////////////////////////////////////  ENCODE  //
 98
 99 int bp::encode(vecpair& v)
100 {
101     float   maxdiff;
102
103     //
104     // h=F(W1 i) get vector that is dot  product of input v.a
105     // and weight matrix
106     //
```

```
107       *h = (*W1) * (*(v.a));
108
109       //
110       // apply sigmoid activation function to result
111       //
112       h->sigmoid(*thresh1);
113
114       *o = (*W2) * (*h);              // Step 2) o=F(W2 h)
115
116       if(trace)
117       {
118           cout << "\nUnsquashed guess: " << *o
119           << "\nOutput layer threshold " << *thresh2;
120       }
121
122       o->sigmoid(*thresh2);
123
124       if (epoch)
125       {
126           //
127           // adjust weights at the end  of the cycle
128           // d = o (1-o) (o-t);  use existing overloaded
129           // operators from vector class
130           //
131           *d = (*(v.b) - *o);
132
133            if(trace){
134            cout.precision(2);
135            cout << "\nOutput: " << *(v.b) << " Guess: " << *o ;
136           }
137
138           maxdiff=d->maxval();
139
140           *d  = *d *  o->d_logistic();
141
142           //
143           //  e = b (1-b) W2 d matrix x vector = vector
144           // returns dot product of vec & complement
145           //
146           *e = ((*W2) * *d) *  h->d_logistic();
147
148           *totd += *d;                // weights will be adjusted at
149
150           *tote += *e;                // end of cycle with following
151                                       // totals
152
153           totd->maxerror();
154       }
155       else                            // pattern-by-pattern training
156       {
157
158           //
159           // calculate forward pass error between hidden layer and
160           // output layer; d = o (1-o) (o-t) use existing
161           // overloaded operators from the vector class
162           //
163           *d = (*(v.b) - *o);
```

```
164
165         if(trace)
166         {
167          cout.precision(2);
168          cout << "\nOutput: " << *(v.b) << " Guess: " << *o ;
169         }
170
171         maxdiff=d->maxval();
172         d->maxerror();
173
174         *d  = *d *  o->d_logistic();
175
176         //
177         // Backward pass error calculation between input layer
178         // and  hidden layer; e = b (1-b) W2 d
179         //
180         *e = ((*W2) * *d) * h->d_logistic();
181
182         // matrix x vector = vector returns dot product of
183         // vec & complement
184         //
185         // W2 =  W2 + a h d + dW2(i-1) " a h d " part
186         // dW2 is temporary work matix add the weight change
187         // to the existing weight matrix
188         //
189         initvals(*dW2,(*h),*d,learnrate,momentum);
190         (*W2) += *dW2;
191
192         // calculate new threshold vector
193         *thresh2 += ( (*d) * learnrate );
194
195         // W1 = W1 + a i e + W2(i-1)
196         initvals(*dW1,*(v.a),*e,learnrate,momentum);
197
198         // add the weight change to the existing weight matrix
199          (*W1) +=  *dW1;
200
201         // calculate new threshold vector
202         *thresh1 += ( (*e) * learnrate);
203     }
204
205     if(trace)
206         cout << "\nMaximum difference: " << maxdiff;
207     if(maxdiff < tolerance )
208     {
209         return 1;
210     } else
211     {
212         return 0;
213     }
214 }
215
216
217 ///////////////////////////////////////////////////  INITVALS  //
218 //
219 // Used to initialize a matrix to the vector product
220 // of v1 and v2 times the learn rate
```

```
221  // also adding in the previous contents of the matrix
222  // multiplied by a momentum term.
223  //
224
225  void bp::initvals(matrix& m,const vec& v1,const vec& v2,
226                 const float rate,const float momentum)
227  {
228      for(int i=0;i<m.depth();i++)
229          for(int j=0;j<m.width();j++)
230              m.setval(i,j,(m.getval(i,j)*momentum)+
231                              (v1.v[i]*v2.v[j])*rate);
232  }
233
234
235  /////////////////////////////////////////////////////  RECALL  //
236
237  vec bp::recall(vec& v)
238  {
239      //
240      // Step 1: h = F(W1 i) get vector that is dot
241      // product of input and weight matrix apply sigmoid activation
242      // function to result
243      //
244      *h=(*W1)*v;
245      h->sigmoid(*thresh1);
246
247      // o = F (W2 h)
248      vec out(this->p);
249      out=(*W2)*(*h);
250      out.sigmoid(*thresh2);
251
252      return out;
253  }
254
255
256  //////////////////////////////////////////////////////  CYCLE  //
257  //
258  // This will get called from the neural network train since train
259  // will call the most derived cycle method.
260  // We need to override the network cycle since backpropagation
261  // may require the weights to be update at the end of a cycle.
262  //
263
264  float bp::cycle(istream& s)
265  {
266      vecpair v(n,p);
267      float good,total;
268
269      good = 0.0;
270      total = 0.0;
271      s >> *minvecs;
272
273      s >> *maxvecs;
274
275      //
276      // initialize error accumulation vectors
277      //
```

```
278     if(epoch){
279         for(int i=0;i<totd->length();i++)
280         totd->set(i);
281         for(i=0;i<tote->length();i++)
282     tote->set(i);
283     }
284
285     skipcmt(s);
286     for(;;)
287     {
288         s >> v;
289         if(s.eof()||s.fail())break;
290
291         v.scale(*minvecs,*maxvecs);
292
293         if(encode(v))
294         {
295             good+= 1.0;
296         }
297         total+= 1.0;
298     }
299
300     if(epoch)
301     {
302         //
303         // adjust weights at end of cycle
304         // W2 = dW2 + a h d (total) " a h d " part
305         //
306         initvals(*dW2,(*h),*totd,learnrate,momentum);
307         (*W2) += *dW2;
308
309         *thresh2 += ( (*totd) * learnrate );
310
311         //
312         // W1 = dW1 + a i e (total)
313         //
314         initvals(*dW1,*(v.a),*tote,learnrate,momentum);
315         (*W1) +=  *dW1;
316
317         *thresh1 += ( (*tote) * learnrate );
318     }
319     return good/total;
320 }
321
322
323 /////////////////////////////////////////////////////////////////
324 //
325 // BACKPROP LEVEL INPUT/OUTPUT METHODS:
326 // Saving and loading weights, skipping comments.
327 //
328 ////////////////////////////////////////////
329
330
331
332 //////////////////////////////////////////////// SAVEWEIGHTS  //
333
334 int bp::saveweights()
```

```
335  {
336      FILE *f;
337      char fn[32];
338
339      sprintf(fn,"%s.WTS",name);
340      f=fopen(fn,"wb");
341
342                                      // couldn't open the file
343      if(f == NULL)
344          return 0;                   // save failed!
345
346      fwrite((char *)(&cycleno),sizeof(int),1,f);
347
348      if
349      (
350          !(W1->save(f)) ||
351          !(W2->save(f)) ||
352          !(thresh1->save(f)) ||
353          !(thresh2->save(f))
354      )
355      {
356          fclose(f);
357          return 0;
358      }
359      else
360      fclose(f);
361                                      // put matrices into ".MAT"
362                                      // in readable form
363      if(trace)
364      {
365          sprintf(fn,"%s.MAT",name);
366          ofstream matf(fn,ios::out);
367          matf << "First matrix contains: \n"
368          << *W1
369          << "Second matrix contains: \n"
370          << *W2;
371      }
372      return 1;
373  }
374
375
376  ////////////////////////////////////////////////  LOADWEIGHTS  //
377
378  int bp::loadweights()
379  {
380      FILE *f;
381      char fn[32];
382      sprintf(fn,"%s.WTS",name);
383      f=fopen(fn,"rb");
384
385                                      // couldn't open the file
386      if(f == NULL)
387          return 0;                   // save failed!
388
389      fread((char *)(&cycleno),sizeof(int),1,f);
390
391      if
```

```
392      (
393          !(W1->load(f)) ||
394          !(W2->load(f)) ||
395          !(thresh1->load(f)) ||
396          !(thresh2->load(f))
397       )
398      {
399          fclose(f);
400          return 0;
401      }
402      else
403          fclose(f);
404      return 1;
405  }
```

```
#####   #####           #    #
#    #  #    #          #    #
#####   #    #          ######
#    #  #####    ###    #    #
#    #  #        ###    #    #
#####   #        ###    #    #
```

```
  1  // FILE: bp.h
  2  // Header file for backprop implementation
  3  // Copyright (c) 1990, Adam Blum
  4  //
  5  // UNIX port by Al Piszcz 1993
  6
  7  #include  "net.h"
  8
  9
 10  class bp: public net {                  // backpropagation
 11                                          // network derived from
 12  private:
 13      int q;                              // size of hidden layer
 14      matrix *W1,*W2;                     // synapse weight matrices
 15      matrix *dW1,*dW2;                   // used to compute
 16                                          // changes to matrices
 17      vec *h,*o,*d,*e,*thresh1,*thresh2;  // used to store
 18                                          // h - hidden-layer
 19                                          //     neuron activations
 20                                          // o - output-layer result
 21                                          // d - the target result
 22                                          // e - the error vector
 23
 24      int epoch;                          // indicates whether to perform
 25                                          // epoch based update of weights
 26
 27      vec *totd,*tote;                    // accumulate the totals
 28                                          // for epoch-based updating
 29
 30      vecpair *minvecs,*maxvecs;          // used to normalize inputs
 31                                          // between min and max range
 32
 33      float momentum,initrange;           // weight change in previous
 34                                          // time period affects weight
 35                                          // change in current time period
 36
 37                                          // private member functions
 38                                          // these are helper
 39                                          // member functions
 40      void initvals(matrix& m,const vec& v1,const vec& v2,
 41                  const float rate=1.0,const float momentum=0.0);
 42      int bp::saveweights();
 43      int bp::loadweights();
 44      float bp::cycle(istream& s);
 45
 46  public:
 47                                          // public member functions
 48      bp(char *s);                        // constructs based on
 49                                          // <name>.DEF file
 50      ~bp();                              // destructor
```

```
51                                          // override pure virtual
52                                          // functions
53      int encode(vecpair& v);             // store one pattern pair
54      vec recall(vec& v);                 // recall an output pattern
55                                          // given an input
56  };
```

```
#    #  ######   #####            ####    ####
##   #  #          #             #    #  #    #
# #  #  #####      #             #       #
#  # #  #          #      ###    #       #
#   ##  #          #      ###    #    #  #    #
#    #  ######     #      ###     ####    ####
```

```
  1  /////////////////////////////////////////////////////////////////
  2  // FILE: net.cc
  3  // Source code for abstract neural network base class
  4  //
  5  // Ported to UNIX by Al Piszcz 1993
  6  // precision has been extended to support smaller learning rates
  7  #include "net.h"
  8  #include <iostream.h>
  9  #include <time.h>
 10
 11  #define     _MAX_PATH   120
 12
 13
 14
 15  ///////////////////////////////////
 16  // Parameter table functions
 17
 18  ///////////////////////////////////////////////////// READPARMS  //
 19
 20  int readparms(int n,PARM *p,char *name)
 21  {
 22      char fn[16];
 23      sprintf(fn,"%s.DEF",name);
 24      ifstream def(fn,ios::in);
 25      if(!def)
 26      {
 27          cerr << "Failed to find definition file.\n";
 28          return 0;
 29      }
 30      while  (readparm(def,n,p) && !def.eof());
 31      return 0;
 32  }
 33
 34
 35  ////////////////////////////////////////////////////// READPARM  //
 36  // This streams extraction operator takes input from network
 37  // definition file for one definition parameter.
 38  // It reads in the name of the parameter
 39  // and then looks up which entry in the parameter table to
 40  // instantiate with a value.
 41  //
 42
 43  istream& readparm(istream& s,int noparms,PARM *p)
 44  {
 45      char keyword[NAMELEN],val[16];
 46      s >> keyword;
 47      if(!s || s.eof() || s.fail())       // end of file or failure
 48                                          // to read keyword
 49      return s;
 50
```

```
 51     for(int i=0;i<noparms;i++)
 52     if(!strcmp(keyword,p[i].name))
 53         break;
 54
 55     if(i < noparms)                     // recognized parameter
 56     switch(p[i].type)
 57     {
 58         case string:   s >> p[i].val.s; break;
 59         case integer:  s >> p[i].val.i; break;
 60         case real:
 61         cin.precision(6);
 62         s >> p[i].val.f; break;
 63     }
 64     else
 65         s >> val;
 66
 67      return s;
 68 }
 69
 70
 71 /////////////////////////////////////////////////////////////////
 72 //                    NET CLASS
 73 // Abstract neural net class methods
 74 //
 75
 76 ////////////////////////////////////////////////////////  NET  //
 77
 78 net::net(char *s)
 79 {
 80     name=new char[strlen(s)+1];
 81     strcpy(name,s);
 82     const NOPARMS=6;
 83     static PARM parms[6]=
 84     {
 85         {"INPUTS",  integer},
 86         {"OUTPUTS", integer},
 87         {"RATE",    real},
 88         {"DECAY",   real},
 89         {"ITERS",   integer},
 90         {"DISPLAY",   integer}
 91     };
 92     readparms(NOPARMS,parms,name);
 93     n = parms[0].val.i;
 94     p = parms[1].val.i;
 95     learnrate = parms[2].val.f;
 96     decayrate = parms[3].val.f;
 97     iters = parms[4].val.i;
 98     display = parms[5].val.i;
 99     return;
100 }
101
102
103 ///////////////////////////////////////////////////////  ~NET  //
104
105 net::~net()
106 {
107     delete name;
```

```
108  }
109
110  /////////////////////////////////////////////////////  TRAIN  //
111
112  void net::train()
113  {
114      char            oline[256];         // output line
115      char            tstamp[13];         // holds yymmddhhmmss
116      double          timeinterval;
117      float           ret;
118      float           ge;
119      ifstream        *s;
120      static unsigned long    timestart, timefinish;
121      struct tm       *date;
122      time_t          now;
123      vec             *error;
124
125      error=new vec(p);                   // error vector total value
126
127
128      if(loadweights())
129        cout << "Training from stored weights.\n";
130
131      char fn[_MAX_PATH];
132      sprintf(fn,"%s.FCT",name);
133
134      cout << "Training from:" << fn << "\n";
135      cout << "       INPUTS: " << n << "\n";
136      cout << "      OUTPUTS: " << p << "\n";
137      cout << "   LEARN RATE: " << learnrate << "\n";
138      cout << "   DECAY RATE: " << decayrate << "\n";
139      cout << "        ITERS: " << iters << "\n";
140      cout << "      DISPLAY: " << display << "\n";
141
142      timestart = time (( long * ) 0 );   // load start value of seconds
143
144      cout << "\n";
145      cout << "------- ---------  ------------  --------  ---------\n";
146      cout << " CYCLE   ELAPSED    TIME STAMP   ACCURACY    GLOBAL \n";
147      cout << "  (#)    (HOURS)   yymmddhhmmss              ERROR  \n";
148      cout << "------- ---------  ------------  --------  ---------\n";
149      for(;;)
150      {
151          s=new ifstream(fn,ios::in);
152
153          if(!*s){
154             cout << "Failed to open fact file.\n";
155             return;
156          }
157          error->maxerrorreset();
158            ret=cycle(*s);
159          ge=error->getglobalerror();
160            delete s;
161
162          if (( cycleno % display ) == 0 )
163          {
164                                       // calculate the elapsed time
```

