ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

FACULTY OF TECHNOLOGY

NEURAL NETWORK IMPLEMENTATION OF CHARACTER RECOGNITION

by

Tezazu Bireda

Addis Ababa

May 1998

NEURAL NETWORK IMPLEMENTATION OF CHARACTER RECOGNITION

A thesis presented to the School of Graduate Studies in partial fulfilment of the requirements of the degree of Master of Science in Electrical Engineering.

by

Tezazu Bireda

Addis Ababa

May 1998

ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

"Neural Network Implementation of Character Recognition"

By

Tezazu Bireda
Faculty of Technology

Approval by Board of Examiners

Dr. Eneyew Adugna Chairman Department Graduate Committee	Signature
Dr. Eneyew Adugna Advisor	Signature
Dr. Wolde-Ghiorgis W.M Examiner	Signature
Prof. G. Devarajan External Examiner	Signature

ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

"Neural Network Implementation of Character Recognition"

By

Tezazu Bireda
Faculty of Technology

Approval by Board of Examiners

Dr. Eneyew Adugna Chairman Department Graduate Committee	Signature
Dr. Eneyew Adugna Advisor	Signature
Dr. Wolde-Ghiorgis W.M Examiner	Signature
Prof. G. Devarajan External Examiner	Signature

ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

"Neural Network Implementation of Character Recognition"

By

Tezazu Bireda
Faculty of Technology

Approval by Board of Examiners

Dr. Eneyew Adugna	
Chairman Department Graduate Committee	Signature
Dr. Eneyew Adugna	
Advisor	Signature
Dr. Wolde-Ghiorgis W.M	
Examiner	Signature
Prof. G. Devarajan	
External Examiner	Signature

TABLE O F CONTENT

List of Figures and Tables

List of Figures

List Of Tables

ACKNOWLEDGEMENT

I am very grateful to my project advisor Dr. Eneyew Adugna who through discussions starting from the proposal and providing me very useful materials on neural networks he himself owns during which no valuable alternative materials were present at the start of the project neither in the AAU nor outside and hence this project would otherwise not have been possible, contributed to the successful completion of this thesis project.

I am also very grateful to the Alemaya University and DAAD for offering me the in-country scholarship which enabled me to successfully complete my M.Sc. study and hence this thesis.

I again deeply appreciate my advisor's concern and collaboration to make the DAAD office to cooperate in maintaining my scholarship grant fund whenever problems were encountered and without this, the obstacles that might follow would have made me quit the project. I also thank Dr. Haferkon on similar collaboration.

My special thanks are also due to Ato Mekonnen Abebe who gave me his valuable book on neural networks which further enhanced my insight to the field.

I would also like to thank Ato Assefa Dagne for providing me his BSc. project paper and informed me the location of other materials and at last but not least all, particularly the Electrical Engineering Department Staff, who supported me in many other ways and means.

Abstract

Artificial neural networks, as they are usually called, currently gained much popularity in the design of "*intelligent*" machines and in programs which are used for automatic pattern recognition as pattern classifiers. In contrast to symbolic-oriented methods in artificial intelligence (AI), artificial neural networks are computing systems that use mathematical algorithms and "*imitate*" the way the brain, the biological neural network, functions. They are made up of a number of simple, highly connected non linear processing elements and process information by their dynamic state response to external inputs. They are characterized by the ability to learn and generalize, massive parallelism which gives rise to greater speed on computers with parallel processors or on a dedicated analogue VLSI circuit chip, tolerance to significant erroneous data or network fault, and some models exhibit self organization in the learning phase giving optimum network architecture. Their greatest asset compared to other recognition methods, however, is their ability to learn and generalize.

In this paper a feed-forward Back Propagation Network (BPN) architecture, which is one of the several network architectures available, is implemented to recognize printed multifont alpha-numeric (English and Amharic or Geez) characters and its performance is investigated. The network model has three layers and is trained in a supervised training mode.

In the research, two independent sets of pattern classes of characters were formed each pattern class having four training and two testing sample character patterns. The first set deals with randomly selected pattern classes and the second set deals with very similar pattern classes. And in both sets of training and testing schemes, the relative recognition performance of the network is evaluated. The network recognition rate or performance, *only* for the test patterns, in percentage for the first set is about 85% and for the second set is about 67%. The overall recognition rate accounting tests with both the training and test patterns is 93% for the first set and 87% for the second.

When the test patterns from these two sets were corrupted with noise, the recognition performance of both sets degraded steadily.

Test was also made with tilted test patterns on the first set and the performance was unaffected up to a tilt angle of 4.4 degrees from the vertical.

A steady improvement in performance was observed as the dimension of the input pattern vectors, the number of training patterns in a pattern class, and network size were increased.

CHAPTER 1. INTRODUCTION

Pattern recognition concepts have become increasingly recognized as an important factor in the design of modern computerized information systems and interest in this area is still growing at a rapid rate, having been a subject of interdisciplinary study and research in such varied fields as engineering, computer science, information science, statistics, physics, chemistry, linguistics, psychology, biology, physiology and medicine. Each of these fields emphasizes certain aspects of the problem, ranging from the modelling of physiological processes to the development of analytical techniques for automatic decision making.

A pattern is the mathematical and/or geometrical description of any member of a category representing a pattern class. A pattern class is a class in which all its members have identical (mathematical and/or geometrical) properties or features. In practice when a set of patterns falling into disjoint pattern classes is available, it is desired to categorize these patterns into their respective classes through the use of some automatic device, or in a precise term, an automatic pattern recognition system .

Pattern recognition can thus simply be defined as the categorization of input data into identifiable classes via the extraction of significant features or attributes of the data from a background of irrelevant detail. And the basic functions of a pattern recognition system are to detect and extract common features from the patterns describing the objects that belong to the same pattern class, and to recognize this pattern in any new environment and classify it as a member of one of the pattern classes under consideration.

The classification capability, implementation simplicity, general-purposedness, and robustness of neural networks may thus be exploited in the vast application areas mentioned above.

An artificial neural network, ANN, is a computing system [10] that imitates intelligent behaviour in the categorization of pattern classes in automatic pattern recognition systems; it is made up of a number of simple non-linear, highly connected processing elements and processes information by its dynamic state response to external inputs.

In this paper a neural network implementation of a digital image pattern recognition system is designed to recognize printed two multifont alpha-numeric characters (Arabic numeral, English (Latin), and Amharic (Geez)).

The skeleton of the paper will now follow: The second chapter deals with the theoretical foundation of automatic pattern recognition. It discusses basic concepts of pattern recognition, what its fundamental problems are and how it is implemented to solve practical problems. It explains how each input pattern is represented, how pre-processing is done on pattern classes, and how decision functions used in pattern classification and identification of an automatic pattern recognition system behave. Pattern classifiers, such as distance functions, likelihood functions (Byesian classifier) and the trainable classifiers are also presented.

The third chapter deals with the historical and theoretical background and implementation details of ANN, which itself is a deterministic trainable pattern classifier, making emphasis to the BPN architecture since it is used in this thesis.

The fourth chapter presents the design, implementation, training and testing details of the BPN and the results of the project.

Chapter five treats the discussion, conclusion and recommendation part.

The appendixes A, B, C and D consist of the C++ codings of the training and testing pattern pre-processing module, the BPN simulator module, the Recognition or (Run/Test/) module, and a gaussian noise generating and pattern corrupting module respectively.

CHAPTER 2. PATTERN RECOGNITION FUNDAMENTALS

2.1 Introduction

2.1.1 Background

Recognition is regarded as a basic attribute of human beings, as well as other living organisms. A pattern is the description of an object. We are performing an act of recognition every instant of our waking lives. We can spot a friend in a crowd, recognize the voice of a known individual, read hand-writing and analyse fingerprints, distinguish smiles from gestures of anger. A human being is thus a very sophisticated information system, partly because he possesses a superior pattern recognition capability.

According to the nature of patterns to be recognized, we may divide our acts of recognition into two major types: ***the recognition of concrete items*** and ***the recognition of abstract items***. We recognize characters, pictures, music, and the objects around us. This may be referred to as **sensory recognition**, which includes ***visual*** and ***aural pattern recognition***. This recognition process involves the identification and classification of ***spatial*** and ***temporal patterns***. Examples of spatial patterns are characters, fingerprints, weather maps, physical objects and pictures. Temporal patterns include speech waveforms, electrocardiograms (ECG), target signatures, and time series. Some patterns may still be both spatial and temporal and an example is a moving picture on a TV screen and are usually treated in *dynamic scene analysis systems.*

On the other hand, we can recognize an old argument, or a solution to a problem, with our eyes and ears closed. This process involves the recognition of abstract items and can be termed **conceptual recognition**, in contrast to visual or aural recognition.

Recognition of concrete patterns by human beings may be considered as psycho-physiological problem which involves a relationship between a person and a physical stimulus. When a person perceives a pattern, he makes an inductive inference and associates this perception with some general concepts or clues which he has derived from his past experience. Human recognition is in reality a question of estimating the relative odds that the input data can be associated with one of a set of known statistical populations which depend on our past experience and which form the clues and the *a priori* information for recognition.

Thus, the problem of pattern recognition may be regarded as one of discriminating the input data, not between individual patterns but between populations, via the search for features or invariant attributes among members of the population.

The study of pattern recognition problems may be logically divided into two major categories:

i) The study of the pattern recognition capability of human beings and other living organisms.

ii) The development of theory and techniques for the design of devices capable of performing a given recognition task for a specific application, i.e. for machine perception or computer vision.

The first subject area is concerned with such disciplines as psychology, physiology, and biology. The second area deals primarily with engineering, computer, and information science. This paper is concerned with the computer, information science, and engineering aspects of the design of automatic pattern recognition systems.

2.1.2 Basic Terms & Definitions of Pattern Recognition [15]

A pattern is the mathematical and/or geometrical description of any member of a category representing a pattern class. A pattern class is a class in which all its members are supposed to have identical (mathematical and/or geometrical) properties. For instance if alphanumeric characters are considered, there are 62 pattern classes representing 26 upper-case letters, 26 lower-case letters, and 10 numerals and the different fonts and styles of a particular letter or numeral form patterns in that pattern class. In practice when a set of patterns falling into disjoint pattern classes is available, it is desired to categorize these patterns into their respective classes through the use of some automatic device, or in a precise term, an automatic pattern recognition system. Pattern recognition can thus simply be defined as the categorization of input data into identifiable classes via the extraction of significant features or attributes of the data from a background of irrelevant detail. And the basic functions of a pattern recognition system are to detect and extract common features from the patterns describing the objects that belong to the same pattern class, and to recognize this pattern in any new environment and classify it as a member of one of the pattern classes under consideration.

2.1.3 Pattern Recognition System Design Concepts and Methodologies [15]

The fundamental problems in an automatic pattern recognition system design generally involve several major areas. Among them, the most ones are: *input data representation, pattern pre-processing* , and *the determination of optimum decision procedures*. The representation of input data involves the measuring, sensing or detection and digital encoding of the data which can be done from the patterns or objects to be recognized and is discussed in Sec 2.2. Any object or pattern which can be recognized and classified possesses a number of discriminatory attributes or features and pattern pre-processing deals with the selection and extraction of these features which is discussed in Sec. 2.3. The determination of optimum decision procedures, discussed in Sec 2.4 and Sec. 2.5, deal with the development of decision functions from sets of finite sample patterns of the pattern classes so that the functions will partition the measurement space in to regions each of which contains the sample pattern points belonging to one class and thus they are needed in the identification and classification process of pattern recognition systems.

Design Concepts

The design concepts for automatic pattern recognition are motivated by the ways in which pattern classes are characterized and defined. The three most basic possibilities of pattern class characterization design concepts are :

i) The Membership-Roster Concept

ii) The Common-Property Concept and

iii) The Clustering Concept

i) The Membership-Roster Concept: Characterization of a pattern class by a roster of its members suggests automatic pattern recognition by *template matching* . The set of patterns belonging to the same pattern class is stored in the pattern recognition system . When an unknown pattern is shown to the system, it is compared with the stored patterns one by one. The pattern recognition system classifies this input pattern as a member of a pattern class if it matches one of the stored patterns belonging to that pattern class. For instance, if letters of different fonts are stored in the pattern recognition system, such letters may be recognized by

the membership-roster approach as long as they are not distorted by noise due to smear, bad inking, porous paper, or the like. Clearly this is a simple-minded method. The membership-roster approach will work satisfactorily under the condition of nearly perfect pattern samples. However, this concept can lead to the design of inexpensive recognition schemes which serve the purpose in certain applications.

ii) The Common-Property Concept : Characterization of a pattern class by common properties shared by all of its members suggests an automatic pattern recognition via the detection and processing of similar features. The basic assumption in this method is that patterns belonging to the same class posses certain common properties or attributes which reflect similarities among these patterns. The common properties, for example, can be stored in the pattern recognition system. When an unknown pattern is observed by the system, its features are extracted and sometimes coded and then are compared with the stored features. The recognition scheme will classify the new pattern as belonging to the pattern class with similar features. Thus, the main problem in this approach is to determine common properties from a finite set of sample patterns known to belong to the pattern class to be recognized. It appears that this concept excels the membership-roster approach in many respects : the storage requirement for features of a pattern class is much less severe than that for all patterns in the class; since features of a pattern class are invariant, comparison of features allows variation in individual patterns as opposed to the template matching or membership-roster approach which does not tolerate significant pattern variations. If all the features of a class can be determined from sample patterns, the recognition process reduces simply to *feature matching* . However, it is extremely difficult, if not impossible , to find the complete set of discriminating features for a pattern class and in usual practice, restrictions in time, space, and cost dictate the development of realistic approaches. Utilization of this concept, therefore, often necessitates the development of feature selection techniques which are optimum in some sense as will be seen in sec. 2.3. The common-property concept is also fundamental in pattern recognition by means of formal language theory (syntactic pattern recognition).

iii) The Clustering Concept : When the patterns of a class are vectors whose components are real numbers, a pattern class can be characterized by its clustering properties in the pattern space. The design of a pattern recognition system based on this general concept is guided by

the relative geometrical arrangement of the various pattern clusters. If the classes are characterized by clusters which are far apart, simple recognition schemes such as the minimum-distance classifiers presented in section 2.5.1 may be successfully employed. When the clusters overlap, however, it becomes necessary to utilize more sophisticated techniques for partitioning the pattern space, such as the methods presented in sections 2.5.2 to 2.5.3.

Overlapping clusters are the result of a deficiency in observed information and the presence of measurement noise. Hence, the degree of overlapping can often be minimized by increasing the number and the quality of measurements performed on the patterns of a class.

Implementing Methodologies [15]

The basic design concepts for automatic pattern recognition described above may be implemented by three principal categories of methodology : *heuristic, mathematical, and syntactic (or linguistic)*. It is not uncommon to find a combination of these methods in a pattern recognition system.

i) Heuristic Methods : The heuristic approach relies too heavily on the past experience and intuition of the system designer, making use of the membership-roster and common-property concepts. A system designed using this approach generally consists of a set of *ad hoc* procedures developed for specialized recognition tasks. An example of this approach with the problem of character recognition is the classification of a character pattern based on the detection of features such as the number and sequence of particular strokes of the character.

Although the heuristic approach is an important branch of pattern recognition system design, little can be said about generalized principles in this area since each problem requires the application of specifically tailored design rules.

ii) The Mathematical Methods : The mathematical approach is based on classification rules which are formulated and derived in a mathematical framework, making use of the common-property and clustering concepts. This is in contrast with the heuristic approach, in which

decisions are based on *ad hoc* rules. The mathematical approach may be subdivided into two categories: *deterministic and statistical*.

The deterministic approach is based on a mathematical framework which does not employ explicitly the statistical properties of the pattern classes under consideration. Examples of the deterministic approach are the minimum distance classifiers presented in section 2.5.1. and the trainable or adaptive learning algorithms presented in section 2.5.3.1.

The statistical approach is based on mathematical classification rules which are formulated and derived in a statistical framework. The design of a statistical pattern classifier is generally based on the Bayes classification rule and its variations. This rule yields an optimum classifier when the probability density function of each pattern population and the probability of occurrence of each pattern class are known. Examples of this approach are the likelihood pattern classifiers presented in section 2.5.2. and the statistical trainable or adaptive pattern classifiers presented in section 2.5.3.2.

iii) The Syntactic (Linguistic) Methods : Characterization of patterns by primitive elements (subpatterns) and their relationships suggests automatic pattern recognition by the syntactic or linguistic approach making use of the common-property concept. A pattern can be described by a hierarchical structure of subpatterns analogous to the syntactic structure of languages. This permits application of formal language theory to the pattern recognition problem.

A pattern grammar is considered as consisting of finite sets of elements called variables, primitives, and productions. The rules of production determine the type of grammar. Among the most studied grammars are the regular grammars, context-free grammars, and context-sensitive grammars. The essence of this approach lies in the selection of pattern primitives, the assembling of the primitives and their relationships into pattern grammar, and analysis and recognition in terms of these grammars. This approach is particularly useful in dealing with patterns which can not be conveniently described by numerical measurements or are so complex that local features can not be identified and global properties must be used.

Supervised and Unsupervised Pattern Recognition Modes [6], [8], [15]

Once a specific design method has been selected, one is still faced with the actual design and implementation problem. In most cases, representative patterns from each class

under consideration are available. In these situations, ***supervised*** pattern recognition techniques are applicable. In a supervised learning environment, the system is "*taught*" to recognize patterns by means of various adaptive schemes. The essentials of this approach are a set of *training* patterns of known classification and the implementation of an appropriate learning procedure.

In some applications, only a set of training patterns of unknown classification may be available. In these situations, ***unsupervised*** pattern recognition techniques are applicable. As mentioned above, supervised pattern recognition is characterized by the fact that the correct classification of every training pattern is known. In the unsupervised case, however, one is faced with the problem of actually learning the pattern classes present in the given data. This problem is also known as ***learning without a teacher***.

It is important to keep clearly in mind that learning or training takes place only during the design (updating) phase of a pattern recognition system. Once acceptable results have been obtained with the training set of patterns, the system is applied to the task of actually performing recognition on samples drawn from the environment in which it is expected to operate. Of course, the quality of the recognition performance will be largely determined by how closely the training patterns resemble the actual data with which the system will be confronted during normal operation.

2.1.4 Functional Block Diagram of a Pattern Recognition System [15]

The main functional components of a pattern recognition system are a data

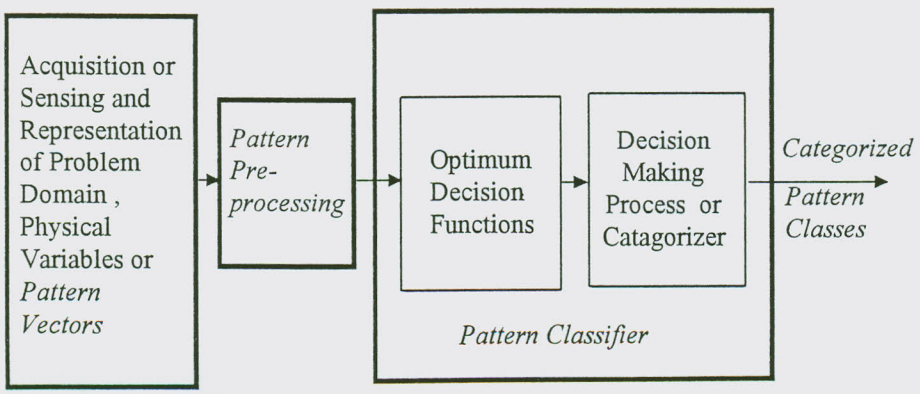


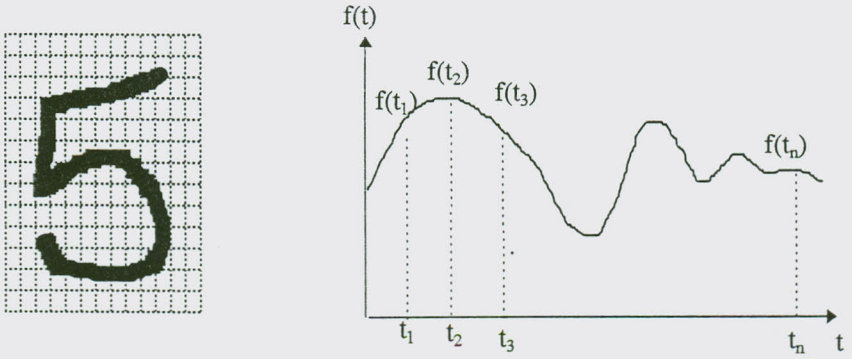
Figure 2.1 Functional Block Diagram of an Automatic Pattern Recognition System

acquisition or pattern input part, a pattern pre-processor part and a pattern classifier part. The data acquisition part senses or detects the problem domain and represents the associated physical variables or *pattern vectors* suitable for computer processing, such as digitaization. The pattern pre-processing part does feature extraction and dimensionality reduction of the input pattern vectors. The patterns are then classified by the pattern classifier part which has two functional parts, optimum decision functions and the associated categorizer. The optimum decision functions in the pattern classifier may be generated using either the minimum distance functions, the likelihood functions or the trainable or adaptive pattern classifiers.

In this paper the adaptive or iterative classifiers are used to do so and the complete block diagram representation is given in Chapter 4, Fig.4.1.

2.2. Input Data Representation [15]

This is the measuring, sensing or detection and encoding of the input data which can be done from the patterns or objects to be recognized. Each measured quantity describes a characteristic of the pattern or object. Suppose, for example, that the patterns in question are alphanumeric characters (which are spatial in nature). In this case, a grid measuring scheme



$\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ where $x_i = 0$ or 1

(a) Alphanumeric character

$\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ where $x_i = f(t_i)$

(b) A continuous function

Figure 2.2. Two simple schemes for the generation of pattern vectors

such as the one in Fig. 2.2(a) can be effectively used in the sensor. If we assume that the grid has n elements, the measurements can be arranged in the form of a *measurement* or *pattern*

vector: $\mathbf{x} = (x_1, x_2, \ldots, x_n)^t$ where each element x_i is, for example, assigned the value 1 if the *i*th cell contains a portion of the character, and is assigned the value 0 otherwise.

A second example is shown in fig 2.2(b) in which the patterns are continuous functions (such as acoustic signals) of a variable time *t*, i.e. temporal in nature. If these functions are sampled at discrete points $t_1, t_2, \ldots, t_n$, a pattern vector may be formed by letting $x_1 = f(t_1)$, $x_2 = f(t_2)$, $\ldots$, $x_n = f(t_n)$.

The pattern vectors contain all the measured information available about the patterns. The measurements performed on the objects of a pattern class may be regarded as a coding process which consists of assigning to each pattern a characteristic symbol from the alphabet set $\{x_i\}$. When the measurements yield information in the form of *real numbers*, it is often useful to think of a pattern vector as a point in an n-dimensional Euclidean space.

The set of patterns belonging to the same class corresponds to an ensemble of points scattered within some region of the measurement space. In practical situations, however, one is not always able to specify measurements that will result in neatly disjoint sets without slight overlapping of classes.

2.2.1 Computer Representation of Digital Images [2], [9], 13]

In this paper since the patterns used are scanned digital character images, few will be said about the representation and manipulation of digital images.

The term image refers to a two dimensional light intensity function, say, $f(x,y)$, where x and y denote spatial co-ordinate and the value of f at any point (x,y) is proportional to the brightness (or grey level) of the image at that point.

A digital image is an image that has been discretized or quantized both in spatial co-ordinates and brightness and thus can be considered as a matrix $\mathbf{f_D} = (f_{ij})_{n\times m}$ whose row and column indices i and j identify a spatial point in the image and the value of the corresponding matrix element f_{ij} identifies the brightness level at that point. The element of such a digital array are called *image elements, picture elements, pixels* or *pels*.

Digital representation of images in computers usually requires a very large number of bits. In many applications, it is important to consider the techniques for representing an image, or information, contained in the image, with fewer bits. In the terminology of information theory, this is referred to as *source encoding*. Applications of source encoding in

the field of image processing generally fall into one of three categories: *image data compression*, *image transmission* and *feature extraction*.

Digital Image File Formats [2],[13]

A number of bitmapped digital image file encoding or compressing formats have been developed with the aim for flexible and powerful digital image representation, manipulations or processing as well as efficient computer system resource utilization, particularly computer memory and speed. The digital image file formats include the Macpaint file format (by Apple Macintosh); the Graphics Interchange Format, GIF, (by Compuserve); the GEM/IMG file format (associated with Digital Research); the Tagged Image File Format, TIFF; the Joint Photographic Experts Group format, JPEG, and the PCX file format (by PC Paintbrush).

A bit mapped image can be anything that can be displayed in the graphics mode of a display card. For the sake of initial discussion, let's assume that the image file under discussion will store a picture like the size of the picture when it is in the graphics mode. All the data in the encoded image file will be of three types : *key bytes*, *index bytes*, and *data*.

(i) *key bytes* : These tell the decoding what to do with the next packet of data it uses.

(ii) *index bytes* : These tell the decoder how much of the rest of the file should be included in the current packet.

(iii) *data* : This is what will be decoded to make up the stored picture.

A packet is a collection of bytes of information that are all dealt in the same way. A packet can be as long as a scan line, or it can be quite a lot shorter if the scan line in question lend itself to being encoded more effectively by using a combination of several different types of packets. We have two types of packets, *a run of byte packet* and *a string packet*.

The run of byte packet is three bytes long. The first byte will be 0xff (*note that this is a C++ syntax in which the first two letters 0x or 0X tell the number is hexadecimal and the last two letters represent hexadecimal digits*), telling the decoding routine that it is looking at a run of byte packets. The second byte will be a number between 1 and 80, to indicate how many bytes on the screen are involved. Obviously a zero length packet would be of no use. The third byte is the actual data byte. If the decoder routine encountered the following packet

: 0xff 0x09 0x55 , it would write 9 bytes of 0x55 to the screen. The value 0x55 is useful as screen data in image files, as has every odd numbered bit set and thus paints a 50% grey area on any part of the screen filled with it.

The string packet deals with screen areas that can not be compressed into run of byte packets. This packet can be from 3 to 82 bytes long. The first byte is 0x00, indicating that the packet is a string packet. The second byte contains the number of subsequent bytes in the packet, and therefore to be written in the screen. The rest of the packet contains data that could not be compressed, and is to be written on the screen "as is". If the decoder encountered the packet : 0x00 0x09 0x65 0x12 0xa6 0x77 0x01 0x76 0x69 0x98, it would copy 9 bytes from 0x65 through 0x98 to the screen.

2.2.2 The PCX Image File Format [2], [13]

In this paper the PCX image file format is used in the scanning and storage, processing and preparation of character pattern data as an input for the neural network designed and thus is discussed in some detail.

