

INSTANTANEOUSLY TRAINED
NEURAL NETWORKS
WITH
COMPLEX INPUTS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Pritam Rajagopal
B.E., University of Madras, 2000
May 2003

Acknowledgements

I would like to take this opportunity to thank my advisor Dr. Subhash C. Kak for all his invaluable guidance, patience and support throughout the course of this work, without which this thesis would not have been possible.

I would also like to thank Dr. Hsiao-Chun Wu and Dr. Ashok Srivastava for taking the time to be on my examination committee.

Table of Contents

| | |
|---------------------------------------------------------------------------------------|----|
| ACKNOWLEDGEMENTS | ii |
| ABSTRACT | iv |
| CHAPTER | |
| 1 INTRODUCTION | 1 |
| 2 CC4: CORNER CLASSIFICATION TRAINING ALGORITHM FOR NEURAL NETWORKS | 4 |
| 3 INPUT ENCODING | 15 |
| 3.1 Length of the Codewords | 16 |
| 4 THE PROBLEM OF INSTANTANEOUS TRAINING OF COMPLEX NEURAL NETWORKS | 19 |
| 4.1 Restrictions on the Inputs | 26 |
| 5 THE 3C ALGORITHM: INSTANTANEOUS TRAINING FOR COMPLEX INPUT NEURAL NETWORKS | 37 |
| 6 TIME SERIES PREDICTION | 53 |
| 7 CONCLUSION | 67 |
| BIBLIOGRAPHY | 69 |
| VITA | 71 |

Abstract

Neural network architectures that can handle complex inputs, such as backpropagation networks, perceptrons or generalized Hopfield networks, require a large amount of time and resources for the training process. This thesis adapts the time-efficient corner classification approach to train feedforward neural networks to handle complex inputs using prescriptive learning, where the network weights are assigned simply upon examining the inputs. At first a straightforward generalization of the CC4 corner classification algorithm is presented to highlight issues in training complex neural networks. This algorithm performs poorly in a pattern classification experiment and for it to perform well some inputs have to be restricted. This leads to the development of the 3C algorithm, which is the main contribution of the thesis. This algorithm is tested using the pattern classification experiment and the results are found to be quite good. The performance of the two algorithms in time series prediction is illustrated using the Mackey-Glass time series. Quaternary input encoding is used for the pattern classification and the time series prediction experiments since it reduces the network size significantly by cutting down on the number of neurons required at the input layer.

Chapter 1

Introduction

Neural network architectures based on the biological neuron model have been developed for various applications over the years. Though research in such architectures has wavered from time to time, their usefulness in many applications has ensured them a secure niche in AI research. Neural networks are used in speech recognition, electronics and telecommunications, aerospace, medicine, banking and financial analysis, pattern recognition and other analytical areas.

Many neural network architectures operate only on real data. But there are applications where consideration of complex inputs is quite desirable. For example, complex numbers in the frequency domain define the amplitude and phase of a process as a function of frequency. Prior complex neural network models [11, 16, 18] have generalized the Hopfield model, backpropagation and the perceptron learning rule to handle complex inputs. Noest [15, 16, 17] formulated the Hopfield model for inputs and outputs falling on the unit circle in the complex plane. Georgiou [3] described the complex perceptron learning rule. Also, Georgiou and others [2, 12, 14] described the complex domain backpropagation algorithm. More recently, work presented by Li, Liao and Yu [13] uses digital filter theory to perform fast training of complex-valued recurrent neural networks.

The training processes used by the different architectures mentioned above are iterative, requiring a large amount of computer resources and time for the training. This

may not be desirable in some applications. The corner classification approach [5, 6, 7, 8, 9] (algorithms CC1 to CC4), speeds up the training process of neural networks that handle binary inputs, achieving instantaneous training. A generalization of this approach that maps non-binary inputs to non-binary outputs is presented by Kak and Tang [10, 20].

The corner classification approach utilizes what is known as prescriptive learning, which requires that the training samples be presented only once during training. In this procedure, the network interconnection weights are assigned based entirely on the inputs without any computation. The corner classification algorithms such as CC3 and CC4 are based on two main ideas that enable the learning and generalization of inputs:

1. The training vectors are mapped to the corners of a multidimensional cube.

Each corner is isolated and associated with a neuron in the hidden layer of the network. The outputs of these hidden neurons are combined to produce the target output.

2. Generalization using the *radius of generalization* enables the classification of any input vector within a Hamming Distance from a stored vector as belonging to the same class as the stored vector.

Due to its generalization property, the CC4 algorithm can be used efficiently for certain AI problems. The results of pattern recognition and time series prediction experiments using CC4 are presented by Tang [19]. When sample points from a pattern are presented to the network, the CC4 algorithm trains it to store these samples. The network then classifies the other input points based on the radius of generalization, allowing for the network to recognize the pattern with good accuracy. In time-series prediction, some of

the samples from the series are used for training, and then the network can predict future values in the series.

This thesis generalizes the corner classification approach to handle complex inputs and presents two modifications of the CC4 algorithm. These generalizations require new procedures of weight assignment. Chapter 2 discusses the CC4 algorithm, its working and performance for different types of problems. Chapter 3 describes a new encoding scheme called the quaternary encoding, which will be used in different experiments to analyze the new algorithms. Chapter 4 presents the first generalization of the CC4 algorithm and describes its performance and limitations. Chapter 5 presents the 3C algorithm, which overcomes the limitations of the first algorithm. In the next chapter, the performance of the two algorithms is tested using the time series prediction experiment and results comparing the efficiency of the two algorithms are presented. Finally, the last chapter provides the conclusions related to the use of complex binary inputs in corner classification and the future of the 3C algorithm.

Chapter 2

CC4: Corner Classification Training Algorithm for Neural Networks

The CC4 algorithm is similar to its predecessor the CC3 algorithm, and uses the same feedforward network architecture and features like the prescriptive learning procedure and radius of generalization. Both algorithms map binary inputs to binary outputs using similar computations at each layer in the network, but the CC4 has a far better classificatory power.

There are two main differences between the CC3 and the CC4 algorithms. Firstly, in CC4 a hidden neuron is created for each and every training sample that is presented. Secondly, the output layer is fully connected, i.e. all hidden neurons are connected to all the output neurons. In the CC3 algorithm, hidden neurons are created only for training samples that produce a binary “1” output. If there are multiple output neurons, hidden neurons are created only if their corresponding training samples produce at least one binary “1” in the output vector. Also a hidden neuron is connected to an output neuron, only if the training sample that the hidden neuron corresponds to, leads to a binary “1” at that output neuron. This interconnection is assigned a fixed weight of 1. But in the CC4, since the output layer is fully connected, an appropriate weight assignment scheme is required at this layer to ensure that the right output neuron fires. Like the CC3 algorithm, the CC4 uses the binary step activation function at the hidden layer and the output layer. The activation function determines the output of a neuron in any layer, depending on whether or not the input to the neuron crosses a threshold of 0.

The different features of the CC4 algorithm are as follows:

1. The number of input neurons is one more than the number of input elements in a training sample. The extra neuron is called the bias neuron and receives a constant input (bias) of 1.
2. A hidden neuron is created for each training sample presented to the network. The first hidden neuron corresponds to the first training sample; the second hidden neuron corresponds to the second sample and so on.
3. The output layer is fully connected. All hidden neurons are connected to each and every output neuron.
4. The weights for interconnections between the input neurons and the hidden neuron corresponding to the training sample presented, are assigned using the prescriptive learning procedure.
5. For every training sample, if an input element is a binary “1”, the appropriate input interconnection is assigned a weight of 1. Similarly, if the input element is a binary “0”, an inhibitory weight of -1 is assigned to the interconnection.
6. The interconnection weight from the bias neuron to the hidden neurons for each training sample is assigned as $r - s + 1$. Here r is the radius of generalization and s is the number of ones in the sample being presented.
7. The output interconnection weight from each hidden neuron to an output neuron is assigned based on the desired output at the output neuron. If the desired output for the training sample corresponding to the hidden neuron is a binary “1”, then the weight assigned is 1. If the desired output is a binary “0” the weight assigned is an inhibitory -1.

8. The activation functions used at both the hidden and output layers, is the binary step activation function presented below. Here x is the input and y is the output.

$$y[n] = \begin{cases} 1 & \text{if } x[n] > 0 \\ 0 & \text{otherwise} \end{cases}$$

The algorithm can be stated using a set of simple **if - then** rules. The formal algorithm is as follows:

```

for each training vector  $x_m[n]$  do
     $s_m = \text{no of 1's in } x_m[1:n-1]$ ;
    for index = 1 to n-1 do                                //  $w_m[ ]$ : input weights
        if  $x_m[\text{index}] = 0$  then
             $w_m[\text{index}] = -1$ ;
        else
             $w_m[\text{index}] = 1$ ;
        end if
    end for
     $w_m[n] = r - s_m + 1$ ;

    for index1 = 1 to k do                                //  $k = \text{no of outputs } y$ 
        if  $y_m[\text{index1}] = 0$  then
             $ow_m[\text{index1}] = -1$ ;                          //  $ow_m[ ]$ : output weights
        else
             $ow_m[\text{index1}] = 1$ ;
        end if
    end for
end for

```

The working of the algorithm may be best understood when we take r as 0. Now when a training sample is presented, the vector combines with the input interconnection weights as the sum of the products of each input element with its corresponding interconnection weight element. The hidden neuron corresponding to the training sample presented receives a +1 input from all the input neurons which are presented with a binary “1”. There are s such input neurons, which receive a +1 input from the training sample since

there are s ones in the sample. Since r is zero, the input to the hidden neuron from the bias neuron is $-s + 1$ and the net input received by the hidden neuron is $s - s + 1 = 1$. The other hidden neurons will receive a negative or 0 input because the positions of the $+1$ and -1 weights don't match the positions of all the binary "1" inputs and the binary "0" inputs respectively. This ensures that only the corresponding hidden neuron fires and produces a binary "1" output since the threshold of the activation function is set as zero. Thus the similar combination of the hidden neuron outputs and the output layer weights produces the desired final outputs. The general network architecture for the CC4 is shown below.