```
165                                     // since the last check point
166             timefinish= time(( long * ) 0 );
167             timeinterval = ( (timefinish - timestart) / 3600.0 ) ;
168             sprintf(oline,"%7d  %7.3f  ",cycleno,timeinterval);
169
170                                     // create a sortable timestamp
171             time (&now);
172             date = localtime(&now);
173             strftime ( tstamp, 13, "%y%m%d%H%M%S", date);
174
175             sprintf(oline,"%7d  %7.3f   %s    %6.2f  %8.4f\n",
176                     cycleno,timeinterval,tstamp,ret,ge);
177
178             cout << oline;          // send the line out
179             cout.flush();           // for sure
180             saveweights();
181         }
182         ++cycleno;
183
184         if( ret>=1.0 )
185         {
186             cout    << "Training suspended at "
187                     << cycleno << " cycles.\n";
188             break;
189         }
190     }
191     saveweights();
192     return;
193 }
194
195
196
197 //////////////////////////////////////////////////////  CYCLE  //
198
199 float net::cycle(istream& s)
200 {
201     vecpair v(n,p);
202     float good,total;
203     good = 0.0;
204     total = 0.0;
205
206     skipcmt(s);
207     for(;;)
208     {
209         s >> v;
210         if(s.eof()||s.fail())break;
211
212         if(encode(v))
213             good+= 1.0;
214
215         total+= 1.0;
216     }
217     return good/total;
218 }
219
220
221 ////////////////////////////////////////////////////  SKIPCMT  //
```

```
222
223 int net::skipcmt(istream& inf)
224 {
225     int c;
226     inf.unsetf(inf.skipws);
227     if(inf.peek()==':')
228     {
229         do
230         {
231             c=inf.get();
232             if(c<0)
233                 return 0;
234         } while( (c!=0xd) && (c!=0xa) );
235         inf.setf(inf.skipws);
236         return 1;
237     }
238     else
239     {
240         inf.setf(inf.skipws);
241         return 0;
242     }
243 }
244
245
246 //////////////////////////////////////////////////////  TEST  //
247
248 float net::test()
249 {
250     float accuracy = 0.0, good=0.0,total=0.0;
251     char tstfn[32];
252     vecpair v;
253     vec out;
254
255     if(!loadweights())
256     {
257         cout << "No stored network to test.";
258         return 0;
259     }
260
261     sprintf(tstfn,"%s.TST",name);
262     ifstream tstf(tstfn,ios::in);
263
264     skipcmt(tstf);                  // skip comment
265      for(;;)
266     {
267         if(!(tstf>>v))break;
268         out=recall(*(v.a));
269
270         if( (*(v.b)-out).maxval() < tolerance )
271             good+= 1.0;
272
273         total+= 1.0;
274     }
275     cout.precision(5);
276     accuracy = ( good/total ) * 100.0;
277     cout <<  good/total*100.0 << " percent correct.\n";
278     return good/total;
```

```
279  }
280
281  ////////////////////////////////////////////////////////  RUN  //
282
283  void net::run()
284  {
285      char ifn[16],ofn[16];
286      vec in(n),out(p);
287
288      if(!loadweights())
289      {
290          cout << "No stored network to run.\n";
291          return;
292      }
293
294      sprintf(ifn,"%s.IN",name);
295      sprintf(ofn,"%s.OUT",name);
296      cout << "Running from " << ifn << "\n";
297      cout << "Output to " << ofn << "\n";
298      ifstream inf(ifn,ios::in);
299      ofstream outf(ofn,ios::out);
300
301      skipcmt(inf);
302      for(;;)
303      {
304          if(!(inf>>in))break;;
305          if(!inf || inf.eof()|| inf.fail())break;
306          cout.precision(3);
307          outf << recall(in);
308      }
309      return;
310  }
```

```
#    #  ######   #####          #    #
##   #  #          #            #    #
# #  #  #####      #            ######
#  # #  #          #     ###    #    #
#   ##  #          #     ###    #    #
#    #  ######     #     ###    #    #

   1  /////////////////////////////////////////////////////////////////
   2  // FILE: net.h
   3  // UNIX port Al Piszcz 1993
   4  // Header file for abstract neural network base class
   5  // To be used as parent to specific neural network implementations.
   6  // The encode and recall methods are defined as pure virtual functions
   7  // making this an abstract class than can never be instantiated.
   8  // Details of encode and recall must depend on the topology
   9  // itself. However the methods "train", "test", and "run"
  10  // can be defined since they are substantively the same for each
  11  // of the classes. The constructor can be defined and will be used
  12  // by child classes in their own constructors to instantiate
  13  // common elements of derived classes.
  14
  15  #include "vecmat.h"
  16
  17  // parameter class used to point to variable to be initialized
  18  // and specify string to be used in definition file to initialize it
  19
  20  enum vartype {real,integer,string};
  21  const NAMELEN=16;
  22
  23
  24  typedef struct {
  25      char name[16];                      // string to init value
  26      vartype type;
  27      union {
  28          char s[8];
  29          float f;
  30          int i;
  31      } val;
  32  } PARM;
  33
  34  istream& readparm(istream& s,int noparms,PARM *p);
  35  int readparms(int n,PARM *p,char *name);
  36
  37  /////////////////////////////////////////////////////////////
  38  //              NET CLASS
  39  //
  40  class net {
  41  protected:
  42      char *name;                         // string used as basename
  43                                          // for files
  44
  45      int n;                              // size of input layer
  46      int p;                              // size of output layer
  47      float learnrate;                    // learning rate (defined as 1
  48                                          // where not gradual)
  49      float decayrate;                    // decay
  50      int display;                        // Cycle interval for display
```

```
51                                          // and weight update
52                                          // (default constructed zero
53                                          //  if not applicable)
54     float tolerance;
55     int iters;
56     int cycleno;
57
58                                          // weight saving methods since
59                                          // topology is not known yet,
60                                          // they must be pure virtual
61     virtual int saveweights(void) = 0;
62     virtual int loadweights(void) = 0;
63     int skipcmt(istream& s);
64
65 public:
66     enum parmtype {inputs,outputs,learn,decay};
67      net(){};
68     net(char *s);
69     net(char *s,int noparms,PARM *p);
70     ~net();
71
72                                              // encode and recall and
73                                              // "pure virtual" which
74                                              // makes the net class
75                                              // abstract
76     virtual int encode(vecpair& v) = 0;
77     virtual vec recall(vec& v) = 0;          // recall an output
78                                              // pattern given an input
79     virtual float cycle(istream& s);
80     virtual void train();
81     int getiters(void){return iters;}
82     virtual float test();                    // floating point value
83                                              // indicates percentage
84                                              // correct of test
85     virtual void run();
86 };
```

```
#####  ######   ####    #####  #####   #####           ####    ####
  #    #       #          #    #    #  #    #         #    #  #    #
  #    #####    ####      #    #####   #    #         #       #
  #    #            #     #    #    #  #####    ###   #       #
  #    #       #    #     #    #    #  #        ###   #    #  #    #
  #    ######   ####      #    #####   #        ###    ####    ####

 1 /////////////////////////////////////////////////////////////////
 2 // FILE: testbp.cpp
 3 // Interactive BP System Demonstration Program
 4 // Used to verify BP system algorithms
 5 // Developed with Turbo C++ 1.0
 6 // Copyright (c) 1990 Adam Blum
 7 //
 8 // Ported to SUN C++ Al Piszcz 1993
 9
10 #define NDEBUG 1                        // ANSI method to enable or
11                                         // disable debugging
12 #include"bp.h"
13
14 void getargs(int argc,char **argv);
15
16 char netname[16]="BP";                  // network file name
17 char mode=0;                            // default to learn or mode
18 int trace=0;                            // SET TRACE=<whatever>
19                                         // at UNIX prompt to
20                                         // turn trace on
21 char *p;
22
23 ///////////////////////////////////////////////////////  MAIN  //
24
25 main(int argc,char **argv)
26 {
27
28     cout.setf(cout.unitbuf);            // turn "unitbuf" on to force
29                                         // flushing after each char
30     cout.unsetf(cout.scientific);       // turn skipws and
31                                         // scientific off
32     cout.precision(2);
33
34    cout << "TESTBP - Interactive Backpropagation Network Tester\n";
35    trace=(p=getenv("TRACE"))?1:0;
36
37    getargs(argc,argv);
38
39    bp b(netname);
40    switch(mode){
41    case 'L':
42       b.train();
43       break;
44    case 'T':
45       b.test();
46       break;
47    case 'R':
48       b.run();
49       break;
50    default:
```

```
51         break;
52     }
53     return 1;
54 }
55
56
57 ////////////////////////////////////////////////////  GETARGS  //
58
59 void getargs(int argc,char **argv)
60 {
61     for(int i=1;i<argc;i++)
62     if(argv[i][0]=='-')
63         mode=argv[i][1];
64     else
65         strcpy(netname,argv[i]);
66     }
```

```
#    #  ######   ####   #    #    ##     #####           ####    ####
#    #  #       #    #  ##  ##   #  #      #            #    #  #    #
#    #  #####   #       # ## #  #    #     #            #       #
#    #  #       #       #    #  ######     #     ###    #       #
 #  #   #       #    #  #    #  #    #     #     ###    #    #  #    #
  ##    ######   ####   #    #  #    #     #     ###     ####    ####
```

```
  1  ////////////////////////////////////////////////////////////////
  2  // FILE: vecmat.cc
  3  // vector and matrix class methods
  4  // Copyright (c) 1990, Adam Blum
  5  //
  6  // Modified for UNIX by Al Piszcz
  7  // Added maxerrorreset, maxerror, and getglobalerror methods
  8  // 1993
  9
 10  #include"vecmat.h"
 11
 12
 13  ////////////////////////////////////////////////////////////////
 14  // vector class member functions
 15
 16
 17
 18  ////////////////////////////////////////////  VEC constructor  //
 19  vec::vec(int size,int val)
 20  {
 21      v = new float[n=size];
 22      for(int i=0;i<n;i++)
 23          v[i]=val;
 24  }                                        // constructor
 25
 26
 27
 28
 29  /////////////////////////////////////////////  VEC destructor  //
 30  vec::~vec() { delete v;}                 // destructor
 31
 32
 33
 34
 35  ////////////////////////////////////////////  VEC to VEC copy  //
 36  vec::vec(vec& v1)                        // copy-initializer
 37  {
 38      v=new float[n=v1.n];
 39      for(int i=0;i<n;i++)
 40          v[i]=v1.v[i];
 41  }
 42
 43
 44
 45  //////////////////////////////////////  VEC to VEC assignment  //
 46  vec& vec::operator=(const vec& v1)
 47  {
 48      delete v;
 49      v=new float[n=v1.n];
 50      for(int i=0;i<n;i++)
```

```
 51          v[i]=v1.v[i];
 52      return *this;
 53  }
 54
 55
 56
 57  ////////////////////////////////////////  VEC to VEC addition  //
 58  vec vec::operator+(const vec& v1)
 59  {
 60      vec sum(v1.n);
 61      for(int i=0;i<v1.n;i++)
 62          sum.v[i]=v1.v[i]+v[i];
 63      return sum;
 64  }
 65
 66
 67
 68  ////////////////////////////////////  VEC and  FLOAT addition  //
 69  vec vec::operator+(const float d)
 70  {
 71      vec sum(n);
 72      for(int i=0;i<n;i++)
 73          sum.v[i]=v[i]+d;
 74      return sum;
 75  }
 76
 77
 78
 79  //////////////////////////////////  VEC to VEC add and assign  //
 80  vec& vec::operator+=(const vec& v1)
 81  {
 82     for(int i=0;i<v1.n;i++)
 83                  v[i]+=v1.v[i];
 84          return *this;
 85  }
 86
 87
 88
 89  ////////////////////////////////////  VEC to VEC dot product  //
 90  float vec::operator*(const vec& v1)     // dot-product
 91  {
 92      float sum=0;
 93      for (int i=0; i<min(n,v1.n); i++)
 94          sum+=(v1.v[i]*v[i]);
 95      return sum;
 96  }
 97
 98
 99
100  ///////////////////////////////////  VEC to VEC equality test  //
101  int vec::operator==(const vec& v1)
102  {
103      if (v1.n!=n) return 0;
104      for (int i=0;i<min(n,v1.n);i++){
105          if(v1.v[i]!=v[i]){
106              return 0;
107          }
```

```
108      }
109      return 1;
110  }
111
112
113
114  ///////////////////////////  VEC [] operator, extract element  //
115  float vec::operator[](int x)
116  {
117      if( ( x < length() ) && ( x >= 0) )
118          return v[x];
119      else
120          cerr << "vec index out of range";
121      return 0;
122  }
123
124
125
126  ///////////////////////////////////////////  LENGTH of vector  //
127  int vec::length(){return n;}            // length method
128
129
130
131  /////////////////////////  GARBLE randomly load vector values  //
132  vec& vec::garble(float noise)           // corrupt vector w/random
133  {                                        // noise
134      time_t t;
135      time(&t);
136      srand((unsigned)t);
137      for(int i=0;i<n;i++){
138          if((rand()%10)/10<noise)
139          v[i]=1-v[i];
140      }
141      return *this;
142  }
143
144
145
146  ///////////////////////////////////////////  NORMALIZE vector  //
147  vec& vec::normalize()                   // normalize by length
148  {
149      for(int i=0;i<n;i++)
150          v[i]/=n;
151      return *this;
152  }
153
154
155
156  ////////////////  NORMALIZEON normalize by non zero elements  //
157  vec& vec::normalizeon()
158  {
159      int on=0;
160      for(int i=0;i<n;i++)
161          if(v[i])
162              on++;
163      for(i=0;i<n;i++)
164          v[i]/=on;
```