The PCX format uses run length encoding to compress its image data. Its encoding is, in fact, the least efficient of the other formats in terms of compression. All PCX files carry around a 128-byte *program header* that defines parameters such as the size of the image, bits per pixel, the colour palette if the image is not monochrome and several other bits of information that the decoder or an uncompressor program can identify as being its own signature.

The general structure of a PCX image file header in a C++ syntax or representation is shown below and its description follows.

```
typedef struct {
    char   manufacturer;      // Always 0xa0
    char   version;           // Version number
    char   encoding;          // Always 1
    char   bits-per-pixel;    // Colour bits
    int    Xmin,Ymin;         // Image orgin
    int    Xmax,Ymax;         // Image Dimension[Lower corner]
    int    hres;              // Horizontal Resolution
    int    vres;              // Vertical Resolution
    char   palette[48];       // Colour pallet
```

```
    char    reserved;
    char    color_planes;       // Colour planes
    int     bytes_per_line;     // Line buffer size
    int     pallete_type;       // Grey or colour palette
  [byte]char  filler[58];
} PCXHEAD;
```

It is pretty easy to read the first 128 bytes of a PCX image file into a PCXHEAD structure from C++. One can just load them into a buffer and cast a PCXHEAD pointer to point to the buffer, or one can do this:

```
PCXHEAD   h;  //declare a variable h with PCXHEAD data structure
if ((fread((char*)&h,1,sizeof(PCXHEAD),fp)= =sizeof(PCXHEAD))
  {  // Some code goes here  }
else puts ("Error reading header");
```

This assumes that the PCX file whose header is being read has been opened for reading with file pointer variable fp.

The definition and use of the information mentioned in the header will now be explained although some are irrelevant to monochrome files.

The *manufacturer* byte of the PCX file header always will be 0xa0. This is really the only check that the format provides for software that reads a PCX file. If the first isn't 0xa0, the file is not a PCX file.

The *version* tells which version of PC Paintbrush created the file.

The *encoding* byte should always contain the value 1 and this indicates the file has been compressed using the PCX run length encoding scheme.

The *bits_per_pixel* value has to do with colour images.

Xmin, Xmax, Ymin, and *Ymax* values define the dimensions of the image file in question.

The *hres* and *vres* member define the resolution of the device that created the image.

The *palette* buffer contains the colour palette if the image has 16 or fewer colours. It is 48 bytes long.

The *color_planes* value also has to do with colour image files.

The *bytes_per_line* is extremely useful because it obviates the need to change the pixel to byte to find out how many bytes are required to contain each line of image data in the file.

The *palette_type* filled is fairly new one - its only significance is with the advent of VGA cards, which can do meaningful grey scales. This field will contain one byte for grey scales and two bytes for full colour.

Unpacking The Image Data For Image Manipulation

The first byte immediately after the header of a PCX file is the beginning of the compressed image data. Fig. 2.3 illustrates the basic decoding procedure for a PCX file. If the

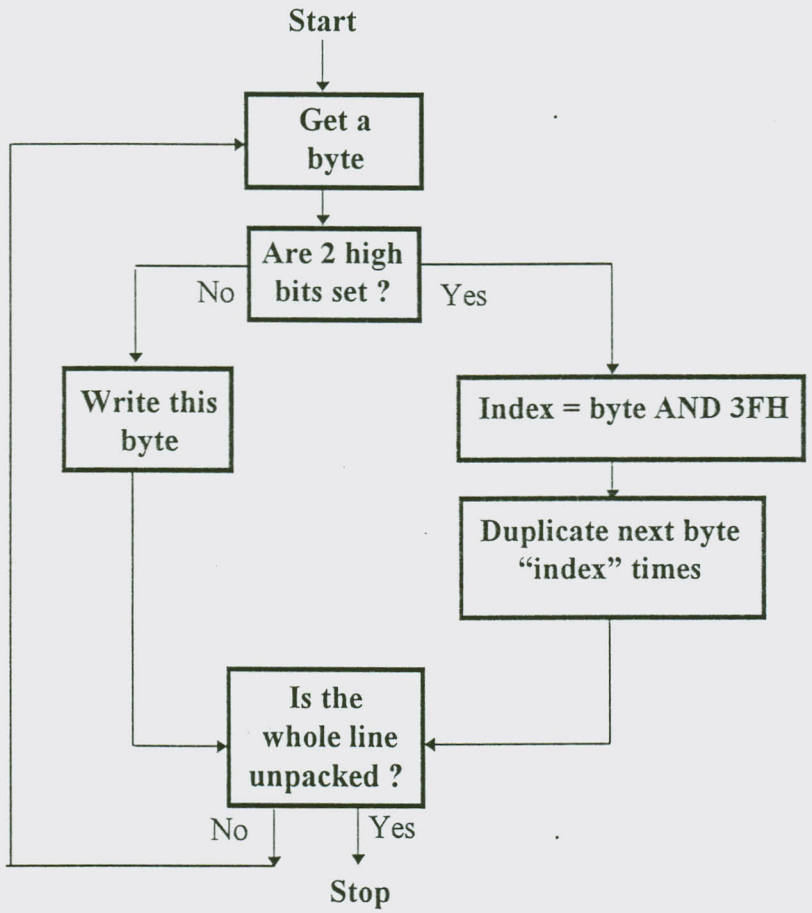


Figure 2.3 Flow Chart Illustrating The Basic Decoding Procedure Of A Line Of A PCX Image File.

upper two bits of a key byte are set, then the index of the current image packet can be found in the lower six bits. The next byte is specified in the index. Because the index can only use six bits, a run of bytes field in a PCX file can be only a maximum of 63 bytes long. Longer

fields would require two packets. If the upper two bits are not set, the bytes are written to the image "as is."

The above flow chart shows the procedure for unpacking a single line of the PCX file. This will be done by a simple program to read and decode a PCX line into a buffer variable p. The program function for doing this is ***ReadPcxLine()*** which is part of the ***Smplimg()*** program module which is used to open, unpack, sample and store PCX image files and is obtained in appendix A.

2.3. Pattern Pre-processing [15]

The extraction of characteristic features or attributes from the received input data and the data dimensionality reduction problems, usually termed as *pre-processing,* have been recognized as important problems in the design of pattern recognition systems.

Any object or pattern which can be recognized and classified possesses a number of discriminatory attributes or features. The first step in any recognition process, performed either by a human being or by a machine, is to consider the problem of what discriminatory features to select and how to extract (measure) theses features. In speech recognition, for example, we may discriminate vowel-like sounds from fricative and certain other consonants by measuring the distribution of energy over frequency in the spectra.

The number of features [15] needed to successfully perform a given recognition task depends on the discriminatory qualities of the chosen features. For instance, the commonly used features used for speech recognition are the duration of sound, the ratios of energy in various frequency bands, the location of spectral peaks, or formants, and the movement of these peaks in time. However, the problem of feature selection is usually complicated by the fact that the most important features are not necessarily easily measurable, or, in many cases, their measurement is inhibited by economic considerations. For instance, because of the high cost of drilling for oil prospecting, oil industry scientists and engineers must settle for other features which, though conveying less information, are more economical to obtain, say, from seismic signal processing. Unfortunately, this trade-off between feature selection and classification performance is a constraint present in most pattern recognition problems of practical significance. In fact, the selection of an appropriate set of features which take into account the difficulties present in the extraction or selection process, and at the same time

result in acceptable performance, is one of the most difficult tasks in the design of pattern recognition systems.

The features or the characterizing attributes of a pattern class common to all patterns belonging to that class are often referred to as ***intraset features*** and the features which represent the differences between pattern classes may be referred to as ***interset features*** .

The elements of intraset features which are common to all pattern classes under consideration carry no discriminatory information and can be ignored. If a complete set of discriminatory features for each pattern class can be determined from the measured data, the recognition and classification of patterns will present little difficulty and an automatic recognition may be reduced to a simple matching process or a table look-up scheme. However, in most pattern recognition problems which arise in practice, the determination of a complete set of discriminatory features is extremely difficult, if not impossible. Fortunately, we can often find some of the discriminatory features from the observed data. These features may be used to advantage in the simplification of the automatic recognition process. For instance, we may reduce the dimensionality of the measurement vector through a transformation, with minimum loss of information.

To facilitate the analysis of this problem, features will be classified into three categories: (i) physical features, (ii) structural features, and (iii) mathematical features.

Physical And Structural Features : Physical and structural features are commonly used by human beings in the recognition of patterns because these features are easily detected by the eye, by touch or other sensory organs. Although the distinction between physical and structural features are arbitrary, for convenience we give, say, examples of physical features as colour and fragrance and that of structural features as shape, texture, and other geometrical properties. These features are also used in automatic pattern recognition systems, primarily in digital image processing and computer vision, although they are strongly problem oriented in the sense that their use involves the development of special algorithms which fit the problem at hand and it is almost impossible to formulate general guidelines regarding the selection of physical and structural features. For instance, the selection of physical features (e.g. colour) would be meaningful to identify crops by means of aerial photography and structural features analysis is used for the identification of objects such as trucks, buildings, and highways.

Mathematical Features : Machines can be designed to extract mathematical features of patterns which human beings may have some difficulty in determining without mechanical aid. These features are more general in scope and lend themselves readily for machine implementation. Examples of such features are statistical means, correlation coefficients, eigen values and eigen vectors of covariance matrices, and other invariant properties.

Once a set of attributes has been selected, the extraction process consists of simply of extracting these attributes from the pattern classes under consideration reducing the dimensionality of the pattern vectors by means of a linear transformation. In this paper mathematical features are emphasized.

Clustering Transformation and Feature Ordering

Pattern pre-processing generally involves two major tasks : *clustering transformation* and *feature selection.* Clustering transformation is made on the measurement space in order to cluster the points representing samples of the class. Such transformation will maximize the *interset distance*, (the mean-square distance between pattern points of two different classes), while minimizing *the intraset distance*, (the mean-square distance between pattern points of the same class).

Feature selection independent of classification performance is dictated by the optimization of a criterion function giving absolute features. While in performance-dependent feature selection, the effectiveness of the feature is directly related to the performance of the classification system, usually in terms of the probability of correct recognition.

The measurements of a pattern which are represented by the different co-ordinate axes x_k are not all equally important in influencing the definition of the category to which similar patterns belong. In comparing two patterns feature by feature, measurements with decreasing significance should be assigned decreasing weights. The process of feature weighing may be realized through a linear transformation, which will cluster most highly the transformed pattern points in the new space.

Consider the pattern vectors **a** and **b**, which are transformed to pattern vectors **a*** and **b*** through a transformation **W**. Then

$$\mathbf{a}^* = \mathbf{W}\,\mathbf{a} \quad \text{and} \quad \mathbf{b}^* = \mathbf{W}\,\mathbf{b}$$

where

$$\mathbf{W} = (w_{kj})_{n\times n}$$

in which w_{kj} are the weighing coefficients.

Thus, we have

$$a_k^* - b_k^* = \sum_{j=1}^{n} w_{kj}(a_j - b_j)$$

Each element of the transformed pattern vector is a linear combination of the elements of the original pattern vector. The Euclidean distance between **a*** and **b*** in the new space is then given by

$$D(\mathbf{a}^*, \mathbf{b}^*) = \sqrt{\sum_{k=1}^{n}(a_k^* - b_k^*)^2}$$

$$= \sqrt{\sum_{k=1}^{n}\left[\sum_{j=1}^{n} w_{kj}(a_j - b_j)\right]^2} \qquad \text{.....(2.1)}$$

When the linear transformation involves only scale-factor changes of the co-ordinates, we may let **W** be a diagonal matrix with only the elements on the main diagonal nonzero. Thus the Euclidean distance reduces to

$$D(\mathbf{a}^*, \mathbf{b}^*) = \sqrt{\sum_{k=1}^{n} w_{kk}^2 (a_k - b_k)^2} \qquad \text{.....(2.2)}$$

where w_{kk} represent the feature-weighing coefficients.

The clustering transformation problem is to determine the coefficients w_{kk} so that the intraset distance between { $\mathbf{a}^i$, i=1,2,...,k} and { $\mathbf{a}^j$, j=1,2,...,k} is minimized, subject to a specified constraint on w_{kk}. It can be shown that the intraset distance for pattern points in the new space is

$$\bar{D}^2 = 2\sqrt{\sum_{k=1}^{n}(w_{kk}\sigma_k)^2} \qquad \text{.....(2.3)}$$

where σ_k^2 is the unbiased sample variance of the components along the x_k co-ordinate direction. In carrying out the minimization procedure, we will consider two cases.

Case 1. Constraint $\sum_{k=1}^{n} w_{kk} = 1$.

Minimizing $\bar{D}^2$ subject to this constraint is equivalent to minimizing

$$S_1 = 2\sum_{k=1}^{n} (w_{kk}\sigma_k)^2 - \rho_1(\sum_{k=1}^{n} w_{kk} - 1) \quad \text{.....(2.4)}$$

Taking the partial derivative of Eq. (2.4) with respect to w_{kk} and equating it to zero yields, upon simplification,

$$w_{kk} = \frac{\rho_1}{4\sigma_k^2} \quad \text{.....(2.5)}$$

where ρ_1 is the Lagrange multiplier, given by

$$\rho_1 = \frac{4}{\sum_{k=1}^{n} \sigma_k^{-2}} \quad \text{.....(2.6)}$$

Thus, the feature weighing coefficient is

$$w_{kk} = \frac{1}{\sigma_k^2 \sum_{k=1}^{n} (1/\sigma_k^2)} \quad \text{.....(2.7)}$$

From Eq. (2.5) it is noted that w_{kk} is small if σ_k^2 is large. This implies that in the distance measure a small weight is to be assigned to a feature of large variation. On the other hand , if σ_k^2 is small, the corresponding feature should be weighed heavily.

In the above analysis, the clustering transformation is accompanied by feature weighing. Intuitively, a small σ_k^2 implies that the *k*th measurements is more reliable; a large σ_k^2, that the *k*th measurement is less reliable. The more reliable measurements are more heavily weighted.

Case 2: Constraint : $\prod_{k=1}^{n} w_{kk} = 1$.

Minimization of $\bar{D}^2$ subject to this constraint is equivalent to minimization of

$$S_2 = 2\sum_{k=1}^{n} (w_{kk}\sigma_k)^2 - \rho_2(\prod_{k=1}^{n} w_{kk} - 1) \quad \text{.....(2.8)}$$

Taking partial derivatives of Eq. (2.8) with respect to w_{kk} and equating it to zero yields,

$$w_{kk} = \frac{\sqrt{\rho_2}}{2\sigma_k} \quad \text{.....(2.9)}$$

where the Lagrange multiplier is ρ_2 is given by

$$\rho_2 = 4(\prod_{k=1}^{n} \sigma_k)^{2/n} \quad \text{.....(2.10)}$$

Thus, the feature- weighing coefficient is

$$w_{kk} = \frac{1}{\sigma_k}\left(\prod_{j=1}^{n} \sigma_j\right)^{1/n} \quad \text{.....(2.11)}$$

which is inversely proportional to the standard deviation of the *k*th measurement.

Equations (2.7) and (2.11) determine the transformation matrix **W** under the constraints specified above.

If the pattern vectors are transformed from space X to space X^* by the transformation

$$\mathbf{x}^* = \mathbf{W}\,\mathbf{x} \quad \text{..... (2.12)}$$

the intraset distance in space X* is minimized.

Feature Selection

There are numerous procedures used in mathematical feature selection and extraction and although most of these methods cover a broad class of problems, the superiority of any one procedure is ultimately determined by the problem at hand. Thus continuous valued feature selection may be done through entropy minimization, orthogonal expansions, functional approximation, or divergence maximization; and for binary patterns, binary feature selection methods can be used.

In this paper since binary patterns are used, only binary feature selection is discussed.

Binary Feature Selection

This deals with the problem of binary features extraction and selection from patterns which are also binary, i.e. an array of 0 and 1 . The more useful aspects of binary feature selection do not deal with dimensionality reduction. Instead, the basic problem is to select a minimum set of binary features of the same dimensionality as the patterns which will be sufficient to reconstruct the original patterns, with the fewest possible errors. Two algorithms represent a reasonable (but not always completely successful) approach to generating useful binary features. The algorithms are *the sequential algorithm* and *the parallel algorithm* which attempt to determine a minimum set of features which are common to a group of patterns.

A Sequential Algorithm

The sequential algorithm generates one binary feature for each iteration through the given patterns. Basically, the procedure consists of establishing a variable threshold and altering the feature being generated during a given iteration whenever the threshold is exceeded.

This algorithm may be formalized as follows. For N binary patterns P_1, P_2, ..., P_N (since these patterns are considered as sets, they are represented by the symbol P_i rather than the familiar vector notation $\mathbf{x}_i$), the feature at the ith step in the kth iteration through the patterns is given by

$$f_k(i) = \begin{cases} f_k(i-1) \cap P_i & \text{if } \left\| f_k(i-1) \cap P_i \right\| \geq \theta^* \\ \{f_k(i-1) & \text{otherwise} \end{cases} \quad ,i = 1,2,\ldots,N \quad \text{.....(2.13)}$$

where

$$\theta^* = \theta + \left\| f_1(M) \cap [f_k(i-1) \cap P_i] \right\| + \left\| f_2(M) \cap [f_k(i-1) \cap P_i] \right\| + \ldots + \left\| f_{k-1}(M) \cap [f_k(i-1) \cap P_i] \right\| \quad \text{.....(2.14)}$$

where θ is an arbitrary initial threshold and θ^* is the updated threshold for the new iteration.

Two questions [15] to be answered in connection with this algorithm are how to select the threshold θ and how many features need to be generated. Unfortunately neither question can, at this time, be answered in general. Since the real problem centers about the selection of θ, however, it is often practical to repeat the procedure for several values of θ and to choose the threshold which yields the best results.

A Parallel Algorithm

In the parallel algorithm, instead of determining one feature per iteration, it is possible to determine several features at the same time in a parallel manner. This algorithm to be presented below starts with a single feature as before, but it introduces new features whenever they are needed to reconstruct a pattern at a given step. After a pattern has been presented and changes in the features have been carried out, a test is made to see whether the union of the new features is sufficient to reconstruct the pattern under consideration. If it is, the next

pattern is presented. If it is not, a new feature, identically equal to the pattern being considered, is created, and then the next pattern is presented.

Consider N binary patterns $P_1, P_2, \ldots, P_N$ and assume that in the ith step of iteration through the patterns j features, $f_1(i), f_2(i), \ldots, f_j(i)$, are being considered. When pattern P_{i+1} is presented, the new features are determined as follows:

1. $f_1(i+1) = f_1(i) \cap P_{i+1}$ if $\| f_1(i) \cap P_{i+1} \| \geq \theta$. Otherwise $f_1(i+1) = f_1(i)$.

2. $f_2(i+1) = f_2(i) \cap P_{i+1}$ if $\| f_2(i) \cap P_{i+1} \| \geq \theta + \theta_2$. Otherwise $f_2(i+1) = f_2(i)$.

The parameter θ_2 is given by $\theta_2 = \| f_1(i+1) \|$ if

(a) $\| f_1(i) \cap P_{i+1} \| \geq \theta$, *and*

(b) $f_1(i+1)$ is contained in $f_2(i)$, that is, $f_1(i+1) \subseteq f_2(i)$. Otherwise $\theta_2 = 0$.

3. In general, $f_l(i+1) = f_l(i) \cap P_{i+1}$ if $\| f_l(i) \cap P_{i+1} \| \geq \theta + \theta_1$. Otherwise $f_l(i+1) = f_l(i)$.

The parameter θ_l is given by $\theta_l = \sum_k \| f_k(i+1) \|$, (for $k < l$), where a term $\| f_k(i+1) \|$ is included in the summation only if (a) $\| f_k(i) \cap P_{i+1} \| \geq \theta + \theta_k$, and (b) $f_k(i+1) \subseteq f_l(i)$.

After the j new features have been computed, the union of the *features which were changed* is formed. If this union yields the pattern P_{i+1}, the next pattern is presented. If it does not, a new feature, $f_{j+1} = P_{i+1}$, is created and then the next pattern is presented.

The parallel algorithm, which is somewhat more complicated than its sequential counterpart, yields a set of features in a fewer number of iterations. This speed is gained by creating new features before old ones have been completely determined. This complicates the threshold-raising mechanism, however, and frequently leads to the determination of more features than are needed for reconstruction.

As was the case with the sequential algorithm, the only way to obtain an acceptable set of features is, in general, by running the algorithm with various values of θ and choosing the best results.

2.4. Decision Functions [15]

2.4.1 Introduction

A central problem in pattern recognition systems is the development of decision functions from sets of finite sample patterns of the pattern classes so that the functions will partition the measurement space into regions each of which contains the sample pattern points belonging to one class. That is, decision functions are needed in the identification and classification process of pattern recognition systems and they are treated here because the aim of any pattern classifier, discussed in the next sections, is to give an optimum decision function.

Decision functions can be generated in a variety of ways. When complete a priori knowledge about the patterns to be recognized is available, the decision functions may be determined with precision on the basis of this information. When only qualitative knowledge about the patterns is available, reasonable guesses of the forms of the decision functions can be made. In this case the decision boundaries may be far from correct, and it is necessary to design the machine to achieve satisfactory performance through a sequence of adjustments.

The more general situation [15] is that there exists little, if any, a priori knowledge about the patterns to be recognized. Under these circumstances pattern recognizing machines are best designed using *a training or learning procedure* discussed in Sec.2.5 and Chapter 3 in which arbitrary decision functions are initially assumed, and through a sequence of iterative training steps the decision functions are made to approach optimum or satisfactory forms.

Formulation of Decision Functions

After the observed data from patterns to be recognized have been expressed in the form of pattern points or measurement vectors in the pattern space, we want the machine to decide to which pattern class these data belong. Assume that the machine is to be designed to recognize M different pattern classes, denoted by $\omega_1, \omega_2, \ldots, \omega_M$. Then the pattern space can be considered as consisting of M regions, each of which encloses the pattern points of a class. The recognition problem can now be viewed as that of generating the decision boundaries which separate the M pattern classes on the basis of the observed measurement vectors. Let

the decision boundaries be defined, for example, by decision functions, $d_1(\mathbf{x})$, $d_2(\mathbf{x})$,..., $d_M(\mathbf{x})$. These functions, which are also called *discriminant* functions, are scalar and single valued functions of the pattern $\mathbf{x}$. If $d_i(\mathbf{x}) > d_j(\mathbf{x})$ for i, j = 1,2, . . . ,M, and $j \neq i$, the pattern belongs to pattern class ω_i. In other words, if the *i*th decision function, $d_i(\mathbf{x})$, has the largest value for a pattern $\mathbf{x}$, then $\mathbf{x} \in \omega_i$.

The success of the foregoing pattern classification scheme depends on two factors : (i) the form of $d(\mathbf{x})$ and (ii) one's ability to determine its coefficients. The first problem is directly related to the geometrical properties of the pattern classes under consideration. If the dimensionality of the patterns is higher than three, our powers of visualization are no longer of assistance in determining the geometrical class boundaries. Under these conditions, the only reasonable recourse is a strictly analytical approach. Unfortunately, unless some *a priori* information is available, the only way to establish the effectiveness of a chosen decision function is by *direct trial.* As will be seen in subsequent sections, adaptive and training schemes can bear on this problem.

Once certain decision functions are selected, the problem becomes the determination of the coefficients, normally determined by using the available sample patterns. If the pattern classes under consideration are separable by the specified decision functions, it is possible to utilize sample patterns to determine the coefficients which characterize these functions.

In this section we shall see two types of decision functions, Linear Decision Functions and Generalized Decision Functions. They give us a good theoretical insight as to how pattern classifiers classify the decision boundaries of pattern classes and the technical constraints, if any, in them in doing so.

2.4.2 Linear and Generalized Decision Functions

Linear Decision Functions

A general linear decision function used to separate or classify classes of an n-dimensional vectors is of the form

$$\begin{aligned} d(\mathbf{x}) &= w_1.x_1 + w_2.x_2 + \ldots + w_n.x_n + w_{n+1} \\ &= \mathbf{w_o}^T \mathbf{x} + w_{n+1} \qquad \ldots\ldots (2.15) \end{aligned}$$

where $\mathbf{w}_o = (w_1, w_2, \ldots, w_n)^T$. This vector is referred to as the *weight* or *parameter* vector. It is a widely accepted convention to append a 1 after the last component of *all* pattern vectors and express Eq. (2.15) in the form

$$d(\mathbf{x}) = \mathbf{w}^T \mathbf{x} \qquad \text{.....(2.16)}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n, 1)^T$ and $\mathbf{w} = (w_1, w_2, \ldots, w_n, w_{n+1})^T$ are called the *augmented* pattern and weight vectors or simply pattern and weight vectors, respectively. Since the same quantity is equally appended to all patterns, the basic geometrical properties of the pattern classes are not disturbed.

For instance, in a two class case of classes ω_1 and ω_2, the decision function will be assumed to have the property

$$d(\mathbf{x}) = \mathbf{w}^T \mathbf{x} \begin{cases} > 0 & if\ \mathbf{x} \in \omega_1 \\ < 0 & if\ \mathbf{x} \in \omega_2 \end{cases} \qquad \text{.....(2.17)}$$

When we have more than two classes , denoted by $\omega_1, \omega_2, \ldots, \omega_M$, we consider the following multiclass cases.

Case 1: Each class is separable from the other classes by a single decision surface. In this case there are M decision functions with the property

$$d_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} \begin{cases} > 0 & if\ \mathbf{x} \in \omega_i \\ < 0 & otherwise \end{cases}, i = 1,2,\ldots,M \qquad \text{.....(2.18)}$$

where $\mathbf{w}_i = (w_{i1}, w_{i2}, \ldots, w_{in}, w_{i,n+1})'$ is the weight vector associated with the ith decision function.

Case 2: Each pattern class is separable from every other individual class by a distinct decision surface, that is, the classes are pairwise separable. In this case there are $M(M-1)/2$ (the combination of two M classes taken two at a time) decision surfaces. The decision functions here are of the form $d_{ij}(\mathbf{x}) = \mathbf{w}_{ij}^T \mathbf{x}$ and have the property that, if $\mathbf{x}$ belongs to class ω_i, then

$$d_{ij}(\mathbf{x}) > 0 \quad \text{for all } j \neq i \qquad \text{.....(2.19)}$$

These functions also have the property that $d_{ij}(\mathbf{x}) = - d_{ji}(\mathbf{x})$.

It is not uncommon to find problems involving a combination of Cases 1 and 2. These situations require fewer than the $M(M - 1)/2$ decision surfaces which would be needed if all the classes were only pairwise separable.