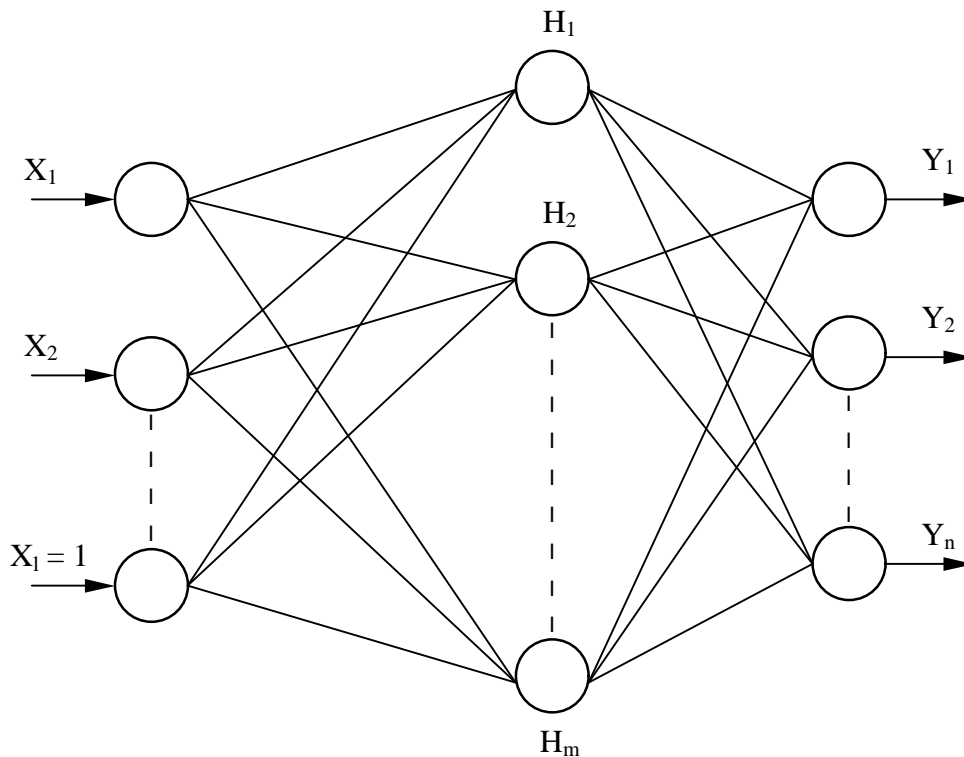


Figure 2.1: General network architecture for CC4

The working of the algorithm can be illustrated by the following examples. The first is the simple XOR example. The second example maps inputs to multiple outputs.

Example 2.1

In this example the network is trained to perform the XOR function. The truth table is shown in Table 2.1. There are two input elements in each input vector. Thus three input neurons are required including the bias neuron. No generalization is desired here. Hence the radius of generalization r is set as zero. Here all four input vectors are required to train the network. Thus four hidden neurons are used. These hidden neurons are connected to a single output neuron.

Table 2.1: Inputs and outputs for Example 2.1; XOR truth table

| Inputs | | Output |
|----------------|----------------|--------|
| X ₁ | X ₂ | Y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The weights at the input layer and output layer are assigned according to the algorithm. The input vectors are then presented one at a time and the combination of each with the weights causes only one hidden neuron to fire. For the vector (0 1 1), the number of binary ones, s , is 1, without the bias bit 1. Hence we get the weight vector to the hidden neuron corresponding to the sample as (-1 1 0). Thus the total input to the hidden neuron is $(0 * -1) + (1 * 1) + (1 * 0) = 1$. The input to the other hidden neurons is either 0 or negative. Similarly the network is trained by presenting the other vectors one after the other and then tested successfully. The network diagram is shown in Figure 2.2, and Table 2.2 contains the network parameters obtained during the training.

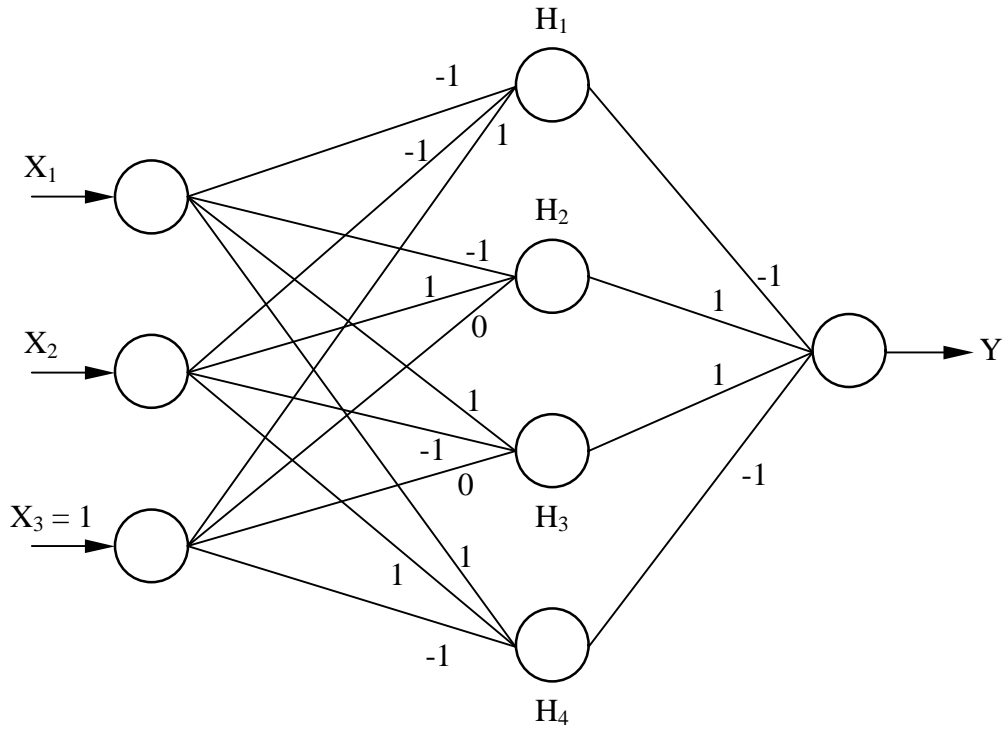


Figure 2.2: The Network Architecture for Example 2.1

Table 2.2: Network Parameters in the input/output mapping of Example 2.1

| | | | | | | | Input to | | | | Output of | | | | Input | Output |
|--------|---|---|---|---------|----|----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-------|--------|
| Inputs | | | s | Weights | | | H ₁ | H ₂ | H ₃ | H ₄ | H ₁ | H ₂ | H ₃ | H ₄ | to Y | Y |
| 0 | 0 | 1 | 0 | -1 | -1 | 1 | 1 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | -1 | 0 |
| 0 | 1 | 1 | 1 | -1 | 1 | 0 | 0 | 1 | -1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | -1 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 2 | 1 | 1 | -1 | -1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | -1 | 0 |

Example 2.2: Two-output neuron network

In this example the CC4 algorithm is used to train a network, which has two output neurons. The truth table below holds the inputs and the outputs.

Table 2.3: Inputs and outputs for Example 2.2

| Inputs | | | | | Outputs | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| X ₁ | X ₂ | X ₃ | X ₄ | X ₅ | Y ₁ | Y ₂ |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 |

Here the input vectors have five elements each. Thus six input neurons are required including the bias. All three vectors need to be used for training, thus the number of hidden neurons is three. The output vector is two bits long; so two output neurons need to be used. As each sample is presented, the network weights are assigned according to the algorithm for the input and the output layer.

The network is then tested with all of the inputs and it can be seen that the network successfully maps all the inputs to the desired outputs. The architecture of the network required to implement the input/output mapping for this example is shown in Figure 2.3, and the different parameters of the network are tabulated in Table 2.4.

These above examples show how well the network can be trained to store vectors and then associate the vectors with their appropriate outputs when the vectors are presented to the network again. However the generalization property cannot be observed since in both examples r is set to 0. This property of the CC4 algorithm can be analyzed by a pattern classification experiment. The algorithm is used to train a network to separate two regions of a spiral pattern.