```
165     return *this;
166 }
167
168
169
170 //////////////////////  RANDOMIZE vector with specific range  //
171 vec& vec::randomize(float range)
172 {
173     time_t t;
174     int pct,val,rnd;
175     if(range){
176         time(&t);
177         srand((unsigned)t);
178     }
179     for(int i=0;i<n;i++){
180         rnd=rand();
181         pct=(int) (range * 100.0);
182         val= rnd % pct;
183         v[i]= (float) val / 100.0 ;
184         if(range<0)
185         v[i] = fabs(range) - (v[i] * 2.0);
186     }
187     return *this;
188 }
189
190
191
192 //////////////////////////  GETGLOBALERROR  return current value  //
193 float vec::getglobalerror()                     // total error
194 {
195     return global_error;
196 }
197
198
199
200 ////////////////////////////////  MAXERRORRESET zero out value //
201 float vec::maxerrorreset()                  // total error 0.0
202 {
203     global_error = 0.0;
204     return global_error;
205 }
206
207
208
209 /////////////////////////////////////  MAXERROR sum of vector  //
210 float vec::maxerror()                   // total error SUM
211 {
212     float errortotal=0;
213     for(int i=0;i<n;i++)
214     {
215         errortotal+=fabs(v[i]);
216     }
217     global_error = global_error + errortotal;
218 //  cout << "GE:  " << global_error << "\n";
219 //  cout << "ET:  " << errortotal << "\n";
220     return errortotal;
221 }
```

```
222
223
224 /////////////////  MAXVAL absolute maximum value from vector  //
225 float vec::maxval()                     // maximum ABSOLUTE value
226 {
227     float mx=0;
228     for(int i=0;i<n;i++)
229     if(fabs(v[i])>mx){
230         mx=fabs(v[i]);
231     }
232     return mx;
233 }
234
235
236
237 /////////////////////  SCALE vector from  min and max vector  //
238 vec& vec::scale(vec& minvec,vec& maxvec)
239 {
240     for(int i=0;i<n;i++){
241     if(v[i]<minvec.v[i])
242         v[i]=0;
243         else if(v[i]>maxvec.v[i])
244             v[i]=1;
245         else if((maxvec.v[i]-minvec.v[i])==0)
246             v[i]=1;
247         else
248             v[i]=(v[i]-minvec.v[i])/(maxvec.v[i]-minvec.v[i]);
249     }
250     return *this;
251 }
252
253
254
255 ///////////////////  D_LOGISTIC returns derivative of vector  //
256 float vec::d_logistic()                 // returns vec * (1-vec)
257 {
258     float sum=0.0;
259     for(int i=0;i<n;i++)
260         sum+=(v[i]*(1-v[i]));
261     return sum;
262 }
263
264
265
266 ///////////////////////////////  DISTANCE Euclidean distance  //
267 // Euclidean distance
268 // function ||A-B||
269 float vec::distance(vec& A)
270 {
271     float sum=0,d;
272     for(int i=0;i<n;i++){
273         d=v[i]-A.v[i];
274         if(d)sum+=pow(d,2);
275     }
276     return sum?pow(sum,0.5):0;
277 }
278
```

```
279
280
281 ////////////////////////////////////////////////////  MAXINDEX  //
282 // index of the highest
283 // item in vector
284 int vec::maxindex()
285 {
286     int idx,i;
287     float mx;
288     for(i=0,mx=-INT_MAX;i<n;i++)
289     if(v[i]>mx){
290         mx=v[i];
291         idx=i;
292     }
293     return idx;
294 }
295
296
297
298 //////////////////////////////////////////////  LOGISTIC function  //
299 double logistic(double activation)
300 {
301 // These underflow limits were copied from McClelland's
302 // bp implementation.
303 // We had problems with underflow with numbers that should have been
304 // small enough in magnitude. McClelland seems to have encountered this
305 // and established the numbers below as reasonable limits. - AB
306
307     if(activation>11.5129)
308             return 0.99999;
309     if(activation<-11.5129)
310             return 0.00001;
311     return 1.0/(1.0+exp(-activation));
312 }
313
314
315
316 //////////////////////////////////////////////////////  GETSTR  //
317 vec& vec::getstr(char *s)
318 {
319     for(int i=0;i<MAXVEC&&s[i];i++){
320     if(isalpha(s[i]))
321         v[toupper(s[i])-'A']=1;
322     }
323     return *this;
324 }
325
326
327
328 //////////////////////////////////////////////////////  PUTSTR  //
329 void vec::putstr(char *s)
330 {
331      int ct=0;
332     for(int i=0;i<26;i++)
333         if(v[i]>0.9)
334             s[ct++]='A'+i;
335 }
```

```
336
337
338
339 /////////////////////////////////  DIFFERENCE between vectors  //
340 //
341 //
342 vec vec::operator-(const vec& v1)
343 {
344     vec diff(n);
345     for(int i=0;i<n;i++)
346         diff.v[i]=v[i]-v1.v[i];
347     return diff;
348 }
349
350
351
352 ////////////////////  DIFFERENCE between vector and constant  //
353 vec vec::operator-(const float d)       // subtraction of constant
354 {
355     vec diff(n);
356     for(int i=0;i<n;i++)
357         diff.v[i]=v[i]-d;
358     return diff;
359 }
360
361
362
363 ////////////////////  MULTIPLY every element vector by float  //
364 vec vec::operator*(float c)
365 {
366     vec prod(length());
367     for(int i=0;i<prod.n;i++)
368         prod.v[i]=v[i]*c;
369     return prod;
370 }
371
372
373
374 //////////////////  operator *= MULTIPLY multiply and assign  //
375 vec& vec::operator*=(float c)
376 {
377     for(int i=0;i<n;i++)
378         v[i]*=c;
379     return *this;
380 }                                    // vector multiply by constant
381
382
383
384 ////////////////////////////////////////////////////  SIGMOID  //
385 // this is the sigmoid activation function we have chosen for
386 // our backprop implementation. It happens to use the logistic
387 // function: 1/(1+e^-x)
388 const SCALE=5;
389
390 vec& vec::sigmoid(vec& thresh)
391 {
392     for(int i=0;i<n;i++)
```

```
393         v[i] = (float) logistic( (double) (SCALE * (v[i]+thresh[i])) );
394     return *this;
395 }
396
397
398
399 //////////////////////////////  SET specific element to float  //
400 vec& vec::set(int i,float f)
401 {
402     v[i]=f;
403     return *this;
404 }
405
406
407
408 /////////////////////////////////////////// >> stream operator  //
409 // format: list of floating point numbers followed by ','
410 istream& operator>>(istream& s,vec& v1)
411 {
412     float d;int i=0,c;
413     for(;;){
414         s>>d;
415         if(s.eof())
416             return s;
417         if(s.fail()){
418             s.clear();
419             do
420                 c=s.get();
421             while(c!=',' && c);
422             return s;
423         }
424         v1.v[i++]=d;
425
426         if(i==v1.n){
427             do
428                 c=s.get();
429             while(c!=',');
430             return s;
431         }
432     }
433 }
434
435
436
437 //////////////////////////////////////////  << stream operator  //
438 // format: list of floating point numbers followed by ','
439 ostream& operator<<(ostream& s,vec& v1)
440 {
441     s.precision(2);
442     for(int i=0;i<v1.n;i++)
443         s << v1[i] <<" ";
444     s << ",";
445     return s;
446 }
447
448
449
```

```
450 ////////////////////////////////////////////////  SAVE matrix  //
451 // save binary values of matrix to specified file
452 int vec::save(FILE *f)
453 {
454    int success=1;
455    for(int i=0;i<n;i++)
456        if(fwrite((char *)(&(v[i])),sizeof(v[i]),1,f) < 1)
457             success=0;
458    return success;
459 }
460
461
462
463 ////////////////////////////////////////////////  LOAD matrix  //
464 // load binary values from file to  matrix
465 int vec::load(FILE *f)
466 {
467    int success=1;
468    for(int i=0;i<n;i++)
469         if(fread((char *)(&(v[i])),sizeof(v[0]),1,f) < 1)
470             success=0;
471    return success;
472 }
473
474
475
476 /////////////////////////////////////////////////////////////////
477 // matrix  member functions
478
479 /////////////////////////////  LOAD matrix with random floats  //
480 matrix::matrix(int n,int p,float range)
481 {
482     int i,j,rnd;time_t t;
483     int pct;
484     m=new float *[n];
485     if(range){
486         time(&t);
487         srand((unsigned)t);
488     }
489     for(i=0;i<n;i++){
490         m[i]=new float[p];
491         for(j=0;j<p;j++){
492             if(range){
493                 rnd=rand();
494                 pct=(int) (range * 100.0);
495                 m[i][j]= (float)(rnd % pct) / 100.0 ;
496                 if(range<0)
497                 m[i][j] = fabs(range) - (m[i][j] * 2.0);
498             }
499             else
500                 m[i][j]=0;
501         }
502     }
503     r=n;
504     c=p;
505 }
506
```

```
507
508
509 ////////////////////////////////////  LOAD matrix with floats  //
510 matrix::matrix(int n,int p,float value,float range)
511 {
512     int i,j;
513     i=(int)range;
514     m=new float *[n];
515     for(i=0;i<n;i++){
516         m[i]=new float[p];
517         for(j=0;j<p;j++)
518             m[i][j]=value;
519     }
520     r=n;
521     c=p;
522 }
523
524
525
526 ///////////////  LOAD matrix location with floats from stream  //
527 matrix::matrix(int n,int p,char *fn)
528 {
529      int     i;
530     m=new float *[n];
531     for(i=0;i<n;i++){
532         m[i]=new float[p];
533     }
534     r=n;
535     c=p;
536     ifstream in(fn,ios::in);
537     in >> *this;
538 }
539
540
541
542 ///////////////////////////////////////////  MATRIX constructor  //
543 matrix::matrix(const vecpair& vp)
544 {
545     int j;
546     r=vp.a->length();
547     c=vp.b->length();
548     m=new float *[r];
549     for(int i=0;i<r;i++){
550         m[i]=new float[c];
551         for(j=0;j<c;j++)
552             m[i][j]=((vp.a)->v[i])*((vp.b)->v[j]);
553     }
554 }
555
556
557
558 ///////////////////////////////////////////  MATRIX constructor  //
559 matrix::matrix(vec& v1,vec& v2)
560 {
561     int j;
562     r=v1.length();
563     c=v2.length();
```

```
564     m=new float *[r];
565     for(int i=0;i<r;i++){
566         m[i]=new float[c];
567         for(j=0;j<c;j++)
568             m[i][j]=v1.v[i]*v2.v[j];
569     }
570 }
571
572
573
574 ////////////////////////////////////  MATRIX copy initializer  //
575 matrix::matrix(matrix& m1)
576 {
577     r=m1.r;
578     c=m1.c;
579     m=new float *[r];
580     for(int i=0;i<r;i++){
581         m[i]=new float[c];
582         for(int j=0;j<c;j++)
583             m[i][j]=m1.m[i][j];
584     }
585 }
586
587
588 //////////////////////////////////////////////  MATRIX destructor  //
589 matrix::~matrix()
590 {
591         delete []m;
592 }
593
594
595
596 /////////////////////////  MATRIX = assignment to vector pair  //
597 matrix& matrix::operator=(const vecpair& vp)
598 {
599     int j;double d;
600     r=vp.a->length();
601     c=vp.b->length();
602     for(int i=0;i<r;i++){
603         for(j=0;j<c;j++){
604             d=((vp.a)->v[i])*((vp.b)->v[j]);
605             m[i][j]=(float)d;
606         }
607     }
608     return *this;
609 }
610
611
612
613 //////////////////////  MATRIX = assignment to another matrix  //
614 matrix& matrix::operator=(const matrix& m1)
615 {
616     for(int i=0;i<r;i++)
617         delete m[i];
618     r=m1.r;
619     c=m1.c;
620     m=new float*[r];
```

```
621     for(i=0;i<r;i++){
622         m[i]=new float[c];
623         for(int j=0;j<r;j++)
624             m[i][j]=m1.m[i][j];
625     }
626     return *this;
627 }
628
629
630
631 //////////////////////  MATRIX + assignment to another matrix  //
632 matrix matrix::operator+(const matrix& m1)
633 {
634     int i,j;
635     matrix sum(r,c);
636     for(i=0;i<r;i++)
637         for(j=0;j<r;j++)
638             sum.m[i][j]=m1.m[i][j]+m[i][j];
639     return sum;
640 }
641
642
643
644 /////////////  MATRIX * multiply by float and previous result  //
645 matrix& matrix::operator*(const float d)
646 {
647     int i,j;
648     for(i=0;i<r;i++)
649         for(j=0;j<c;j++)
650             m[i][j]*=d;
651     return *this;
652 }
653
654
655
656 //////////////////////////  COLSLICE column slice into vector  //
657 vec matrix::colslice(int col)
658 {
659     vec temp(r);
660     for(int i=0;i<r;i++)
661          temp.v[i]=m[i][col];
662     return temp;
663 }
664
665
666
667 /////////////////////////////  ROWSLICE row slice into vector  //
668 vec matrix::rowslice(int row)
669 {
670     vec temp(c);
671     for(int i=0;i<c;i++)
672         temp.v[i]=m[row][i];
673     return temp;
674 }
675
676
677
```

```
678
679 ////////////////////////  INSERTCOL insert vector into column  //
680 void matrix::insertcol(vec& v,int col)
681 {
682     for(int i=0;i<v.n;i++)
683         m[i][col]=v.v[i];
684 }
685
686
687
688 ///////////////////////////  INSERTROW insert vector into row  //
689 void matrix::insertrow(vec& v,int row)
690 {
691     for(int i=0;i<v.n;i++)
692         m[row][i]=v.v[i];
693 }
694
695
696
697 ///////////////////////////  DEPTH get depth of matrix (Rows)  //
698 int matrix::depth(){return r;}
699
700
701
702 ////////////////////////  WIDTH get width of matrix (Columns)  //
703 int matrix::width(){return c;}
704
705
706 ///////////////////////////  GETVAL get specific matrix value  //
707 float matrix::getval(int row,int col)
708 {
709    return m[row][col];
710 }
711
712
713
714 /////////////////  SETVAL set specific matrix value to float  //
715 void matrix::setval(int row,int col,float val)
716 {
717    m[row][col] = val;
718 }
719
720
721
722 /////////////////////////////  CLOSESTCOL find closest column  //
723 int matrix::closestcol(vec& v)
724 {
725     int mincol;
726     float d;
727      float mindist=INT_MAX;
728     vec w(r);
729     for(int i=0;i<c;i++){
730         w=colslice(i);
731         if( (d=v.distance(w)) < mindist){
732             mindist=d;
733             mincol=i;
734         }
```