Case 3: There exist M decision functions $d_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x}$, $k = 1,2,.. M$, with the property that, if $\mathbf{x}$ belongs to class ω_i,

$$d_i(\mathbf{x}) > d_j(\mathbf{x}) \qquad \text{for all } j \neq i \qquad \text{.....(2.20)}$$

This is a special instance of case 2 since we may define

$$\begin{aligned} d_{ij}(\mathbf{x}) &= d_i(\mathbf{x}) - d_j(\mathbf{x}) \\ &= (\mathbf{w}_i - \mathbf{w}_j)' \mathbf{x} \\ &= \mathbf{w}_{ij}^T \mathbf{x} \end{aligned} \qquad \text{.....(2.21)}$$

where $\mathbf{w}_{ij} = \mathbf{w}_i - \mathbf{w}_j$. It is easily verified that, if $d_i(\mathbf{x}) > d_j(\mathbf{x})$ for all $j \neq i$, then $d_{ij}(\mathbf{x}) > 0$ for all $j \neq i$, that is, if the classes are separable under Case 3 conditions, they are automatically separable under Case 2. The converse, however, is in general not true.

Generalized Decision Functions

Decision boundaries can always be established between pattern classes which do not share identical pattern vectors. The complexity of these boundaries may range from linear to very non-linear surfaces requiring a large number of terms for their description. Often in practical applications the pattern classes are not truly separable within economic or technical constraints, and it then becomes desirable to seek approximations to decision functions. One convenient way to generalize the linear decision function concept is to consider decision functions of the form

$$\begin{aligned} d(\mathbf{x}) &= w_1 f_1(\mathbf{x}) + w_2 f_2(\mathbf{x}) + \ldots + w_K f_K(\mathbf{x}) + w_{K+1} \\ &= \sum_{i=1}^{K+1} w_i f_i(\mathbf{x}) \end{aligned} \qquad \text{.....(2.22)}$$

where the $\{f_i(\mathbf{x})\}$, i = 1,2,...,K, are real, single valued functions of the pattern $\mathbf{x}$, $f_{K+1}(\mathbf{x}) = 1$, and $K+1$ is the number of terms used in the expansion. Equation (2.22) represents an infinite variety of decision functions, depending on the choice of the functions $\{f_i(\mathbf{x})\}$ and on the number of terms used in the expansion.

One of the most commonly used types of generalized decision functions is that in which the functions $\{f_i(\mathbf{x})\}$ are of polynomial form. In the simplest case, these functions are linear; that is, if $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ then $f_i(\mathbf{x}) = x_i$, with $K = n$. Under this condition we obtain $d(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_{n+1}$.

At the next level in complexity are the second-degree , or quadratic, functions. By considering all combinations of the components of **x** which form terms of degree two or less, that is, if the patterns are n-dimensional the general form is given as,

$$d(\mathbf{x}) = \sum_{j=1}^{n} w_{jj} x_j^2 + \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} w_{jk} x_j x_k + \sum_{j=1}^{n} w_j x_j + w_{n+1} \quad \text{.....(2.23)}$$

In this equation, the first function on the right hand side consists of n terms, the second function of $n(n - 1)/2$ terms, and the third function of n terms. Hence the total number of terms is $(n + 1)(n + 2)/2$, which is equal to the total number of parameters or weights to be determined.

With reference to Eq. (2.23), if we let

$w_{jj} = a_{jj}$, $w_{jk} = 2a_{jk}$, $w_j = b_j$ and $w_{n+1} = c$, $j, k = 1,2, \ldots, n; j \neq k$

then Eq. (2.23) may be expressed in a compact form as

$$d(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{x}^T\mathbf{b} + c \quad \text{.....(2.24)}$$

in which $\mathbf{A} = (a_{jk})_{n\times n}$ and $\mathbf{b} = (b_1, b_2, \ldots, b_n)^T$.

The properties of matrix **A** determine the shape of the decision boundary. When **A** is the identity matrix, the decision function describes a *hypersphere*. When **A** is positive definite, the decision function describes a *hyperellipsoid* with axes in the directions of the eigen vectors of **A**. When matrix **A** is positive semidefinite, the decision boundary is a *hyperellipsoidal cylinder*, the cross sections of which are lower-dimension hyperellipsoids with axes in the directions of the eigen vectors of **A**, corresponding to nonzero eigen values. When matrix **A** is negative definite, the decision boundary is *hyperhyperboloid.*

Although they will not be treated here, there are also other methods for generating decision functions such as the multivariate functions.

2.4.3 Pattern Space and Geometrical Property of Decision Functions Weight Solution

From the forms of the linear decision function formulation, **w** is the solution to the set of linear inequalities determined by all patterns of all classes in a *pattern space* which is the *n*-dimensional Euclidean space containing the pattern vectors in which the co-ordinate variables are $x_1, x_2, \ldots, x_n$. In this space, **w** is viewed as the set of coefficients which determine a decision surface. The *weight space* is the (n + 1)-dimensional Euclidean space in which the co-ordinate variables are $w_1, w_2, \ldots, w_{n+1}$. We call the weight solution region bounding surface a *convex polyhedral cone*. The total number of cones in addition to the solution cone (if it exists) depends on the number of patterns, N and pattern dimensionality n as will be seen in the next section.

Pattern Dichotomies

Two important geometrical properties of linear decision functions are hyperplane properties and pattern dichotomies. *One measure of the discriminatory power of decision functions* [15] *is the number of ways in which they can classify a given set of patterns.* Consider for example Fig. 2.4. which shows a

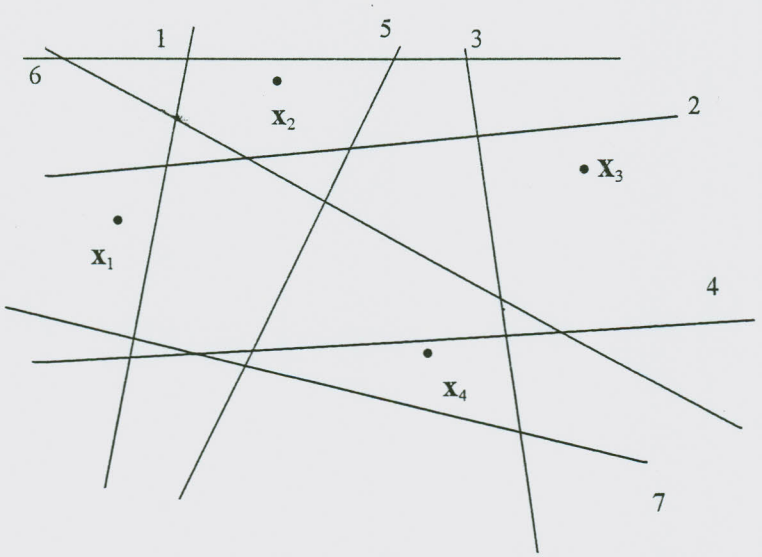


Figure 2.4. Linear dichotomies of 4 well-distributed patterns in two dimensions.

set of four two dimensional patterns, $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$, $\mathbf{x}_4$. Each line in the figure corresponds to a different classification of the patterns into two classes. For example, line 1 separates the group into pattern $\mathbf{x}_1$, and patterns $\mathbf{x}_2$, $\mathbf{x}_3$, $\mathbf{x}_4$. Since we can assign $\mathbf{x}_1$ to ω_1 or ω_2, we see that line 1 produces two possible classifications. In this case the total number of two-class groupings or *dichotomies* is 14. It is interesting to compare this number with the 2^4 ways in which we can group four patterns into two classes. Clearly two of these 16 dichotomies are not linearly implementable.

The number of linear dichotomies of N points (or patterns) in an n-dimensional Euclidean space is equal to twice the number of ways in which the points can be partitioned by an (n - 1)-dimensional hyperplane. It can be shown that, if the points are well distributed, the number of linear dichotomies for N n-dimensional patterns is given by

$$D(N,n) = \begin{cases} 2\sum_{k=0}^{n} C_k^{N-1}, & N > n+1 \\ 2^N, & N \le n+1 \end{cases} \qquad \text{.....(2.25)}$$

where $C_k^{N-1} = (N-1)!/(N-1-k)!k!$. A set of N points in an n-dimensional space is said to be *well distributed* if no subset of $n + 1$ points lies on an $(n - 1)$-dimensional hyperplane. For example, N points in two dimensions are well distributed if no three points lie on a line (or one-dimensional hyperplane). It can be noticed that for various combinations and a moderately increasing N and n, the values of D(N,n) grow dramatically.

It is interesting to associate Eq. (2.25) with the number of convex polyhedral cones in the weight space representation discussed earlier. Any vector **w** inside the convex cones corresponds to a unique classification of the given patterns. Since there are $D(N,n)$ linear dichotomies (assuming that the patterns are well distributed), we conclude that there must be an identical number of convex polyhedral cones in the weight space configuration of N n-dimensional patterns.

Since we are using the number of dichotomies as a measure of classification power, it should be evident that, the greater the number of implementable dichotomies for a given N, the better our chances are of finding a solution to the given inequalities. This, of course, agrees with the fact that the chances of dichotomizing two sets of patterns increase as the nonlinearity of the attempted decision boundary is increased.

Dichotomization Capacity of Generalized Decision Functions

Consider the generalized decision functions of Eq. (2.22), which are characterized by $K+1$ adjustable weights or parameters. Given N transformed, well-distributed patterns, there are 2^N dichotomies, $D(N,K)$ of which are linearly implementable with respect to the K-dimensional space of the transformed patterns. The probability $p_{N,K}$ that a dichotomy chosen at random will be linearly implementable is given by

$$p_{N,K} = \frac{D(N,K)}{2^N} = \begin{cases} 2^{1-N}\sum_{j=0}^{K} C_j^{N-1} & \text{for } N > K+1 \\ 1 & \text{for } N \le K+1 \end{cases} \quad \text{.....(2.26)}$$

In other words, if the number of patterns is less than or equal to $K + 1$, we are assured that, regardless of the way in which we group the given patterns, they will be linearly separable in the K-dimensional pattern space.

The probability $p_{N,K}$ possesses some additional interesting properties. In order to examine these properties it is convenient to let $N = \lambda\,(K + 1)$ and to plot $p_{N,K}$ verses λ. Clearly we can always select λ so that, whatever the value of K, $\lambda(K + 1)$ will equal N. The plot of $p_{\lambda(K+1),K}$ verses λ is shown in Fig 2.5.

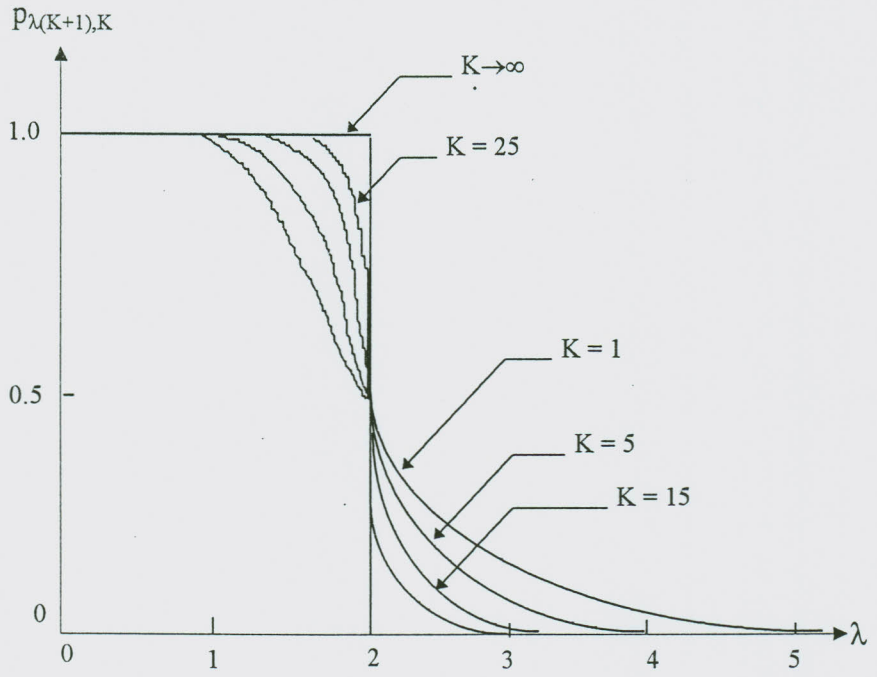


Figure 2.5. Plot of $p_{\lambda(K+1),K}$ versus λ for various values of K

Notice the marked threshold effect that occurs at $\lambda = 2$ for large values of K. We also note that, at this value of λ, $p_{2(K+1),K} = 1/2$ for all values of K. Careful examination of the threshold phenomenon at $\lambda = 2$ shows that, for large values of K, we are almost guaranteed the ability to totally classify $N = 2(K + 1)$ well-distributed patterns with a generalized decision function of $K + 1$ parameters. On the other hand, if N is greater than $2(K + 1)$, we see that the probability of achieving a dichotomy declines sharply for similarly large values of K.

The forgoing considerations lead to define the *dichotomization capacity* of a generalized decision function as

$$C_K = 2(K + 1) \qquad \text{.....(2.27)}$$

We see that the capacity, as defined here, is equal to twice the number of degrees of freedom (adjustable parameters) of the generalized decision functions given in Eq (2.22). Tabulated below for comparison are the dichotomization capacities of some decision functions for n-dimensional patterns.

Decision Boundary	Dichotomization Capacity
Hyperplane	$2(n + 1)$
Hypersphere	$2(n + 2)$
General Quadratic Surface	$(n + 1)(n + 2)$
rth-order polynomial surface	$2\,C_r^{n+r}$

2.4.4 Implementation of Decision Functions

The implementation phase of a pattern classifier based on the decision functions discussed so far consists simply of choosing an acceptable method for mechanizing these functions. In many application, the entire pattern recognition system is implemented in a computer. In other applications where a computer is available only during the design phase, or where very high speed of computation or other specialized requirements are essential factors, it may be necessary to utilize specialized circuitry to do the job.

A schematic diagram of a multiclass pattern classifier based on the general decision functions previously discussed is shown in Fig. 2.6. For simplicity, the discussion is limited here to multiclass Case 3. The other two cases can be implemented with similar system.

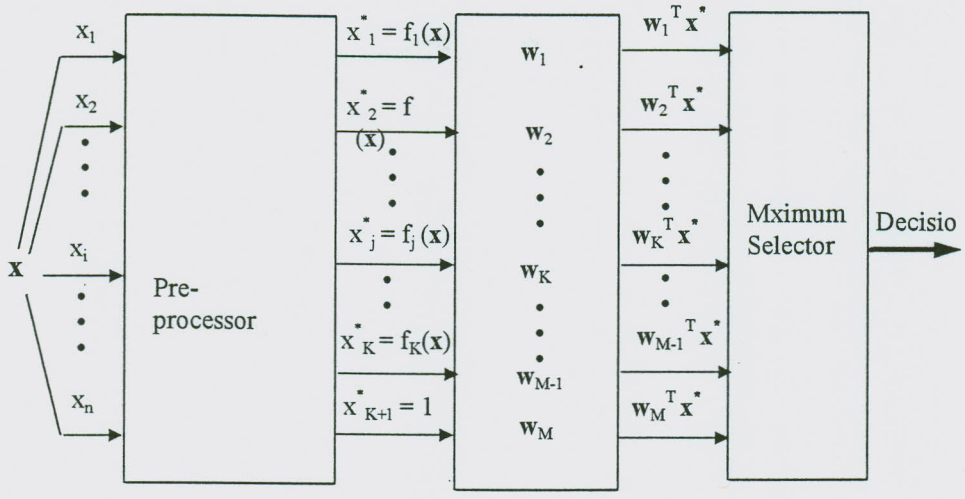


Figure 2.6. Schematic diagram of a multiclass pattern classifier

The pre-processor in this case simply mechanizes Eq. (2.22), the generalized decision function. The box following the pre-processor evaluates the decision functions $d_i(\mathbf{x}^*) = \mathbf{w}_i^T\mathbf{x}^*$, for i = 1,2, . . ., *M*, where *M* is the number of classes. The next stage is a maximum selector. It selects the largest vector product and assigns the unknown pattern to the corresponding class.

2.5. Automatic Pattern Classifications [15]

There are a number of schemes used for pattern classification in an automatic pattern recognition systems. They may be implemented by way of distance functions, likelihood functions (Bayesian classifier) or trainable pattern classifiers. Although these methods are radically different in the approach taken, their basic aim is the same - to develop techniques which can be used to generate, from training patterns, decision functions that can serve as the basis for automatic decision making. Although the classification methods mentioned above represent a varied and powerful set of tools capable of handling a large variety of problems of practical significance, the effectiveness of a given technique is strongly dependent on both the data and the intended application.

In this paper only brief descriptions of the above mentioned classifiers will be given, but the deterministic trainable pattern classifiers are further treated in detail in the next chapter since the thesis is concerned with such an approach to pattern recognition.

2.5.1. Pattern Classification By Distance Functions

The concept of pattern classification by distance functions is the simplest and most intuitive approaches to the problem and is one of the earliest concepts in automatic pattern recognition. The motivation for using distance functions as a classification tool follows naturally from the fact that the most obvious way of establishing a measure of similarity between pattern vectors, which we also consider as points in Euclidean space, is by determining their proximity. Since the proximity of an unknown pattern to the patterns of a class serves as a measure of its classification, the term *minimum-distance pattern classification* is used to characterize this approach. This method can be expected to yield practical and satisfactory results only when the pattern classes tend to have clustering properties i.e. the pattern classes should exhibit a reasonably limited degree of variability. Since clustering properties play an important role in the performance of classifiers based on a distance concept, several clustering algorithms used to estimate cluster centers of pattern classes have been developed by researchers.

2.5.2. Pattern Classification By Likelihood Functions

As its name implies, this approach takes into account the statistical properties of pattern classes in order to arrive at a classification scheme. By means of statistical considerations it is possible to derive a classification rule which is optimal in the sense that, on an average basis, its use yields the lowest probability of committing classification errors. This statistically optimal classification rule, which may also be called the Bayes classification rule, is a generally accepted standard against which the performance of other classification algorithms is often compared.

The structure of the Bayes classifier for normal patterns is completely fixed by the determination of the mean vector and covariance matrix of each class.

2.5.3. Trainable Pattern Classifiers

As pointed earlier, once a type of decision function has been specified, the problem is the determination of the coefficients.

When we wish to design a pattern recognition system which is resistant to distortions, flexible under large pattern deviations, and capable of self-adjustment, we are confronted with the adaptation problem.

The previous two approaches to the design of pattern classifiers are based on direct computations in the sense that the decision boundaries generated by these approaches are derived from sample patterns which determine the coefficients via direct calculation and not based on the interactive, training framework.

In this approach, the coefficients of decision functions used as pattern classifiers are generated from training patterns by means of iterative , *"learning"* algorithms. These algorithms [15] are capable of learning the solution coefficients from the training sets whenever these training pattern sets are separable by the specified decision functions.

Implementation of trainable pattern classifiers have two main approaches: *the deterministic approach* and *the statistical approach*. The statistical approach make use of the statistical properties of the pattern classes in the formulation and derivation of the trainable pattern classifier algorithms. But in the deterministic approach, no statistical properties of the pattern classes are used. In the final analysis seen in literature, both the statistical and deterministic algorithms were found surprisingly similar in form.

2.5.3.1. The Deterministic Approach to Trainable Pattern Classifier

The origin of the pattern classification algorithm in this approach may be traced to early efforts in the field of ***bionics*** (the application of biological concepts to electronic machines), which were concerned with problems in animal and machine learning. During the mid 1950's and early 1960's a class of machines, originated by Rosenblat (1957) and frequently called ***perceptrons***, seemed to offer what many researchers thought was a natural and powerful model of machine learning.

Since Chapter 3 on Artificial Neural Networks says on this point and much more on other related issues, this section is made relieved from saying anything rather than being redundant.

2.5.3.2. Statistical Approach to Trainable Pattern Classifier

In this approach, all the algorithms for pattern classification are derived from statistical and probabilistic considerations of pattern classes. Also, since the Bayes classification rule sets the standard of optimum classification performance, it is logical that a statistical formulation of pattern classification algorithms should be centered on this rule. The Bayes decision functions minimize the average cost of misclassification as well as yielding the lowest probability of error.

Stochastic approximation methods and the method of potential functions are only representative of the spectrum of available schemes in formulating trainable statistical algorithms for the determination of the estimation of optimum decision functions used for classification.

Unlike their deterministic counterparts, the statistical algorithms developed using the above two methods converge in the limit to an approximation of the Bayes classifier. It is worth noting that the poor convergence rates of statistical classifiers tend to overshadow their potential for optimum performance. As is true with deterministic approaches, however, the quality of the decision functions generated by statistical methods is, in general, strongly dependent on the complexity of the approximation chosen for these decision functions.

CHAPTER 3. NEURAL NETWORKS

3.1 Theoretical Background.

The immense computational capability, memory storage capacity and huge execution speed of modern digital computers, continuously boosted up by modern hardware and software technologies, enabled them to solve hundreds of simultaneous differential equations with speed and accuracy. With regard to solving scientific, engineering, economic, and social problems confronting mankind, this immense capability imply that much effort has to be expended in the re-examination and revolutionization of the traditional problem assessment and formulations, mathematical analysis and computational techniques [15].

In some fields, it is not yet possible to write mathematical equations which accurately describe process of interest. Here, the computer may be used simply to simulate a process and, perhaps, to observe the efficacy of different control processes. In others, a mathematical description may be available, but the equations are frequently difficult to solve numerically. In such cases, the difficulties may be faced squarely and possibly overcome; alternatively formulations may be sought which are more compatible with the inherent capability of computers. Mathematics itself nourishes and is nourished by such developments. Most practical pattern recognition problems being the most difficult of this kind, are the most potential areas that should be tackled with the computer [15].

Furthermore, among these pattern recognition problems, there are many potential computer applications that are difficult to implement because they are unsuited to solution by a sequential process. Applications that must perform some complex data translation, yet have no predefined mapping function to describe the translation process, or those that must provide a *best guess* as output when presented with noisy input data are but two examples of problems of this type [8].

In this chapter a neural network implementation, which is a parallel processing model, of a pattern recognition system is presented.

3.1.1 Introduction [8], [10] [12]

Artificial Neural Networks (ANN) models or simply "neural networks" go by many names such as connectionist models, Parallel Distributed Processing models, and neuromorphic systems. They have been studied for many years in the hope of achieving human-like performance in the fields of speech and image recognition. They are used in pattern recognition systems as pattern classifiers. These models are composed of many non-linear computational elements operating in parallel and arranged in patterns reminiscent to biological neural networks (BNN) based on our present understanding of the biological nervous systems. The computational elements or nodes are connected via weights that are typically adapted during use or performance. Thus, instead of performing a program of instructions sequentially as in a Von Neumann computer architecture, neural network models explore many competing hypotheses simultaneously using massively parallel networks composed of many computational elements connected by links with variable or adaptable weights.

They have greatest potential in areas such as speech and image recognition where many hypotheses are pursued in parallel, high computations are required, and the current best systems are far from equalling human performance.

Characteristics

The potential benefits of neural networks extend beyond the high computation rates provided by massive parallelism. Neural networks typically provide a greater degree of robustness or fault tolerance than sequential algorithms, because they have many processing nodes each with primarily local connections. Few erroneous training data or damage to a few nodes or links thus need not impair overall performance significantly. Most neural network algorithms also adopt connection weights in time to improve performance based on current computation results.

Adaptation or learning where training data is limited, such as in speech and image recognition, is a major focus of neural network research. Adaptation also provides a degree of robustness by compensating for minor variablities in characteristics of processing elements.

Neural network classifiers are non parametric and make weaker assumptions concerning the shapes of underlying distributions than traditional statistical classifiers. They may thus prove to be more robust when distributions are generated by non-linear processes and are strongly non gaussian.

Neural network models are specified by the network topologies or architectures, node characteristics, and training or learning rules. These rules specify an initial set of weights and indicate how weights should be adapted during training. Both design procedures and training rules are the topic of much current research. The ANN is trained to classify a given classification task in a supervised or unsupervised learning mode, depending on the particular architecture.

3.1.2. The Neuro-Physiology Of The Brain [6], [8]

Studying real biological neural networks lead to new insights and algorithmic improvements in designing and implementing artificial neural networks. Studies over the past few decades have shed some light on the construction and operation of our brains and nervous systems.

The Organization Of The Nervous System

The mental activity of an organism is effected through the agency of a multitude of special bodily devices. One group of such devices serves as receivers of external influences, a second group transforms them into signals, constructs a plan of behaviour and controls its execution, a third group imparts the necessary energy and impetuosity to the behaviour, a fourth group actuates the muscles, etc. This complex activity performed by an organism, permits its active orientation in a specific situation and enables it to solve vital tasks.

In contrast to a unicellular organism , higher representatives of the animal world exhibit a very fast and accurate response to their environment owing to the specialization of their organs. Some of these specialized organs have, for instance, specialized functions due to their special cells. Cells whose sole function is to receive signals are called *receptors* and are sensitive to certain kinds of external stimulation. Other cells form *effectors* - muscles or glands which respond to motor impulses. However, specialization tends to separate organs

and functions, whereas life calls for their constant communication, for the co-ordination of movements with the flows of signals from the surrounding objects and from the organism itself. This integration is achieved through the "master control desk" - ***the central nervous system*** acting as an integral whole.

The basic building block of the nervous system is the *nervous* cell or *neuron* whose main function is to conduct excitation. The major components of a neuron include *a cell body*, *dendrites* or ramifying fibres conveying impulses toward this body , and the *axon* which conducts impulses away from the cell body to other neurons.