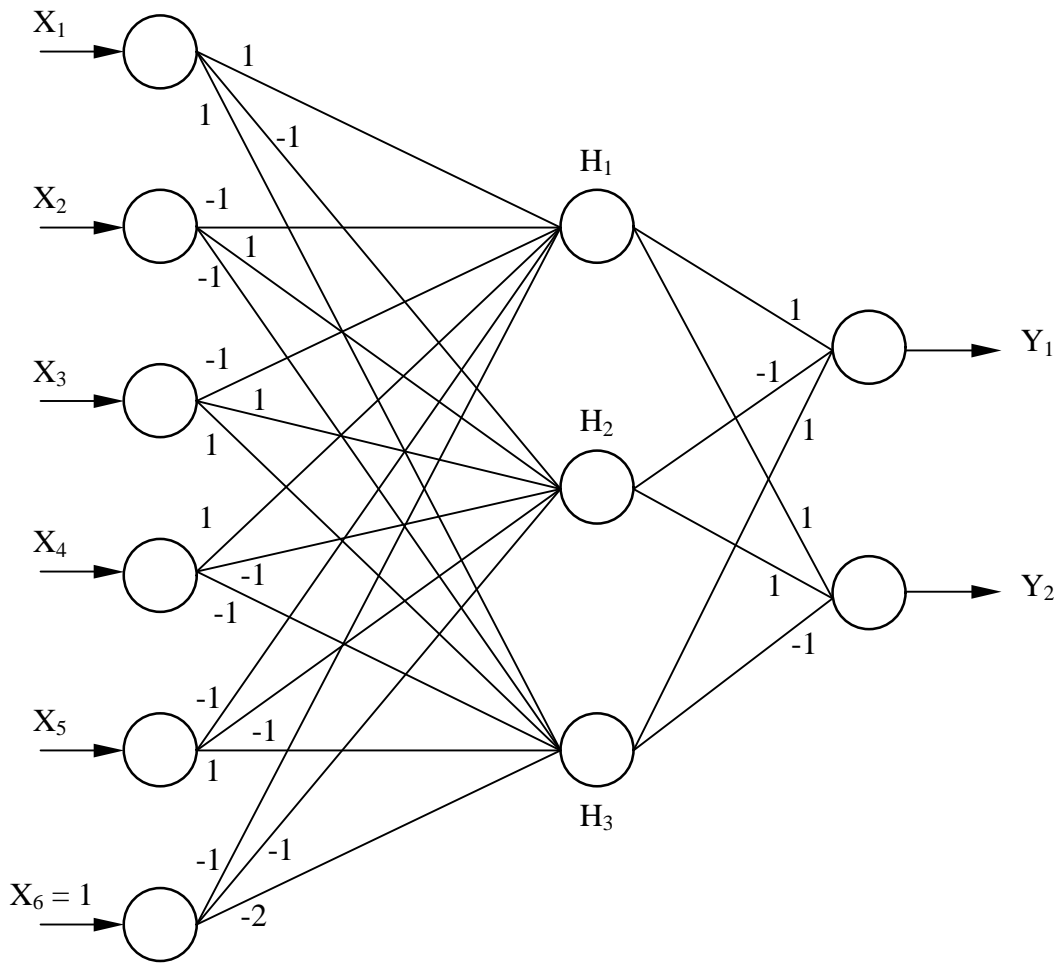


Figure 2.3: The Network Architecture for Example 2.2

Table 2.4: Network Parameters in the input/output mapping of Example 2.2

| | | | | | | Input to | | | Output of | | | Output | | | | | | | | |
|--------|---|---|---|---|---|----------|---------|----|-----------|----|----|--------|----------------|----------------|----------------|----------------|----------------|----------------|----|----|
| Inputs | | | | | | s | Weights | | | | | | H ₁ | H ₂ | H ₃ | H ₁ | H ₂ | H ₃ | y1 | y2 |
| 1 | 0 | 0 | 1 | 0 | 1 | 2 | 1 | -1 | -1 | 1 | -1 | -1 | 1 | -3 | -2 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 2 | -1 | 1 | 1 | -1 | -1 | -3 | 1 | -2 | 0 | 1 | 0 | 0 | 1 | |
| 1 | 0 | 1 | 0 | 1 | 1 | 3 | 1 | -1 | 1 | -1 | 1 | -2 | -2 | 1 | 0 | 0 | 1 | 1 | 0 | |

The original pattern is shown in Figure 2.4 (a). The pattern consists of two regions, dividing a 16 by 16 area into a black spiral shaped region and another white region. A point in the black spiral region is represented as a binary “1” and a point in the white region is represented by a binary “0”. Any point in the region is represented by row and column coordinates. These coordinates, simply row and column numbers, are encoded using 16 - bit unary encoding and fed as inputs to the network. The corresponding outputs are 1 or 0, to denote the region that the point belongs to.

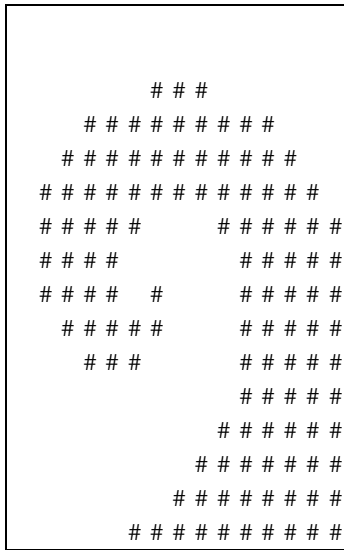
The unary encoding used converts numbers 1 through 16 into strings, each of length 16 bits. As an example the number 1 is represented by a string that has fifteen 0’s followed by a single 1. The number 2 is represented by fourteen 0’s followed by two 1’s and so on. Thus the set of numbers from 1 through 16 is represented by strings belonging to the range 0000 0000 0000 0001 to 1111 1111 1111 1111. To represent a point in the pattern, the 16 - bit strings of the row and column coordinates are concatenated together. To this, another bit that represents the bias is added and the resulting 33-element vector is given as input to the network.

The training samples are randomly selected points from the two regions of the pattern. The samples used here are shown in Figure 2.4 (b). The points marked “#” are the points from the black region and the points marked “o” are points from the white region. A total of 75 points are used for training. Thus the network used for this pattern classification experiment has 33 neurons in the input layer and 75 neurons in the hidden layer. The output layer requires only one neuron to display a binary “0” or “1”.

After the training is done the network is tested for all 256 points in the 16 by 16 area of the pattern as the value of r is varied from 1 to 4. The results for the different

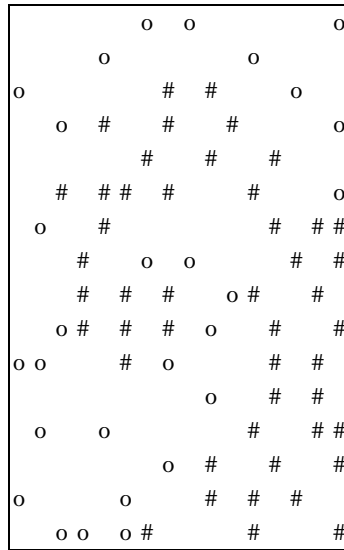
levels of generalization achieved are presented in Figure 2.4 (c), (d), (e) and (f). It can be seen that as the value of r is increased the network tends to generalize more points as belonging to the black region. This over generalization is because during training, the density of the samples presented from the black region was greater than the density of samples from the white region.

Original Spiral Pattern



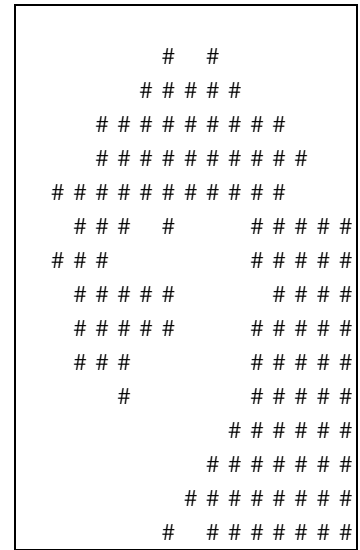
(a)

Training Samples



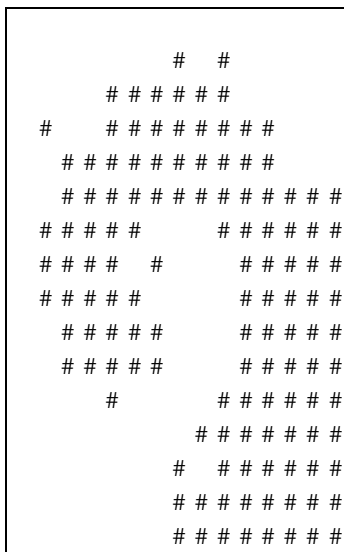
(b)

r = 1



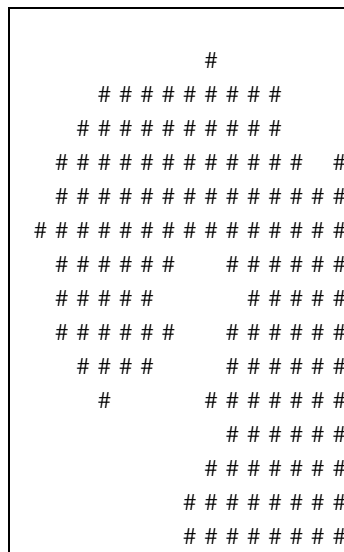
(c)

r = 2



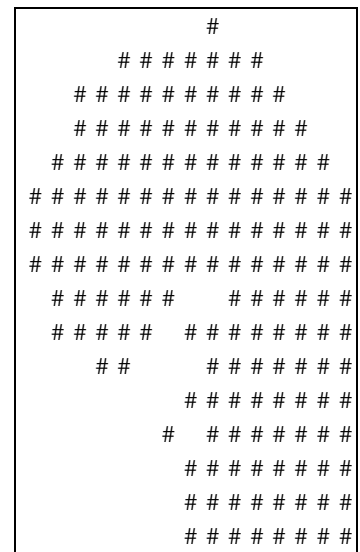
(d)

r = 3



(e)

r = 4



(f)

Figure 2.4: Results of spiral pattern classification using CC4 algorithm

Chapter 3

Input Encoding

For the pattern classification problem using the CC4 algorithm, the unary encoding scheme is preferred over binary encoding because it provides an ordered and linear measure of distance, related to the number of ones in the coded sequence. The 16 row and column coordinates of the pattern are integers represented by 16 - bit unary codes. Therefore 1 is represented as 0000 0000 0000 0001, 2 is represented as 0000 0000 0000 0011 and so on such that 16 is represented as 1111 1111 1111 1111. The encoded coordinates are then concatenated to denote a point in the pattern. This means that after including the bias, each input vector is 33 bits in length and the same number of neurons are required at the input layer of the network. If binary encoding were used instead, the number of input neurons required would be 9, as the row and column coordinates would each require four bits and one bit would be required for the bias. Nevertheless since unary encoding enhances generalization it is considered an appropriate choice for the corner classification family of networks.