```
735     }
736     return mincol;
737 }
738
739
740
741 ////////////////////////////////  CLOSESTROW find closest row  //
742 int matrix::closestrow(vec& v)
743 {
744     int minrow;
745     float d;
746     float mindist=INT_MAX;
747     vec w(c);
748     for(int i=0;i<r;i++){
749         w=rowslice(i);
750         if( (d=v.distance(w)) < mindist){
751             mindist=d;
752             minrow=i;
753         }
754     }
755     return minrow;
756 }
757
758
759
760 //////////////////////////////////////////////////  CLOSESTROW  //
761 int matrix::closestrow(vec& v,int *wins,float scaling)
762 {
763     int minrow;
764     float d;
765     float mindist=INT_MAX;
766     vec w(c);
767     for(int i=0;i<r;i++){
768         w=rowslice(i);
769         d=v.distance(w);
770         d*=(1+((float)wins[i]*scaling));
771         if( d < mindist){
772             mindist=d;
773             minrow=i;
774         }
775     }
776     return minrow;
777 }
778
779
780 ////////////////////////////////////////  SAVE matrix to file  //
781 // save binary values of matrix to specified file
782 int matrix::save(FILE *f)
783 {
784    int success=1;
785    for(int i=0;i<r;i++)
786       for(int j=0;j<c;j++)
787          if(fwrite((char *)(&(m[i][j])),sizeof(m[0][0]),1,f) < 1)
788             success=0;
789    return success;
790 }
791
```

```
792
793
794 /////////////////////////////////////  LOAD matrix from  file  //
795 // load binary values of matrix from specified file
796 int matrix::load(FILE *f)
797 {
798     int success=1;
799     for(int i=0;i<r;i++)
800         for(int j=0;j<c;j++)
801             if(fread((char *)(&(m[i][j])),sizeof(m[0][0]),1,f) < 1)
802                 success=0;
803         return success;
804 }
805
806
807
808 #if MULTIWINNER
809 int _Cdecl intcmp(const void* i1,const void *i2)
810 {
811         if(*(int *)i1 > *(int *)i2)
812             return 1;
813         if(*(int *)i1 < *(int *)i2)
814             return -1;
815         return 0;
816 }
817 #endif
818
819
820 ///////////////////  += MATRIX operator assign and increment  //
821 matrix& matrix::operator+=(const matrix& m1)
822 {
823     int i,j;
824     for(i=0;i<r&&i<m1.r;i++)
825         for(j=0;j<c&&j<m1.c;j++)
826             m[i][j]+=(m1.m[i][j]);
827     return *this;
828 }
829
830
831
832 ////////////////////  *= MATRIX operator multiply and assign  //
833 matrix& matrix::operator*=(const float d)
834 {
835     int i,j;
836     for(i=0;i<r;i++)
837         for(j=0;j<c;j++)
838             m[i][j]*=d;
839     return *this;
840 }
841
842
843
844 ///////////////////  * MATRIX operator dotproduct with vector  //
845 vec matrix::operator*(vec& v1)
846 {
847     vec temp(v1.n==r?c:r),temp2(v1.n==r?r:c);
848     for(int i=0;i<((v1.n==r)?c:r);i++){
```

```
849         if(v1.n==r)
850             temp2=colslice(i);
851         else
852             temp2=rowslice(i);
853         temp.v[i]=v1*temp2;
854     }
855     return temp;
856 }
857
858
859
860 /////////////////////////////////////////////  INITVALS matrix  //
861 void matrix::initvals(const vec& v1,const vec& v2,const float rate, const
float momentum)
862 {
863    int j;
864    for(int i=0;i<r;i++)
865       for(j=0;j<c;j++)
866          m[i][j]=(m[i][j]*momentum)+((v1.v[i]*v2.v[j])*rate);
867 }
868
869
870
871 //////////////////////////  OPERATOR << MATRIX print a matrix  //
872 ostream& operator<<(ostream& s,matrix& m1)
873 {
874     for(int i=0;i<m1.r;i++){
875         for(int j=0;j<m1.c;j++){
876             s << m1.m[i][j] << " ";
877         }
878         s << "\n";
879     }
880     return s;
881 }
882
883
884
885 //////////////////////////  OPERATOR >> MATRIX read a matrix  //
886 istream& operator>>(istream& s,matrix& m1)
887 {
888     for(int i=0;i<m1.r;i++){
889         for(int j=0;j<m1.c;j++){
890             s >> m1.m[i][j];
891         }
892     }
893     return s;
894 }
895
896
897
898 ////////////////////////////////////////////////////////////////
899 // vecpair  member functions
900
901 ////////////////////////////////////////////////////////////////
902 // constructors
903 vecpair::vecpair(int n,int p,int val)
904 {
```

```
905     a=new vec(n,val);b=new vec(p,val);
906 }
907
908
909
910 //////////////////////////////////////  vector pair constructor //
911 vecpair::vecpair(vec& A,vec& B)
912 {
913     a=new vec(A.length());
914     *a=A;
915     b=new vec(B.length());
916     *b=B;
917 }
918
919
920
921 //////////////////////////////////////  vector pair initializer //
922 vecpair::vecpair(const vecpair& AB)             // copy-initializer
923 {
924     a=new vec((AB.a)->length());
925      b=new vec((AB.b)->length());
926     *a=*(AB.a);
927     *b=*(AB.b);
928 }
929
930
931
932 ///////////////////////////////////////  vector pair destructor //
933 vecpair::~vecpair() {
934     delete a; delete b;
935 }                                               // destructor
936
937
938
939 ///////////////////////////////////////////////////  = Operator //
940 vecpair& vecpair::operator=(const vecpair& v1)
941 {
942     *a=*(v1.a);
943     *b=*(v1.b);
944     return *this;
945 }
946
947
948
949 /////////////////////////////////////////////////////  scale  //
950 vecpair& vecpair::scale(vecpair& minvecs,vecpair& maxvecs)
951 {
952     a->scale(*(minvecs.a),*(maxvecs.a));
953     b->scale(*(minvecs.b),*(maxvecs.b));
954     return *this;
955 }
956
957
958
959 ////////////////////////////////////////////  == Equality test  //
960 int vecpair::operator==(const vecpair& v1)
961 {
```

```
962     return  (*a == *(v1.a)) && (*b == *(v1.b));
963 }
964
965
966
967 ////////////////////////////////////  >> input stream operator //
968 istream& operator>>(istream& s,vecpair& v1)     // input a vector pair
969 {
970     s>>*(v1.a)>>*(v1.b);
971     return s;
972 }
973
974
975
976 ///////////////////////////////////  >> output stream operator //
977 ostream& operator<<(ostream& s,vecpair &v1)     // print a vector pair
978 {
979     return s<<*(v1.a)<<*(v1.b)<<"\n";
980 }
```

```
#    #  ######   ####   #    #    ##     #####          #    #
#    #  #       #    #  ##  ##   #  #      #            #    #
#    #  #####   #       # ## #  #    #     #            ######
#    #  #       #       #    #  ######     #     ###    #    #
 #  #   #       #    #  #    #  #    #     #     ###    #    #
  ##    ######   ####   #    #  #    #     #     ###    #    #
```

```
  1  ///////////////////////////////////////////////////////////////
  2  // FILE: vecmat.h
  3  // Vector and matrix classes
  4  // Copyright (c) 1990, Adam S. Blum
  5  //
  6  // UNIX port by Al Piszcz 1993
  7
  8
  9  #include<stdlib.h>
 10  #include<fcntl.h>
 11  #include<stdio.h>
 12  #include<string.h>
 13  #include<limits.h>
 14  #include<ctype.h>
 15  #include<math.h>
 16  #include<time.h>
 17  #include<float.h>
 18  #include<sys/stat.h>
 19  #include<iostream.h>
 20  #include<memory.h>
 21  #include<iostream.h>
 22  #include<fstream.h>
 23  #include<iomanip.h>
 24
 25  #define max(a,b)        (((a) > (b)) ? (a) : (b))
 26                              // C++ doesn't have min/max
 27  #define min(a,b)        (((a) < (b)) ? (a) : (b))
 28
 29  #include "debug.h"
 30
 31  double logistic(double activation);
 32  //int _Cdecl intcmp(const void* i1,const void *i2);
 33
 34                              // will be changed to much higher
 35                              // than these values
 36  const ROWS=128;             // number of rows (length of first pattern)
 37  const COLS=10;              // number of columns (length of second pattern)
 38  const MAXVEC=128;           // default size of vectors
 39
 40
 41  class matrix;
 42
 43  class vec {
 44      friend ostream& operator<<(ostream& s,vec& v1);
 45      friend class matrix;
 46      friend class bp;
 47      friend istream& operator>>(istream& s,vec& v1);
 48      int n;
 49      float *v;
 50  public:
```

```
  51      vec(int size=MAXVEC,int val=0);             // constructor
  52      ~vec();                                     // destructor
  53      vec(vec &v1);                               // copy-initializer
  54      int length();
  55      float distance(vec& A);
  56      vec& normalize();
  57      vec& normalizeon();
  58      vec& randomize(float initrange=1.0);
  59      vec& scale(vec& minvec,vec& maxvec);
  60      float d_logistic();                         // dot product of
  61                                                  // vector and complement
  62      float maxval();                             // get maximum element
value
  63      float maxerror();                           // calculate maximum error
  64      float maxerrorreset();                      // reset the error
accumulator
  65      float getglobalerror();                     // get the global error
value
  66      static float    global_error;               // contains the error value
  67       vec& garble(float noise);
  68      vec& operator=(const vec& v1);              // vector assignment
  69      vec operator+(const vec& v1);               // vector addition
  70      vec operator+(const float d);
  71      vec& operator+=(const vec& v1);             // vector
  72                                                  // additive-assignment
  73                                                  // supplied for
  74                                                  // completeness,
  75                                                  // but we don't use
  76                                                  // this now
  77      vec& operator*=(float c);                   // vector multiply by
  78                                                  // constant
  79                                                  // vector transpose
  80                                                  // multiply
  81                                                  // needs access to v
  82                                                  // array
  83      int operator==(const vec& v1);
  84      float operator[](int x);
  85      int vec::maxindex();
  86      vec& getstr(char *s);
  87      void putstr(char *s);
  88
  89      vec operator-(const vec& v1);               // subtraction
  90      vec operator-(const float d);               // subtraction
  91      float operator*(const vec& v1);             // dot-product
  92      vec operator*(float c);                     // multiply by constant
  93      vec& sigmoid(vec& thresh);
  94      vec& set(int i,float f=0);
  95
  96      int load(FILE *f);
  97      int save(FILE *f);
  98
  99  };                                              // vector class
 100
 101  class vecpair;
 102
 103  class matrix {
 104  // we only allow access here to improve backpropagation's performance
```

```
105     friend ostream& operator<<(ostream& s,matrix& m1);
106     friend istream& operator>>(istream& s,matrix& m1);
107     protected:
108             float **m;                  // the matrix representation
109             int r,c;                    // number of rows and columns
110     public:
111                                         // constructors
112             matrix(int n=ROWS,int p=COLS,float range=0);
113             matrix(int n,int p,float value,float range);
114             matrix(int n,int p,char *fn);
115             matrix(const vecpair& vp);
116             matrix(vec& v1,vec& v2);
117             matrix(matrix& m1);         // copy-initializer
118             ~matrix();
119             int depth();
120             int width();
121             matrix& operator=(const matrix& m1);
122             matrix& operator=(const vecpair& v);
123             matrix operator+(const matrix& m1);
124
125             vec operator*(vec& v1);
126             vec colslice(int col);
127             vec rowslice(int row);
128             void insertcol(vec& v,int col);
129             void insertrow(vec& v,int row);
130             int closestcol(vec& v);
131             int closestrow(vec& v);
132             int closestrow(vec& v,int *wins,float scaling);
133              int load(FILE *f);
134             int save(FILE *f);
135             float getval(int row,int col);
136             void setval(int row,int col,float val);
137             void initvals(const vec& v1,const vec& v2,
138                     const float rate=1.0, const float momentum=0.0);
139
140             matrix& operator+=(const matrix& m1);
141             matrix& operator*(const float d);
142             matrix& operator*=(const float d);
143 };                                      // matrix class
144
145 class vecpair {
146     friend class matrix;
147     friend istream& operator>>(istream& s,vecpair& v1);
148     friend ostream& operator<<(ostream& s,vecpair& v1);
149     friend matrix::matrix(const vecpair& vp);
150         int flag;                           // flag signalling whether
151                                             // encoding succeeded
152     public:
153         vec *a;
154         vec *b;
155         vecpair(int n=ROWS,int p=COLS,int val=0);       // constructor
156         vecpair(vec& A,vec& B);
157         vecpair(const vecpair& AB);         // copy initializer
158         ~vecpair();
159         vecpair& operator=(const vecpair& v1);
160         int operator==(const vecpair& v1);
161         vecpair& scale(vecpair& minvecs,vecpair& maxvecs);
```

```
162  };
```

APPENDIX B

Raw Bit Map Supporting Source Code

```
                                        #####
 ####           #####     ##     ####  #     #  #####   #####   #    #
#               #    #   #  #   #            #  #    #  #    #  ##  ##
 ####           #    #  #    #   ####   #####   #    #  #####   # ## #
     #          #####   ######       # #        #####   #    #  #    #   ###
#    #          #   #   #    #  #    # #        #       #    #  #    #   ###
 ####  #######  #    #  #    #   ####  #######  #       #####   #    #   ###

  1  #!/bin/sh
  2  #
  3  #  Program  : s_ras2pbm.sh
  4  #  Author   : Al Piszcz
  5  #  Date     : 1993
  6  #  Purpose  : To perform image preprocessing and conversion
  7  #
  8  #sd="/home/suny/99/data/d/l/ras"
  9  #dd="/home/suny/99/data/d/l/pp"
 10  #sd="/home/suny/99/data/u/l/ras"
 11  #dd="/home/suny/99/data/u/l/pp"
 12  sd="/home/suny/99/data/u/t/ras"
 13  dd="/home/suny/99/data/u/t/pp"
 14  #
 15  cd $sd
 16  for filename in `ls ????????.?`
 17  do
 18  #
 19  # 1) anytopnm - convert SUN raster file format to PNM (Portable aNyMap)
 20  # 2) pnmcrop  - crops all white space around image
 21  # 3) pnmscalae- normalize image into a 24x24 pixel array
 22  #
 23      echo "$filename ... "
 24      anytopnm $filename | pnmcrop | pnmscale -width 24 -height 24 \
 25      | pgmtopbm > $dd/$filename
 26  done
```