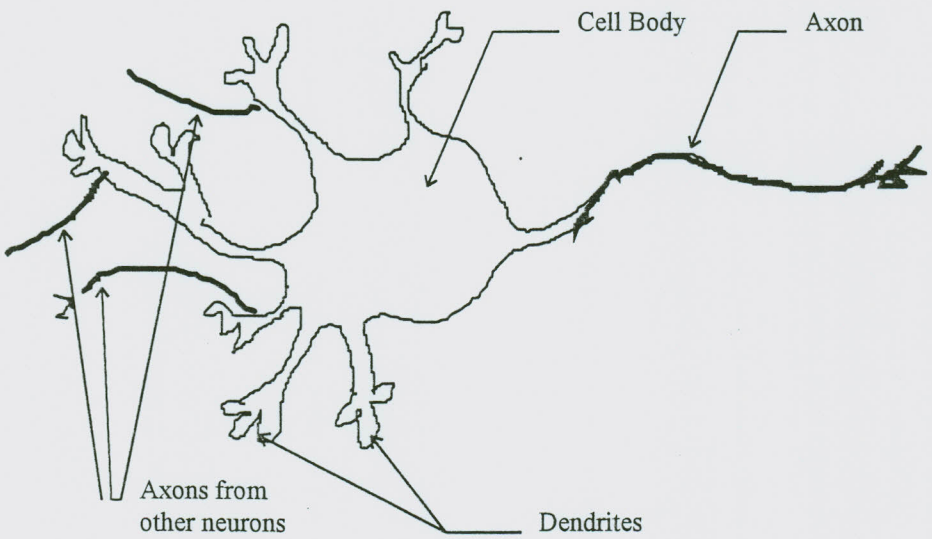


Figure 3.1 The biological neuron

Fig. 3.1, a conceptual diagram of a neuron, is a sketch of only one representation of a neuron. There are many kinds of neurons, with different configuration and functioning. The neuron in this figure probably resembles a motor neuron more than most other types, such as sensory neurons, but it is meant only to convey the basic configuration and terminology. Note that the signal flow goes from left to right, from dendrites, through the cell body, and out through the axon. The signal from one neuron is passed onto another by means of a connection between the axon of the first and the dendrite of the second. This connection is called a *synapse*. Axons often synapse onto the trunk of a dendrite, but they can also synapse directly on to the cell body. The synapse is believed to play the key role in the mechanism

responsible for the establishment of new links in the nervous system. It is presumed that the establishment of such links is accompanied by certain changes (chemical or structural) in the synapses which transmit impulses in a definite direction. According to the physiological theories of memory, any nervous impulse passing through a definite group of neurons leaves behind it a physical trace in the form of electrical and mechanical changes of synapses. Such changes facilitate the secondary passage of the impulse along the trail that has been blazed.

The waves evolving in the brain [6] are electromagnetic oscillations of different frequencies. The lowest frequencies correspond to a state of response when a person is relaxed and sitting with his eyes closed. Once he receives an assignment, for instance, to do a sum, the curve of his biological currents immediately changes and exhibits signs of much higher frequencies.

The human brain is composed of [8] many different parallel, distributed systems, performing well-defined functions, but under the control of a serial-processing system at one or more levels. It has a large number of neurons [6] with typical estimates of in the order of 10 - 500 billions. According to one estimate by Stubbs [6], neurons are arranged into about 1000 main modules, each with about 500 bio-neural networks. Each network has on the order of 100,000 neurons. The axon of each neuron connects to about 100 (but sometimes several thousands) other neurons, and this value varies greatly from neuron to neuron and from neuron type to neuron type. According to a rule called Eccles law [6], [8], each neuron either *excites* or *inhibits* all neurons to which it is connected.

3.1.3 The Historical Development Of The ANN [5] , [10]

Work on artificial neural network models has a long history. Development of detailed mathematical models began in the late 1940 with the work of McCulloch and Pitts, Hebb, Rosenblat, Widrow and others. More recent work by Hopefield, Rumelhart and McClelland, Seinowski, Feldnan, Grossberg and others has led to a new resurgence of the field of artificial neural networks. This new interest is due to the development of new network topologies or architecture and algorithms, new analogue VLSI implementation techniques, and some intriguing demonstrations as well as by a growing fascination with the functioning of the human brain. Recent interest is also driven by the realization that human-like performance in the areas of speech and image recognition will require enormous amounts of processing, one

technique of which may be provided by neural networks due to their inherent massive parallelism .

3.2 Hardware & Software Implementation Of The ANN

3.2.1 The Hardware Implementation [16]

The promise of applying what are known to be extremely effective neural networks and algorithms to elusive problems in science and engineering has historically provided a strong incentive to pursue the design of *neuromines,* electronic circuits that mimic the behaviour of biological nerve cells. Since the advent of semiconductor devices, many such models of neurons and neural processes have been reported.

Mathematical and neuromorphic models dating from 1962 by Crane and Jenik, have given a variety of electronic circuits. Since then, with the onset of discrete semiconductors and the subsequent availability of general purpose IC's, architectures for a variety of comprehensive neural models were explored, and many notable prototype circuits were subsequently developed and reported by many researchers with significant improvement in comprehensiveness or applicability over their predecessors. While many of these were presented as being able to mimic some aspect or aspects of neural functions, another significant long-term rationale for these circuits was to model neural networks rather than individual cells. It was held that, eventually, dedicated "brains" could be applied to difficult problems in signal processing, computation and pattern recognition. Apparently, those prototypes implemented in discrete semiconductors were too cumbersome to apply in large quantity relative to their IC implementation.

More recently, neural network models have entered the realm of custom analogue *very large scale integration* VLSI technology. In VLSI form, these models are equally comprehensive, power efficient, and labour-efficient than their discrete predecessors.

As to applications, for instance, being power-efficient, compact and noise immune, the IC-based artificial neuron is ideal for assembly into neural networks and interfacing to biological counterparts and its two applications are in the areas of rehabilitation technology and robotics.

Research is being carried out to restore or rehabilitate a partial or complete impairment of a biological organ such as the eye or ear with IC-based artificial neurons or neural networks owing to the rich variety of circuits afforded by Complementary Metal Oxide Semiconductor, CMOS, VLSI technology allowing the incorporation of sophisticated filters, amplifiers, and other signal processors into a circuit. This is because the high power efficiency, noise immunity, circuit density, and high input impedance provided by a CMOS circuitry enables its interfacing with living tissue and telemetering signals to virtually any circuit, computer, or electromechanical device.

Robotic controllers are modelled after basic physiological models of neuromuscular control and learning, and thus an IC-based artificial neuron may be used to actuate a joint of conventional robotic arm.

3.2.2. Software Implementation, Sequential & Parallel Processing

Neural network models are usually simulated using a software development environment both in the training and the testing modes. Since the training session takes a very large time and more system resource, a brief overview of the two possible implementations, the sequential and parallel processing, will be made in this section.

Simulation With Sequential Processing

Although neural network models are characterised by massive parallelism which may be tackled with parallel distributed computation easily, even in the training mode most users employ the PC with a hardware architecture characterized by inherent sequential processing mode, i.e. a Von Neumann architecture with a single task processing at a time. This forces one to develop a computational algorithm for training networks on such machines and to improve performance both in speed and accuracy in these machines it will be very good to use a good software development environment tool with efficient data structures, memory management, and high compilation or execution speed as well as a good design of optimal network architecture and training sets.

Simulation With Parallel Processing [6]

In parallel processing, the underlying array of parallel processor hardware architecture, with its multitasking and synchronized communications capabilities, almost completely eliminates the programming hassles associated with controlling several tasks concurrently provided by multitasking operating systems on a PC. As great an increase in performance as possible is obtained by distributing the computational workload as evenly as possible among the array of processors.

As to the software, certain algorithms fit more naturally into parallel implementation than others and ideally we seek to find a computational algorithm that when applied to parallel processors, increases linearly in performance as the number of processors is increased.

In the training phase, an algorithm called ***pipelining*** the computational algorithm of which is described by Pomerleau et. al for use with the Carnegie Mellon University Warp Machine and Chong and Fallside for the transputer proved good. This algorithm can be mapped onto a network of processors in such a manner that any number of processors can be used in a particular training session, with almost linear improvement in speed for every processor added. The algorithm uses a group of processors that are arranged in what is called a *pipeline*. The main concept of a pipeline is that each processor receives intermediate results and/or data from its upstream neighbour, processes the data, and then sends its results/data to its downstream neighbour. For best results, the computational approach should try to perform communications in parallel with the computations. Also, the amount of work that each processor performs should be the same. In this way no processors have to wait for another processor to finish before they can continue with the next stage of processing.

Data Structures Used In An ANN Software Simulation [8]

Since ANN simulators emphasize efficiency in order to reduce the amount of training time needed, an efficient data structure accounting program generality as well as acceptable learning speed should be chosen.

As will be observed later, most data in neural networks [8] is processed as a sum of products (or as the inner product between the weight and input vectors) and this implies that

the network data ought to be arranged in groups of linearly sequential arrays, each containing homogenous data. Two possible data structures to implement this are : *arrays* and *linked-list*.

In the array data structure, a faster sequential stepping through of the data is possible giving rise to increase in speed than it is to have to look up the address of every new value, as would be done if a linked-list approach is used. However, the speed efficiency of arrays is bought at the expense of algorithm generality which is provided with linked-lists. Thus for specific problems arrays, and for general type problems linked-list should be used.

ANN Applications [5],[8]

There are several types of neural network architectures that are in use today. They include the BPN (Back Propagation Network), CPN(Counter Propagation Network), the madaline, the neocognitron and so on. Some, such as the BPN, are general purpose. Some potential areas they are implemented include character recognition, speech and image recognition, medical diagnosis, human face and finger print recognition. In some of these areas, they may be implemented as hybrid with other systems such as expert systems and artificial intelligence.

An expert system is [6] a software-based system that describes the behaviour of an expert in some field by capturing the knowledge of one or more experts in the form of rules and symbols. It is more efficient than neural networks in problems with inadequate data, which might not be enough to train a neural network .

A hybrid system that include both neural networks and expert systems [6] may be called an *expert network* which in some applications may be a very powerful tool. For example for a medical diagnostic system, you might use an expert system, augmented with C++ functions, to interact with the user and provide the initial query capability that guides the system into general area. A series of ANN tools, consisting of networks and subnetworks, could be used to assist in the diagnosis of the more common ailments, and rule-based systems could come into play for the relatively obscure ailments.

System (Hardware & Software) Requirement of ANN [6]

This depends on the complexity of the problem to be solved. Generally, since during their training phase on a PC, neural network systems are CPU speed and RAM memory hungry, training of the networks will be well efficient and user interactive when moderate or easy problems (requiring moderately sized neural architectures) are solved with moderate or high capacity (in speed and memory size) machines and complex problems (requiring large sized neural architectures) are solved with high capacity machines.

However, once the networks are trained, the amount of memory required by the trained version of the network greatly reduces and its speed performance is relatively very high. Thus, one can use high capacity machine for training of a neural network but can implement and use its trained version on a relatively less capacity machine.

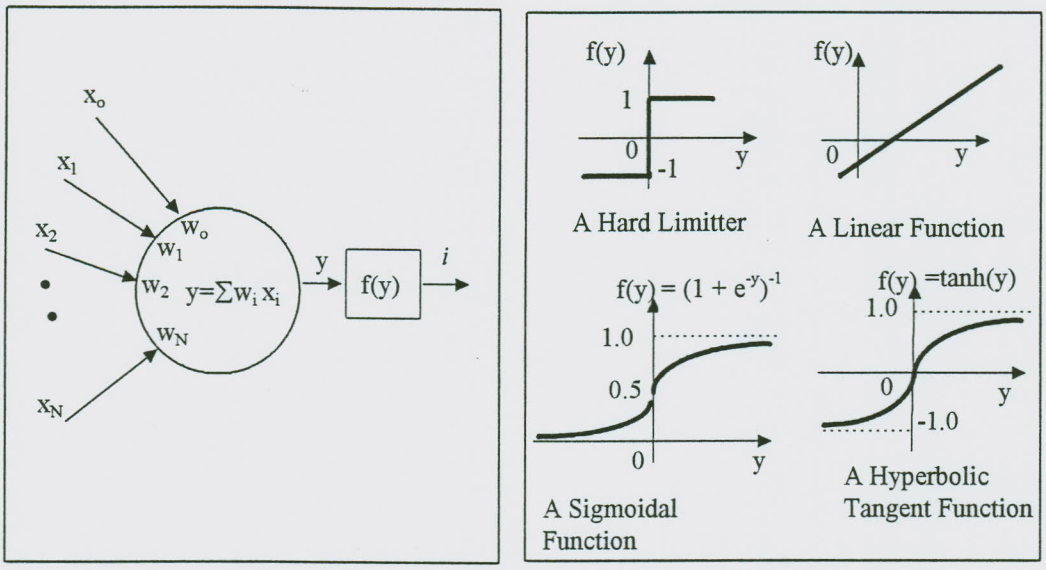
An efficient memory and speed management is also possible by using efficient programming environments such as C++ and Ada. Although tedious and difficult, neural network algorithms may also be coded using machine code programming which greatly increases the speed of training.

3.3. The Back Propagation Network (BPN)

An ANN that is found to be useful [8] in addressing problems requiring recognition of complex patterns and performing nontrivial mapping functions is the BPN, formalized first by Werbos, and later by Parker and by Rummelhart and McClelland. This network is designed to operate as a multilayer, feedforward network, using the supervised mode of learning. In this section, the neuron is discussed and the LMS rule and the general delta rule (GDR) which are used in the design of the learning rules are derived and summarized.

3.3.1 The Neuron Or Processing Element(PE) [5],[8]

The individual computational elements that make up most artificial neural system models are referred to as [8] *nodes*, *units*, *processing elements* (PE) or *artificial neurones* (rarely called since they are only crude models of their biological counterpart).



(a) The Neuron or Processing Element

(b) Different Types of Activation Functions

Figure 3.2. The neuron or PE used in the BPN and the different types of output functions

The artificial neuron or PE is the basic or primitive building block [8] of a neural network. Like a real neuron, the PE has many inputs but has only a single output which can fan out to many other PEs in an ANN. Each input to the PE has associated with it a quantity called a *weight* or ***connection strength.*** All these quantities have analogues in the standard neuron model, i.e., the output of the PE corresponds to the firing frequency of the neuron, and the weight corresponds to the strength of the synaptic connection between neurons. In our models, these quantities will be represented as real numbers. An input connection may be ***excitory***, having positive weights, or ***inhibitory***, having negative weights, and both types are usually considered together constituting the most common forms of input to a PE.

For the BPN the PE used is called the Adaline (ADAptive LInear Neuron) which consists of the ALC (Adaptive Linear Combiner) and an output activation function. The ALC has many weighted inputs, an optional bias term, and a combiner or summer unit. Its input may be continuous (normalized or not) or binary and the weights, or connection strength, via which the inputs are linked to the summer may be positive or negative real numbers. The bias term is a weight on a connection that has its input value always equal to 1 and its inclusion is largely a matter of experience [8] in helping learning convergence during training.

The output function thresholder may be a hard limiter, a linear output, a sigmoid function, a hyperbolic tangent function or another function depending on the problem to be

solved. Fig. 3.2 shows the complete PE or artificial neuron structure with different output functions

The PE performs [8] a sum-of-products calculation using the input and weight vectors, to give the net input y and applies an output function to get a single output value. Using the notation in Fig. 3.2,

$$y = w_0 + \sum_{j=1}^{n} w_j x_j \qquad \text{.....(3.1)}$$

where w_0 is the bias weight. If we make the identification, $x_0 = 1$, we can rewrite the preceding equation as

$$y = \sum_{j=0}^{n} w_j x_j \qquad \text{.....(3.2)}$$

or in vector notation

$$y = \mathbf{w}^t \mathbf{x} \qquad \text{.....(3.3)}$$

Once the net input is calculated, it is converted to an *activation value* or simply activation, for the PE. Since in ALC (as well as in the majority of neuron models) the activation and the net input are the same [8], the two terms are used interchangeably. We can thus determine the output value i of the PE by applying an *output function* :

$$i = f(y) \qquad \text{.....(3.4)}$$

The PE, Adaline (or the ALC) is adaptive in the sense that [8] there exists a well-defined procedure for modifying the weights in order to allow the device to give *correct* output value for the given input and this weight adjusting scheme is dealt next.

3.3.2 The LMS Learning Rule [8]

Before developing the complete network learning algorithm called the Generalized Delta Rule (GDR), we first develop the LMS learning rule for a single PE.

Suppose we have a set of input vectors, $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_L\}$, each having its own, perhaps unique, correct or *desired* output value, d_k, k = 1,...,L. The problem is of finding a single optimum, or best weight vector, $\mathbf{w}^*$, that can successfully associate each input vector with its desired output value. Two methods will be shown to determine $\mathbf{w}^*$:

(i) in analytic or explicit form and

(ii) in adaptive iteration form.

Both methods use the LMS rule. The second method or procedure, which employs the computation of the weight vector explicitly, is termed as the least mean-square (LMS) learning rule to be developed subsequently and the process of finding the weight vector is referred to as *training* the PE, ALC or neuron.

Let us restate the problem a little differently before applying the methods : given examples, $(\mathbf{x}_1,d_1)$, $(\mathbf{x}_2,d_2)$, ..., $(\mathbf{x}_L,d_L)$, of some processing function that associates input vectors, $\mathbf{x}_k$, with (or maps to) the desired output values, d_k, what is the best weight vector, $\mathbf{w}^*$, for a PE or an ALC that performs this mapping?

(i) Analytical or Explicit Determination of w*

This method is based on the minimization of the error, the difference between the desired output and the actual output, for each input vector applied to the PE or ALC. The approach selected here is to minimize the mean squared error for the set of input vectors. If the actual output value is y_k for the kth input vector $\mathbf{x}_k$, then the corresponding error term is $\varepsilon_k = d_k - y_k$.

The mean squared error, or expectation value of the error, is defined by

$$E[\varepsilon_k^2] = \frac{1}{L}\sum_{k=1}^{L}\varepsilon_k^2 \qquad \text{.....(3.5)}$$

where L is the number of input vectors in *the training set*, or exemplars.

Using Eq. (3.3), we can expand the mean squared error as follows:

$$E[\varepsilon_k^2] = E[(d_k - \mathbf{w}^t\mathbf{x}_k)^2] \qquad \text{.....(3.6)}$$

$$= E[d_k^2] + \mathbf{w}^t E[\mathbf{x}_k\,\mathbf{x}_k^t]\,\mathbf{w} - 2E[d_k\,\mathbf{x}_k^t]\,\mathbf{w} \qquad \text{.....(3.7)}$$

In going from Eq. (3.6) to Eq. (3.7), we have made the assumption [8] that the training set is statistically stationary, meaning that any expectation values vary slowly with respect to time. This assumption allows us to factor out the weight vectors from the expectation value terms in Eq. (3.7).

Now define a matrix $\mathbf{R} = E[\mathbf{x}_k, \mathbf{x}_k^t]$, called the input correlation matrix, and a vector $\mathbf{p} = E[d_k \mathbf{x}_k]$ and make the identification $\xi = E[\varepsilon_k^2]$. Using these definitions, we can rewrite Eq. (3.7) as,

$$\xi = E[d_k^2] + \mathbf{w}^t \mathbf{R}\mathbf{w} - 2\,\mathbf{p}^t \mathbf{w} \qquad \text{.....(3.8)}$$

This equation shows ξ as an explicit function of the weight vector, **w**. In other words, $\xi = \xi(\mathbf{w})$.

To find the weight vector corresponding to the minimum mean squared error, we differentiate Eq. (3.8), evaluate the result at **w***, and set the result equal to zero:

$$\frac{\partial \xi(\mathbf{w})}{\partial \mathbf{w}} = 2\mathbf{R}\mathbf{w} - 2\mathbf{p} \qquad \text{.....(3.9a)}$$

$$2\mathbf{R}\mathbf{w}^* - 2\mathbf{p} = 0$$

$$\mathbf{R}\mathbf{w}^* = \mathbf{p} \qquad \text{.....(3.9b)}$$

or

$$\mathbf{w}^* = \mathbf{R}^{-1}\,\mathbf{p} \qquad \text{.....(3.9c)}$$

Notice that, although ξ is a scalar , $\partial\xi(\mathbf{w})/\partial\mathbf{w}$ is a vector. Equation (3.9a) is an expression of the gradient of ξ , $\nabla\xi$., which is the vector

$$\nabla\xi = \left[\frac{\partial\xi}{\partial w_1}, \frac{\partial\xi}{\partial w_2}, \ldots, \frac{\partial\xi}{\partial w_n}\right]^t \qquad \text{.....(3.10)}$$

All that we have done by the procedure is to show that we can find a point where the slope of the function, ξ (**w**), is zero. In general, that point may be a minimum or a maximum point. This result is general and is obtained regardless of the dimension of the weight vector. In the case of two dimensions, the graph of ξ (**w**) is a paraboloid and it must be a concave upward surface since all combinations of weights must result in a nonnegative value for the mean squared error, ξ. For dimensions higher than two the paraboloid is known as a hyperboloid.

(ii) The Adaptive or Iterative Procedure of the Determination of w*

This method of computing the optimum weight **w*** allows us to avoid the often-difficult calculations necessary to determine the weights manually and is the basis by which most neural network learning algorithms are designed. It employs the ***method of steepest descent*** .

The previous method used to determine **w*** is rather difficult in general. Not only does the matrix manipulation get cumbersome for large dimensions, but also each component of **R** and **p** is itself an expectation value. Thus, explicit calculations of **R** and **p** require knowledge of the statistics of input signals. A better approach would be to let the PE or ALC find the optimum weights itself by having it search over the weight surface to find the minimum. A purely random search might not be productive or efficient, so we shall add some ***intelligence*** to the procedure [8].

Begin by assigning arbitrary values to the weights. From that point on the weight surface, determine the direction of the steepest slope in the down ward direction. Change the weights slightly so that the new weight vector lies farther down the surface. Repeat the process until the minimum has been reached.

Implicit in this method is the assumption that we *know* what the weight surface looks like in advance. We do not know the weight surface, however, and we will shortly see the solution to this problem leads to our learning algorithm of the PE.

Because the weight vector is variable in this procedure, we write it as an explicit function of time step, *t*. The initial weight vector is denoted **w**(0), and the weight vector at time step t is **w**(t). At each step, the next weight vector is calculated according to

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\,\mathbf{w}(t) \qquad \text{.....(3.11)}$$

where $\Delta\,\mathbf{w}(t)$ is the change in **w** at the *t*th time step.

We are looking for the direction of the steepest descent at each point on the surface, so we need to calculate the gradient of the surface (which gives the direction of the steepest *upward* slope). The negative of the gradient is in the direction of the steepest descent. To get the magnitude of the change, multiply the gradient by a suitable constant, η . This procedure results in the following expression:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta\nabla\xi(\mathbf{w}(t)) \quad \text{.....(3.12)}$$

All that is necessary to complete the discussion is to determine the value of $\nabla\xi(\mathbf{w}(t))$ at each successive iteration step. The value of $\nabla\xi(\mathbf{w}(t))$ was determined analytically previously. Eq. (3.9) or Eq. (3.10) could be used here to determine $\nabla\xi(\mathbf{w}(t))$, but we would have the same problem that we had with the analytical determination of $\mathbf{w}^*$; we would need to know both **R** and **p** in advance This knowledge is equivalent to knowing what the weight surface looks like in advance. To circumvent this difficulty [8], we use an approximation for the gradient that can be determined from information that is known explicitly at each iteration. For each step in the iteration process, we perform the following :

i) Apply an input vector, $\mathbf{x}_k$, to the ALC inputs.

ii) Determine the value of the error squared, $\varepsilon_k^2(t)$, using the current value of the weight vector

$$\varepsilon_k^2(t) = (d_k - \mathbf{w}^t(t)\mathbf{x}_k)^2 \quad \text{.....(3.13)}$$

.iii) Calculate an approximation to $\nabla\xi(t)$, by using $\varepsilon_k^2(t)$ as an approximation for $E[\varepsilon_k^2]$:

$$\nabla\varepsilon_k^2(t) \approx \nabla E[\varepsilon_k^2] \quad \text{.....(3.14)}$$

$$\nabla\varepsilon_k^2(t) = -2\,\varepsilon_k(t)\,\mathbf{x}_k \quad \text{.....(3.15)}$$

where we have used Eq. (3.13) to calculate the gradient explicitly.

iv) Update the weight vector according to Eq. (3.12) using Eq. (3.15) as the approximation for the gradient

$$\mathbf{w}(t+1) = \mathbf{w}(t) + 2\,\eta\varepsilon_k(t)\mathbf{x}_k \quad \text{.....(3.16)}$$

v) Repeat steps 1 through 4 with the next input vector, until the error has been reduced to an acceptable value.

Equation 3.16 is an expression of the LMS algorithm. Changes in the weight vector must be kept relatively small on each iteration. If changes are too large, the weight vector

could wander about the surface, never finding the minimum, or finding it only by accident rather than as a result of a steady convergence toward it.

The parameter η, which is used to prevent this aimless searching [8], is called the *learning rate* which has a significant effect on training and determines the stability and speed of convergence of the weight vector toward the minimum-error value. If the statistics of the input signal are known, it is possible to show that its value is restricted to the range

$$0 < \eta < 1/\lambda_{max} \quad \text{.....(3.17)}$$

where λ_{max} is the largest eigen value of the matrix **R**, the input correlation matrix discussed in Sec 2.4.1 so that stability and convergence are assured. Since there is no a general guideline as to how to choose an appropriate value of η in the absence of the statistics of the input data, experience appears to be the best teacher for selecting its appropriate value [8].

3.3.3. The Generalized Delta Rule [8]

In this section the formal mathematical description of BPN operation is presented with a detailed derivation of the *generalized delta rule* which is the learning algorithm of the network. The generalized delta rule is a generalization of the LMS rule applied to two or more dimensional problems. Fig. 3.3 serves as reference for most of the discussion.

The BPN is a feedforward network that is fully interconnected by layers. There are no feedback connections that bypass one layer to go directly to later layer. Although only three layers are used in the discussion, more than one hidden layer is permissible.

A neural network is called a ***mapping network*** if it is able to compute some functional relationship between its input and its output. For a simple mapping or function we do not need a neural network; however, we might want to perform a complicated mapping where we do not know how to describe the functional relationship in advance, but we do know of examples of the correct mapping. In this situation, the power of a neural network to discover its own algorithms is extremely useful.

Suppose we have a set of P vector-pairs, $(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_P, \mathbf{y}_P)$, which are examples of a functional mapping $\mathbf{y} = \phi(\mathbf{x}) : \mathbf{x} \in e\ \mathbf{R}^N, \mathbf{y} \in \mathbf{R}^M$ .We want to train the network so that it will learn an approximation $\mathbf{o} = \mathbf{y}' = \phi'(\mathbf{x})$. We shall derive [8] a method

of doing this training that usually works, provided the training-vector pairs have been chosen properly and there is a sufficient number of them.