For the new algorithms, one can take advantage of the two dimensional nature of complex numbers for the encoding problem. First the 16 row indices are represented using unary strings of the range 0000 0000 0000 0001 through 1111 1111 1111 1111. The 16 column indices can be represented in a similar manner, using i instead of 1. The column index 1 can be represented as 0000 0000 0000 000i, the column index 2 can be represented as 0000 0000 000 000ii and so on. Now the encoded row and column indices can be added together to denote a point, resulting in a 16 - character string. For example

the point (5, 8) is encoded as 0000 0000 0001 1111 0000 0000 1111 1111 using the unary code. This point can now be represented as 0000 0000 iii1+i 1+i1+i1+i1+i. This cuts down the number of input neurons required from 33 to 17.

It is possible to reduce the network size even further by using a new scheme called quaternary encoding. This scheme requires only 11 input neurons for the pattern classification problem. It is a simple modification of the unary scheme and accommodates two additional characters i and $l+i$. The row and column indices ranging from 1 to 16 can be represented using quaternary *codewords* only five characters in length. The integers 1 to 6 are represented with codewords from the range 00000 through 11111. This follows from the unary scheme. Now the character i is introduced and the integers 7 through 11 are represented as 1111*i* through *iiii*. At this point $l+i$ is introduced and the integers 12 through 16 are represented with codewords from *iiii*1+i to 1+i1+i1+i1+i1+i. Thus for each point, when its row and column codewords are concatenated and the bias is added, we get an 11 element vector. Table 3.1 holds all the codewords used to represent integers 1 to 16.

3.1 Length of the Codewords

An important issue is to decide the length of the codewords required to represent a desired range of integers. Let l be the length of the codewords for a range of C integers. Consider the integers in Table 3.1. For this range $C = 16$ and $l = 5$. We can now examine how 16 codewords can be formed with $l = 5$. The 16 codewords can be classified into three groups. The first group represents integers 1 to 6, where the codewords are constructed without using characters i or $l+i$. The codewords in the second group represent integers 7 to 11 and don't use $l+i$, while in the third group the

Table 3.1: Quaternary codewords for integers 1 to 16

| Integer | Quaternary code | | | | |
|---------|-----------------|-----|-----|-----|-----|
| 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 |
| 3 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 1 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 | 1 | i |
| 8 | 1 | 1 | 1 | i | i |
| 9 | 1 | 1 | i | i | i |
| 10 | 1 | i | i | i | i |
| 11 | i | i | i | i | i |
| 12 | i | i | i | i | 1+i |
| 13 | i | i | i | 1+i | 1+i |
| 14 | i | i | 1+i | 1+i | 1+i |
| 15 | i | 1+i | 1+i | 1+i | 1+i |
| 16 | 1+i | 1+i | 1+i | 1+i | 1+i |

codewords representing integers 12 to 16 use $l+i$. We see here that the first group has 6 codewords. The other two have 5 each, corresponding to the length of the codewords.

For any C , the set of codewords would consist of three such groups where the first group has $l + 1$ codewords, the second and third have l codewords each. This is summarized in Equation 3.1 as follows:

$$C = (l + 1) + l + l \quad [3.1]$$

$$C = 3 * l + 1 \quad [3.2]$$

$$l = (C - 1) / 3 \quad [3.3]$$

Equation 3.3 is valid only when $(C - 1)$ is divisible by 3. For cases when this is not true, it is safe to generalize as follows:

$$l = \mathbf{ceil} [(C - 1) / 3] \quad [3.4]$$

When $(C - 1)$ is not divisible by 3, the number of codewords that can be formed using the l obtained from Equation 3.4 is more than required. In this case any C consecutive codewords from the complete set of words of length l can be used.

Example 3.1

Suppose we require 10 codewords to represent a range of 10 integers. We can deduce l using Equation 3.4. We have $C = 10$.

$$l = \text{ceil} [(10 - 1) / 3]$$

$$l = \text{ceil} [9 / 3]$$

$$\therefore l = 3$$

From Table 3.2 we see that indeed 10 codewords can be formed using $l = 3$. This verifies Equation 3.4.

Table 3.2: All possible quaternary codewords when $l = 3$

| Integer | Quaternary code | | |
|---------|-----------------|-----|-----|
| 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 1 | 1 |
| 5 | 1 | 1 | i |
| 6 | 1 | i | i |
| 7 | i | i | i |
| 8 | i | i | 1+i |
| 9 | i | 1+i | 1+i |
| 10 | 1+i | 1+i | 1+i |

Similarly when $C = 11$, from Equation 3.4 we have $l = 4$. Now 13 different codewords can be constructed in sequence. Of these any 11 consecutive codewords may be used for the desired encoding.

Chapter 4

The Problem of Instantaneous Training of Complex Neural Networks

This chapter presents some issues relating to the development of corner classification algorithms capable of training 3-layered feedforward neural networks with complex inputs. The input alphabet consists of $\{0, 1, i, 1+i\}$, where $i = \sqrt{-1}$ and the outputs are binary 0 and 1. The features of a straightforward generalization of the CC4 algorithm and its network architecture are:

1. The number of input neurons is one more than the number of input elements in a training sample; the extra neuron is the bias neuron and the bias is always set to one as in CC4.
2. A hidden neuron is created for all training samples; the first hidden neuron corresponds to the first training sample, and the second neuron corresponds to the second training sample and so on.
3. The output layer is fully connected; all the hidden neurons are connected to all the output neurons.
4. If an element in the training sample is a high 1, the interconnection between the input neuron receiving this element and the hidden neuron corresponding to the training sample presented is assigned a weight 1. If the input element is a low 0, then the interconnection weight is an *inhibitory* -1. This follows from CC4.
5. The complex input element i is considered to be a low complex input while $1+i$ is taken as a high complex input.

6. If an element in the training sample is i , then the interconnection between the corresponding input and hidden neurons is given a weight of $-i$. If the input element is $1+i$, the weight assigned is $1-i$. Thus the weights for complex inputs are simply their complex conjugates.
7. The input weights from the bias neuron are assigned as $r - s + 1$. The value of s is computed differently from the CC4. Here, s is assigned the value equal to the sum of the number of ones, i 's, and twice the number of $(1+i)$ s. For example, for the following vector;

$$x [3] = [0 \ i \ 1 \ 1+i]$$

$$s = 4$$

8. The output weights are assigned as 1 or -1 if the desired outputs are 1 or 0 respectively, as done in CC4.
9. This algorithm uses a binary step activation function at the output layer as in the CC3 and CC4 algorithms. However the hidden neuron activation function is different. The i^{th} - hidden neuron produces an output $oh_i [n]$ for a training sample n , when it receives an input $ih_i [n]$, by the following activation function:

$$oh_i [n] = \begin{cases} 1 & \text{if } (\mathbf{Im} [ih_i [n]] = 0 \text{ and } \mathbf{Re} [ih_i [n]] > 0) \\ 0 & \text{otherwise} \end{cases}$$

Since complex inputs have to be mapped to binary outputs, a simple binary step activation function would not be sufficient. When the complex input vector combines with all the interconnection weights, only the appropriate hidden neuron receives a real positive input. The other hidden neurons receive either negative real inputs or complex inputs. The activation function presented above will filter out all the undesirable

complex inputs. Thus only one hidden neuron produces a real positive output when its corresponding training vector is presented to the network.

This generalized algorithm like the members of the Corner Classification family can be implemented by simple **if-then** rules. The formal algorithm is presented below:

```

for each training vector  $x_m[n]$  do
     $s_m = \text{no of } 1\text{'s} + \text{no of } i\text{'s} + 2*(\text{no of } (1+i)\text{s}) \text{ in } x_m[1:n-1];$ 
    for index = 1 to n-1 do                                     //  $w_m[ ]$ : input weights
        if  $x_m[\text{index}] = 0$  then
             $w_m[\text{index}] = -1;$ 
        else
            if  $x_m[\text{index}] = 1$  then
                 $w_m[\text{index}] = 1;$ 
            end if
        end if
        if  $x_m[\text{index}] = i$  then
             $w_m[\text{index}] = -i;$ 
        else
            if  $x_m[\text{index}] = 1+i$  then
                 $w_m[\text{index}] = 1-i;$ 
            end if
        end if
    end for
     $w_m[n] = r - s_m + 1;$ 

    for index1 = 1 to k do                                     // k = no of outputs y
        if  $y_m[\text{index1}] = 0$  then
             $ow_m[\text{index1}] = -1;$                                //  $ow_m[ ]$ : output weights
        else
             $ow_m[\text{index1}] = 1;$ 
        end if
    end for
end for

```

The value of s is assigned based on the total contribution of each input element to achieve a high output. This follows from CC4. The complex input i combines with its weight assignment $-i$, to produce a contribution of 1 and the input $1+i$ combines with its own weight $1-i$ to produce a contribution of 2. This explains why s is set as equal to the sum of the number of ones, the number of i 's, and twice the number of $(1+i)$ s. The s value is

expected to ensure that only one hidden neuron fires for each input, as was the case in CC4. But sometimes more than one neuron fires even for a logically appropriate value of s . It will be seen that this is because of the inputs and the weights. Thus some inputs have to be restricted.