```
                        #####
#####   #####   #    # #     #  #    #  ######   ####            ####
#    #  #    #  ##  ##       #  #    #  #       #    #          #    #
#    #  #####   # ## #  #####   #    #  #####   #               #
#####   #    #  #    # #        #    #  #       #        ###    #
#       #    #  #    # #         #  #   #       #    #   ###    #    #
#       #####   #    # #######    ##    ######   ####    ###     ####
```

```
    1  /************************************************************************
    2  **
    3  **  Program      : pbm2vec.c
    4  **  Author       : Al Piszcz
    5  **  Date         : 1993
    6  **  Purpose      : Convert a Raw Bit map to 576 length feature vector
    7  **
    8  ***********************************************************************/
    9  #include <string.h>
   10  #include <stdio.h>
   11
   12  main(argc,argv)
   13  int argc;
   14  char **argv;
   15  {
   16      FILE    *in_file;
   17
   18      char    buffer1[1024];
   19      char    buffer2[1024];
   20      char    class;                          /* character class definition
*/
   21      char    input_file_name[128];
   22      char    *success = 0;
   23      char    type[3];
   24
   25      int     class_int;                      /* character class value
integer*/
   26      int     elem_index;                     /* element index
*/
   27      int     elem_index_max;                 /* element index
*/
   28      int     image_cell[128][128];
   29      int     maxgrayval;                     /* maximum gray value
*/
   30      int     oih;                            /* output index high
*/
   31      int     oil;                            /* output index low
*/
   32      int     output_flag;                    /* flag to indicate generation
   33                                                 of output vector desired
*/
   34      int     pixel;                          /* pixel value
*/
   35      int     pixelbyte;
   36      int     r,c;                            /* row and column index
*/
   37      int     rmax,cmax;                      /* row and column in image
*/
```

```
  38      int     totalpixels;                    /* total pixels in image
*/
  39
  40      /*
  41      ** START OF CODE
  42      */
  43
  44      switch(argc)
  45      {
  46      case 2:
  47              strcpy( input_file_name,argv[1]);
  48              output_flag = 0;
  49              break;
  50      case 3:
  51              strcpy( input_file_name,argv[2]);
  52              output_flag = 1;
  53              break;
  54      default:
  55              printf("\n\nusage: pbm2vec [o] inputfile\n");
  56              break;
  57      }
  58
  59      if ( ( in_file = fopen(input_file_name,"r") ) == NULL )
  60      {
  61          fprintf(stderr, "ERROR: INPUT FILE PROBLEMS");
  62              fclose(in_file);
  63              exit(-1);
  64      }
  65
  66      strcpy(buffer1,"");
  67       strcpy(buffer2,"");
  68      success = fgets(buffer1,1023,in_file);
  69      if (success)  sscanf( buffer1, "%s", type);
  70      success = fgets(buffer1,1023,in_file);
  71      if (success)  sscanf( buffer1, "%d %d", &cmax, &rmax);
  72      /* printf("TYPE:[%s]   WIDTH:[%d]   HEIGHT:[%d]\n", type,cmax,rmax); */
  73
  74      /*
  75      ** Decode pixels, which are stored in bytes
  76      ** Load image cell array
  77      */
  78      for ( r = 0; r < rmax ; r++ )
  79      {
  80          for ( c = 0; c < cmax ; c+=8 )
  81          {
  82              pixelbyte = getc(in_file);
  83              image_cell[r][c+7] = pixelbyte & 0x01;
  84              image_cell[r][c+6] = pixelbyte & 0x02;
  85              image_cell[r][c+5] = pixelbyte & 0x04;
  86              image_cell[r][c+4] = pixelbyte & 0x08;
  87              image_cell[r][c+3] = pixelbyte & 0x10;
  88              image_cell[r][c+2] = pixelbyte & 0x20;
  89              image_cell[r][c+1] = pixelbyte & 0x40;
  90              image_cell[r][c] = pixelbyte & 0x80;
  91          }
  92      }
  93
```

```
 94      /*
 95      ** output image cell vector
 96      */
 97      for ( r = 0; r < rmax ; r++ )
 98      {
 99          for ( c = 0; c < cmax ; c++ )
100          {
101              if ( image_cell[r][c] > 0 )
102                  printf("1.0 ");
103              else
104                  printf("0.0 ");
105          }
106      }
107
108      /*
109      ** input vector terminator
110      */
111      printf(" , ");
112
113      /*
114      ** Determine character class
115      ** from input file name, 00000022.A
116      */
117      class = input_file_name[ ( strlen (input_file_name) - 1)];
118      /* printf("CLASS: %c\n",class); */
119      class_int = (int) class;
120
121      if (output_flag)
122      {
123          /*
124          ** Digits
125          */
126          if ( ( class_int < 58 ) && ( class_int > 47 ) )
127          {
128              oil = 48;
129              oih = 57;
130          }
131          else
132
133           /*
134          ** Upper
135          */
136          if ( ( class_int < 91 ) && ( class_int > 64 ) )
137          {
138              oil = 65;
139              oih = 90;
140          }
141          else
142
143          /*
144          ** Lower
145          */
146          if ( ( class_int < 123 ) && ( class_int > 96 ) )
147          {
148              oil = 97;
149              oih = 122;
150          }
```

```
151
152          /*
153          ** Generate output vector
154          */
155          for ( elem_index = oil; elem_index <= oih; elem_index++)
156          {
157              if ( elem_index == class_int ) printf("1.0 ");
158              else printf("0.0 ");
159          }
160
161          /*
162          ** terminate output vector
163          */
164          printf(", ");
165      }
166      printf("\n");
167
168  }
```

```
 ####    ####    ####   #####   ######     #     #####          ####
#       #    #  #    #  #    #  #          #       #           #    #
 ####   #       #    #  #    #  #####      #       #           #
     #  #       #    #  #####   #          #       #     ###    #
#    #  #    #  #    #  #   #   #          #       #     ###    #    #
 ####    ####    ####   #    #  ######     #       #     ###     ####
```

```
  1  /*******************************************************************
  2  **
  3  **  Author: Al Piszcz
  4  **    Date: 1993
  5  ** Purpose: Determine peak output of element in output vector for
  6  **          a 10 node output.
  7  **
  8  *******************************************************************/
  9  #include <stdio.h>
 10
 11  main()
 12  {
 13      FILE    *input;
 14
 15      char    fname[128];
 16      char    buffer[1024];
 17      int     bad, good, highest, success, total;
 18      int     expected;
 19      int     goodone, i;
 20      float   accuracy, maxval;
 21      float   actual[10];
 22
 23      strcpy(fname,"DIG1000.OUT.CR");
 24
 25      /*
 26      ** Open file
 27      */
 28      if ( ( input = fopen(fname,"r") ) == NULL )
 29      {
 30          fprintf(stderr, "ERROR: INPUT FILE PROBLEMS");
 31              fclose(input);
 32              exit(-1);
 33      }
 34
 35      expected=0;
 36      good=0;
 37      bad=0;
 38      total=0;
 39      /*
 40      ** Perform this loop until end of file
 41      */
 42      while ( !(feof(input)))
 43      {
 44          /*
 45          ** read input
 46          ** convert it into 10 floating point values
 47          */
 48          success=fgets(buffer,1023,input);
 49  /*      printf("LINE[%d]:%s",expected,buffer);*/
 50          sscanf(buffer,"%f %f %f %f %f %f %f %f %f %f",
```

```
 51                   &actual[0],&actual[1],&actual[2],&actual[3],&actual[4],
 52                   &actual[5],&actual[6],&actual[7],&actual[8],&actual[9]);
 53           /*
 54           for (i=0;i<10;i++)
 55           {
 56               printf("actual[%d]:[%f]\n",i,actual[i]);
 57           }
 58           */
 59
 60           /*
 61           ** Find peak or highest activation value of vector
 62           */
 63           goodone=0;
 64           maxval = 0.0;
 65           for (i=0;i<10;i++)
 66           {
 67                if ( actual[i] > maxval )
 68               {
 69                   maxval = actual[i];
 70                   goodone=i;
 71               }
 72           }
 73
 74           /*printf("goodone: %d\n",goodone);*/
 75
 76           /*
 77           ** Tally or score expected result with
 78           ** actual highest value
 79           */
 80           switch (expected)
 81           {
 82               case 0:
 83                       if ( goodone == 0 ) good++;
 84                       else bad++;
 85                       break;
 86               case 1:
 87                       if ( goodone == 1) good++;
 88                       else bad++;
 89                       break;
 90               case 2:
 91                       if ( goodone == 2) good++;
 92                       else bad++;
 93                       break;
 94               case 3:
 95                       if ( goodone == 3) good++;
 96                       else bad++;
 97                       break;
 98               case 4:
 99                       if ( goodone == 4) good++;
100                       else bad++;
101                       break;
102               case 5:
103                       if ( goodone == 5) good++;
104                       else bad++;
105                       break;
106               case 6:
107                       if ( goodone == 6) good++;
```

```
108                         else bad++;
109                         break;
110                 case 7:
111                         if ( goodone == 7) good++;
112                         else bad++;
113                         break;
114                 case 8:
115                         if ( goodone == 8) good++;
116                         else bad++;
117                         break;
118                 case 9:
119                         if ( goodone == 9) good++;
120                         else bad++;
121                         break;
122             }
123             total++;
124             expected=(total % 10);
125 /*          printf("expected: %d\n",expected); */
126         }
127 
128         /*
129         ** calculate and display accuracy
130         */
131         accuracy = (float)good/total;
132         printf("ACCURACY: %f\n",accuracy);
133 }
```

APPENDIX C

Side Contour Profile Supporting Source Code

```
                                        #####
 ####           #####   #####   #    # #     #  #    #  #    #           ####
#               #    #  #    #  ##  ##       #  ##  ##  ##  ##          #
 ####           #    #  #####   # ## #  #####   # ## #  # ## #           ####
     #          #####   #    #  #    # #        #    #  #    #   ###         #
#    #          #       #    #  #    # #        #    #  #    #   ###    #    #
 ####  #######  #       #####   #    # #######  #    #  #    #   ###     ####
```

```
  1  #!/bin/sh
  2  #
  3  #  Program  : s_pbm2mm.sh
  4  #  Author   : Al Piszcz
  5  #  Date     : 1993
  6  #  Purpose  : To perform image preprocessing and conversion
  7  #
  8  sd="/suny/99/data/d/l/before"
  9  dd="/suny/99/data/d/l/mm"
 10  #
 11  cd $sd
 12  for filename in `ls ????????.?`
 13  do
 14  #
 15  # 1) pnmcrop  - crops all white space around image
 16  # 2) pnmscale- normalize image into a 32x32 pixel array
 17  #
 18      echo "$filename ... "
 19      pnmcrop $filename | pnmscale -width 30 -height 30 \
 20      | pnmmargin -white 1 | pgmtopbm > $dd/$filename
 21  done
```

```
 ####            ####   ######  #    #  #    #  #    #  ######  ######    ##
#               #    #  #       ##   #  ##  ##  ##  ##  #       #        #  #
 ####           #       #####   # #  #  # ## #  # ## #  #####   #####   #    #
     #          #  ###  #       #  # #  #    #  #    #  #       #       ######
#    #          #    #  #       #   ##  #    #  #    #  #       #       #    #
 ####  #######   ####   ######  #    #  #    #  #    #  #       ######  #    #
```

```
  1  #!/bin/sh
  2  #
  3  #  Program  : s_genmmfeat.sh
  4  #  Author   : Al Piszcz
  5  #  Date     : 1993
  6  #  Purpose  : To create feature vectors for training or testing
  7  #
  8  sd="/suny/99/data/d/t/pp32"
  9  dd="/suny/99/data/d/t/mmfv"
 10  #
 11  cd $sd
 12  for filename in `ls ?????1??.?`
 13  do
 14      mm2vec $filename >> $dd/DIGMM1.IN
 15  done
```

```
                #####
#    #  #    # #     #  #    #  ######   ####            ####
##  ##  ##  ##       #  #    #  #       #    #          #    #
# ## #  # ## #  #####   #    #  #####   #               #
#    #  #    # #        #    #  #       #        ###    #
#    #  #    # #         #  #   #       #    #   ###    #    #
#    #  #    # #######    ##    ######   ####    ###     ####

   1  /***********************************************************************
   2  **
   3  **  Program      : mm2vec.sh
   4  **  Author       : Al Piszcz
   5  **  Date         : 1993
   6  **  Purpose      : Convert a Raw Bit map to a 48 vector mesh map
   7  **               : description vector
   8  **
   9  ** Mesh map vector description.
  10  ** Mesh element is measured from
  11  ** respective side inwards until
  12  ** first black pixel is detected.
  13  **
  14  **
  15  **      Side 0
  16  **          |
  17  **          v
  18  **
  19  **         +-------+
  20  **         |.. #...|
  21  **  v1 --> |..# #..| <--
  22  **         |.#   #.|
  23  ** Side    |#     #|   Side
  24  **  3      |#######|     1
  25  **         |#     #|
  26  **         |#     #|
  27  **         +-------+
  28  **             ^
  29  **             |
  30  **
  31  **          Side 2
  32  ***********************************************************************/
  33  #include <string.h>
  34  #include <stdio.h>
  35
  36  #define FALSE 0
  37  #define TRUE 1
  38
  39  main(argc,argv)
  40  int argc;
  41  char **argv;
  42  {
  43      FILE    *in_file;
  44
  45      char    buffer1[1024];
  46      char    buffer2[1024];
  47      char    class;                          /* character class definition
*/
  48      char    input_file_name[128];
```