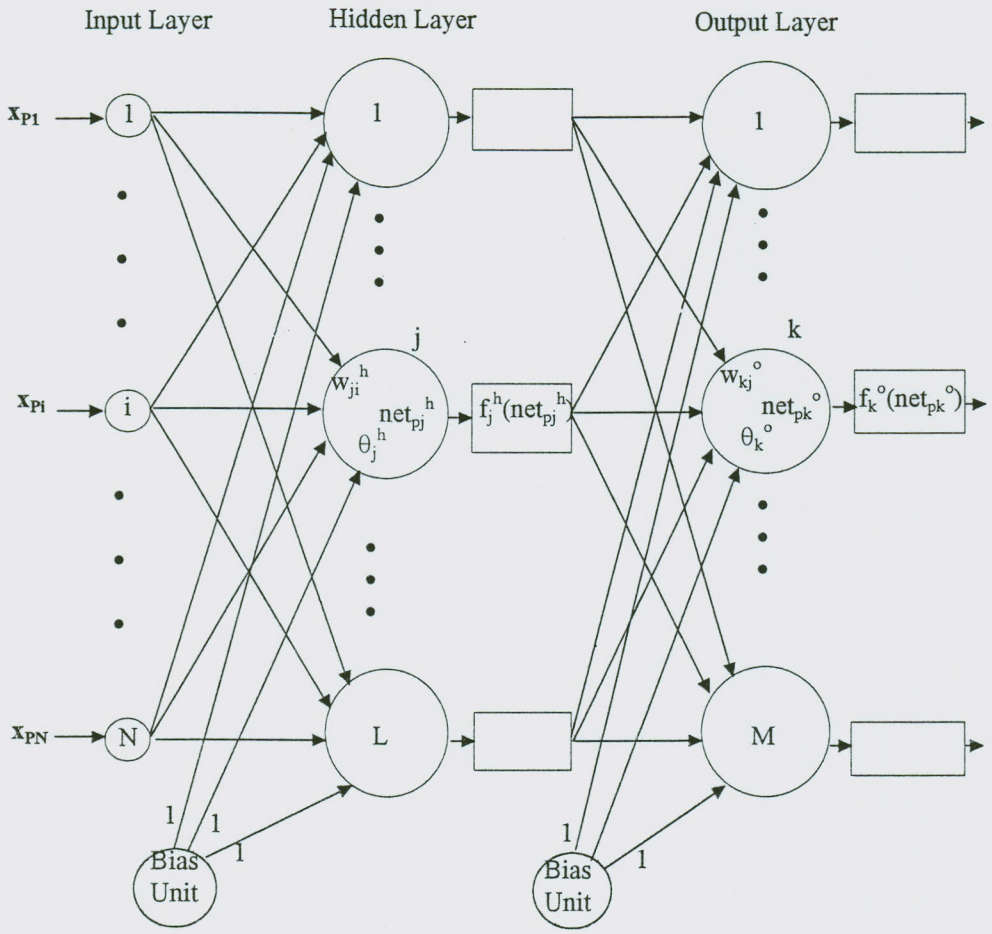


Figure 3.3 The Three Layer Feed-Forward BPN Architecture.

It should be noted that learning in a neural network means finding an appropriate set of weights. The learning technique that we describe here resembles the problem of finding the equation of a line that best fits a number of known points. For a line-fitting problem, we would probably use a least squares approximation. However, since the relationship we are trying to map is likely to be non-linear, as well as multidimensional, we employ an iterative version of the simple least-squares method, called a *steepest-descent technique*.

To begin, let's review the equations for information processing in the three-layer network in Fig 3.3. An input vector, $\mathbf{x}_p = (x_{p1}, x_{p2}, \ldots, x_{pN})^t$, is applied to the input layer of the network. The input units distribute the values to the hidden-layer units. The net input to the *j*th hidden unit is

$$net_{pj}^{h} = \sum_{i=1}^{N} w_{ji}^{h} x_{pi} + \theta_{j}^{h} \quad \text{.....(3.18)}$$

where w^{h}_{ji} is the weight on the connection from the *i*th input unit, and θ_j^{h} is the bias term. The "h" subscript refers to quantities on the hidden layer. Assume that the activation of this node is equal to the net input; then, the output of this node is

$$i_{pj} = f_j^{h}(net^{h}_{pj}) \quad \text{.....(3.19)}$$

The equations for the output nodes are

$$net_{pk}^{o} = \sum_{j=1}^{L} w_{kj}^{o} i_{pj} + \theta_{k}^{o} \quad \text{.....(3.20)}$$

$$o_{pk} = f_k^{o}(net^{o}_{pk}) \quad \text{.....(3.21)}$$

where the "o" superscript refers to quantities on the output layer.

The initial set of weight values represents a first guess as to the proper weights for the problem. Unlike some methods, the technique we employ here does not depend on making a *good* first guess. There are guidelines for selecting the initial weights, however.

The basic procedure for training the network is embodied in the following description.

i) Apply an input training vector to the network and calculate the corresponding output values.

ii) Compare the actual outputs with the correct outputs and determine a measure of the error.

iii) Determine in which direction (+ or -)to change each weight in order to reduce the error.

iv) Determine the amount by which to change each weight.

v) Apply the corrections to the weights.

vi) Repeat items 1 through 5 with all training vectors until the error for all vectors in the training set is reduced to an acceptable value.

The iterative weight-change law for network with no hidden units and linear output units, called the LMS rule or delta rule, treated in Sec 3.3.2 and given in Eq. (3.12) in matrix form and here given in component form is :

$$w(t+1)_i = w(t)_i + 2\,\eta\varepsilon_k\, x_{ki} \quad \text{.....(3.22)}$$

where η is a positive constant (learning rate), x_{ki} is the ith component of the kth training vector, and ε_k is the difference between the actual output, y_k, and the desired or correct value, d_k , that is, $\varepsilon_k = (d_k - y_k)$.

A similar equation results when the network has more than two layers, or when the output functions are non-linear. The results will be derived explicitly in the next section.

Updates of Output-Layer Weights

In this derivation of the delta rule, since there are multiple units in a layer of a BPN, we shall define the error at a single output unit to be $\delta_{pk} = (y_{pk} - o_{pk})$, where the subscript "p" refers to the pth training vector, and "k" refers to the kth output unit. In this case, y_{pk} is the desired output value, and o_{pk} is the actual output from the kth unit. The error that is minimized by the GDR is the sum of the squares of the errors for all output units:

$$E_p = \frac{1}{2}\sum_{k=1}^{M}\delta_{pk}^2 \quad \text{.....(3.23)}$$

The factor of 1/2 in Eq. (3.23) is there for convenience in calculating derivatives later. Since an arbitrary constant will appear in the final result, the presence of this factor does not invalidate the derivation.

To determine the direction in which to change the weights, we calculate the negative of the gradient of E_p , ∇E_p , with respect to the weights, w_{kj}. Then, we can adjust the values of the weights such that the total error is reduced. It is often useful to think of E_p as a surface in weight space. To keep things simple, we consider each component of ∇E_p separately. From Eq. (3.23) and the definition of δ_{pk} ,

$$E_p = \frac{1}{2}\sum_k (y_{pk} - o_{pk})^2 \quad \text{.....(3.24)}$$

and

$$\frac{\partial E_p}{\partial w^o_{kj}} = -(y_{pk} - o_{pk}) \frac{\partial f^o_k}{\partial (net^o_{pk})} \frac{\partial (net^o_{pk})}{\partial w^o_{kj}} \quad \text{.....(3.25)}$$

where we have used Eq. (3.21) for the output value, o_{pk}, and the chain rule for partial derivatives. For the moment, we shall not try to evaluate the derivative of f_k^o, but instead will write it simply as $f_k^{o\prime}$ (net^o_{pk}). The last factor in Eq. (3.25) is

$$\frac{\partial (net^o_{pk})}{\partial w^o_{kj}} = \left(\frac{\partial}{\partial w^o_{kj}} \sum_{j=1}^{L} w^o_{kj} i_{pj} + \theta^o_k \right) = i_{pj} \quad \text{.....(3.26)}$$

Combining Eqs. (3.25) and (3.26), we have for the negative gradient

$$-\frac{\partial E_p}{\partial w^o_{kj}} = (y_{pk} - o_{pk}) f_k^{o\prime}(net^o_{pk}) i_{pj} \quad \text{.....(3.27)}$$

As far as the magnitude of the weight change is concerned, we take it to be proportional to the negative gradient. Thus, the weights on the output layer are updated according to

$$w^o_{kj}(t+1) = w^o_{kj}(t) + \Delta_p w^o_{kj}(t) \quad \text{.....(3.28)}$$

where

$$\Delta_p w^o_{kj}(t) = \eta (y_{pk} - o_{pk}) f_k^{o\prime}(net^o_{pk}) i_{pj} \quad \text{.....(3.29)}$$

The factor η is called the *learning-rate parameter*. The value of η is discussed in Sec 3.3.2. It is positive and is usually less than 1.

Let's go back to look at the function $f_k^{o\prime}$. First, notice the requirement that the function f_k^o be differentiable. This requirement eliminates the possibility of using a linear threshold unit, since the output function for such a unit is not differentiable at the threshold value.

There are two forms of the output function that are of interest here:

(i) $f_k^o\ (\text{net}^o_{jk}) = \text{net}^o_{jk}$ .....(3.30)

(ii) $f_k^o\ (\text{net}^o_{jk}) = (1 + \exp(-\text{net}^o_{jk}))^{-1}$ .....(3.31)

The first function defines the linear output unit. The latter function is called a *sigmoid*, or logistic function; it is illustrated in Fig. (3. 2b). The choice of output function depends on how you choose to represent the output data. For example, if you want the output units to be binary, you use a sigmoid output function, since the sigmoid is output-limiting and quasi-bistable but also differentiable. In other cases, either a linear or a sigmoid output function is appropriate. In the first case, $f_k^{o\,|} = 1$; in the second case, $f_k^{o\,|} = f_k^o\ (1 - f_k^o) = o_{pk}\ (1 - o_{pk})$. For these two cases, we have

$$w^o_{kj}\,(t+1) = w^o_{kj}\,(t) + \eta\,(y_{pk} - o_{pk})\,i_{pj} \quad \text{.....(3.32)}$$

for the linear output, and

$$w^o_{kj}\,(t+1) = w^o_{kj}\,(t) + \eta\,(y_{pk} - o_{pk})\,o_{pk}\,(1 - o_{pk})\,i_{pj} \quad \text{.....(3.33)}$$

for the sigmoidal output.

We now summarize the weight-update equations by defining a quantity

$$\delta^o_{pk} = (y_{pk} - o_{pk})\,f_k^{o|}\ (\text{net}^o_{pk}) \quad \text{.....(3.34)}$$

$$= \delta_{pk} f_k^{o\,|}\ (\text{net}^o_{pk}) \quad \text{.....(3.35)}$$

We can then write the weight-update equation as

$$w^o_{kj}\,(t+1) = w^o_{kj}\,(t) + \eta\delta^o_{pk}\,i_{pj} \quad \text{.....(3.36)}$$

regardless of the functional form of the output function, f_k^o .

We wish to make a comment regarding the relationship between the gradient-descent method described here and the least-squares technique. If we were trying to make the

generalized delta rule entirely analogous to a least-squares method, we would not actually change any of the weight values until all of the training patterns were had been presented to the network once. We would simply accumulate the changes as each pattern was processed, sum them, and make one update to the weights. We would then repeat the process until the error was acceptably low. The error that this process minimizes is

$$E = \sum_{p=1}^{P} E_p \qquad \text{.....(3.37)}$$

where P is the number of patterns in the training set. This procedure is called ***batch*** or ***epoch training*** [6] . In practice, little advantage is found to this strict adherence to analogy with the least-squares method. Moreover, you must store a large amount of information to use this method. It is recommended, therefore that, you perform weight updates as each training pattern is processed and this procedure is called ***on-line training*** [6].

Updates of Hidden-Layer Weights

We would like to repeat for the hidden layer the same type of calculation as we did for the output layer. A problem arises when we try to determine a measure of the error of the outputs of the hidden-layer units. We know what the actual output is, but we have no way of knowing in advance what correct output should be for these units. Intuitively, the total error, E_p , must somehow be related to the output values on the hidden layer. We can verify our intuition by going back to Eq. (3.24) :

$$\begin{aligned} E_p &= \frac{1}{2}\sum_k (y_{pk} - o_{pk})^2 \\ &= \frac{1}{2}\sum_k (y_{pk} - f_k^o(net_{pk}^o))^2 \\ &= \frac{1}{2}\sum_k (y_{pk} - f_k^o(\sum_j w_{kj}^o i_{pj} + \theta_k^o))^2 \end{aligned} \qquad \text{.....(3.38)}$$

We know that i_{pj} depends on the weights on the hidden layer through Eqs. (3.18) and (3.19). We can exploit this fact to calculate the gradient of E_p with respect to the hidden-layer weights.

$$\frac{\partial E_p}{\partial w_{ji}^h} = \frac{1}{2}\sum_k \frac{\partial}{\partial w_{ji}^h}(y_{pk} - o_{pk})^2$$

$$= -\sum_k (y_{pk} - o_{pk}) \frac{\partial o_{pk}}{\partial (net_{pk}^o)} \frac{\partial (net_{pk}^o)}{\partial i_{pj}} \frac{\partial i_{pj}}{\partial (net_{pj}^h)} \frac{\partial (net_{pj}^h)}{\partial w_{ji}^h} \quad \text{.....(3.39)}$$

Each of the factors in Eq. (3.39) can be calculated explicitly from previous equations. The result is

$$\frac{\partial E_p}{\partial w_{ji}^h} = -\sum_k (y_{pk} - o_{pk}) f_k^{o\prime}(net_{pk}^o) w_{kj}^o f_j^{h\prime}(net_{pj}^h) x_{pi} \quad \text{.....(3.40)}$$

We update the hidden-layer weights in proportion to the negative of Eq. (3.40):

$$\Delta_p w_{ji}^h = \eta f_j^{h\prime}(net_{pj}^h) x_{pi} \sum_k (y_{pk} - o_{pk}) f_k^{o\prime}(net_{pk}^o) w_{kj}^o \quad \text{.....(3.41)}$$

where η is once again the learning rate.

We can use the definition of δ^o_{pk} given in the previous section[previously] to write

$$\Delta_p w_{ji}^h = \eta f_j^{h\prime}(net_{pj}^h) x_{pi} \sum_k \delta_{pk}^o w_{kj}^o \quad \text{.....(3.42)}$$

Notice that every weight update on the hidden layer depends on *all* the error terms, δ^o_{pk}, on the output layer. This result is where the notion of ***backpropagation*** arises [8]. The known errors on the output are ***propagated*** back to the hidden layer to determine the appropriate weight changes on that layer. By defining a hidden-layer error term

$$\delta_{pj}^h = f_j^{h\prime}(net_{pj}^h) \sum_k \delta_{pk}^o w_{kj}^o \quad \text{.....(3.43)}$$

we cause the weight update equations to become analogous to those for the output layer:

$$w^{h}_{ji}(t+1) = w^{h}_{ji}(t) + \eta\delta^{h}_{pj} x_{pi} \quad \text{.....(3.44)}$$

3.3.4 Summary Of The BPN Learning Algorithm [8]

The discussion on the GDR will be closed by giving the summary of relevant equations for the BPN in the order in which they would be used during training for a single training-vector pair.

i) Apply the input vector $\mathbf{x}_p = (x_{p1}, x_{p2}, \ldots, x_{pN})^t$ to the input units.

ii) Calculate the net-input values to the hidden layer units:

$$net^{h}_{pj} = \sum_{i=1}^{N} w^{h}_{ji} x_{pi} + \theta^{h}_{j}$$

iii) Calculate the outputs from the hidden layer:

$$i_{pj} = f^{h}_{j}(\text{net}^{h}_{pj})$$

iv) Move to the output layer. Calculate the net-input values to each unit:

$$net^{o}_{pk} = \sum_{j=1}^{L} w^{o}_{kj} i_{pj} + \theta^{o}_{k}$$

v) Calculate the outputs:

$$o_{pk} = f^{o}_{k}(\text{net}^{o}_{pk})$$

vi) Calculate the error terms for the output units:

$$\delta^{o}_{pk} = (y_{pk} - o_{pk}) f^{o\prime}_{k}(\text{net}^{o}_{pk})$$

vii) Calculate the error terms for the hidden units:

$$\delta^{h}_{pj} = f^{hl}_{j}(net^{h}_{pj}) \sum_{k} \delta^{o}_{pk} w^{o}_{kj}$$

Notice that the error terms on the hidden layer units are calculated *before* the connection weights to the out-put layer units have been updated.

viii) Update weights on the output layer :

$$w^{o}_{kj}(t+1) = w^{o}_{kj}(t) + \eta\delta^{o}_{pk} i_{pj}$$

ix) Update weights on the hidden layer :

$$w^{h}_{ji}(t+1) = w^{h}_{ji}(t) + \eta\delta^{h}_{pj} x_{pi}$$

The order of the weight updates on an individual layer is not important.

x) Finally calculate the error term

$$E_p = \frac{1}{2}\sum_{k=1}^{M}\delta_{pk}^2$$

since this quantity is the measure of how well the network is learning. When the error is acceptably small for each of the training-vector pairs, training can be discontinued.

3.3.5 Practical Considerations In ANN Architecture & Parameters Selection

There are several questions [8] to consider when we are attempting to use the BPN to solve a particular problem:

. How is the BPN architecture is designed, that is how many layers and nodes on each layer is required?
. What output function should be used at the hidden and output layer nodes?
. How many training vectors are required to solve a particular problem ?
. What is the appropriate dimension of each of these training vectors ?
. What is the appropriate dimension and initial values of the weight vectors ?
. Is a bias weight or term necessary ?
. What is the appropriate value of the learning rate η and the momentum term α ?
. How do we determine when to stop training ?

The answers to these questions depend on the specific problem being addressed [8], so it is difficult to give well-defined responses that apply in all cases. Moreover, for a specific case, the answers are not necessarily independent. Thus we shall try to present general considerations.

BPN Architectural Parameters

In general the BPN will have an input layer or slab, an output layer and zero, one or more hidden layers and each layer consisting of a number of nodes. However, there are no strict guidelines [8] as to how to determine the number of layers and their associated number of nodes for a particular problem. Sometimes a problem seems to be easier (meaning the network learns faster) to solve with more than one hidden layer. For most application,

however, three layers are sufficient, that is, the BPN will have an input layer, one hidden layer and an output layer.

The number of nodes required on the input layer of the BPN equals the dimension of an input (training/test) vector. The number of nodes required on the output layer [8] can often be determined by deciding whether analogue or binary values are desired on the output units or nodes. For binary values this equals to the number of pattern classes to be learned by the network.

Determining the number of units to use on the hidden layer [5], [6], [8] is not usually as straightforward as it is for the input and output layers. The main idea is to use as few hidden-layer units as possible, because each unit adds to the load on the CPU during simulations. Although there is no a general guide line as to how to chose the number of nodes on the hidden layer, Freeman and Skapura [8] offer from their experience for networks of reasonable size (hundreds or thousands of inputs), the size of the hidden layer needs to be only a relatively small fraction of that of the input layer; and still one of the two other uses the square root of the number of input plus output layer neurons and adding a few which they suppose as a reasonable number to start with in many cases [6]; and the other used the input and output layer neurons sum average approximate [9]. If the network fails to converge to a solution [6], [8], it may be that more hidden nodes are required and if it does converge, you may try fewer hidden nodes and settle on a size on the basis of overall system performance.

It is also possible to remove hidden units that are superfluous [8]. If the weight values on the hidden nodes are examined periodically as the network trains, it will be seen that weights on certain nodes change very little from their starting values. These nodes may not be participating in the learning process, and fewer hidden units may suffice. There is also an automatic method [8], developed by Rumelhart, for pruning unneeded nodes from the network.

Output Function Used At The Hidden And Output Layer Nodes [5],[8],[12]

If the output function is sigmoidal, then the output values have to be scaled. The network outputs can never reach 0 or 1 because of the form of the sigmoidal function and therefore values such as 0.1 and 0.9 should be used to represent the smallest and largest output values respectively. The sigmoid function can also be shifted so that, for example, the

limiting values become ±0.4. Moreover the slope of the linear portion of the sigmoid function can be changed by including a multiplicative constant in the exponential. There are many such possibilities that depend largely on the problem being solved.

Number Of Training Vectors And Their Dimensions [8]

Unfortunately, there is no single definition that applies to all cases regarding the *proper choice* of training-vector pairs and their *sufficient number* of them for the BPN. As with many aspects of neural-network systems, experience is the best teacher. As facility with using networks is gained, an appreciation for how to select and prepare training sets is also gained. Thus only few guidelines are given here. In general one can use as many data as available to train the network, although all may not be needed. From the available training data, a small subset is often all that is needed to train a network successfully. The remaining data can be used to test the network to verify that the network can perform the desired mapping on input vectors it has never encountered during training.

If the network is being trained to perform in a noisy environment, then some noisy training vectors are included in the training data set. Sometimes the addition of noise to the input vectors during training helps the network to converge even if no noise is expected on the inputs.

If a satisfactory result is not obtained with a small number of training vectors, a few more should be added. On the other hand, if good convergence and satisfactory result is obtained with the first number of training vectors chosen, it may be tried with fewer training vectors to see whether a significant speed up in convergence and still satisfactory results are obtained.

The Dimension Of The Weight Vectors And Their Initial Values [5],[6],[8]

The dimension of weight vectors are interrelated to the dimension of the training vectors (or equivalently the number of input layer nodes), the number of hidden layer and output layer nodes. For a three layer BPN, there will be two sets of weights: the *input layer-to-hidden layer weight vectors* and *the hidden layer-to-output layer weight vectors*. The dimensions of the weight vector sets are determined on the basis of performance criterion

since they are interrelated with the network architecture which are themselves determined by experimentation. Since generally more weights mean longer training times and more weights does not always result in a better solution [8], the weight dimensions should be balanced against other factors, such as the acceptability of the solution.

Weights should be initialized to small, random values -say between ±0.5 or less [6], [8].

The Bias Term Or Weight [5], [6], [8]

As described earlier, including the bias term sometimes helps convergence of weights to an acceptable solution. It is perhaps best thought of as [8] an extra *degree of freedom,* and its use is largely a matter of experimentation with the specific application. Assumed to be included, say, in the hidden layer, the bias term, θ_i , that appear in the equations for the net input is set and fixed to a constant value of 1. Thus, for computational convenience it is common practice to treat this bias value as another weight, which is connected to a fictitious unit that always has an output of 1. To see how this scheme works, recall Eq. (3.18) :

$$net^{o}_{pk} = \sum_{j=1}^{L} w^{o}_{kj}\, i_{pj} + \theta^{o}_{k}$$

By making the definitions, $\theta_k^{\circ} \equiv w^{\circ}_{k(L+1)}$ and $i_{p(L+1)} \equiv 1$, we can write

$$net^{o}_{pk} = \sum_{j=1}^{L+1} w^{o}_{kj}\, i_{pj} \qquad \text{.....(3.45)}$$

So θ_k° is treated just like a weight, and it participates in the learning process as weight, that is its value also changes in the learning process. Similar modifications may be made for bias terms on the output layer.

Alternatively the bias terms may simply be removed altogether since their use is optional.

The Learning Rate η And The Momentum Parameter

(i) The Learning Rate η

The function, property and limit value of this parameter is discussed in Sec 3.3.2. The value of this parameter has a significant effect on the network performance, both in training time and acceptability of the solution [8] and it was stated in Sec 3.3.2. that experience is the

best teacher as to its selection. Usually, η must be a [6], [8], [11] small number- on the order of 0.05 to 0.25- to ensure that the network will settle to a solution. A small value of η means that the network will have to make a large number of iteration, but that is the price to be paid if accuracy of solution is desired.

It is often possible to increase the size of η as learning proceeds [6], [8], [11]. This is because, as training proceeds, the error value will diminish (hopefully), resulting in smaller and smaller weight changes, and hence, in slower convergence toward the minimum of the weight surface.

Increasing η as the network error decreases will often help to speed convergence by increasing the step size as the error reaches a minimum, but the network may bounce around too far from the actual minimum value if η gets too large.

(ii) The Momentum Parameter α [5], [6] & [8]

Another way to increase the speed of convergence is to use a technique called *momentum*. When calculating the weight-change value, $\Delta_p w$, we add a fraction of the *previous* change. This additional term tends to keep the weight changes going in the same direction- hence the term momentum.

The new weight-change or update equations on the output layer then become for the output layer,

$$w^{o}_{kj}(t+1) = w^{o}_{kj}(t) + \eta\delta^{o}_{pk} i_{pj} + \alpha\Delta_p w^{o}_{kj}(t-1) \quad \text{.....(3.46)}$$

and for the hidden layer,

$$w^{h}_{ji}(t+1) = w^{h}_{ji}(t) + \eta\delta^{h}_{pj} x_{pi} + \alpha\Delta_p w^{h}_{ji}(t-1) \quad \text{.....(3.47)}$$

In the above two equations, α is the momentum parameter, and usually set to a positive value less than 1. The use of the momentum is optional like the bias term.

Determination Of Stopping Condition Of Learning [5],[8]

The question of when to stop training is largely a matter of the requirements on the output of the system. The amount of error that can be tolerated on the output signal is determined, and a network is trained until the observed error is consistently less than the required value. Since the mean squared error is the value used to derive the training algorithm, that is the quantity that usually determines when a network has converged to its minimum error solution. Alternatively, observing individual errors is often necessary, since the system performance may have a requirement no error exceed a certain amount. Nevertheless, a mean squared error that falls as the iteration number increases is probably a best indication that the system is converging toward a solution.

3.3.6 Few Points on Neural Network Analysis [6]

It is not enough that a neural network can be trained to solve a problem of interest. The question invariably arises : " How does the network do it ?" In other words what problem solving strategy did the network discover for the problem of interest ? Unfortunately, there are no strict guidelines for performing a network analysis.

Because feed forward neural networks, such as the BPN, use their hidden layer nodes to form internal representations of the map that the network learns, analysis is mostly done on them to understand how the network uses them.

Distribution of Hidden Layer Nodes Activity Levels

Analysis of the distribution of activity levels of hidden layer nodes gives some insight as to how they classify a given categorization task. In this procedure following the training, all the training patterns are presented to the network and histograms of each hidden layer nodes activity levels are plotted. The hidden layer nodes activation level distribution or histograms are then compared with the performance of the network and an interpretation is given as to which hidden nodes detect which features of the patterns. Unfortunately, the interpretation of this analysis is simple only for patterns having a small number of discriminant features.

Analyzing Weights in Trained Networks

It is sometimes possible to gain some understanding of the network's strategy by examining the weights and weight patterns after training is complete. In general, if a given input node has weights for relatively high magnitude fanning out from it to the node of the hidden layer, then that parameter may play a relatively important role in the network decision-making process. If the magnitudes of normalized inputs to a given input node are sufficiently large, then it is more likely that large fan-out weights from the input node will be meaningful. If, however, because of input normalization or some other reason, the input magnitudes are small, it is less likely that significance can be attached to the weight magnitudes.

CHAPTER 4. THE PROJECT IMPLEMENTATION DETAILS

In this chapter the neural network implementation details of character recognition, which is the objective of the paper, will be presented and described.

4.1. Functional Modules Description

The project has three main functional modules: *the detection, the learning and the recognition modules* which is shown below in the block diagram of Fig. 4.1.