Assuming $r = 0$, when an input vector is presented to the network, it combines with the interconnection weights and only the corresponding hidden neuron receives a positive input (the s value for that input vector) from the input neurons presented with non-zero inputs. This hidden neuron also receives a $-s + 1$ input from the bias neuron. Thus the total input to the hidden neuron is $s - s + 1 = 1$. The working of the algorithm may be better understood by the following two examples. The first example considers only complex elements as inputs. The second example considers both complex and real elements as inputs.

Example 4.1

Consider the truth table given below. The algorithm will now be used to train a network to map the inputs to the outputs in the truth table.

Table 4.1: Inputs and outputs for Example 4.1

| Inputs | | Output |
|--------|-------|--------|
| X_1 | X_2 | Y |
| i | i | 0 |
| i | 1+i | 1 |
| 1+i | i | 1 |
| 1+i | 1+i | 0 |

The network architecture required for the input/output mapping is shown in Figure 4.1.

The different network parameters obtained during training are tabulated in Table 4.2.

The example is based on the XOR function with i as a low input and $1+i$ as a high input.

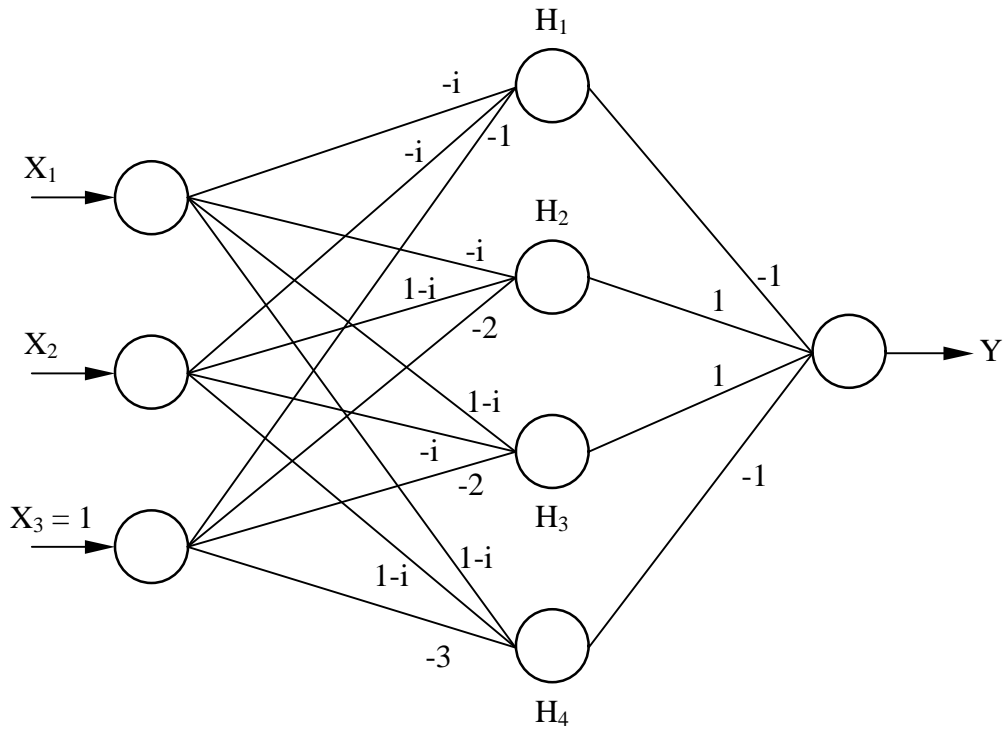


Figure 4.1: The Network Architecture for Example 4.1

Table 4.2: Network Parameters in the input/output mapping of Example 4.1

| | | | | | | | Input to | | | | Output of | | | | Input to Y | Output Y |
|--------|-------|---|-----|---------|-------|------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|------------|----------|
| Inputs | | | s | Weights | | | H ₁ | H ₂ | H ₃ | H ₄ | H ₁ | H ₂ | H ₃ | H ₄ | | |
| i | i | 1 | 2 | $-i$ | $-i$ | -1 | 1 | i | i | $-1+2i$ | 1 | 0 | 0 | 0 | -1 | 0 |
| i | $1+i$ | 1 | 3 | $-i$ | $1-i$ | -2 | $1-i$ | 1 | 0 | i | 0 | 1 | 0 | 0 | 1 | 1 |
| $1+i$ | i | 1 | 3 | $1-i$ | $-i$ | -2 | $1-i$ | 0 | 1 | i | 0 | 0 | 1 | 0 | 1 | 1 |
| $1+i$ | $1+i$ | 1 | 4 | $1-i$ | $1-i$ | -3 | $1-2i$ | $1-i$ | $1-i$ | 1 | 0 | 0 | 0 | 1 | -1 | 0 |

Each input vector has two elements. Thus including a bias neuron, three input neurons are required. All input vectors need to be used for training and so the number of hidden neurons required is four. Only one output neuron is required and all four hidden neurons are connected to it. The value of r is set as zero as no generalization is required here.

The input and output weights are assigned according to the algorithm.

Consider the second training vector $(i \ 1+i \ 1)$ where 1 is the bias. The total contribution s , by this vector is 3. Since $r = 0$, the weight associated with the link connecting the bias neuron to the second hidden neuron, which corresponds to this training vector is -2. Thus the weights are $(-i \ 1-i \ -2)$. The training vector and the weights combine to provide the hidden neuron an input of 1. The other hidden neurons receive a negative or complex input and fail to fire. Similarly for each sample only one hidden neuron fires. Thus the input/output mapping is accomplished.

Example 4.2: Two-output neuron network

The generalized algorithm can be used to map inputs to a network with multiple output neurons. This example maps five-element vectors to two outputs. The inputs and outputs are shown in Table 4.3.

Table 4.3: Inputs and outputs for Example 4.2

| Inputs | | | | | Outputs | |
|--------|-------|-------|-------|-------|---------|-------|
| X_1 | X_2 | X_3 | X_4 | X_5 | Y_1 | Y_2 |
| 0 | $1+i$ | $1+i$ | 0 | i | 1 | 1 |
| $1+i$ | 0 | 1 | $1+i$ | 1 | 0 | 1 |
| 1 | 1 | i | 0 | 1 | 1 | 0 |

The network architecture accomplishing the mapping for this example is shown in Figure 4.2. All the network parameters obtained during the training are presented in Table 4.4.