```
   49      char    *success = 0;
   50      char    type[3];
   51
   52      float   normalized_value = 0.0;         /* node value for input
*/
   53      int     class_int;                      /* character class value
integer*/
   54      int     edge;                           /* boolean indicator for pixel
*/
   55      int     elem_index;                     /* element index
*/
   56      int     elem_index_max;                 /* element index
*/
   57      int     image_cell[128][128];
   58      int     mm[4][32];                      /* mesh map values
*/
   59      int     maxgrayval;                     /* maximum gray value
*/
   60      int     oih;                            /* output index high
*/
   61      int     oil;                            /* output index low
*/
   62      int     output_flag;                    /* flag to indicate generation
   63                                                 of output vector desired
*/
   64      int     pixel;                          /* pixel value
*/
   65      int     pixelbyte;
   66      int     r,c;                            /* row and column index
*/
   67       int     rmax,cmax;                      /* row and column in image
*/
   68      int     side;                           /* measurement side
*/
   69      int     side_index;                     /* side index               */
   70      int     totalpixels;                    /* total pixels in image
*/
   71
   72      /*
   73      ** START OF CODE
   74      */
   75
   76      switch(argc)
   77      {
   78      case 2:
   79              strcpy( input_file_name,argv[1]);
   80              output_flag = 0;
   81              break;
   82      case 3:
   83              strcpy( input_file_name,argv[2]);
   84              output_flag = 1;
   85              break;
   86      default:
   87              printf("\n\nusage: pbm2vec [o] inputfile\n");
   88              break;
   89      }
   90
```

```
 91      if ( ( in_file = fopen(input_file_name,"r") ) == NULL )
 92      {
 93          fprintf(stderr, "ERROR: INPUT FILE PROBLEMS");
 94              fclose(in_file);
 95              exit(-1);
 96      }
 97
 98      strcpy(buffer1,"");
 99      strcpy(buffer2,"");
100      success = fgets(buffer1,1023,in_file);
101      if (success)  sscanf( buffer1, "%s", type);
102      success = fgets(buffer1,1023,in_file);
103      if (success)  sscanf( buffer1, "%d %d", &cmax, &rmax);
104      /* printf("TYPE:[%s]   WIDTH:[%d]   HEIGHT:[%d]\n", type,cmax,rmax); */
105
106      /*
107      ** Decode pixels, which are stored in bytes
108      ** Load image cell array
109      */
110      for ( r = 0; r < rmax ; r++ )
111      {
112          for ( c = 0; c < cmax ; c+=8 )
113          {
114              pixelbyte = getc(in_file);
115              image_cell[r][c+7] = pixelbyte & 0x01;
116              image_cell[r][c+6] = pixelbyte & 0x02;
117              image_cell[r][c+5] = pixelbyte & 0x04;
118              image_cell[r][c+4] = pixelbyte & 0x08;
119              image_cell[r][c+3] = pixelbyte & 0x10;
120              image_cell[r][c+2] = pixelbyte & 0x20;
121              image_cell[r][c+1] = pixelbyte & 0x40;
122              image_cell[r][c] = pixelbyte & 0x80;
123          }
124      }
125
126      /*
127      ** Determine side 3 values
128      */
129      elem_index_max = 31;
130      side = 3;
131      for ( r = 0; r < rmax ; r++ )
132      {
133           c = 0;
134          edge = FALSE;
135          while ( ( c < cmax ) && ( edge == FALSE ) )
136          {
137              if (image_cell[r][c] > 0)
138              {
139                  edge = TRUE;
140                  /*
141                  ** If c is the max value it means nothing
142                  ** was detected in this slice
143                  */
144                  if ( c == elem_index_max )
145                      c = 0;
146                  mm[side][r] = c;
147              }
```

```
148             else
149                 c++;
150         }
151     }
152
153     /*
154     ** Determine side 1 values
155     */
156     side = 1;
157     elem_index_max = 31;
158     for ( r = 0; r < rmax ; r++ )
159     {
160         c = cmax - 1;
161         edge = FALSE;
162         while ( ( c > 0 ) && ( edge == FALSE ) )
163         {
164             if (image_cell[r][c] > 0)
165             {
166                 edge = TRUE;
167                 if ( c == elem_index_max )
168                     c = 0;
169                 mm[side][r] = cmax - c;
170             }
171             else
172                 c--;
173         }
174     }
175
176     /*
177     ** Determine side 2 values
178     */
179     side = 2;
180     for ( c = 0; c < cmax ; c++ )
181     {
182         r = rmax - 1;
183         edge = FALSE;
184         while ( ( r > 0 ) && ( edge == FALSE ) )
185         {
186             if (image_cell[r][c] > 0)
187             {
188                 edge = TRUE;
189                 if ( r == 0 )
190                     r = 31;
191                 mm[side][c] = rmax - r;
192             }
193             else
194                 r--;
195         }
196     }
197
198     /*
199      ** Determine side 0 values
200     */
201     side = 0;
202     for ( c = 0; c < cmax ; c++ )
203     {
204         r = 0;
```

```
 205          edge = FALSE;
 206          while ( ( r < rmax ) && ( edge == FALSE ) )
 207          {
 208              if (image_cell[r][c] > 0)
 209              {
 210                  edge = TRUE;
 211                  if ( r == elem_index_max )
 212                      r = 0;
 213                  mm[side][c] = r;
 214              }
 215              else
 216                  r++;
 217          }
 218      }
 219
 220      /*
 221      ** Output element values
 222      */
 223      for ( side_index = 0; side_index <4; side_index++)
 224      {
 225          /*
 226          ** Since it is a square image cell, either
 227          ** side, row or column will be adequate as an index
 228          */
 229          for ( r = 0; r < rmax; r++)
 230          {
 231              if ( mm[side_index][r] > 0 )
 232                  normalized_value = (float)(mm[side_index][r] / 30.0);
 233              else
 234                  normalized_value = 0.0;
 235
 236              printf("%4.3f ",normalized_value);
 237              /*printf("mm[%d][%d]:[%d]\n",
side_index,r,mm[side_index][r]);*/
 238
 239          }
 240      }
 241
 242
 243
 244      /*
 245      ** input vector terminator
 246      */
 247      printf(" , ");
 248
 249      /*
 250      ** Determine character class
 251      ** from input file name, 00000022.A
 252      */
 253      class = input_file_name[ ( strlen (input_file_name) - 1)];
 254      /* printf("CLASS: %c\n",class); */
 255      class_int = (int) class;
 256
 257      if (output_flag)
 258      {
 259          /*
 260          ** Digits
```

```
261         */
262         if ( ( class_int < 58 ) && ( class_int > 47 ) )
263         {
264             oil = 48;
265              oih = 57;
266         }
267         else
268 
269         /*
270         ** Upper
271         */
272         if ( ( class_int < 91 ) && ( class_int > 64 ) )
273         {
274             oil = 65;
275             oih = 90;
276         }
277         else
278 
279         /*
280         ** Lower
281         */
282         if ( ( class_int < 123 ) && ( class_int > 96 ) )
283         {
284             oil = 97;
285             oih = 122;
286         }
287 
288         /*
289         ** Generate output vector
290         */
291         for ( elem_index = oil; elem_index <= oih; elem_index++)
292         {
293             if ( elem_index == class_int ) printf("1.0 ");
294             else printf("0.0 ");
295         }
296 
297         /*
298         ** terminate output vector
299         */
300         printf(", ");
301     }
302     printf("\n");
303 }
```

APPENDIX D

Quadrant Method Supporting Source Code

```
 ####            ####   ######  #    #   ####   #    #  ######  ######    ##
#               #    #  #       ##   #  #    #  ##  ##  #       #        #  #
 ####           #       #####   # #  #  #    #  # ## #  #####   #####   #    #
     #          #  ###  #       #  # #  #  # #  #    #  #       #       ######
#    #          #    #  #       #   ##  #   #   #    #  #       #       #    #
 ####  #######   ####   ######  #    #   ### #  #    #  #       ######  #    #

   1   #!/bin/sh
   2   #
   3   #  Program  : s_genqmfeat.sh
   4   #  Author   : Al Piszcz
   5   #  Date     : 1993
   6   #  Purpose  : To create feature vectors for training or testing
   7   #
   8   sd="/suny/99/data/u/t/pp32"
   9   dd="/suny/99/data/u/t/qmfv"
  10   #
  11   cd $sd
  12   for filename in `ls ????????.?`
  13   do
  14   #   qm2vec o $filename >> $dd/QMDIG1.part
  15       qm2vec $filename >> $dd/QMDIG1.IN
  16   done
```

```
                #####
 ####   #    # #     #  #    #  ######   ####            ####
#    #  ##  ##       #  #    #  #       #    #          #    #
#    #  # ## #  #####   #    #  #####   #               #
#  # #  #    # #        #    #  #       #        ###    #
#   #   #    # #         #  #   #       #    #   ###    #    #
 ### #  #    # #######    ##    ######   ####    ###     ####

    1  /*********************************************************************
    2  **
    3  **  Program      : qm2vec.sh
    4  **  Author       : Al Piszcz
    5  **  Date         : 1993
    6  **  Purpose      : Convert a Raw Bit map to a 64 element vector quadrant
    7  **               : description
    8  **
    9  **
   10  *********************************************************************/
   11  #include <string.h>
   12  #include <stdio.h>
   13
   14  #define FALSE 0
   15  #define TRUE 1
   16
   17  main(argc,argv)
   18  int argc;
   19  char **argv;
   20  {
   21      FILE    *in_file;
   22
   23      char    buffer1[1024];
   24      char    buffer2[1024];
   25      char    class;                          /* character class definition
*/
   26      char    input_file_name[128];
   27      char    *success = 0;
   28      char    type[3];
   29
   30      float   qm04[64];                       /* quadrant map totals
*/
   31      float   quad_weight;                    /* quadrant weight value
*/
   32
   33      int     class_int;                      /* character class value
integer*/
   34      int     elem_index;                     /* element index
*/
   35      int     i, j;                           /* general purpose indexes
*/
   36      int     image_cell[128][128];
   37      int     quadrant;                       /* current quadrant
*/
   38      int     oih;                            /* output index high
*/
   39      int     oil;                            /* output index low
*/
   40      int     output_flag;                    /* flag to indicate generation
```

```
  41                                                 of output vector desired
*/
  42      int     pixel;                          /* pixel value
*/
  43      int     pixelbyte;
  44      int     r,c;                            /* row and column index
*/
  45      int     rih,cih;                        /* row and column high limit
*/
  46      int     ril,cil;                        /* row and column low limit
*/
  47      int     rmax,cmax;                      /* row and column in image
*/
  48
  49      /*
  50      ** START OF CODE
  51      */
  52
  53      switch(argc)
  54      {
  55      case 2:
  56              strcpy( input_file_name,argv[1]);
  57              output_flag = 0;
  58              break;
  59      case 3:
  60              strcpy( input_file_name,argv[2]);
  61              output_flag = 1;
  62              break;
  63      default:
  64              printf("\n\nusage: pbm2vec [o] inputfile\n");
  65              break;
  66      }
  67
  68      if ( ( in_file = fopen(input_file_name,"r") ) == NULL )
  69      {
  70          fprintf(stderr, "ERROR: INPUT FILE PROBLEMS");
  71              fclose(in_file);
  72              exit(-1);
  73      }
  74
  75      strcpy(buffer1,"");
  76      strcpy(buffer2,"");
  77      success = fgets(buffer1,1023,in_file);
  78      if (success)  sscanf( buffer1, "%s", type);
  79      success = fgets(buffer1,1023,in_file);
  80      if (success)  sscanf( buffer1, "%d %d", &cmax, &rmax);
  81      /* printf("TYPE:[%s]   WIDTH:[%d]   HEIGHT:[%d]\n", type,cmax,rmax); */
  82
  83      /*
  84      ** Decode pixels, which are stored in bytes
  85      ** Load image cell array
  86      */
  87      for ( r = 0; r < rmax ; r++ )
  88      {
  89          for ( c = 0; c < cmax ; c+=8 )
  90          {
  91              pixelbyte = getc(in_file);
```

```
 92             image_cell[r][c+7] = pixelbyte & 0x01;
 93             image_cell[r][c+6] = pixelbyte & 0x02;
 94             image_cell[r][c+5] = pixelbyte & 0x04;
 95             image_cell[r][c+4] = pixelbyte & 0x08;
 96             image_cell[r][c+3] = pixelbyte & 0x10;
 97             image_cell[r][c+2] = pixelbyte & 0x20;
 98             image_cell[r][c+1] = pixelbyte & 0x40;
 99             image_cell[r][c] = pixelbyte & 0x80;
100         }
101     }
102
103     /*
104     ** Initialize quadrant matrices
105     */
106     for ( i = 0; i<64; i++ ) qm04[i] = 0.0;
107
108
109
110
111
112     /*
113     ** Determine quadrant map with (64) 4x4 vector values
114     */
115     quad_weight = (1.0 / 16.0);
116
117     for (quadrant = 0; quadrant < 64 ; quadrant++)
118     {
119         /*
120         ** calculate quadrant coordinate limits
121         */
122         if (quadrant > 0)
123             ril = ( (int)( quadrant / 8) ) * 4;
124         else
125             ril = 0;
126
127         rih = ril + 4;
128
129         cil = ( quadrant % 8 ) * 4;
130         cih = cil + 4;
131
132         /*printf("Quadrant[%03d]:   ",quadrant);*/
133          /*printf("ril:%2d   cil:%2d   rih:%2d   cih:%2d\n",
134                      ril,cil,rih,cih); */
135         for ( r = ril; r < rih ; r++ )
136         {
137             for ( c = cil; c < cih ; c++ )
138             {
139                 if (image_cell[r][c] > 0)
140                 {
141                     qm04[quadrant]+=quad_weight;
142                 }
143             }
144         }
145     }
146
147
148
```

```
149     /*
150     ** Output element values
151     */
152     for ( i = 0; i<64; i++ )
153     {
154         printf("%4.3f ",qm04[i]);
155     }
156
157
158     /*
159     ** input vector terminator
160     */
161     printf(" , ");
162
163     /*
164     ** Determine character class
165     ** from input file name, 00000022.A
166     */
167     class = input_file_name[ ( strlen (input_file_name) - 1)];
168     /* printf("CLASS: %c\n",class); */
169     class_int = (int) class;
170
171     if (output_flag)
172     {
173         /*
174         ** Digits
175         */
176         if ( ( class_int < 58 ) && ( class_int > 47 ) )
177         {
178             oil = 48;
179             oih = 57;
180         }
181         else
182
183         /*
184         ** Upper
185         */
186         if ( ( class_int < 91 ) && ( class_int > 64 ) )
187         {
188             oil = 65;
189             oih = 90;
190         }
191         else
192
193         /*
194         ** Lower
195         */
196         if ( ( class_int < 123 ) && ( class_int > 96 ) )
197         {
198             oil = 97;
199              oih = 122;
200         }
201
202         /*
203         ** Generate output vector
204         */
205         for ( elem_index = oil; elem_index <= oih; elem_index++)
```

```
206          {
207               if ( elem_index == class_int ) printf("1.0 ");
208               else printf("0.0 ");
209          }
210
211          /*
212          ** terminate output vector
213          */
214          printf(", ");
215     }
216     printf("\n");
217 }
```

APPENDIX E

Hough Transform Method Supporting Source Code