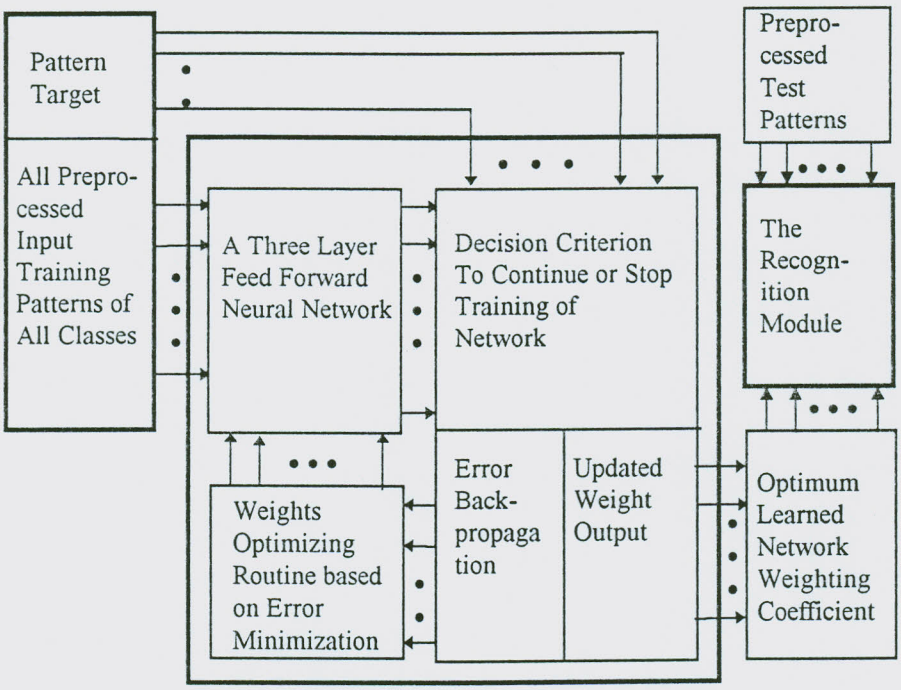


Figure 4.1 Functional block diagram representation of the project main parts

4.1.1 Training and Test Patterns Classification, Representation, & Pre-processing

Input Pattern Classification

Character pattern classes to be used by the network are first selected and formed with each class having representative training and test character patterns chosen among the available number of font styles. In this case two sets of training and testing pattern classes were formed.

Table 4.1. The Set of Randomly Chosen Sample Printed Pattern Classes

Font type/ Item	Times New Roman Train	Arial Train	**Britanic Bold Train**	Century Gothic Train	Albertville Test	Footlight MT Light Test	Pattern Class Name
1	A	A	A	A	A	A	ClassA
2	C	C	C	C	C	C	ClassC
3	Q	Q	Q	Q	Q	Q	ClassQ
4	f	f	f	f	f	f	Class-f
5	m	m	m	m	m	m	Class-m
6	w	w	w	w	w	w	Class-w
7	5	5	5	5	5	5	Class-5
Font type / Item	Geez Train	AGF Rejim Train	Geezigna Train	AGF Yeji Tsihuf Train	Geezigna Test	AGF Zemen Test	Patern Class Name
8	ሀ	ሀ	ሀ	ሀ	ሀ	ሀ	Class ሀ
9	ለ	ለ	ለ	ለ	ለ	ለ	Class ለ

The first set of pattern classes, shown in Table 4.1, are formed from 9 randomly selected individual character pattern classes of 3 upper case English characters, 3 lower case English characters, one numeric character and 2 Geez characters. Each pattern class has four training

patterns and two testing patterns chosen among about the 40 available fonts representing the English characters and among the 7 available fonts representing the Geez characters.

The second set of pattern classes, shown in Table 4.2, are formed from 6 very similar character pattern classes of 2 upper case English characters, 2 lower case English characters, and 2 Geez characters.

Table 4.2. The Set of Very Similar Sample Printed Pattern Classes

Font type/ Item	Times New Roman Train	Arial Train	**Britanic Bold Train**	Century Gothic Train	Albertville Test	Footlight MT Light Test	Pattern Class Name
1	C	C	C	C	C	C	ClassC
2	G	G	G	G	G	G	ClassG
3	f	f	f	f	f	f	Class-f
4	t	t	t	t	t	t	Class-t
	Geez Train	AGF-Rejim Train	Geeezigna Train	AGF-Eeji Tsihuf Train	Geeezigna Test	AGF-Zemen Test	Patern Class Name
5	ለ	ለ	ለ	ለ	ለ	ለ	Classለ
6	ሰ	ሰ	ሰ	ሰ	ሰ	ሰ	Classሰ

Input Pattern Detection, Representation & Pre-processing

Each training and testing pattern of all the pattern classes is scanned or detected and represented as a PCX image file, which is compressed to optimize storage disk space and each such PCX image file is first opened and then unpacked, uncompressed or decoded into its original representation so that further manipulations and processing are possible on the image. These further manipulations and processing in this case are the pre-processing and storage of each unpacked PCX character image file.

Since each character pattern is scanned as a binary image, i.e. each pixel is represented as either black ("0") or white ("1"), its uncompressed image is uniformly sampled such that the number of samples equals the number of nodes on the input layer of the neural network and stored as a different text file for easy access by the network. In appendix A, the Borland C++ language coding is given for opening, uncompressing, and sampling of a PCX image file and then storing it as a numeric text file.

Organization of The File Containing All Training Patterns

All the uncompressed, sampled and stored training patterns of all pattern classes are organized and combined together along with their associated targets in a single file such that the pattern classes are uniformly or randomly dispersed in this single file so that the neural network learns all the patterns without any bias from one or few classes. Otherwise, if patterns of one class are clustered together, the neural network may forget that class when it learns the other subsequent classes. In this paper this single file is organized in such a way that one member of each pattern class is alternatively arranged uniformly starting from the first class through the last class.

4.1.2 The BPN Learning The Training Patterns

Network Architecture and Parameters

The neural network implemented in this paper is the Back Propagation Network (BPN) which is a three-layer feed forward network to be trained using the supervised learning mode.

The first layer is the input layer and is used to distribute or propagate training patterns to the subsequent layers. The number of nodes or neurons on it, determined only by experimentation, equals the number of sample points of a training pattern that gives satisfactory performance both in, or a compromise between, speed of convergence and accuracy of solution.

The second and third layers are the hidden layer and the output layer which play the role of classification or generating decision boundaries used in the categorization of the training patterns into their true classes via the adaptive adjustment and convergence of the input-to-hidden layers

and the hidden-to-output layers weighting coefficients to an acceptable values determined by the output layer SSE error value at the last iteration.

The output layer outputs are also used to monitor the learning performance of the network at each iteration. The number of nodes on the output layer equals the number of pattern classes to be learned by the network, the *i*th node being assigned to the *i*th character pattern class. Its value is 9 for the first set and 6 for the second set of pattern classes.

The number of nodes on the hidden layer is determined in conjunction to the number of input layer nodes, output layer nodes and other network parameters together with the evaluation of the compromise between learning performance(i.e., convergence speed) and test performance (i.e., acceptance of the solution arrived).

Initial input-to-hidden and hidden-to-output layers weighting coefficients, learning rate, momentum rate, the error limit and the bias term are the other network parameters included and experimented.

How the BPN Learns Or Is Trained

First an initial network architecture and parameters, such as the number of input-, hidden-, and output layer nodes, initial input-to-hidden and hidden-to-output layers weighting coefficients as well as learning rate, momentum rate, and error limit are determined and/or assumed and set in the software simulation, in this paper a Borland C++ coding, module of the BPN. The software module then accesses all the training patterns and their associated targets from the organized file containing them all, previously pre-processed, and copies the training patterns into an input array memory buffer and the associated corresponding targets into another input array memory of the module.

Training Pattern Feed-Forwarding To The BPN

In the first cycle of its iteration, the module sends the first training pattern data, actually a vector, of the first class into the input layer of the BPN. Each component of this first vector data is then propagated to the hidden layer weighted by the initial input-to-hidden layers weighting

coefficients corresponding to each node in the hidden layer. At each node of the hidden layer, the corresponding weighted node input components are summed and this sum is then thresholded to a value between 0.0 and 1.0 by a sigmodal thresholding output activation function. Each such thresholded hidden layer node sum, collectively forming a hidden layer output vector data, is then propagated to the output layer weighted by the initial hidden-to-output layer weighting coefficients corresponding to each node in the output layer. Here again, at each node of the output layer, the corresponding weighted node input components are summed and this sum is then thresholded to a value between 0.0 and 1.0 by a sigmodal thresholding output function.

The sum of the squares of the errors, i.e. the difference between the expected target and the actually computed result, for all output units (SSE) is then computed at the output layer.

Error BackPropagation and Weight Updating

The SSE at the output layer forms the objective or cost function to be minimized. The adaptive weight updating equations for the input-to-hidden and hidden-to-output layers depends on the negative gradient of this SSE and we use an approximation for the gradient that can be determined from information that is known explicitly at each iteration. Moreover, since SSE is directly related to the hidden-to-output layers weighting coefficients and indirectly to the input-to-hidden layers weighting coefficients, its gradient or minimization with respect to these two sets of weighting coefficients gives the in-/de-cremental values to be added to the previous corresponding weights producing new updated sets of weighting coefficients at the hidden-to-output and input-to-hidden layers.

Now that the hidden-to-output and the input-to-hidden layers weighting coefficients are updated, the second training pattern, say the first training pattern from the second class, along with its corresponding target is fed to the input of the BPN and propagated through the network giving a new SSE as above. The SSE is then manipulated as above to give the new updated sets of weighting coefficients at the hidden-to-output and input-to-hidden layers.

In this way the cyclical training pattern feed- forwarding to the BPN and error backpropagation for weight updating iteration continues until an acceptable error limit is reached.

Training Period of The BPN

The training period of the BPN simulator on the IBM, Aptiva PC with a 66 Mhz CPU and a 16 Mbyte RAM is about 37 for the first set and 28 minutes for the second set of pattern classes.

Generation and Storage of the Trained Optimum Weights

The last updated optimum input-to-hidden layers and the hidden-to-output layers weighting coefficients generated from the adaptively trained BPN at the last iteration point are finally stored in a single text file to be used then by the recogniton module which is discussed in the next section. In appendix B, the Borland C++ language coding or simulation of the BPN is given. The function of this coding is represented in the block diagram of Fig. 4.1 as the center heavy-line bounded rectangle which itself is functionally divided into four parts: *training pattern forward propagation part, decision making part to stop or continue training, error back propagation and weight optimizing part,* and *optimized weight output part.*

4.1.3 The Recognition or Run/Test Module

The recognition module is the main and ultimate objective of the project. It is used for optimum classification or categorization of test input character patterns into their respective true classes. It has two main parts: *pattern propagation part* and *a look up table (LUT) part.*

The Pattern Propagation Part

This part or sub-module is a three-layer (input, hidden & output layers) feed forward network having the same architectural and some network parameters as the BPN. It has the same number of input-layer nodes, hidden-layer nodes, output-layer nodes, bias terms, sigmoidal thresholding function and so on as the BPN but no backpropagation or learning parameters. Its input-to-hidden layers and hidden-to-output layers weighting coefficients are those fixed optimum corresponding weight sets generated by the BPN in its learning mode and stored in a single text

file which are accessed by the recognition module each time it is used to recognize a given test character pattern.

During testing or use, a pre-processed test character pattern *without* an associated target is propagated through this part weighted (successively by the optimum input-to-hidden layers and the hidden-to-output layers weighting coefficients), summed and thresholded at each node of the hidden and output layers. The final result at the output layer nodes are then input to the LUT part.

The Look Up Table (LUT) Part

This part or sub-module categorizes an input test character pattern as belonging to an existing or no class based on an acceptable difference value between an expected target and the output value at the output layer as a result of propagation of the test character pattern through the pattern propagation part or sub-module.

The LUT is designed such that if a propagated given test character pattern activates (node output ≥ 0.75 in this paper) one and only one node of the output layer and all remaining nodes are deactivated (node output ≤ 0.35), then that character is recognized as or belongs to the pattern class assigned to that activated node; otherwise the test character is unrecognized or belongs to none of the available pattern classes.

Overall Process Time Required by the Recognition Module

Unlike the BPN which takes about 37 minutes in its training of nine classes of the first set and 28 minutes for the second set of six classes, the recognition module takes an overall of only about one second (1.2029 Sec. for the recognition module of the first set and 1.1657 Sec. for the second set) to recognize a test character pattern and output the result.

Designer Interaction After Each BPN Training Phase

The performance or recognition rate of the recognition module is evaluated and based on this evaluation, the BPN as well as the recognition module architecture and parameters, and/or the input training and testing pattern pre-processing are readjusted until an overall good performance of the character recognition system is obtained. This, of course, is the crucial but tedious part of the project since due to the relatively lengthy training period of the BPN, the interaction with the simulator to adjust and modify parameters and then see the results and performance is considerably slowed down. The solution to this problem is attempted by minimizing the number of pattern classes as well as the number of training patterns in each class to only few significant quantities, and taking only sufficient samples or pixels of each character pattern from its scanned form.

4.1.4 Summary of Implementation Details

The following summary are arrived and given after a number of interactions have been made until a satisfactory performance of the recognition module is obtained.

(i) Input patterns

. Each pre-processed training or test pattern is 10(Width) × 12(Height) = 120 pixel points each pixel being either black(0) or white(1).

. Two Sets Of Pattern Classes are used in the experimentation.

The First Set

- Is formed from 9 pattern classes consisting of randomly selected (3)upper case & (3) lower case English (Latin), one Arabic numeral and (2) Geez characters as shown in table 4.1.
- Each pattern class has 4 training and 2 test patterns. The two test patterns to be used are made pure, noisy or tilted (rotated).
- All the training patterns of all pattern classes of this set are organized and combined together along with their associated targets in a single file in such a way that one member of each pattern class is alternatively arranged periodically starting from the first class

through the last class.

- From this same set 9 pattern classes each having 7 patterns were formed in which four of them are pure training patterns, one of them is 5% noisy training pattern , and two of them are pure, noisy or tilted test patterns.

The Second Set

- Is formed from 6 pattern classes consisting of very similar (2) upper case and (2) lower case English (Latin) and (2) Geez characters as shown in table 4.2 in page..
- Each pattern class has 4 training and 2 test patterns. Here the two test patterns may either be pure or noisy.
- The single file organization containing the training patterns and their corresponding targets is similar to that of in the first set.

(ii) The BPN Network

Architecture

- Has three (input, hidden and output) layers and bias units at the hidden & output layers.
- The input layer has 120 input nodes to which either 0 or 1 is propagated.
- The output layer has 9 nodes for the first set of pattern classes and 6 for the second.
- The hidden layer has 40 nodes for both sets of pattern classes.
- The output thresholding function at each node in the hidden and output layers is sigmoidal.

Parameters

- The learning rate eta (η) is 0.075 for both sets of pattern classes.
- The momentum factor alpha(α) is 0.002 for both sets of pattern classes.
- The initial input-to-hidden and hidden-to-output layer weighting coefficients are set to Gaussian random numbers bounded between ± 0.45 for both sets of pattern classes.
- The Errorlimit(ε) is 2.5×10^{-8} for both sets of pattern classes.
- The maximum number of training iterations is set to 2000.

(iii) The Recognition Module

Propagation Part

- Has the same architecture as that of the BPN network.
- Has the optimum input-to-hidden and hidden-to-output layer weighting coefficients generated by the BPN in its learning mode.

The LUT Part - Is a simple sequential search algorithm.

4.2 SIMULATION RESULT

To observe the state of the learning process by the BPN, it will be helpful to see some sample outputs representing the main parameters such as the output layer outputs at the start and end of learning iterations and weight coefficients verses iterations generated by the Borland C++ BPN simulator software at a given epoch iteration in its learning mode.

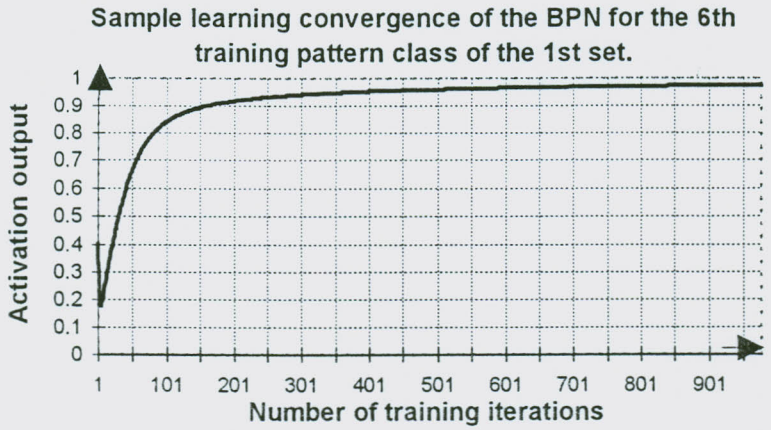
	ClassA	ClassC	ClassQ	Class-f	Class-m	Class-w	Class-5	Class-υ	Class-ʌ
Node1	0.3632	0.4888	0.4296	0.4104	0.3145	0.3247	0.3808	0.2767	0.1909
Node2	0.6583	0.5723	0.6813	0.6136	0.5567	0.4557	0.3477	0.4700	0.4688
Node3	0.6095	0.5878	0.5762	0.5925	0.4745	0.3858	0.3731	0.3242	0.4409
Node4	0.3694	0.3788	0.2660	0.2136	0.3486	0.3440	0.2455	0.4206	0.2514
Node5	0.3593	0.3896	0.3472	0.3137	0.3170	0.2677	0.2299	0.2977	0.2685
Node6	0.6593	0.5232	0.4650	0.4492	0.4833	0.3639	0.3328	0.3658	0.3465
Node7	0.4010	0.3803	0.3569	0.3485	0.3244	0.3724	0.2032	0.1637	0.2559
Node8	0.3465	0.3057	0.2524	0.2164	0.1842	0.2184	0.2264	0.2228	0.2577
Node9	0.7917	0.7202	0.6817	0.6442	0.5612	0.6307	0.4509	0.4223	0.3870

Table 4.3 The activation output at all nodes of the output layer at the first training iteration for the first 9 training patterns one from each class.

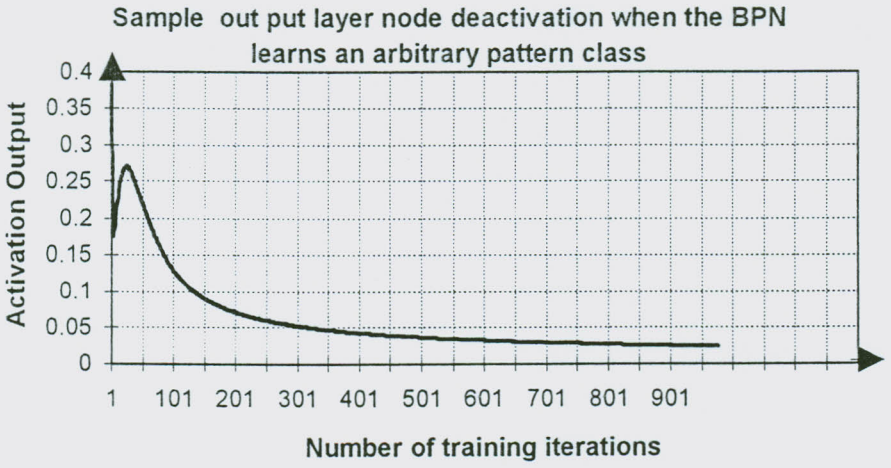
	ClassA	ClassC	ClassQ	Class-f	Class-m	Class-w	Class-5	Class-υ	Class-ʌ
Node1	0.9780	0.0003	0.0001	0.0068	0.0104	0.0135	0.0139	0.0007	0.0083
Node2	0.0004	0.9833	0.0092	0.0049	0.0006	0.0012	0.0028	0.0076	0.0025
Node3	0.0001	0.0033	0.9835	0.0006	0.0028	0.0101	0.0126	0.0129	0.0081
Node4	0.0065	0.0063	0.0038	0.9789	0.0154	0.0056	0.0031	0.0000	0.0029
Node5	0.0109	0.0014	0.0070	0.0081	0.9829	0.0105	0.0011	0.0009	0.0001
Node6	0.0151	0.0005	0.0054	0.0147	0.0024	0.9827	0.0021	0.0012	0.0012
Node7	0.0053	0.0034	0.0085	0.0091	0.0002	0.0022	0.9881	0.0034	0.0023
Node8	0.0005	0.0069	0.0066	0.0000	0.0112	0.0032	0.0126	0.9854	0.0053
Node9	0.0094	0.0022	0.0010	0.0048	0.0032	0.0061	0.0027	0.0093	0.9839

Table 4.4 The activation outputs as in Table 4.3 but at convergence point.

Table 4.3 and table 4.4 show the output layer outputs at the start and end of learning iterations respectively of the randomly chosen set of pattern classes. As can be observed from table 4.4 that only one node representing a pattern class activates (converged to 1.0) while all other nodes are deactivated (converged to 0.0) at the end of the training process. This can be observed in relation to the initial activation outputs as shown in table 4.3. The next curves in Fig. 2(a) and (b) best show the learning convergence of the neural network for the 6th pattern class of the first set.



(a)



(b)

Fig. 4.2 Sample plots of the values of the output layer activation output as the BPN is learning the training pattern classes.

The curves in Fig. 4.3 and Fig. 4.4 below show the convergence performance of samples of an input-to-hidden and a hidden-to-output layer weighting coefficients versus training iteration while learning the set of very similar pattern classes for eta = 0.085, alpha = 0.005 and the number of input and hidden layer nodes are 120 and 36 respectively.

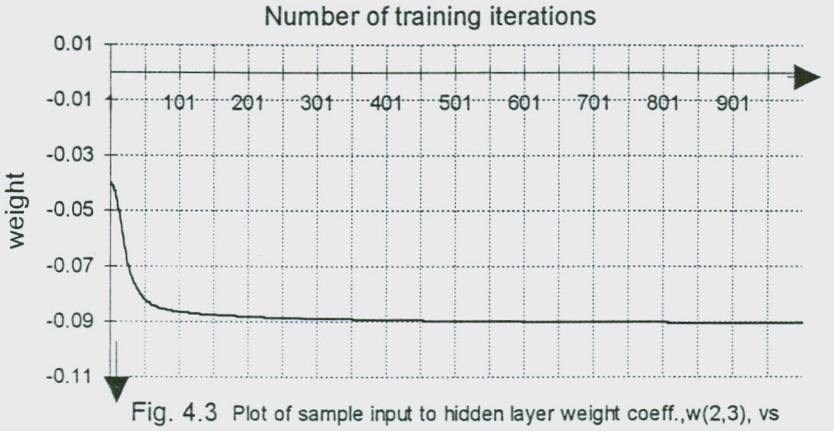


Fig. 4.3 Plot of sample input to hidden layer weight coeff.,w(2,3), vs number of training iterations

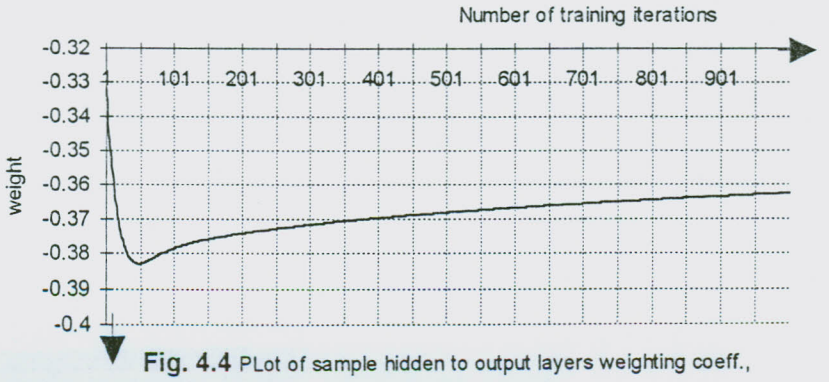
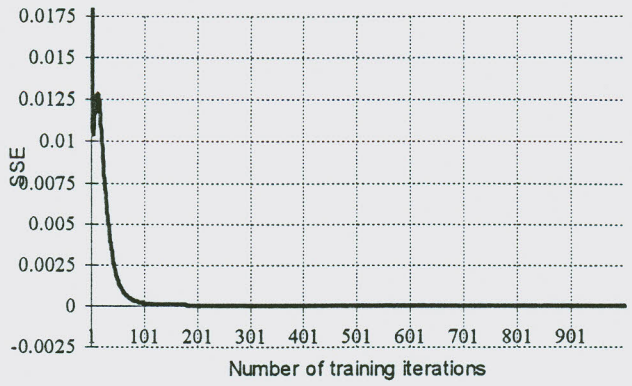


Fig. 4.4 PLot of sample hidden to output layers weighting coeff., w(5,3), vs training iteration

The next plot, Fig. 4.5., also shows how the sum squared error (SSE) converges to zero as the network learns for similar pattern classes and network architecture and parameters.

Fig. 4.5 Sample plot of SSE vs training iterations.
eta = 0.085 and alpha = 0.005



Recognition Performance of the Recognition Module

The recognition performance of the recognition module here is measured in terms of the percentage of recognition *only* of the test character patterns which may be either of the three types or sets: character patterns already prepared for testing, a noise corrupted version of them, and their tilted version. All these tests are used to evaluate the "generalizing" capability of the BPN on the untrained test patterns and implicitly its "memorizing" capability as well.

Table 4.5. Recognition performance of the BPN for pure, nosy and tilted test character patterns.

Pattern Class Set/ Recognition Rate *With*		Set1 (Randomly Chosen)	Set 1B (Randomly Chosen)	Set 2 (Very Similar)
Pure Test Patterns		85 %	-	66.7%
Noisy Test Patterns With Shown Noise Levels	2.5%	77.8%	Unaffected	-
	5.0%	66.7%	77.8%	Unaffected
	7.0%	66.7%	77.8%	58.3%
	8.5%	55.5%	66.7%	66.7%
	10.0%	55.5%	61.1%	58.3%
	12.5%	50.0%	55.5%	41.7%
	15.0%	33.3%	38.9%	33.3%
	17.5%	33.3%	38.9%	33.3%
	20.0%	27.8%	16.7%	-
Rotated Test Pattern with angle from the vert., in deg	2.2	Unaffected	Unaffected	-
	4.4	Unaffected	Unaffected	-
	10.9	66.6%	66.6%	-
	19.8	Unrecognizable	Unrecognizable	-

Keys:

1. Set1 is the run module (with parameters M = 40, η = 0.075, α= 0.002) trained with pure, randomly chosen training patterns only.
2. Set1B is the run module (with parameters M = 42, η = 0.095, α= 0.005) trained with pure *plus* noisy randomly chosen training patterns.
3. Set 2 is the run module (with parameters M = 40, η = 0.075, α= 0.002) trained with pure & very similar training patterns.