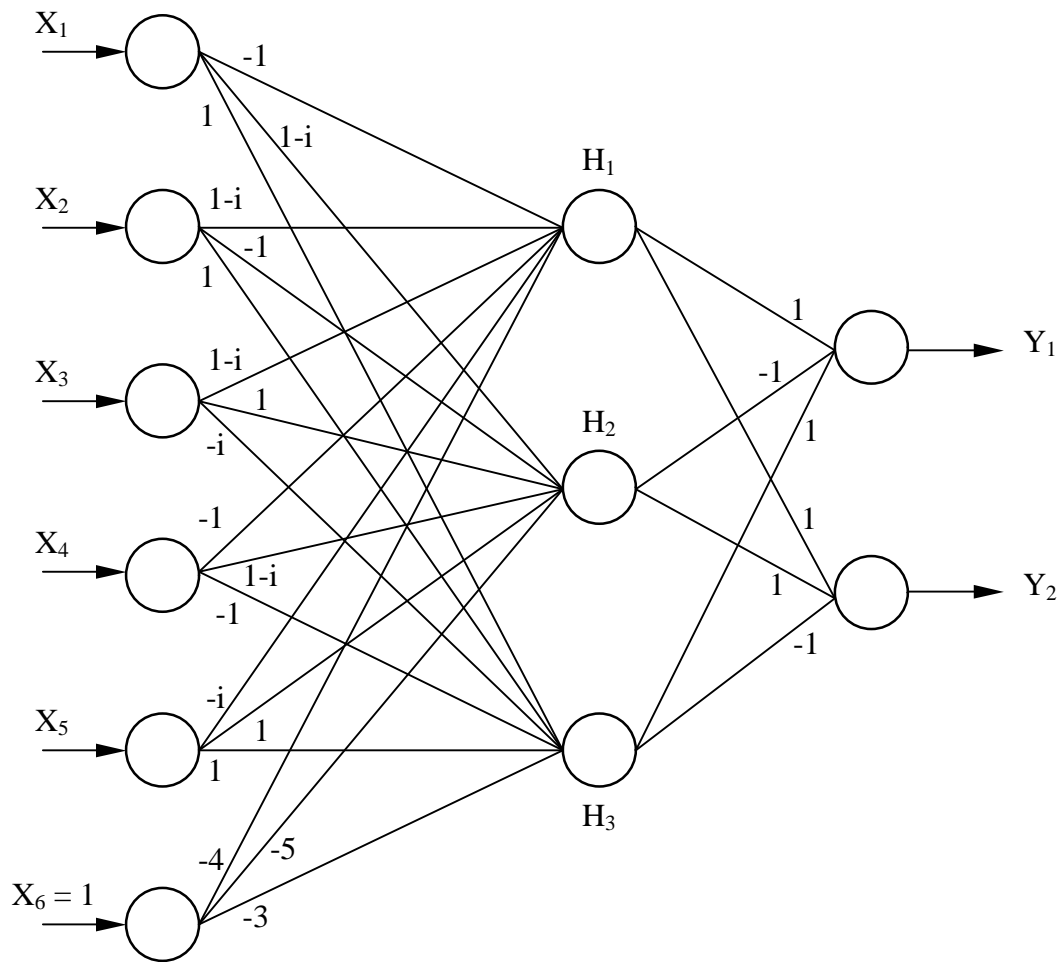


Figure 4.2: The Network Architecture for Example 4.2

Table 4.4: Network Parameters in the input/output mapping of Example 4.2

| | | | | | | | | | | | Input to | | | Output of | | | Output | | | |
|--------|-----|-----|-----|---|---|-----|---------|-----|-----|-----|----------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---|
| Inputs | | | | | | s | Weights | | | | | H ₁ | H ₂ | H ₃ | H ₁ | H ₂ | H ₃ | y ₁ | y ₂ | |
| 0 | 1+i | 1+i | 0 | i | 1 | 5 | -1 | 1-i | 1-i | -1 | -i | -4 | 1 | -5+i | -1+i | 1 | 0 | 0 | 1 | 1 |
| 1+i | 0 | 1 | 1+i | 1 | 1 | 6 | 1-i | -1 | 1 | 1-i | 1 | -5 | -5-4i | 1 | -2-i | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | i | 0 | 1 | 1 | 4 | 1 | 1 | -i | -1 | 1 | -3 | -3-i | -4 | 1 | 0 | 0 | 1 | 1 | 0 |

Each input vector has five elements, which means that six input neurons are required including the bias neuron. All three samples need to be used for training and so three hidden neurons are required, which are connected to two output neurons. The input and output interconnection weights are assigned according to the algorithm. Only one hidden neuron fires for each input accomplishing a successful mapping.

4.1 Restrictions on the Inputs

As mentioned earlier some inputs vectors fire more than one hidden neuron. This undesirable property is caused because the complex nature of one input vector is neutralized by the weights corresponding to some other vector. Thus certain inputs have to be restricted for the algorithm to work and accomplish successful mapping. There are two restrictions and they are as follows:

1. If an m element vector with n wholly real high and n wholly real low elements is given as an input, then another m element vector with n wholly complex high and n wholly complex low elements cannot be given as an input, if these high and low elements of both the vectors, appear in the same positions, while all other $m - 2n$ elements of both the vectors are identical.

Consider the two sets of vectors below. In each set only the elements in the vectors that are not identical are shown.

- 1) Input 1: ... 1 ... 0 ...
 Input 2: ... 1+i ... i ...
- 2) Input 1: ... i ... 1+i ...
 Input 2: ... 0 ... 1 ...

2. If an m element vector with n complex high elements is given as an input, n being even, then another m element vector with $n/2$ real high elements and $n/2$ complex low elements in the same positions as the n complex high elements of the first vector, cannot be given as an input, if all the other $m - n$ elements of both the vectors are identical.

For instance, in both sets of vectors below, one vector's weights neutralize the complex nature of the other.

- 1) Input 1: ... 1 i 1 i ...
 Input 2: ... 1+i 1+i 1+i 1+i ...
- 2) Input 1: ... 1+i 1+i 1+i 1+i ...
 Input 2: ... 1 i 1 i ...

The above two restrictions are illustrated in the following two examples. Example 4.3 handles vectors restricted by the first restriction and Example 4.4 handles vectors restricted by the second restriction.

Example 4.3: Network with Inputs under Restriction I

The following table displays inputs and their corresponding outputs where the second or the third input vector should be restricted. In this case the network cannot map the second input vector to the corresponding output. The network diagram is shown in Figure 4.3, and the network parameters are in Table 4.6.

Table 4.5: Inputs and outputs for Example 4.3

| Inputs | | | | | Outputs |
|----------------|----------------|----------------|----------------|----------------|---------|
| X ₁ | X ₂ | X ₃ | X ₄ | X ₅ | Y |
| 1 | i | 1 | 0 | 1+i | 1 |
| i | 1 | 1+i | 0 | i | 1 |
| i | 1 | 1 | 0 | 0 | 0 |

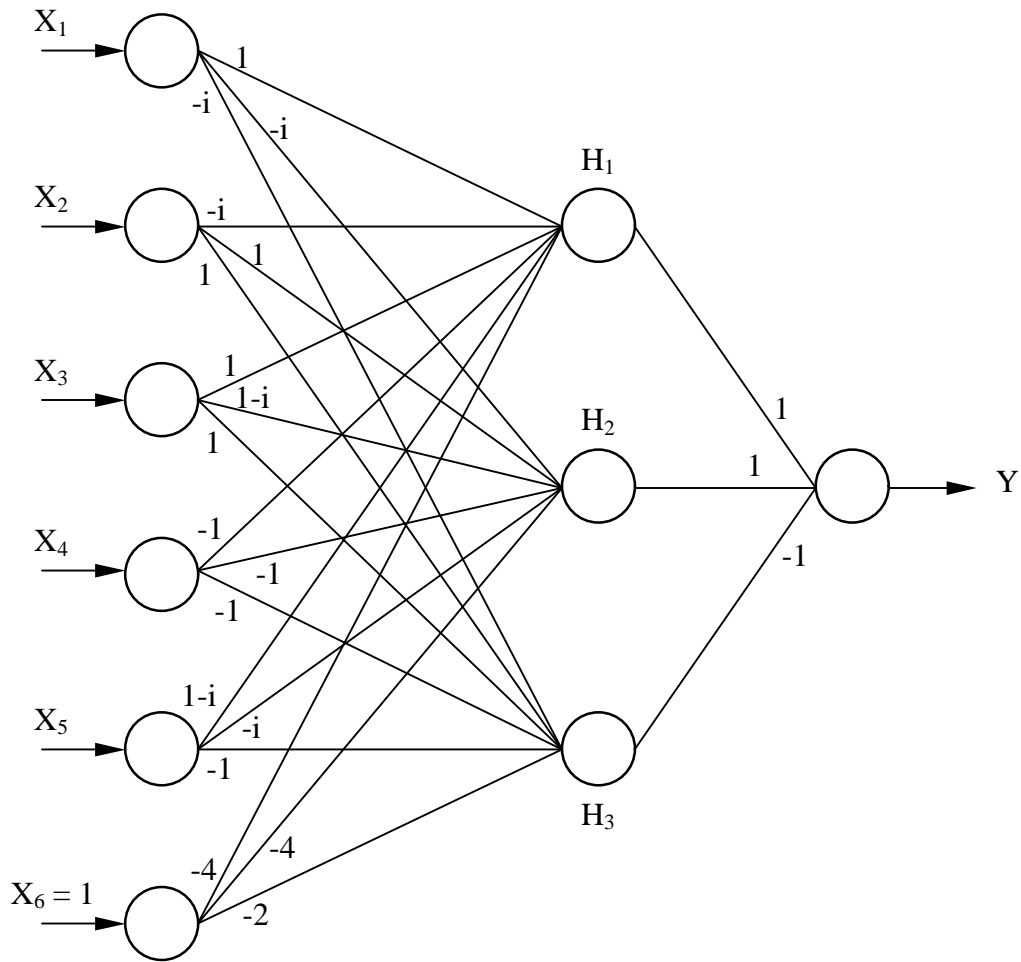


Figure 4.3: The Network Architecture for Example 4.3

Table 4.6: Network Parameters in the input/output mapping of Example 4.3

| | | | | | | | | | | | | Input to | | | Output of | | | Output | |
|--------|---|-----|---|-----|---|---|---------|----|-----|----|-----|----------------|----------------|----------------|----------------|----------------|----------------|----------|----------|
| Inputs | | | | | | s | Weights | | | | | H ₁ | H ₂ | H ₃ | H ₁ | H ₂ | H ₃ | Y | |
| 1 | i | 1 | 0 | 1+i | 1 | 5 | 1 | -i | 1 | -1 | 1-i | -4 | 1 | -2-2i | -2-i | 1 | 0 | 0 | 1 |
| i | 1 | 1+i | 0 | i | 1 | 5 | -i | 1 | 1-i | -1 | -i | -4 | -2+2i | 1 | 1 | 0 | 1 | 1 | 0 |
| i | 1 | 1 | 0 | 0 | 1 | 3 | -i | 1 | 1 | -1 | -1 | -2 | -3 | -1-i | 1 | 0 | 0 | 1 | 0 |

The two vectors differ only by elements 3 and 5. The third element in both is high, but in the second vector it is complex and the third it is real. The fifth element in both vectors is low, but again in the second vector it is complex and in the third it is real. Thus the complex nature of the second vector gets neutralized when multiplied by the weights of the third vector. This causes the faulty output for the second vector. The shaded cells in Table 4.6 contain the undesired input to the third hidden neuron and the erroneous output element produced by this input.

Example 4.4: Network with Inputs under Restriction II

The table below displays inputs and their corresponding outputs for this example, where the second or the third vector should be restricted. Here the network fails to map the third input vector to the corresponding output.