```
                                        #####
 ####           #####   #####   #    # #     #  #    #   #####           ####
#               #    #  #    #  ##  ##       #  #    #     #            #
 ####           #    #  #####   # ## #  #####   ######     #             ####
     #          #####   #    #  #    # #        #    #     #     ###         #
#    #          #       #    #  #    # #        #    #     #     ###    #    #
 ####  #######  #       #####   #    # ####### #    #     #     ###     ####

   1  #!/bin/sh
   2  #
   3  #  Program  : s_pbm2ht.sh
   4  #  Author   : Al Piszcz
   5  #  Date     : 1993
   6  #  Purpose  : To perform image preprocessing and conversion
   7  #
   8  sd="/suny/99/data/es/t/pp32"
   9  dd="/suny/99/data/es/t/ht"
  10  #
  11  cd $sd
  12  for filename in `ls ????????.?`
  13  do
  14  #
  15  # 1) Take Hough of Raster file and convert to PBM
  16  #
  17      echo "$filename ... "
  18      pnmtorast $filename > $filename.ras
  19      /hd3/imagepro/bin/hough $filename.ras  $filename.hgh
  20      anytopnm  $filename.hgh > $dd/$filename
  21      /bin/rm $filename.ras* $filename.hgh
  22  done
```

```
 ####           ####   ######  #    #  #    #   #####  ######  ######    ##
#              #    #  #       ##   #  #    #     #    #       #        #  #
 ####          #       #####   # #  #  ######     #    #####   #####   #    #
     #         #  ###  #       #  # #  #    #     #    #       #       ######
#    #         #    #  #       #   ##  #    #     #    #       #       #    #
 ####  #######  ####   ######  #    #  #    #     #    #       ######  #    #
```

```
  1  #!/bin/sh
  2  #
  3  #  Program  : s_genhtfeat.sh
  4  #  Author   : Al Piszcz
  5  #  Date     : 1993
  6  #  Purpose  : To create feature vectors for training or testing
  7  #
  8  sd="/suny/99/data/es/t/ht"
  9  dd="/suny/99/data/es/t/htfv"
 10  #
 11  cd $sd
 12  for filename in `ls ????????.?`
 13  do
 14      ht2vec $filename >> $dd/HTUP1.IN
 15  done
```

```
 #    #   ####   #    #   ####   #    #           ####
 #    #  #    #  #    #  #    #  #    #          #    #
 ######  #    #  #    #  #       ######          #
 #    #  #    #  #    #  #  ###  #    #   ###    #
 #    #  #    #  #    #  #    #  #    #   ###    #    #
 #    #   ####    ####    ####   #    #   ###     ####

     1  #include <stdio.h>
     2  #include <math.h>
     3
     4  #define DEG2RAD (M_PI/180.0)
     5  /*
     6  **
     7  ** Proof of concept and understanding program
     8  ** for Hough Transform
     9  */
    10  main()
    11  {
    12
    13      double      cos(), sin();
    14
    15      int pixmap[24][24];     /* Source pixel image       */
    16
    17      int hmax;               /* Maximum value in hough   */
    18      int hs[180][45];        /* Hough space map          */
    19
    20      int image_w, image_h;   /* pixel image dimensions   */
    21      int maxr;               /* maximum rectangle        */
    22      int r, theta;           /* Hough Space indexes      */
    23      int x,y;                /* pixel map indexes        */
    24      int xoff,yoff;          /* pixel map center         */
    25
    26      int c[180], s[180];
    27
    28      image_h = 24;
    29      image_w = 24;
    30      xoff = image_w/2;
    31      yoff = image_h/2;
    32
    33      maxr = (int) (sqrt ((double) (image_h * image_h + image_w * image_w ))
/ 2 );
    34
    35      printf("maxr: %d\n",maxr);
    36
    37      /*
    38      ** Build sin and cos table
    39      ** for fast computation of sine and cosine
    40      */
    41      for (theta=0; theta < 180; theta++)
    42      {
    43          c[theta] = (int) (cos (DEG2RAD *theta) * 1000);
    44          s[theta] = (int) (sin (DEG2RAD *theta) * 1000);
    45      }
    46
    47      /*
    48      ** Initialize both spaces
    49      */
```

```
 50     for (x=0;x<image_w;x++)
 51     {
 52         for (y=0;y<image_h;y++)
 53         {
 54             pixmap[x][y] = 0;
 55         }
 56     }
 57
 58     for (r=0;r<2*maxr + 1;r++)
 59     {
 60         for (theta=0;theta<180;theta++)
 61         {
 62             hs[theta][r] = 0;
 63         }
 64     }
 65
 66     /*
 67      ** load pixel image
 68     */
 69     for (x=0;x<image_w;x++)
 70     {
 71             pixmap[x][(image_w - 1) - x] = 1;
 72             pixmap[x][x] = 1;
 73     }
 74
 75     /*
 76     ** Display pixel image
 77     */
 78     for (y=0;y<image_h;y++)
 79     {
 80         for (x=0;x<image_w;x++)
 81         {
 82             if ( pixmap[x][y] == 0 )
 83                 printf(" .");
 84             else
 85                 printf(" *");
 86         }
 87         printf("\n");
 88     }
 89     printf("\n");
 90
 91     /*
 92     ** HOUGH
 93     */
 94     for ( y=0; y<image_w; y++)
 95     {
 96         for ( x=0; x<image_h; x++)
 97         {
 98             if ( pixmap[x][y] == 1 )
 99             {
100                 for (theta=0; theta < 180; ++theta)
101                 {
102                     r = ( y - yoff ) * s[theta]
103                         -
104                         ( x - xoff ) * c[theta] ;
105                     r = r < 0 ? r / 1000 : r / 1000 + 1;
106                     hs[theta][maxr + r] = hs[theta][maxr + r] + 1;
```

```
  107                 }
  108             }
  109         }
  110     }
  111
  112     /*
  113     ** Find Maximum Value in Hough Space Matrix
  114     */
  115     hmax=0;
  116     for (r=0; r < 2 * maxr + 1; r++)
  117     {
  118         for (theta=0;theta<180;theta++)
  119         {
  120             if ( hs[theta][r] > hmax )
  121             {
  122                 hmax = hs[theta][r];
  123             }
  124 /*          printf("hs[%d][%d]: %d  hmax:%d\n",theta,r,hs[theta][r],hmax);
*/
  125         }
  126     }
  127     printf("hmax: %d\n",hmax);
  128
  129     /*
  130     ** Normalize Hough Space Matrix to 255 maximum value
  131     */
  132     for (r=0;r< 2 * maxr + 1;r++)
  133      {
  134         for (theta=0;theta<180;theta++)
  135         {
  136             if ( hs[theta][r] > 0 )
  137             hs[theta][r] = (int)
  138                             ((float)(hs[theta][r] / (float)hmax) * 255.0);
  139         }
  140     }
  141
  142     /*
  143     ** Display Hough Space Matrix Values
  144     ** Create a .pgm file format
  145     */
  146     printf("P2\n");
  147     printf("%d %d\n", theta, 2 * maxr+1 );
  148     printf("255\n");
  149     for (r=0;r< 2 * maxr + 1;r++)
  150     {
  151         for (theta=0;theta<180;theta++)
  152         {
  153             if ( hs[theta][r] == 0 )
  154                 printf("  0");
  155             else
  156                 printf(" %3d",hs[theta][r]);
  157         }
  158         printf("\n");
  159     }
  160     printf("\n");
  161
  162 }
```

```
                #####
#    #   ##### #     #  #    #  ######   ####            ####
#    #     #         #  #    #  #       #    #          #    #
######     #    #####   #    #  #####   #               #
#    #     #   #        #    #  #       #        ###    #
#    #     #   #         #  #   #       #    #   ###    #    #
#    #     #   #######    ##    ######   ####    ###     ####

    1  /***********************************************************************
    2  **
    3  **  Program      : ht2vec.sh
    4  **  Author       : Al Piszcz
    5  **  Date         : 1993
    6  **  Purpose      : Convert a hough transform image to a 8100 element
    7  **               : vector
    8  **
    9  ***********************************************************************/
   10  #include <string.h>
   11  #include <stdio.h>
   12
   13  #define FALSE 0
   14  #define TRUE 1
   15
   16  main(argc,argv)
   17  int argc;
   18  char **argv;
   19  {
   20      FILE    *in_file;
   21
   22      char    buffer1[1024];
   23      char    buffer2[1024];
   24      char    class;                          /* character class definition
*/
   25      char    input_file_name[128];
   26      char    *success = 0;
   27      char    type[3];
   28
   29      float   normalized_value = 0.0;         /* node value for input
*/
   30      int     class_int;                      /* character class value
integer*/
   31      int     edge;                           /* boolean indicator for pixel
*/
   32      int     elem_index;                     /* element index
*/
   33      int     elem_index_max;                 /* element index
*/
   34      int     image_cell[45][180];
   35      int     ht[8100];                       /* hough values
*/
   36      int     maxgrayval;                     /* maximum gray value
*/
   37      int     oih;                            /* output index high
*/
   38      int     oil;                            /* output index low
*/
   39      int     output_flag;                    /* flag to indicate generation
```

```
  40                                                  of output vector desired
*/
  41      int     pixel;                          /* pixel value
*/
  42      int     pixelbyte;
  43      int     r,c;                            /* row and column index
*/
  44      int     rmax,cmax;                      /* row and column in image
*/
  45
  46
  47      int     totalpixels;                    /* total pixels in image
*/
  48
  49      /*
  50      ** START OF CODE
  51      */
  52
  53      switch(argc)
  54      {
  55      case 2:
  56              strcpy( input_file_name,argv[1]);
  57              output_flag = 0;
  58              break;
  59      case 3:
  60              strcpy( input_file_name,argv[2]);
  61              output_flag = 1;
  62              break;
  63      default:
  64              printf("\n\nusage: pbm2vec [o] inputfile\n");
  65              break;
  66      }
  67
  68      if ( ( in_file = fopen(input_file_name,"r") ) == NULL )
  69      {
  70          fprintf(stderr, "ERROR: INPUT FILE PROBLEMS");
  71              fclose(in_file);
  72              exit(-1);
  73      }
  74
  75      strcpy(buffer1,"");
  76      strcpy(buffer2,"");
  77      success = fgets(buffer1,1023,in_file);
  78      if (success)  sscanf( buffer1, "%s", type);
  79      success = fgets(buffer1,1023,in_file);
  80      if (success)  sscanf( buffer1, "%d %d", &cmax, &rmax);
  81      /* printf("TYPE:[%s]   WIDTH:[%d]   HEIGHT:[%d]\n", type,cmax,rmax); */
  82
  83      /*
  84      ** Decode pixels, which are stored in bytes
  85      ** Load image cell array
  86      */
  87      for ( r = 0; r < rmax ; r++ )
  88      {
  89          for ( c = 0; c < cmax ; c++ )
  90          {
  91              pixelbyte = getc(in_file);
```

```
 92             image_cell[r][c] = pixelbyte;
 93             /*printf("imagecell[%d][%d]: %d\n",r,c,image_cell[r][c]);*/
 94             if ( image_cell[r][c] > 0 )
 95                 normalized_value = (float)(image_cell[r][c] / 255.0);
 96             else
 97                 normalized_value = 0.0;
 98 
 99             printf("%4.3f ",normalized_value);
100 
101         }
102     }
103 
104 
105     /*
106     ** input vector terminator
107     */
108     printf(" , ");
109 
110     /*
111     ** Determine character class
112     ** from input file name, 00000022.A
113     */
114     class = input_file_name[ ( strlen (input_file_name) - 1)];
115     /* printf("CLASS: %c\n",class); */
116     class_int = (int) class;
117 
118     if (output_flag)
119     {
120         /*
121         ** Digits
122         */
123         if ( ( class_int < 58 ) && ( class_int > 47 ) )
123         {
124             oil = 48;
125             oih = 57;
126         }
127         else
128 
129         /*
130         ** Upper
131         */
132         if ( ( class_int < 91 ) && ( class_int > 64 ) )
133         {
134             oil = 65;
135             oih = 90;
136         }
137         else
138 
139         /*
140         ** Lower
141         */
142         if ( ( class_int < 123 ) && ( class_int > 96 ) )
143         {
144             oil = 97;
145             oih = 122;
146         }
147 
```

```
148         /*
149         ** Generate output vector
150         */
151         for ( elem_index = oil; elem_index <= oih; elem_index++)
152         {
153             if ( elem_index == class_int ) printf("1.0 ");
154             else printf("0.0 ");
155         }
156
157         /*
158         ** terminate output vector
159         */
160         printf(", ");
161     }
162     printf("\n");
163 }
```

ALV CODE

```
 #    #   ####   #    #   ####   #    #           ####
 #    #  #    #  #    #  #    #  #    #          #    #
 ######  #    #  #    #  #       ######          #
 #    #  #    #  #    #  #  ###  #    #   ###    #
 #    #  #    #  #    #  #    #  #    #   ###    #    #
 #    #   ####    ####    ####   #    #   ###     ####

    1   #include <stdio.h>
    2   #include "defs.h"
    3
    4   #define BLACK   1
    5   #define DEG2RAD (M_PI/180.)
    6
    7   char           *progname;
    8   char           *filename;
    9   Pixrect        *pr1, *pr2;
   10
   11   #ifdef STANDALONE
   12   main(argc, argv, envp)
   13   #else
   14   hough_main(argc, argv, envp)
   15   #endif
   16       int             argc;
   17       char           **argv;
   18       char           **envp;
   19   {
   20       register int    i, j;
   21       int             levels;
   22       Pixrect *hough();
   23
   24       progname = strsave(argv[0]);
   25       parse_profile(&argc, argv, envp);
   26
   27       while ((gc = getopt(argc, argv, " ")) != EOF)
   28           switch (gc) {
   29           case '?':
   30               errflag++;
   31               break;
   32           }
   33
   34       if (errflag)
   35           error((char *) 0, "Usage: %s: [infile] [outfile]\n", progname);
   36
   37       for (stream = 0; optind < argc; stream++, optind++)
   38           if (stream < 2 && strcmp(argv[optind], "-") != 0)
   39               if (freopen(argv[optind], mode[stream], f[stream]) == NULL)
   40                   error("%s %s", PR_IO_ERR_INFILE, argv[optind]);
   41
   42       if ((pr1 = pr_load(stdin, NULL)) == NULL)
   43           error(PR_IO_ERR_RASREAD);
   44
   45       if (pr1->pr_depth != 1)
   46           error("Input image must be 1 bit deep");
   47
   48       pr2 = hough(pr1);
```