As shown in table 4.5, the performance rated 85% for the randomly chosen set of pattern classes and 67% for the set of very similar pattern classes with the first test set. Assuming the recognition rate is 97% for both sets over the training patterns only, the overall recognition rate accounting both the training and test patterns is 93% for the first set and 87% for the second.

A steady degradation was observed for both sets with the noisy test sets. The performance of the randomly chosen set with the tilted test sets was unaffected up to an angle of 4.4 degrees from the vertical.

A similar table of performance rate, previously measured with pure test patterns only, as a function of network parameters, i.e., network size (the number of hidden layer neurons M, the number of input and output layer neurons being constant), learning rate (η) & momentum term (α) values and the maximum bound ($\pm w_{max}$) of the initial weight coefficients is also shown in Table 4.6 for few sets of these parameters.

Table 4.6. Recognition performance of the BPN for different sample network parameters sets.

	Network Parameters Set				Recognition Rate (in %)	
Sample Trial	M	η	α	w_{max}	Run Module From Set 1 (Trained from pure training sets)	Run Module From Set 2
1	36	0.085	0.005	±0.50	83.5	62.7
2	40	0.075	0.002	±0.45	85.0	67
3	45	0.075	0.002	±0.30	83.5	65

CHAPTER 5. DISCUSSION , CONCLUSION AND RECOMMENDATION

Discussion

The BPN trained on the two sets of pattern classes - one set randomly chosen and the other set formed from very similar patterns performed satisfactorily. The recognition rate on the randomly chosen set of patterns is 85% and on the very similar patterns set is 67% both tested *only with* the test patterns. The overall recognition rate would be higher if the test were made with both the training sets as well as the test sets; for instance assuming the recognition rate is 97% for both sets over the training patterns only, the overall recognition rate accounting both the training and test patterns is 93% for the first set and 87% for the second, however, what here desired is to rate the generalizing capability of the BPN directly and its memorizing capability implicitly.

The network was also tested with various percentage noise corrupted test patterns and its recognition performance degraded steadily when the noise percentage on the patterns is increased.

Tilted test patterns were also tolerable upto 4.4 degrees from the vertical.

It is noted here beforehand in the discussion that there are no formalized or generalized easily implementable rules present currently as to how to change the network architecture and parameters as well as the dimensionality of the input pattern vectors and number of training patterns in a class that give a satisfactory network performance. To do so, the neural network field is currently under intensive research in neural systems mathematical analysis and formulation, algorithm development, implementation or synthesis and performance measurement in both software and hardware modes. Empirical methodologies, generalizations and guidelines often based on qualitative analysis are therefore mostly used to properly modify, and then test and measure the performance of the trained network. Thus, experimentation were conducted with different values of the dimension of input vectors, number of hidden layer neurons, the initial input-to-hidden and the hidden-to-output layers weighting coefficients, the learning parameter(η) and momentum term(α) to obtain optimal empirical sets of these parameters of the network that give an optimally performing recognition module in relation to the general empirical guidelines.

In this respect, the performance was observed to improve when the above parameters were varied sometimes one at a time and at another two or more of them simultaneously.

From the experimentation a satisfactory performance was found when the number of hidden layer neurons was around 40, the initial weights were uniformly distributed random numbers bounded between around $\pm$ 0.45, η is 0.075 and α is 0.002.

It was observed that small values of the learning parameter slows down the optimal weight convergence as monitored via the SSE display on the screen of the computer but is supposed to give a more precise learning. On the other hand, a relatively large learning parameter speeds convergence but with relatively non smooth learning. The momentum term has also played the role of enabling to use a compromise between a large value of the learning parameter which gives a faster convergence and a desired accuracy which can otherwise be obtained from a small value of the learning parameter.

The dimensionality of the input character pattern vectors or equivalently the number of input layer nodes was also varied. First, an 8 $\times$ 8 (= 64 pixels) matrix representation of the pattern vectors was used in both the training and testing modes. But this dimension was found insufficient to represent the character patterns precisely and as a result, the performance of the network is relatively not good although a relatively higher training speed is possible. On an 8 $\times$ 10 matrix representation, a relative performance improvement, which may be attributed to the relatively precise representation of the patterns, was obtained although the training speed reduced in compensation. Finally an 10 $\times$ 12 matrix representation was used and the network as expected performed better although again the training speed reduced further.

Since a relatively good performance is observed with input pattern vectors represented in a 10 $\times$ 12 matrix, no further attempt was made to increase the dimensionality of the vectors; instead, other parameters were varied and their effects were observed. This is because even for the 10 $\times$ 12 matrix, it takes an average training time of about 37 minutes for the first set and 27 minutes for the second set of pattern classes on the IBM PC, Aptiva with 66 MHz CPU and 16 Mb RAM. This limits one significantly to interact continuously and efficiently with the BPN simulator software in the modify-train-test-measure performance cycle.

Neither the BPN was trained on exhaustive number of pattern classes nor the number of training patterns in a given pattern class was very large. What was aimed is that, if the network perform very well on a relatively significant representative number of patterns, it can easily be scaled up to perform on a complete set of characters either from the English alphabets (about 40 font styles), numeric characters, special symbols, Geez characters(about 7 font styles) , or a combination of two or more of them.

When the number of training patterns in a pattern class is increased, the network "generalizing " capability and its performance increases [8]. However, when there are significant number of similar pattern classes in the training and testing sets, the network may misclassify the similar pattern classes as observed, and as a result the network performance degrades. But under the presence of insignificant number of similar pattern classes, increasing the number of training patterns in a class improves the generalizing capability of the network.

The generalizing capability is proved not only by testing the network with test patterns not used in the training but also with noise corrupted forms and tilted versions of these test patterns.

One other important parameter that should be taken in to account in the design and implementation of neural network applications for character recognition is the speed of the recognition module to recognize a given test character pattern. This will be significant when a page of large number of characters are scanned, document analysed and fed sequentially to the recognition module to be subsequently recognized. If it takes significant time for one character, it obviously will take quite a long total time to recognize all characters in the document. Thus, to circumvent this problem, an integrated design approach should be used accounting tolerable speed of training, satisfactory performance in both accuracy and speed of recognition and easy implementation.

Using a single precision number rather than a double precision number not only gives a satisfactory accuracy in computation but also makes the training and testing/running times to reduce significantly since a single precision number requires half the memory size required by a double precision number.

The performance was greatly affected by problems associated with the scanning procedure. If the input character patterns are misaligned by some horizontal and/or vertical lines, a complete rejection or misclassification of the pattern resulted. This is partly due to the low dimensionality of each pattern vector and the small number of training patterns in a given class and it can be significantly rectified by increasing the above parameters.

Conclusion

A steady improvement in performance was observed as the dimension of the input pattern vectors is increased and the network was accordingly tuned as its size is increased. The

scanning procedure may also be improved by using automatic means rather than using manual adjustment which is usually prone to error. For this, of course, the PC paintbrush graphics application would have been a good solution but in the final analysis it is the scanner that will be directly used in OCR systems to input and document analyze the scanned page of characters which after that are input to the recognition module for further processing.

Recommendation

The recognition module developed from the trained BPN may be used as the main module or layer in the design of a complete automated OCR system that converts a scanned text document into a word processor document for editing purposes. The main additions to it to be made are an input interface part which is a document analyser whose main function is to open a scanned document image file and convert each character in the document into suitable numeric data to be sequentially input to the recognition module for recognition, and an output interface part which invokes a word processor to store all the recognized characters and a representative character symbol of all the unrecognized characters as a word processor document for further editing. This completion as well as enhancing the BPN may be carried out by other researchers.

This same BPN may also be implemented by tuning its network architecture and parameters to recognize other temporal or spatial patterns such as speech, medical signals (such as EEG and ECG), finger print and so on after proper pattern pre-processing is done on the patterns.

Appendix A: The C++ coding for opening, decoding, sampling and storing a PCX image file.

```
/*************************************************************************
! Develpment Environment : Turbo C++, ver. 3.0,                          !
!                  Copy right 1990,92,                                   !
!                  Borland International, Inc.                           !
! A C++ program (SAMPLECH.CPP) to read (scan) in and unpack a pcx image   !
! file and then sample and store it under a different file name so that   !
! it can be used by the BPN NN as an input at its input layer.            !
!                                                                         !
! To use this program, execute it and enter valid command line arguments !
! at the DOS command prompt, say, c:\users\tezazu>, e.g :                 !
!                                                                         !
! SAMPLECH \epscan2\amh_l1.pcx \users\tezazu\dtfdir\amhl1.bin             !
!                                                                         !
! Or you may put this as it is in a batch file for doing the same task.   !
!                                                                         !
! Note: SAMPLECH - is the executable form of this program (SAMPLECH.exe). !
!                  i.e. SAMPLECH = argv[0].                               !
!          amh_l1.pcx - is a scanned pcx image file of the amharic letter "li"  !
!                  in the directory c:\epscan2, where PC paintbrush       !
!                  graphics editor & application software resides.        !
!                   i.e. \epscan2\amh_l1.pcx = argv[1] .                  !
!          amhl1.bin - is the sampled vesion of the above pcx file        !
!                  stored in the datafile directory "c:\users\tezazu\dtfdir"  !
!                  as a text file (integer content representing the letter  !
!                  feature or pattern in 0 and 1 ) to be used by the ANN  !
!                  as its input for learning or testing.                  !
!                  i.e. \users\tezazu\dtfdir\amhl1.bin = argv[2] .        !
!                                                                         !
*************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <alloc.h>
#include <dos.h>
#include <mem.h>
#include <conio.h>
#include <graphics.h>
#include <process.h>
#include <ctype.h>

//---- P R O T O T Y P E   F U N C T I O N S ---/

int  ReadPcxLine(char *q, FILE *fp);
void UnpackPcxFile(char *q, char ch1, FILE *fp);
void Threshold();
void DisplayImage();
void DisplayTextImage();
void DsplyHderParm();
void SaveSampledImag(char *img,char *pp);
void breakout(char *s);
```

```
void  SetPalets();

//-------------- Global Variables --------------/
typedef  struct {
                    char manufacturer;
                    char version;
                    char encoding;
                    char bits_per_pixels;
                    int  xmin,ymin;
                    int  xmax,ymax;
                    int  hres,vres;
                    char palette[48];
                    char reserved;
                    char colour_planes;
                    int  bytes_per_line;
                    int  palette_type;
                    char filler[58];
                     } PCXHEAD; // a PCX file header
PCXHEAD      header;  // a pcx image file header
                      // (parameters) structure

int bytes;   // No. of  bytes in one horiz. line of a pcx img file.
             // to hold the graphics information on that line.
int width,depth;   // The width & depth of a pcx image in pixels.
int nXsampls,nYsampls;  // Desired no. of sample pixels in
                        // the x- & y-dir.'s.
int width_sf,depth_sf;      // x- & y-dir. image sampling factors.

unsigned long size;       // Memory size for sampled image
                          // storing buffer.
char far *p;              // Memory buffer array to store sampled image.
unsigned int  qbytebits[100 /* = #bytes * 8 bits */];
                          // a temporary array to hold all uncompressed
                          // bytes in one pcx image line.
unsigned int  qbyte;      //  a temp. var. to copy one
                          //  byte info. in an image line.
unsigned int  mask[8] = {0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
                          // Used to mask out the information in
                          // each of the 8 bits of a byte.

FILE        *fp;  // a file pointer variable

//-----------------------------------------------------/
//-Reads, decodes and stores one line of a pcx
// image file into buffer q[].

int ReadPcxLine(char far *q, FILE *fp)
 {
          int n = 0,c,i;
//         memset(q,0,bytes);  // null the buffer q
          _fmemset(q,0,bytes);
          do {
                    c = fgetc(fp) & 0xff;        //-Get a key bytes.
                    if ((c & 0xc0) == 0xc0){  // If it's a run of bytes
```

```
                    i = c & 0x3f;              // field and of the high
                    c = fgetc(fp);             // bit get the run of byte
                    while(i--) q[n++] = c;     // run of byte
                    }
                  else q[n++] = c;             // else just store it
          }
         while(n < bytes);
         return(n);
}
//-------------------------------------------------------------/
// UNPACKS AND SAMPLES A PCX IMAGE FILE
// Reads,uncompress,samples and stores all
// pcx file lines into buffer p[] after
// which it can be displayed &/or saved
//under a different filename.

void UnpackPcxFile(char *q,char ch1, FILE *fp)
{                  // open and print
      int i,j,k,x,b,bb = -1;

      if (ch1 != 'y' && ch1 != 'n') {
          puts("\n Please enter y,Y,n or N and no other.");
          exit(1);
        }
      if (ch1 == 'y') {      // if yES ,i.e,if sampling is desired
         printf("\n Enter desired horiz.img sampls, nXampls [From 1 to %d{= %d*bytes(8) - %d}] :
                    ",width,bytes,8*bytes -width);
         scanf("%d",&nXsampls);
         printf("\n Enter desired vert.img sampls, nYampls [From 1 to %d] : ",depth);
         scanf("%d",&nYsampls);
         depth_sf = (int)((float)depth/(float)nYsampls);
         width_sf = (int)((float)width/(float)nXsampls);
          }
         else {  // if (ch1 == 'n')
             nXsampls = width ;
             nYsampls = depth;
         }
         if ((p =(char far *) farmalloc(8*size)) != NULL) {
               for(i = 0; i < nYsampls; i++) {   // 8*size->byte=8bits
                    //ReadPcxLine(q,fp);  /*read lines in inverse order*/
                   if (ch1 == 'y')
                      for(k = 0; k < depth_sf - 1; k++) {
                         q += bytes;
                         ReadPcxLine(q,fp); //n=ReadPcxLine(q,fp);
                        }
                    ReadPcxLine(q,fp);

          //  if (8*bytes < width) {  // i.e if image has 1bit/pixel (=1byte/8pixels)
                                      // uncompresses each byte of an img line
                                      // into 8 diff. bit infor. giving (8 * bytes-per-line)
                                      // of pixel information.
                   for(b = 0; b < bytes; b++) {
                      qbyte = q[b];
                         for(j = 0; j < 8; j++) {
```

```
                        int AllBytesBits = b * 8 + j;
                        qbytebits[AllBytesBits] = (qbyte & mask[j]) >> 7-j;
                        if (AllBytesBits == width - 1) break;
                      }
                }
/*   }
          else {   // if (8*bytes >= width), i.e if image
                   // has 8bits/pixel(=1byte/pixel)
                   for(b = 0; b < bytes; b++)
                       qbytebits[b] = q[b];
                }            */

//-------------------------------------------------------------
          // copies uncompressed & img samples into
          // buffer p[] for later use.

          for(x = 0; x < nXsampls; x++)
                   p[i*nXsampls + x] =(ch1=='y')?qbytebits[width_sf * x]:qbytebits[x];
         } // for of i loop
          free(q);
          free(qbytebits);
          }
         /*else if(bb) {
                   perror("\nNo enough memory. Thus it'll be sampled.");
                   bb = ~bb;
                   UnpackPcxFile(q,'y',fp);
         } else breakout(": sampling is also impossible1"); */
          else { perror("\nNo enough memory for size. Don't exceed the smaple ranges.");
                  exit(1);
                }
}
//-----------------------------------------------------/
//--Thresholds the Buffer content of p[]
// Sets the numeric value of each pixel of an
// image either black ( 0 ) or white ( >=15 ),
// [ signed FF = 255 or -1 in this case ].Each
// pixel of a B&W(2 gray level)image is 1 bit.

void Threshold()
{
  int i,j;          unsigned int m1,m2,m3,thresh;
 // printf("Enter a threshold value to hard limit Blk & Wht:  ");
 // scanf("%u",&thresh);
   thresh = 1;
   for(j = 0; j < nYsampls; j++)
      for(i = 0; i < nXsampls; i++) {
         m1 = p[j*nXsampls + i] & 0xff;   // m1 is set to an int b/n 0 & 255(-1)
         m2 = p[j*nXsampls + i] & 0x00;  // m2 is set to 0 (int)
         m3 = (p[j*nXsampls + i] | 0xff)/*>> 0*/ ;  // m3 is set to -1 (sgnd int)
         p[j*nXsampls + i] = (m1 < thresh) ? m2 : m3;
         } // depending on the thresh value, the
          // buffer(p[]) content is thresholded either
 }         // to 0(blk) or -1 (255) (wht)
//-------------------------------------------------
```

```
//---------------------------------
void breakout(char *s)
 {
          if (p != NULL) farfree(p);
          if (fp != NULL) fclose(fp);
          if (*s == '.') puts(s + 1);
          exit(0);
 }
//-----------------------------------------------/
// The VGA graph driver in the VGAHI mode divides
//  the screen with 640 by 480. This will result in
//  square pixel because the ratio of the horizontal
//   size of the screen to that of the vertical
//  (i.e the aspect ratio) is 4 to 3.

void SetPalets()
  {
      int gd = VGA, gm = VGAHI, errorcode;
                    // gd = graphdetector, gm = graphmode
       //int gd = DETECT, gm, errorcode;      // request auto detection
           initgraph(&gd,&gm,"c:\\tc\\bgi");  // initialize graphics mode
           errorcode = graphresult();             // read result of initialization
           if (errorcode != grOk) // an error occurred
             {
                 printf("Graphics error: %s\n", grapherrormsg(errorcode));
                 printf("Press any key to halt, tzex ! :");
                getch();
                exit(1);      // return with error code
             }
           setpalette(15,63);   setpalette(14,55);
           setpalette(13,31);    setpalette(12,30);
           setpalette(11,22);    setpalette(10,51);
           setpalette(9,19);     setpalette(8,53);
           setpalette(7,29);     setpalette(6,28);
           setpalette(5,49);     setpalette(4,25);
           setpalette(3,56);     setpalette(2,16);
           setpalette(1,40);      setpalette(0,0);
 }
//-------------------------------------------------------------/
// Displays the unsampled or sampled character
// image in graphics mode.

void DisplayImage()
 {
          int x,y;

          clrscr();
          printf(" Please wait ...\n");
          delay(250);
          SetPalets();  // Initializes graphics & set palettes
          for(y = 0; y < nYsampls; y++)
            for(x = 0; x < nXsampls; x++)
                    putpixel(x+300,y+210,p[y*nXsampls + x] >> 0); // >>0 4
          getch();
```

```
        closegraph();
}
//-------------------------------------------------------/
//--Displays the unsampled or sampled character
// image in text mode.

// Each pixel of a B & W  (2 gray level) image is
// one bit & thus the buffer holding it should be
// set either to 0 or 1[= abs(-1) or abs(signd FF=255)]
// by the thresholder so that a text mode bitmapping
// of the sampled character image is displayed.

void DisplayTextImage()
{
        int i,x;

        clrscr();
        puts("Sampled & Thresholded Character Display.\n");
        printf("   ");
        for(x = 0; x < nXsampls; x++)
                printf("%d",x);
        printf("\n");
        for(i = 0; i < nYsampls; i++) {
                printf("%d   ",i);
                for(x = 0; x < nXsampls; x++)
                  printf("%d",abs(p[i*nXsampls + x] >> 0));//>>1,4
                printf("\n");
         }
}
//-------------------------------------------------------/
// Displays the pcx image file header parameters.

void DsplyHderParm()
{
        clrscr();
        printf(" Display of image file header parameters\n");
        printf("----------------------------------------\n");
        printf("manufacturer = %d \n",header.manufacturer);
        printf("version = %d \n",header.version);
        printf("encoding = %d \n",header.encoding);
        printf("horiz.resol = %d ,  vert.resol = %d \n",header.hres,header.vres);
        printf("bits_per_pixels = %d \n",header.bits_per_pixels);
        printf("bytes_per_line = %d \n",header.bytes_per_line);
        printf("colour_planes = %d \n",header.colour_planes);
        printf("xmin   = %d , xmax    = %d , width  = %d  \n",header.xmin,header.xmax,width);
        printf("ymin   = %d , ymax    = %d , depth  = %d \n",header.ymin,header.ymax,depth);
        printf("bytes = %d, size(=8*bytes*depth) = %d \n",bytes,8*size);

        printf("\n User defined image file specification\n");
        printf("----------------------------------------\n");
        printf("x smpls = %d , y smpls = %d, Total smpl pixels=
%d\n",nXsampls,nYsampls,nXsampls*nYsampls);
        printf("width_sf = %d , depth_sf = %d \n",width_sf,depth_sf);
}
```

```
//-----------------------------------------------/
// Saves the sampled pcx image file in to a
// disk file of desired filename

 void SaveSampledImag(char *img,char *pp)
 {
          int i,j;    FILE *fps;

          if ((fps = fopen(img,"w")) != NULL)  {
                    for(j = 0; j < nYsampls; j++)  {
                     for(i = 0; i < nXsampls; i++)
                              fprintf(fps,"%1.0f ", (float)abs(pp[j*nXsampls + i]));
                             // fprintf(fps,"%d ", abs(pp[j*nXsampls + i]));
                     fprintf(fps,"\n");   // Returns to a new line
                                          // after nXsampls of data.
                    }
                    fclose(fps);
          } else puts(" Error creating desination");
 }
//!!!  MAIN  FUNCTION!! ----/
int  main(int argc, char *argv[])
 {
          FILE *fp;
          char *q;
          char ch0;

          if (argc >= 3) {
           if ((fp = fopen(argv[1],"rb")) != NULL) {    //attempt to open the file
//         if ((fp = fopen("c:\\epscan2\\g1.pcx","rb")) != NULL) {    // test for debug
           if (fread((char *) &header,1,sizeof(PCXHEAD),fp)==sizeof(PCXHEAD)) { // read the header
                    if (header.manufacturer == 0x0a) {  // check if it is a picture
                              width = (header.xmax - header.xmin) + 1;
                              depth = (header.ymax - header.ymin) + 1;
                              bytes = header.bytes_per_line;
                              size = (long) bytes * (long) depth;    //Allocate a big buffer.
                              if ((q = (char far *) farmalloc(bytes/*width*/)) != NULL) {
//                                  clrscr();
                                  printf("\n\nDo you want to sample the image? (y/n) : ");
                                  ch0 = tolower(getch());
                                  UnpackPcxFile(q,ch0,fp);
                                  Threshold();
                                  DsplyHderParm();
                                  printf("\nPress any key to see the sampled char.image in text mode : ");
                                  getch();
                                  DisplayTextImage();
                                  SaveSampledImag(argv[2],p);
                                  printf("\nPress any key to see the sampled image in graphics mode : ");
                                  getch();
                                  DisplayImage();
//                                  free(p);
                              }
                    }
```

```
                    else {
                            printf(" %s is not a pcx file !\n",argv[1]);
                            fclose(fp);
                            return(0);
                        }
                }
                else {
                        printf("Error reading %s .\n",argv[1]);
                        fclose(fp);
                        return(0);
                       }
                 }
                 else {
                        printf("Error opening %s .\n",argv[1]);
                        return(0);
                              }
          }
         else {
          perror("\n Please enter at least 3 valid cmd line arguments ! ");
          cprintf("e.g: smplimg5 \\epscan2\\g1.pcx \\users\\tezazu\\dtfdir\\g1.bin \n");
          return(0);
          }
         return(1);
}
```