Table 4.7: Inputs and outputs for Example 4.4

| Inputs | | | | | Outputs |
|--------|-------|-------|-------|-------|---------|
| X_1 | X_2 | X_3 | X_4 | X_5 | Y |
| 1 | i | 1 | 0 | 1+i | 1 |
| i | 1 | i | 1 | 0 | 0 |
| i | 1 | 1+i | 1+i | 0 | 1 |

The second and third vectors differ by the third and fourth elements. The second vector has a complex low and a real high as the third and fourth elements, while the third has complex highs as both the third and fourth elements. In this case the complex nature of the third vector gets neutralized when multiplied with the weights of the second vector. As a result, the vector does not get mapped to the right output. The network architecture for the example is shown in Figure 4.4, and Table 4.8 holds all network parameters. The elements causing the erroneous output are shaded in the table.

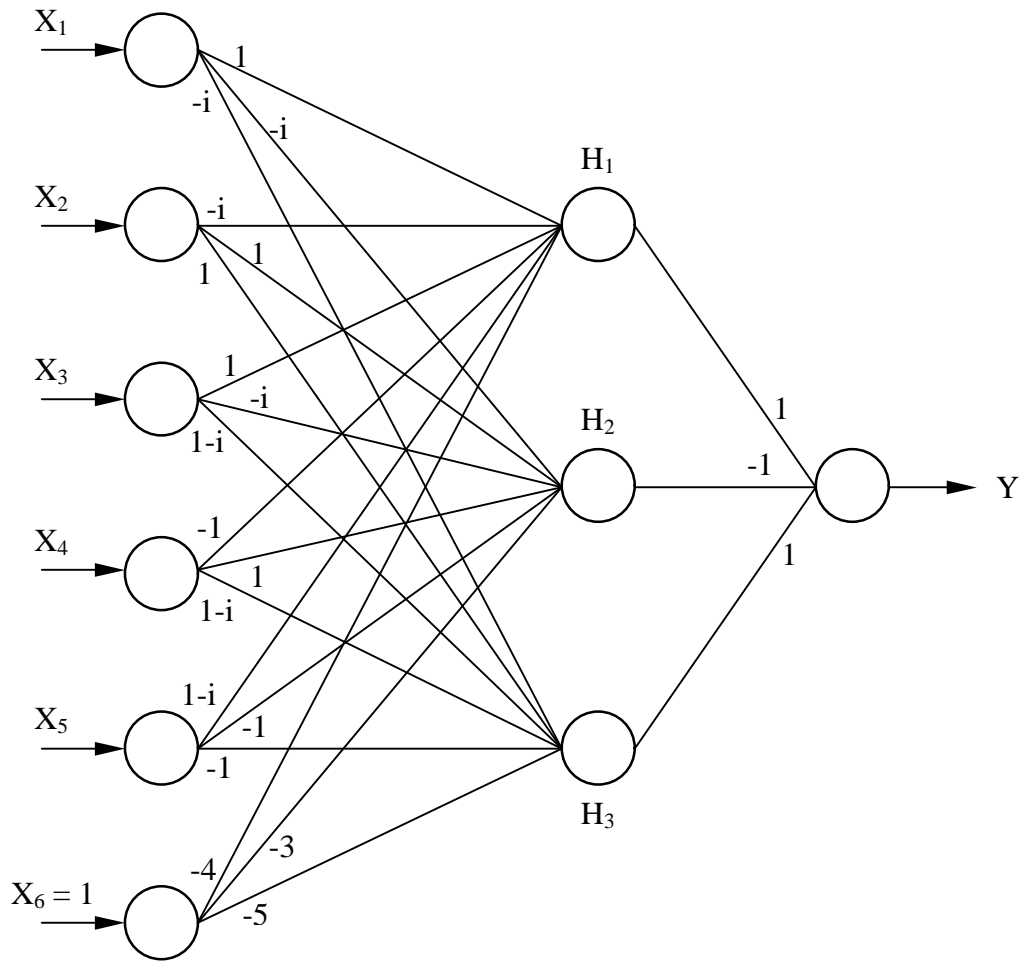


Figure 4.4: The Network Architecture for Example 4.4

Table 4.8: Network Parameters in the input/output mapping of Example 4.4

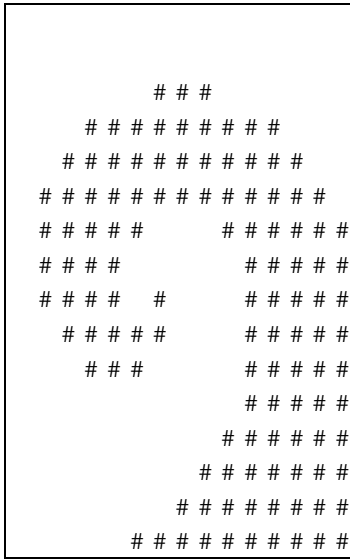
| | | | | | | | | | | | Input to | | | Output of | | | Output | | |
|--------|---|-----|-----|-----|---|---|---------|----|-----|-----|----------|----------------|----------------|----------------|----------------|----------------|----------------|---|----------|
| Inputs | | | | | | s | Weights | | | | | H ₁ | H ₂ | H ₃ | H ₁ | H ₂ | H ₃ | Y | |
| 1 | i | 1 | 0 | 1+i | 1 | 5 | 1 | -i | 1 | -1 | 1-i | -4 | 1 | -4-2i | -5-2i | 1 | 0 | 0 | 1 |
| i | 1 | i | 1 | 0 | 1 | 4 | -i | 1 | -i | 1 | -1 | -3 | -5+i | 1 | -1 | 0 | 1 | 0 | 0 |
| i | 1 | 1+i | 1+i | 0 | 1 | 6 | -i | 1 | 1-i | 1-i | -1 | -5 | -4 | 1 | 1 | 0 | 1 | 1 | 0 |

These above restrictions influence the generalization of the output class. This can be illustrated by repeating the spiral pattern classification experiment of Chapter 2. The row and column coordinates of the points in the 16 by 16 pattern area are encoded using the two schemes discussed in Chapter 3 suitable for this algorithm's alphabet. The first scheme represents row coordinates using 0's and 1's and column coordinates using 0's and i 's. These representations are added together to denote a point in the pattern. In this case the number of input neurons required is only 17. The same 75 training samples used in the CC4 experiment are used here for the training, and then the network is tested with all 256 points. Figure 4.5 (c) to (f) show the different results for the testing, with r varying between 1 and 4.

The experiment is repeated using the quaternary encoding scheme. Here the network size is reduced even further, as only 11 input neurons are required. The network is then tested for all points in the pattern and the results are shown in Figure 4.6 (a) to (d). In both cases we see that the classification is not very good. This is because during testing even restricted inputs are presented to the network causing the poor classification.

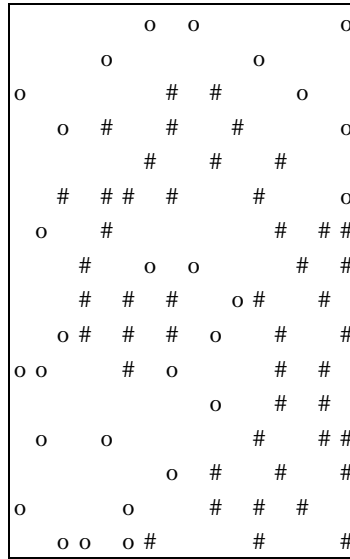
The algorithm was also tested to classify a second pattern shown in Figure 4.7 (a). The asymmetric pattern resembles a hand about to catch a round object. Again a total of 75 points were used in the training, a majority of them being from the black region. The samples are presented in Figure 4.7 (b). Both encoding schemes were used and the results of the testing for the different values of r are shown in Figure 4.7 (c) to (f) and Figure 4.8 (a) to (d). Once more it can be seen that the results are not appreciable. Table 4.9 and 4.10 contain the number of points in the patterns that have been classified and misclassified when both encoding schemes were used.

Original Spiral



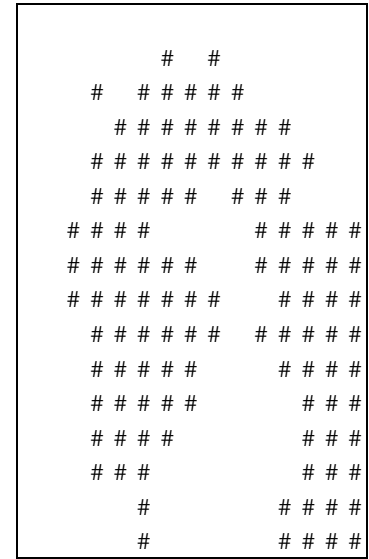
(a)

Training Samples



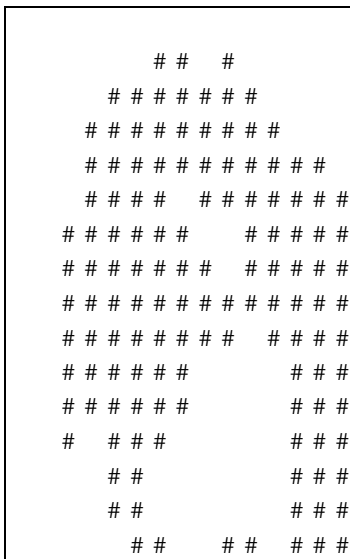
(b)

r = 1



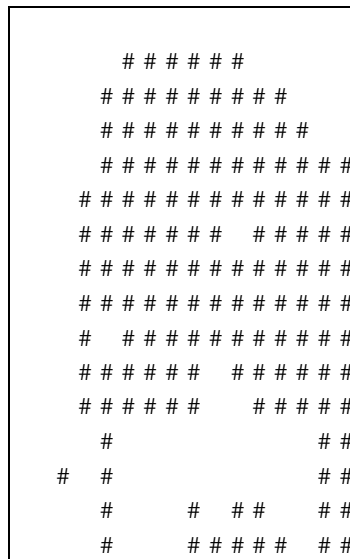
(c)

r = 2



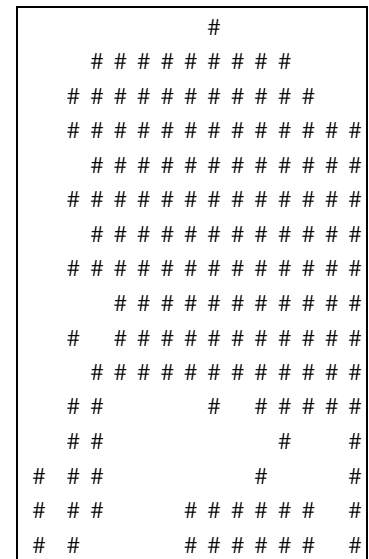
(d)

r = 3



(e)