```
49
50        return(pr_dump(pr2, stdout, NULL, RT_STANDARD, 0));
51    }
52
53
54
55    Pixrect *hough (pr)
56    Pixrect *pr;
57    {
58        double sqrt ();                     /* square root function     */
59        Pixrect *hp;                        /* new image for hough
                                               transformation            */
60        int xoff, yoff;                     /* centre image = origin
                                               hough space */
61        register int r, theta;              /* coords in hough space    */
62        register int x, y;                  /* coords in image */
63        int maxr;                           /* max val of r possible for
                                               image                     */
64        int c[180], s[180];                 /* lookup tables for cos
                                               and sin                   */
65
66        /*  initialise variables  */
67
68        xoff = pr->pr_size.x/2;
69        yoff = pr->pr_size.y/2;
70        maxr = (int) (sqrt ((double) (pr->pr_size.x * pr->pr_size.x
                        + pr->pr_size.y * pr->pr_size.y)) / 2);
71
72        for (theta = 0; theta < 180; theta++) {
73            c[theta] = (int) (cos (DEG2RAD * theta) * 1000);
74            s[theta] = (int) (sin (DEG2RAD * theta) * 1000);
75        }
76
77        /*  create new image  */
78
79        if ((hp = mem_create (180, 2 * maxr + 1, 8)) == NULL)
80            error("mem_create returned NULL");
81
82        for (r = 0; r < 2 * maxr + 1; r++)
83            for (theta = 0; theta < 180; theta++)
84                pr_put(hp, theta, r, 0);
85
86        /*  step through image  */
87
88        for (y = 0; y < pr->pr_size.y; y++)
89            for (x = 0; x < pr->pr_size.x; x++)
90                if (pr_get(pr, x,y) == BLACK)
91                    for (theta = 0; theta < 180; ++theta) {
92                        r = (y - yoff) * s[theta] - (x - xoff) * c[theta];
93                        r = r < 0 ? r / 1000 : r / 1000 + 1;
94                        pr_put(hp, theta, maxr + r, pr_get(hp,
                                theta, maxr + r) +1);
95                    }
96
97        return (hp);
98    }
```

REFERENCES

[1] Fischler and Firchein. Intelligence: The Eye, the Brain, and the Computer. Addison-Wesley. April 1987.

[2] Minsky and Papert. *Perceptrons* . MIT Press. 1969.

[3] Dagli, Cihan H.; Burke, Laura I.; Shin, Yung C. INTELLIGENT ENGINEERING SYSTEMS THROUGH ARTIFICIAL NEURAL NETWORKS, VOLUME 2. ASME PRESS. 1992.

[4] Ibid. Cover.

[5] Tauschek, G. "Reading machine." U.S. Patent 2 026 329. Dec. 1935.

[6] P.W. Handel. "Statistical machine." U.S. Patent 1 915 993. June 1933.

[7] Glauberman, M. H. "Character Recognition for business machines." Electronics. Feb. 1956. pp. 132-136.

[8] Hannan, W.J. "R.C.A. multifont reading machine." Optical Character Recognition. Eds. McGregor & Welmer. 1962. pp. 3-14.

[9] ERA. "An electronic reading automaton." Electronic Eng. Apr 1957. pp. 189-190.

[10] Horwitz, L.P. and Shelton, G.L. "Pattern recognition using autocorrelation." Proc.

IRE. vol. 59. no. 1. 1961. pp. 175-185.

[11] Sato, M.; Yoneyama, K.; Ogata, Y. "Recognition of printed characters." T.R. Radio Wave Research Laboratory. vo. 7. no. 33. Nov. 1961. pp. 489-492.

[12] Gonzolez R.; Woods R; *Digital Image Processing* Addison-Wesley Purblishing Company, Inc. 1992. pp. 619-638.

[13] Mori, Shunji; Suen, Ching Y.; Yamamoto, Kazuhiko. "Historical Review of OCR Research and Development." PROCEEDINGS OF THE IEEE, VOL. 80, NO. 7. July 1992. p. 1032.

[14] Blum, Adam. Neural Networks in C++, An Object Oriented Framework for Building Connectionist Systems. WILEY. 1992.

[15] Nadal, Christine; Legault, Raymond; Suen, Ching Y. "Complementary Algorithms for the Recognition of Totally Unconstrained Handwritten Numerals". PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 443-449.

[16] Lam, L., and Suen, C.Y. "A Dynamic Shape Preserving Thinning Algorithm", in preparation.

[17] Impedovo, S.; Dimauro, G. "An Interactive System for the Selection of Handwritten Numeral Classes." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 563-566.

[18] Lecolinet, E.; Moreau, Jea-Vincent Moreau. "Off-Line Recognition of Handwritten Cursive Script for the Automatic Reading of City Names on Real Mail." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 674-676.

[19] Cun, Y. Le; Matan, O.; Boser, B.; Denker, J.S.; Henderson, D.; Howard, R.E.; Hubbard, W.; Jackel, L.D.; Baird, H.S. "Handwritten Zip Code Recogntion with Multilayer Networks." PROCEEDINGS OF THE IEEE Computer Vision and Pattern

Recognition Systems and Applications. June 1990. pp. 35-39.

[20] Bradford, Roger; Nartker, Thomas. "Error Correlation in Contemporary OCR Systems." PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION. 1991. pp. 516-524.

[21] Downton, A.C.; Tregidgo, R.W.S. "The use of a trie structured dictionary as a contextual aid to recognition of handwritten British postal addresses." PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION. 1991. pp. 542-550.

[22] d'Acierno, A.; De Stefano, C.; Vento, M. "A Structural Character Recognition Method Using Neural Networks." PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION. 1991. pp. 803-811.

[23] Takahashi, Hiroyasu. "Neural Net OCR Using Geometrical And Zonal-pattern Features." PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION. 1991. pp. 827-828.

[24] Gazula, Srinivas; Kabuka, Mansur R. "Design of Supervised Classifiers using Boolean Neural Networks." INTELLIGENT ENGINEERING SYSTEMS THROUGH ARTIFICIAL NEURAL NETWORKS, VOLUME 2. ASME PRESS. 1992. pp. 5-14.

[25] Fuller, Thomas H. Jr.; Kimura, Takayuki D. "Supervised Competitive Learning Part I: SCL with Backpropagation Networks." INTELLIGENT ENGINEERING SYSTEMS THROUGH ARTIFICIAL NEURAL NETWORKS, VOLUME 2. ASME PRESS. 1992. pp. 185-190.

[26] Graf, Hans P.; Jackel, Lawrence D.; Denker, John S. "Analog Electronic Neural Networks For Pattern Recognition Applications." Neural Networks Concepts, Applications, and Implementations. Volume I. PRENTICE HALL. 1991. pp. 155-171.

[27] KRYZYZAK, A.; DAI, W.; SUEN, C.Y. "Unconstrained Handwritten Character Classification Using Modified Backpropagation Model." Frontiers in Handwriting Recogntion. CENPARMI, Concordia University. 1990. pp. 155-166.

[28] Solla, Sara A.; Cun, Yann le. "Constrained Neural Networks For Pattern Recognition." Neural Networks Concepts, Applications, and Implementations. Volume IV. PRENTICE HALL. 1991. pp. 142-161.

[29] Ibid.

[30] STRINGA, L. "A Structural Approach to Automatic Primitive Extraction in Hand-Printed Character Recognition." Frontiers in Handwriting Recogntion. CENPARMI, Concordia University. 1990. pp. 65-71.

[31] WARD, Jean Renard. "One View of On-Going Problems in Handwriting Character Recognition." Frontiers in Handwriting Recogntion. CENPARMI, Concordia University. 1990. pp. 101-107.

[32] SRIHARI, S. N. "Reading Unconstrained Handwriting with Bounded Context." Frontiers in Handwriting Recogntion. CENPARMI, Concordia University. 1990. pp. 109-115.

[33] HULL, Jonathan J.; COMMIKE, Alan; HO, Tin-Kam. "Multiple Algorithms for Handwritten Character Recognition." Frontiers in Handwriting Recogntion. CENPARMI, Concordia University. 1990. pp. 117-130.

[34] SUZUKI, Toshihiro; MORI, Shunji. "A Thinning Method Based on Cell Structure." Frontiers in Handwriting Recogntion. CENPARMI, Concordia University. 1990. pp. 39-52.

[35] IMPEDOVO, S.; CASTELLANO, M.; PIRLO, G.; DIMAURO. "An Off-Line Writer Identification System Based on a Syntactic Approach." Frontiers in Handwriting Recogntion. CENPARMI, Concordia University. 1990. pp. 53-64.

[36] Vossepoel, Alber M.; Buys, Jan Paul; Koelewijn, Gert. "Skeletons from chain-coded

contours." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 70-73.

[37] Ilg, Markus. "Knowledge-Based Understanding of Road Maps and other Line Images" PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 282-284.

[38] Sipper, M. and Yeshurun Y. "Pattern Classification Using Teurons". PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 433-437.

[39] Fujisaki, T., Chefalas, T.E.; Kim, Tappert, C.C. "Online Recognizer For Runon Handprinted Characters." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 450-451.

[40] Kabir, E.; Downton, A.C.; and Birch, R. "Recognition and Verification of Postcodes In Handwritten and Handprinted Addresses." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 469-473.

[41] ZHANGE, Shusheng; TACONET, Bruno; FAURE, Alain. "A Complexity Measure based Algorithm for Multifont Chinese Character Recogntion." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 573-577.

[42] Pao, Derek; Li, H. F.; Jayakumar R. "Detecting Parametric Curves Using the Straight Line HOUGH Transfrom." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 620-625.

[43] Eckhardt, Ulrich; Maderlechner, Gerd. "A General Approach for Parametrizing the HOUGH Transform." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 626-630.

[44] AMMAR, MAAN. "Performance of Parametric and Reference Pattern Based Features in Static Signature Verification: A Comparitive Study." PROCEEDINGS OF

THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 646-648.

[45] Belkasim, S.O.; Shridhar, M.; Ahmadi, M. "Shape-Contour Recognition Using Moment Invariants." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 649-651.

[46] Hull, Jonathan j.; Sher, David B. "Quantifying the Unimportance of Prior Probabilities in a c Computer Vision Problem." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 662-664.

[47] Gowely, Khaled El; Dessouki, Ossama El; Nazif, Ahmed. "Multi-Phase Recognition of Multi-Font Photoscript Arabic Text." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. pp 700-702. June 1990. PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 837-841.

[48] Lam, Stephen W.; Wang, Dacheng; Srihari Sargur N. "Reading Newspaper Text." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 703-705.

[49] Wakahara, Toru. "Dot Image Matching Using Local Affine Transformation." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 837-841.

[50] Niblack, Wayne C.; Capson, David W.; Gibbons, Phillip B. "Generating Skeletons and Centerlines from the Medial Axis Transform." PROCEEDINGS OF THE IEEE Computer Vision and Pattern Recognition Systems and Applications. June 1990. pp. 881-885.

[51] Simon, J.C.; Zerhoumi, K. "Robust Description of a Line Image." PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION. 1991. pp. 3-14.

[52] Potier, Christine; Vercken, Christine. "Geometric Modeling of Digitized Curves."

PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION. 1991. pp. 152-160.

[53] NISHIDA, Hirobumi; MORI, Shunji. "An Approach to Automatic Construction of Structural Models for Character Recognition." PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION. 1991. pp. 231-241.

[54] Lee, Seong-Whan; Lam, Lousia; Suen, Ching Y. "Performance Evaluation of Skeletonization Algoritms for Document Image Processing." PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION. 1991. pp. 260-271.

[55] HÖNES, FRANK; HAFFNER, ERNST-GEORG; FEIN, FRANK; DENGEL, ADREAS. "A Hybrid Approach for Document Image Segmentation and Encoding." PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION. 1991. pp. 444-453.

[56] Cun, Y. Le; Boser, B.; Denker, J.S.; Henderson, D.; Howard, R.E.; Hubbard, W.; Jackel, L.D.; Baird, H.S. "Constrained Neural Network For Unconstrained Handwritten Recognition." Frontiers in Handwriting Recogntion. CENPARMI, Concordia University. 1990. pp. 145-154.

[57] G. & C. MERRIAM COMPANY. WEBSTERS New Collegiate Dictionary. G.&C. MERRIAM COMPANY. 1977.

[58] Minsky and Papert. Perceptrons . MIT Press. 1969. Expanded edition 1988.

[59] Hecht-Nielsen, Robert. "Neurocomputing: picking the human brain." IEEE Spectrum. March 1988. pp. 36-41.

[60] Blum, Adam. Neural Networks in C++, An Object Oriented Framework for Building Connectionist Systems. WILEY. 1992. pp. 55-61.

[61] NEURALWARE, INC. NEURAL COMPUTING. 1991. pp. NC89-109

[62] Rumelhart, McClelland, and the PDP Research Group. PARALLEL DISTRIBUTED PROCESSING. MIT Press. 1987.

[63] Stroustrup, B. J. The C++ Programming Language. Prentice-Hall. 1991.

[64] Poskanzer, J. PBMPLUS. 1989.

[65] Gonzolez R.; Woods R; *Digital Image Processing* Addison-Wesley Purblishing Company, Inc. 1992. pp. 574-577.

[66] JAIN, ANIL K. FUNDAMENTALS OF DIGITAL IMAGE PROCESSING, PRENTICE HALL. 1989. pp. 375-376.

[67] HOUGH, P.V.C. "Method and means for recognizing complex patterns." U.S. Patent 3,069,654. 1962.

[68] Baird, Henry S. Model-Based Image Matching Using Location. Massachusetts Institute of Technology. 1985. p. 11.

[69] BALLARD, DANA H. COMPUTER VISION. Prentice-Hall. 1982. pp. 123-131.

[70] Gonzolez R.; Woods R; *Digital Image Processing* Addison-Wesley Purblishing Company, Inc. 1992. pp. 432-535.

[71] Grimson, William Eric Liefur. OBJECT RECOGNITION BY COMPUTER, The Role of Geometric Constraints. Massachusetts Institute of Technology. 1990. pp. 42-44.

[72] Ibid.

[73] Everson, Phillip G. ALV Public Domain Image Processing Toolkit for Sun Workstations. Computer Science Dept. Bristol University United Kingdom. 1989.

[74] Blum, Adam. Neural Networks in C++, An Object Oriented Framework for Building Connectionist Systems. WILEY. 1992.