Appendix B: The C++ coding of the Back Propagation Network (BPN) simulator.

```
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
!^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^!
!        Image pattern recognition using (Artifitial) Neural Network (ANN or NN).   !
!                                                                                   !
!        Image to be recognized: alpha-numeric characters.                          !
!        ANN model used        : the Back Propagation Network (BPN).                !
!        Purpuse of the ANN    : for image pattern classification using             !
!                                a supervised iterative learning algorithm.         !
!        Network Architecture  : The ANN has one input layer, one hidden layer      !
!                                 and one output layer.                             !
!                                                                                   !
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <dos.h>

#define Pmax 36 //= no. of clases * train. ptrns/class
                //= Total no. of all classes' traininig patterns
#define EpochIterations 2000
#define N  120 // No. of feature points of a
               // given sample pattern  of a class
               // given to the input layer of the ANN.
#define M 40   // Number of hidden layer neurons (nodes) of ntk
#define L 9    // Number of output layer neurons (nodes) of ntk.
               //  Should equal to no. of classes to be learned.
#define ErrorLimit  0.0000005

#define wmax 0.45  //0.3

/*------- Global variables -------------- */

int  i;           // Loop & array indexes
int  j;
int  k;
int  p;

float sum;         // Temp. variable

float Xin[Pmax][N];  // i/p matrix feature of a sample
                     // pattern of a given class.
float wihl[M][N];    //wgt. coeff. of input-hidden layers of ntk.
float whol[L][M];    //wgt. coeff. of hidden-output layers of ntk.
float xhl[Pmax][M];  // Propagated & proccessed feature of the
                     // above sample patrn at the o/p of the hidden layer.
```

```
float Ytarget[Pmax][L]; // desired(target) output of
                        // ntk used as a "Teacher".
float Yout[Pmax][L];    // actual computed output of
                        // ntk at its o/p layer.
float err_ol[Pmax][L];  // error term output of the
                        // ntk output-layer.
float err_hl[Pmax][M];  // error term output of the
                        // ntk hidden-layer.
float SSE[Pmax];        // The Sum Squared Error due to one
                        // train. pattern of a gvn class.
float wgt_ith[1000/*EpochIterations*/];
                          //sampl i/p to hid. wgt to plot wgt vs itera.
float wgt_hto[1000/*EpochIterations*/];
                          //sampl hid. to i/p wgt to plot wgt vs itera.
#define  eta    0.095 // Ntk learning rate.
#define  alpha  0.05  // ntk momentum rate.
#define  beta   1.0   // sigmoid function enhancing
                      // (desaturation) parameter.
#define nos_per_line 12  // no. of numeric data items
                         // in one line of a file.

/*------------------- Function prototypes ------------------- */

 float ran(long *iseed); // uniform random number generator
 float ran2(long *iseed,int *init);
 float gran2(float mean,float sigma,long *iseed,int *init);
/*-------------------------------------------------------------..----*/
/*-----The uniform random no. generator ran() subroutine--*/

#define a 25173
#define c 13849
#define m 65536

float ran(long *iseed)
 {
        *iseed=(a**iseed + c)%m;
        return((float)*iseed/(float)m);
 }
/************************************************
*------- ran2() function, a shuffled version of ran() ----------*
* Generates a BETTER uniformly distributed random number *
*-----------------------------------------------------------------------*/
#define  k1 100
float ran2(long *iseed,int *init)
 {
        static float v[k1],
        y;        int i;

        if(*init==0) {
                for(i = 0; i < k1; i++)
                  v[i] = ran(iseed);
                y = ran(iseed);
                *init = 1;
                }
```

```
        i = k1 * y;
        y = v[i];
        v[i] = ran(iseed);
        return y;
       }
/*****************************************************
*--A Gaussian distributed random no. generator gran2() subroutine--*
*                                                                   *
* Generates a white Gaussian noise of a given mean & variance       *
*-------------------------------------------------------------------*/

float gran2(float mean,float sigma,long *iseed,int *init)
 {
        float u;  // float u = 0.0;
        int i;

        u = 0.0;
        for(i = 0;i < 12;i++)
                u= u + ran2(iseed,init);
        return (sigma*(u-6) + mean);
 }

//*****************************************************

void main(void)
 {
  long  *iseed,t1,t3; // Random number generator terms for weighting coeff.
  int   *init,t2,t4;  // initialization & test inputs to the ntk
  float mean,sigma;

  FILE *fpPaterns,  // File where patterns of a class are stored;
                    // may also contain the targets.
                   *fpOptWgts,  // All learned optimum weights
                                // storage file for a class
                   *fpFnlActvOut, // Final activation value,Yout[][],
                                  // o/p storage file of a clas
                   // *fpChTest;    // Scannded,sampled & stored
                                  // test char file to be recognized
                   *fpWgtVsItrIH,  // File to store sampl wihl[][]
                                   // vs. iteration for plotting
                   *fpWgtVsItrHO;  // File to store sampl whol[][]
                                   // vs iteration for plotting
//                 *fpFrstActv;     // File to store actvation o/p's
                                   // of o/p layer at 1st itra.

        char AllPaternsFil[60];  // All i/p training patterns
                                 // file name for an epoch.
        char OptWgtOutFil[60];   // File name of the optimum
                                 // weights o/p after training
        char FnlActvOutFil[60];  // File name of the final activ.
                                 // values o/p after training


        //-------------------------------------------------
```

```
// Sets initial weights of hidden (wihl[M][N]) layer
// & output(whol[L][M]) layer to a random number b/n
// -wmax and wmax. wmax = 0.3 to 0.55
//-------------------------------------------------
//-Initial values of the i/p-hidden & hidden-o/p layers
// weighting coef.

for(i = 0; i < M ; i++) { //Initial Weight coef. mtrx of
 mean = 0.0; //-0.1;    // I/p-hidden Layer- wihl[M][N]
 sigma = 0.32;//0.13;
 t1 = 37;            iseed = &t1;
 t2 = 15;   init = &t2;
 for(j = 0; j <= N ; j++)
       //  wihl[i][j] =  gran2(mean,sigma,iseed,init);
          wihl[i][j] =  wmax*(1.0 - 2.0* float(rand()/32767.0));
}

for(k = 0; k < L; k++) { // Initial Weight coef. mtrx of
 mean = 0.0;          // Hidden-o/p Layer- whol[L][M]
 sigma = -0.15;
 t3 = 13;            iseed = &t3;
 t4 = 29;   init  = &t4;
 for(j = 0; j <= M ; j++)
        // whol[k][j] = gran2(mean,sigma,iseed,init);
          whol[k][j] = wmax*(1.0 - 2.0* float(rand()/32767.0));
}
//-------------------------------------------------------------------
// Reads all patterns and learning targets corresp.
// to each pattern of a given class to be learned
// by the NN in to its i/p Xin[Pmax][N].


//                                          "          argv[1]      "
//      if ((fpPaterns = fopen(AllPaternsFil,"r")) != NULL)  // exit(1);
      if ((fpPaterns = fopen("c:\\users\\tezazu\\dtfdir\\al36ptrn.inp","r")) != NULL)

 for(p = 0; p < Pmax ; p++) {
          Xin[p][N] = 1.0;  // Bias input to i/p layer of nntk
          for(i = 0; i < N ; i++)
                fscanf(fpPaterns,"%f",&Xin[p][i]);

          // read in target outputs for each input pattern read

          for(j=0;j < L ; j++)
                fscanf(fpPaterns,"%f",&Ytarget[p][j]);
         }
fclose(fpPaterns);
```

```
// ! THE ANN LEARNING MODULE ! !

        // ------ Signal Propagation to ANN----------

for(int epoch = 0; epoch < EpochIterations ; epoch++) {

        for(p = 0; p < Pmax ; p++)
        {       //For all Pmax patterns of all classes.
                //Propagation from input to hidden layer

                for(j = 0; j < M; j++) {//j is index of hdn lyr nodes
                 sum = wihl[j][N];
                 for(i = 0; i < N ; i++)
                         sum = sum + wihl[j][i] * Xin[p][i];
                 xhl[p][j] = 1.0/(1.0 + exp(-beta*sum));
                }
                xhl[p][M] = 1.0; // Bias input to o/p layer of nntk


        //Propagation from hidden to output layer

        for(k = 0; k < L; k++) {
                 sum = whol[k][M];
                 for(j = 0; j < M ; j++)
                         sum = sum + whol[k][j] * xhl[p][j];
                         Yout[p][k] = 1.0/(1.0 + exp(-beta*sum));
                }

        //-----   Error back propagation ----------

                // error terms to be backpropagated to the output layer

        for(k = 0; k < L; k++)
                 err_ol[p][k] = (Ytarget[p][k] - Yout[p][k]) * Yout[p][k] * (1.0 - Yout[p][k]);

                // error terms to be backpropagated to the hidden layer

        for(j = 0; j < M; j++) {
                 sum = 0.0;
                 for( k = 0; k < L; k++)
                         sum += err_ol[p][k] * whol[k][j];
                 err_hl[p][j] = xhl[p][j] * (1.0 - xhl[p][j]) * sum;
                 }

        //--- Weight adjustments (network learning phase) ------

                         // hiddden-to-output layer
                         // weight coefficient adjustment

        for(k = 0; k < L; k++) {
                 for(j = 0; j <= M; j++)
                        whol[k][j] += eta * err_ol[p][k] * xhl[p][j] + alpha * err_ol[p][k]*xhl[p][j];
                 }
```

```
                        // input-to-hiddden layer
                        // weight coefficient adjustment


            for(j = 0; j < M; j++) {
                    for(i = 0; i <= N; i++)
                            wihl[j][i] += eta * err_hl[p][j] * Xin[p][i] + alpha * err_hl[p][j]*Xin[p][i];
                    }

 //-- Sum of Squared Errors (SSE) for  evaluation of
 //   learning convergence of the ntk.

            for(sum = 0.0, k = 0 ; k < L; k++) {
                    sum += err_ol[p][k] * err_ol[p][k];
                   }
            SSE[p] = 0.5 * sum;
            sum = 0.0;
           } // Pmax number of loops;
            // Here now on you can use p as an index!

           wgt_ith[epoch] = wihl[2][3]; // Sample i/p to hid. layer
                                        // wgt for ploting vs trainin itr.
           wgt_hto[epoch] = whol[3][5]; //Same for hid. to o/p wgt

 // Storing activation value  Yout[][] of o/p layer
 // at 1st iteration point.

/*         if (epoch == 0)
                   if ((fpFrstActv = fopen("c:\\users\\tezazu\\dtfdir\\actv36y0.out","w")) != NULL) {
                    for(j = 0; j < L; j++)
                            for(i = 0; i < Pmax-27; i++)
                                    fprintf(fpFrstActv,"%3.4f%c", Yout[i][j],j % 10 == 10 - 1 ? '\n':' ');
                           }
                   fclose(fpFrstActv);       */

           //--------------------
           // Dispalys epoch iteration no. & corresponding
           // o/p layer error value every 5 iterations.

           if (epoch % 5 == 0)
            {
              clrscr(); gotoxy(0,2);
              printf("Number of EepochIterations = %d \n",epoch);
              printf("Corresp. o/p layer SSE     = %20.16f : \n",SSE[0/*Pmax-1*/]);
//            printf("Error out-put at each o/p layer of %d nodes, Ytarget[][] - Yout[][]\n\n",L);
//            for(k = 0 ; k < L; k++)
//               printf("%3.4f ",Ytarget[0/*Pmax-1*/][k] - Yout[0/*Pmax-1*/][k]);
//               printf("I/p to hidden layer wgt coeff wihl[j=0][i] conecting the 0th hdn lyr neuron \n\n");
//               for(i = 0 ; i <= N; i++)
//                     printf("%3.4f ",wihl[0][i]);

/*            printf("\n\nHidden to o/p layer wgt coeff whol[k=0][j] conecting the 0th o/p lyr
```

```
                    neuron\n\n");
                for(j = 0 ; j <= M; j++)
                    printf("%3.4f ",whol[0][j]);*/
                printf("\n\nANN LEARNING PROGRESSIVELY WITH O/P,
                    Yout[10][10] = Yout[%d clases by 1 of %d coding]\n\n",L,L);
                for(i = 0; i < Pmax-27; i++)  printf("  clas%d",i+1); printf("\n");
                for(j = 0; j < L; j++) {
                  printf("%d ",j+1);
                  for(p = 0; p < Pmax-27 ; p++)
                    printf("%3.4f ",Yout[p][j]);
                  printf("\n");
                  } printf("\n");

              } //if

            //---------------------------------------
            //-Quit learning phase when error iterated is
            // in acceptable limit

            if (SSE[0/*Pmax*/] < ErrorLimit) {
                        clrscr();
                        printf("\The SSE is %10.8f in %d epochiterations\n",SSE[0/*Pmax*/],epoch);
                        printf("Since SSE is below error limit = %10.8f,learning or iteration "
                                    "is satisfactory.  So quit learning !\n",ErrorLimit);
                        printf("Press any key to quit and return to program.");
                        getch();
                        break;   // gets out of loop Pmax.
                        }
} // end of EpochIterations

//---------------------------------------------------------------------
// Module to store sample input to hidden(wihl[2][3]) and
// a hidden to output (whol[3][5]) connection weights to
// plot them vs iteration no. for observation purposes

    if ((fpWgtVsItrIH = fopen("c:\\users\\tezazu\\dtfdir\\wgtvsitr.ith","w")) != NULL)  {
        for(epoch = 0; epoch < 1000/*EpochIterations*/; epoch++)
            fprintf(fpWgtVsItrIH,"%5.6f \n",wgt_ith[epoch]);
                }
fclose(fpWgtVsItrIH);

    if ((fpWgtVsItrHO = fopen("c:\\users\\tezazu\\dtfdir\\wgtvsitr.hto","w")) != NULL)  {
        for(epoch = 0; epoch < 1000/*EpochIterations*/; epoch++)
            fprintf(fpWgtVsItrHO,"%5.6f \n",wgt_hto[epoch]);
                }
        fclose(fpWgtVsItrHO);
```

```
//-----------------------------------------------------------------
//-- Stores the optimum wgt coeff's learned in to a file.

//                                  "      argv[3]       "
//          if ((fpOptWgts = fopen(OptWgtOutFil,"w")) != NULL)  {
            if ((fpOptWgts = fopen("c:\\users\\tezazu\\dtfdir\\optwgt36.out","w")) != NULL)  {

               for(j = 0; j < M; j++)  {
                    for(i = 0; i <= N; i++)
                       fprintf(fpOptWgts,"%8.6f%c", wihl[j][i],i % nos_per_line == nos_per_line-1 ? '\n':' ');
                }

               for(k = 0; k < L; k++)  {
                   for(j = 0; j <= M; j++)
                       fprintf(fpOptWgts,"%8.6f%c", whol[k][j],j % nos_per_line == nos_per_line-1? '\n':' ');
                   }
            }
     fclose(fpOptWgts);

//--------------------------------------------------------------------
//-- Stores the learned activation o/p's y[Pmax][L] in to a file.
//                                    "      argv[4]          "
//       if ((fpFnlActvOut = fopen(FnlActvOutFil,"w")) != NULL)  {
        if ((fpFnlActvOut = fopen("c:\\users\\tezazu\\dtfdir\\fnlAct36.out","w")) != NULL)  {

          for(p = 0; p < Pmax; p++)
             for(j = 0; j < L; j++)         // Yout[Pmax-1][j]
                fprintf(fpFnlActvOut,"%5.6f%c", Yout[p][j],j % nos_per_line == nos_per_line-1? '\n':' ');
            }
     fclose(fpFnlActvOut);

 } // of main()
```

Appendix C: The C++ coding of the Recognition or Run/Test module of the BPN

```
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^!
! The recognition or Run/Test moddule:                                      !
!                                                                           !
! This is a C++ pattern recognizing module program using the ANN (BPN)      !
! developed on a LUT basis.                                                 !
! To use this program(runmod36), execute it and enter valid command line    !
! arguments at the DOS command prompt, say, at c:\users\tezazu>, e.g :      !
!                                                                           !
! runmod36 A5test.dat \dtfdir\optwgt.out                                    !
!                                                                           !
! Or you may put this as it is in a batch file for doing the same task.     !
!                                                                           !
! Note:                                                                     !
!         runmod36 - is the executable form of this program (runmod36.exe)  !
!                    i.e. runmod36 = argv[0].                               !
!       A5test.dat - is the sampled vesion of the scanned pcx file of the   !
!                    test character "A" stored in the datafile directory    !
!                    "c:\users\tezazu\dtfdir" as a   text file (integer     !
!                    content representing the letter feature or pattern     !
!                    in 0 and 1 ) to be used by the ANN as its input for    !
!                    learning or testing.                                   !
!                    i.e. \users\tezazu\dtfdir\A5test.dat = argv[1].        !
!      optwgt.out - is the optimum weight coeffs. generated by the BPN      !
!                    simulator module in its training mode and is stored    !
!                    in the directory "c:\users\tezazu\dtfdir" in a text    !
!                    file as floating numbers.                              !
!                    i.e. \users\tezazu\dtfdir\optwgt36.dat = argv[1].      !
!                    This file first should be generated and the test       !
!                    character has also to be preprocessed before           !
!                    the recognition module can be used
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>

#define Pmax 36  // = no. of clases*training ptrns/class.
#define N  120    // No. of feature points of a given
                  // sample pattern of a class given
                  // to the input layer of the ANN.
#define M 40      // Number of hidden layer neurons (nodes) of ntk
#define L 9       // Number of output layer neurons (nodes) of ntk
                  // Should equal to no. of classes to be learned.

//------- Global variables --------------

int   i;   // Loop & array indexes
int   j;
int   k;
```

```
int   p;

float sum;        // Temp. variable
float Xtest[N];   // i/p matrix feature of a sample
                  // pattern of a gvn class.
float wihl[M][N]; // wgt. coeff. of input-hidden layers of ntk.
float whol[L][M]; // wgt. coeff. of hidden-output layers of ntk.
float Xhltest[M]; // Propagated & proccessed feature of the
                   // above smpl patrn at the o/p of the hidden layer.
float Youtest[L]; // propagated & proccessed form of Xhltest[]
                  // at the o/p the output layer.
float Yopt[L];    // actual computed output of ntk at its o/p layer.
#define  beta /*0.97*/ 1.0 // sigmoid fn enhancing
                           // or (saturation reduction)parameter.

int   n;        // Index
char testch,tempch;

//!!!!!!!!!- M A I N   P R O G R A M -!!!!!!!!!!!!!!

void main(int argc, char *argv[])
{
     FILE *fpOptWgts,    // All learned optimum weights
                         // storage file for a class.
          *fpFnlActvOut, // Final activation value,Yopt[],
                         // o/p storage file of a class.
          *fpChTest;     // Scannded,sampled & stored test
                         // char file to be recognized.

//-------------------------------------------------

          if ( argc < 2) {
                printf(" Please enter more than %d arguments at cmd line",argc);
                exit(1);
                }
//-----------------------------------------------------------
//-- Reads scanned,unpacked,sampled & stored test
//  character from a file for recognition.

//        if ((fpChTest = fopen("c:\\users\\tezazu\\aatst.bin","r")) != NULL) {
          if ((fpChTest = fopen(argv[1],"r")) != NULL) {
                for(i = 0; i < N ; i++)
                      fscanf(fpChTest,"%f",&Xtest[i]);
                Xtest[N] = 1.0;
                }
          fclose(fpChTest);

//-------------------------------------------------
//-Opens & reads the optimum weights generated by
// the BPN simulator (the learning module) so that
// the test character is propagated through this
// recognition module and the o/p is used by the LUT
// part of this module for recognition or rejection.
```

```
// if ((fpOptWgts = fopen("c:\\users\\tezazu\\dtfdir\\optwgt.out","r")) != NULL)
        if ((fpOptWgts = fopen(argv[2],"r")) != NULL)
         {
            for(j = 0; j < M; j++)
                for(i = 0; i <= N; i++)
                    fscanf(fpOptWgts,"%f",&wihl[j][i]);

            for(k = 0; k < L; k++)
                for(j = 0; j <= M; j++)
                    fscanf(fpOptWgts,"%f",&whol[k][j]);
         }
   fclose(fpOptWgts);

//-------------------------------------------------
//-Opens the final activation o/p at the o/p layer
// obtained after the BPN is trained to be used by
// the LUT for recognition purpose.

// if ((fpFnlActvOut = fopen("c:\\users\\tezazu\\dtfdir\\fnlActvY.out","r")) != NULL)  {
/* if ((fpFnlActvOut = fopen(argv[3],"r")) != NULL)  {
        for(j = 0; j < L; j++)
                fscanf(fpFnlActvOut,"%f",&Yopt[j]);
        }
   fclose(fpFnlActvOut);   */

    //Propagation from input to hidden layer

        for(j = 0; j < M; j++)  {
                sum = wihl[j][N];    //because xtest[N] = 1.0 is a bias i/p
                for(i = 0; i < N ; i++)
                        sum = sum + wihl[j][i] * Xtest[i];
                Xhltest[j] = 1.0/(1.0 + exp(-beta*sum));
               }
        Xhltest[M] = 1.0;

        //Propagation from hidden to output layer

        for(k = 0; k < L; k++)  {
                sum = whol[k][M];
                for(j = 0; j < M ; j++)
                        sum = sum + whol[k][j] * Xhltest[j];
                Youtest[k] = 1.0/(1.0 + exp(-beta*sum));
               }

//-------------------------------------------------
//- The simple look up table (LUT) used to inform
//  whether a test character pattern is recognized
//  or not.

        if (Youtest[0] >= 0.75) {
                tempch = 'A';
                printf("\n The test char is recognized as  %c  !  \n",tempch);}
        if (Youtest[1] >= 0.75) {
```

```
                tempch = 'C';
                printf("\n The test char is recognized as  %c  !  \n",tempch); }
        if (Youtest[2] >= 0.75) {
                tempch = 'Q';
                printf("\n The test char is recognized as  %c  !  \n",tempch);}
        if (Youtest[3] >= 0.75) {
                tempch = 'f';
                printf("\n The test is char recognized as  %c  !  \n",tempch);}
        if (Youtest[4] >= 0.75) {
                tempch = 'm';
                printf("\n The test char is recognized as  %c  !  \n",tempch);}
        if (Youtest[5] >= 0.75) {
                tempch = 'w';
                printf("\n The test char is recognized as  %c  !  \n",tempch);}
        if (Youtest[6] >= 0.75) {
                tempch = '5';
                printf("\n The test char is recognized as  %c  !  \n",tempch);}
        if (Youtest[7] >= 0.75) {
                tempch = 'Ha';
                printf("\n The test char is recognized as  %c  !  \n",tempch);}
        if (Youtest[8] >= 0.75) {
                tempch = 'Le';
                printf("\n The test char is recognized as  %c  !  \n",tempch);}

        if (Youtest[0] < 0.75 && Youtest[1] < 0.75 && Youtest[2] < 0.75 &&
                        Youtest[3] < 0.75 && Youtest[4] < 0.75 && Youtest[5] < 0.75 &&
                        Youtest[6] < 0.75 && Youtest[7] < 0.75 && Youtest[8] < 0.75 )
                        printf("The test character is unrecognized");

        printf("\n\nYoutest[L], trying recognition of test char!\n");
        printf("Each nn node o/p is ON if >= 0.85 and OFF otherwise.\n");
        printf("-----------------------------------------------\n");
        for(j = 0; j < L ; j++) {
                printf("%d  ",j);
                printf(" %5.6f  \n",Youtest[j]);
                }       printf("\n");
//-----------------------------------------------

 } // main
```

Appendix D: The C ++ program coding of a gaussian noise generator & corrupter.

```
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

! This program corrupts k percent of a test character pattern with a gaussian noise of a given  !
! mean and variance. The noise is added to the pure test character pattern file and then        !
! thresholded to obtain a desired percentage of corruption.                                     !
!                                                                                               !
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^


#include  <conio.h>
#include  <ctype.h>
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>
#include <dos.h>

#define N 120  // no. of gaussian random numbers
#define pi 3.1415927

void main(int argc, char *argv[])
 {
        int       i,j;
        double  v[N],meanv,sigmav; // the noise sequence and its mean & variance
        double x1,x2,ChPtrn[N];
        FILE      *fpChTst, // Scannded,sampled & stored test char file to be noise corrupted.
                  *fpNsyCh; // Noise corrupted character

         if ( argc < 2) {
                printf(" Please enter more than %d arguments at cmd line",argc);
                exit(1);
                }
         // Opens and reads scanned,unpacked,sampled & stored  pure
         // test character for corruption with noise.

         if ((fpChTst = fopen(argv[1],"r")) != NULL) {
         //if ((fpChTest = fopen("c:\\users\\tezazu\\aatst.bin","r")) != NULL) {
                for(i = 0; i < N ; i++)
                        fscanf(fpChTst,"%lf",&ChPtrn[i]);
                }
         fclose(fpChTst);

        // Generates the white gaussian noise of meanx and variance sigmax and
        // corrupts k % of the sampled test character file.
        // sigma is .25620, .30090, .33672, .36262, .38222, .42855, .48432 &
        // .51993 for 2.5%, 5%, 7%, 8.5%, 10%, 12.5%, 15% & 17.5% of the test
        // character to be corrupted(i.e, 1->0 & 0->1) respectively.
        //-----------------------------------------------------------------
        // Additive gaussian random noise generator routine; by Muller & Box.
        meanv = 0.2036;   // constant.
        sigmav = 0.58926;
        randomize;
        for(i = 0;i < 120; i++) {
                x1 = double (random(1000)/1000.0);
```

```
            x2 = double (random(2000)/2000.0);
            v[i] = meanv + sigmav * sqrt(-2.0*log(1 - x2)) * cos(2.0*pi*x1);
        }
       //-------------------------------------------------------------------
        // sigmav = sqrt(k)/10.0.  Test with k = 5%, 7%, 8.5%, 10%, 12.5% & 15%
        // Note: k should not be divided by 100.

         for(i = 0 ;i < N; i++)
            ChPtrn[i] = ChPtrn[i] + v[i];

       // Thresholds to obtain the k% corrupted file

        for(i = 0; i < N; i++ ) {
           if(ChPtrn[i] >= meanv)   ChPtrn[i] = 1.0;
           else if(ChPtrn[i] < meanv)  ChPtrn[i] = 0.0;
        }
       // Noisens k % of the sampled test char.
/*      Incr = int(120.0/double(k));
        for(i = 0; i < N; i = i + Incr ) {
            if(ChPtrn[i] == 1.0)  ChPtrn[i] = 0.0;
            else if(ChPtrn[i] == 0.0)   ChPtrn[i] = 1.0;
        } */

       // Test Display For monitoring
        clrscr();
        printf("\n The noise corrupted char. pattern is .\n\n");
        for(j = 0; j < 12; j++) {
            for(i = 0; i < 10; i++)
               printf("%d ",(int)ChPtrn[j*10 + i]);
            printf("\n");
        }
        // Stores the noise corrupted pattern into a separate file.

        if ((fpNsyCh = fopen(argv[2],"w")) != NULL)  {
            for(j = 0; j < 12; j++)  {
                for(i = 0; i < 10; i++)
                   fprintf(fpNsyCh,"%d ", (int)ChPtrn[j*10 + i]);
              fprintf(fpNsyCh,"\n"); // returns to a new line after nXsampls of data.
            }
            fclose(fpNsyCh);
        } else puts(" Error creating desination");
} //main
```

Reference

[1] Annath Sankar, [Book with Neural Networks in Chapter 11.]

[2] Assefa Dagne and Daniel Abebe, *Development of Image Primitives*, A final B.Sc. project, AAU, Electrical Engineering Department, 1993.

[3] Ben Zion Steinberg *et al.*, *A neural network approach to source localization*, J.Acoust.Soc. Am. 90(4), Pt. 1, Oct.1991.

[4] Brown, R. M., Fay, T. H. and Walker, C. L. , *Hand-Printed Symbol Recognition*, Pattern Recognition Society, Vol. 21 No. 2 pp91-118, 1988, Great Britain.

[5] Clarkson, P. M. and Stark, H., *Signal Processing Methods for Audio, Images and Telecommunications*, Academic Press, 1995, Great Britain.

[6] Ebehart, R. C. and Dobbins, R. W., *Neural Network PC Tools, a Practical Guide*, Academic Press, 198?.

[7] Eneyew Adugna, *Source Localization Using the CPN*, Neurocomputer System Design Final project, Dec. 1992. Rutgers University-Piscataway, New Jersey.

[8] Freeman, J. A. and Skapura, D. M. , *Neural Networks : Algorithms, Applications and Programming Techniques*, Addison-Wesley Publishing Co., 1992, USA.

[9] Gonzalez, R. C. & Whinth, R. E. , *Digital Image Processing*, , Addison-Wesley Publishing Co. , 1992, Massachusetts.

[10] Lippmann, R. P. , *An Introduction to Computing with Neural Nets*, IEEE ASSP Magazine, April 1987.

[11] Orphanidis, S. J. *Optimum Signal Processing: An Introduction,* Macmillan Publishing Co., 1988, U.S.A.

[12] Smolensky, P. , Mozer, M. C. and Rumelhart, D. E. , *Mathematical Perspectives on Neural Networks*, Lawrence Erlbaum Associates Publishers, 1996, U.S.A.

[13] Steven Rimmer , *Bit-Mapped Graphics*, McGraw-Hill Co., 1990, U.S.A.

[14] Therrien, C. W. *Decision, Estimation & Classification : An Introduction to Pattern Recognition & Related Topics*, 1994, U.S.A.

[15] Tou, J. T. and Gonzalez, R. C. , *Pattern Recognition Principles,* Addison-Wesley Publishing Co. Inc., 1979, Massachusetts.

[16] Wolpert, S. & Evangelia Michaeli-Tzanakow, *A Neuromine in VLSI*, IEEE Transaction on Neural Networks , March 1996, Vol. 7 No. 2.

[17] *What are Artificial Neural Networks*, Siemens Magazine on Engineering Automation, Vol. XIII, Sep/Oct 5, 1991.

DECLARATION

I, the undersigned, declare that this thesis is my original work performed under the supervision of my research advisor Dr. Eneyew Adugna and has not been presented as a thesis for a degree in any other university. All sources of materials used for the thesis have also been duly acknowledged.

Tezazu Bireda

Addis Ababa

May 1998.