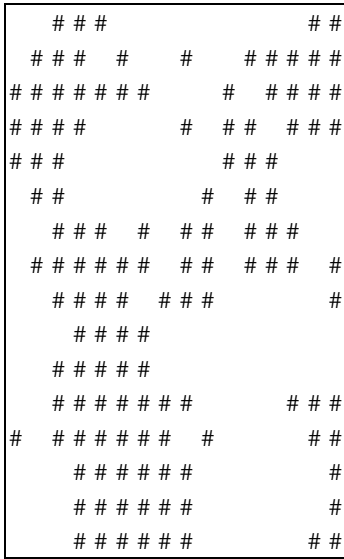
r = 4



(f)

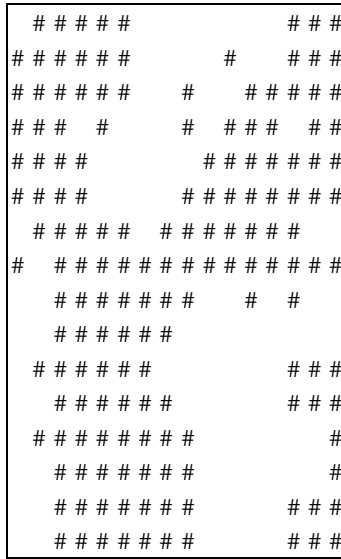
Figure 4.5: Results of spiral pattern classification using the first encoding scheme

r = 1



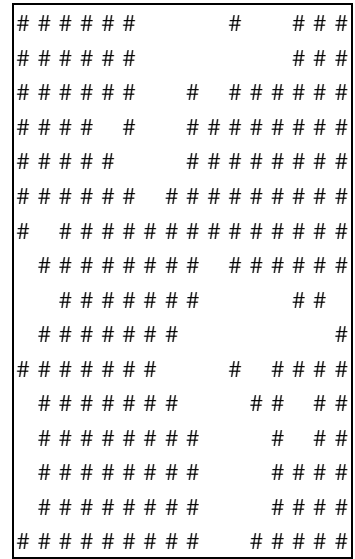
(a)

r = 2



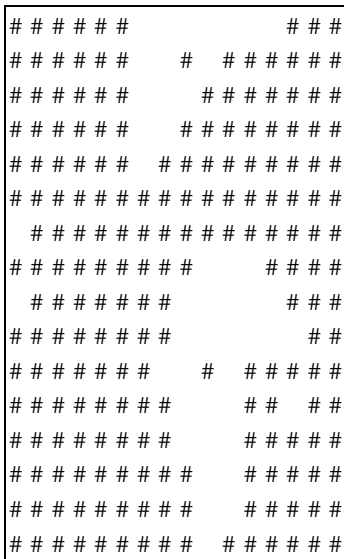
(b)

r = 3



(c)

r = 4



(d)

Figure 4.8: Results of pattern classification using quaternary scheme

Table 4.9: No. of points classified/misclassified in the spiral pattern

| | | No. of points | | | |
|----------------------------|---------------|---------------|---------|---------|---------|
| | | $r = 1$ | $r = 2$ | $r = 3$ | $r = 4$ |
| First Encoding Scheme | Classified | 196 | 191 | 189 | 177 |
| | Misclassified | 60 | 65 | 67 | 79 |
| Quaternary Encoding Scheme | Classified | 191 | 186 | 191 | 182 |
| | Misclassified | 65 | 70 | 65 | 74 |

Table 4.10: No. of points classified/misclassified in the second pattern

| | | No. of points | | | |
|----------------------------|---------------|---------------|---------|---------|---------|
| | | $r = 1$ | $r = 2$ | $r = 3$ | $r = 4$ |
| First Encoding Scheme | Classified | 181 | 166 | 151 | 137 |
| | Misclassified | 75 | 90 | 105 | 119 |
| Quaternary Encoding Scheme | Classified | 193 | 179 | 152 | 133 |
| | Misclassified | 63 | 77 | 104 | 123 |

Neither scheme has produced appreciable results. Thus we can conclude that the algorithm itself does not bring about efficient generalization. This poor performance of algorithm brought about the need for a better algorithm capable of handling complex inputs and lead to the development of the 3C Algorithm.

Chapter 5

The 3C Algorithm: Instantaneous Training for Complex Input Neural Networks

The 3C algorithm (from Complex Corner Classification, CCC) is also a generalization of the CC4 and is capable of training three-layered feedforward networks to map inputs from the alphabet $\{0, 1, i, 1+i\}$ to the real binary outputs 0 and 1. The algorithm presented in the previous chapter placed two restrictions on the inputs thus affecting its performance. The 3C algorithm eliminates these restrictions by altering the assignment of the input interconnection weights and the combination procedure of these weights with the inputs. The features of the algorithm and its network are:

1. The number of input neurons is one more than the number of input elements in a training sample. The extra neuron is the bias neuron; the bias is always set to one.
2. As in CC4, a hidden neuron is created for all training samples, and the first hidden neuron corresponds to the first training sample, the second neuron corresponds to the second training sample and so on.
3. The output layer is fully connected; each hidden neuron is connected to all the output neurons.
4. The interconnection weights from all the input neurons excluding the bias neuron are complex. Each input of the alphabet $\{0, 1, i, 1+i\}$ is treated as complex for the weight assignment.

5. If the real part of the input element is 0, then the real part of the corresponding input interconnection weight assigned is -1. If the real part of the input element is 1, then the real part of the weight is also set as 1.
6. Similarly if the complex part of the input is 0, then the complex part of the weight is assigned as -1 or if the complex part of the input is 1, then the weight is also assigned as 1.
7. The weight from the bias neuron to a hidden neuron is assigned as $r - s + 1$, where r is the radius of generalization. The value of s is assigned as sum of the number of ones, i 's, and twice the number of $(1+i)$ s in the training vector corresponding to the hidden neuron.
8. If the desired output is 0, the output layer weight is set as an inhibitory -1. If the output is 1, then the weight is set as 1.
9. The altered combination procedure of the inputs and the weights causes the hidden neuron inputs to be entirely real. Thus the activation function required at the hidden layer is simply the binary step activation function. The output layer also uses a binary step activation function.

When an input vector is presented to the network, the real and imaginary parts of each input element of the vector is multiplied to the corresponding interconnection weight's real and imaginary parts respectively. The two products are then added to obtain the individual contribution by an input element in the vector. This is done for each element and their individual contributions are then added together to obtain the total contribution of the entire vector. Using this combination procedure, each hidden neuron always receives only real inputs. Thus only a binary step activation function is required at the

hidden layer. As an example consider the vector $(1 \ 1+i \ i)$. The corresponding weight vector is $(1-i \ 1+i \ -1+i)$. The input vector now combines with this weight vector to yield a total contribution of 4. This contribution is computed as follows:

$$\begin{aligned} & (\text{Re}(1) \cdot \text{Re}(1-i) + \text{Im}(1) \cdot \text{Im}(1-i)) \\ & + (\text{Re}(1+i) \cdot \text{Re}(1+i) + \text{Im}(1+i) \cdot \text{Im}(1+i)) \\ & + (\text{Re}(i) \cdot \text{Re}(-1+i) + \text{Im}(i) \cdot \text{Im}(-1+i)) = 4 \end{aligned}$$

The 3C algorithm can also be expressed as a set of simple **if-then** rules. The formal algorithm is as follows:

```

for each training vector  $x_m[n]$  do
     $s_m = \text{no of } 1\text{'s} + \text{no of } i\text{'s} + 2 \cdot (\text{no of } (1+i)\text{s})$  in  $x_m[1:n-1]$ ;
    for index = 1 to n-1 do                                     //  $w_m[ ]$ : input weights
        if  $\text{Re}(x_m[\text{index}]) = 0$  then
             $\text{Re}(w_m[\text{index}]) = -1$ ;
        else
            if  $\text{Re}(x_m[\text{index}]) = 1$  then
                 $\text{Re}(w_m[\text{index}]) = 1$ ;
            end if
        end if
        if  $\text{Im}(x_m[\text{index}]) = 0$  then
             $\text{Im}(w_m[\text{index}]) = -1$ ;
        else
            if  $\text{Im}(x_m[\text{index}]) = 1$  then
                 $\text{Im}(w_m[\text{index}]) = 1$ ;
            end if
        end if
    end for
     $w_m[n] = r - s_m + 1$ ;

    for index1 = 1 to k do                                       //  $k = \text{no of outputs } y$ 
        if  $y_m[\text{index1}] = 0$  then
             $ow_m[\text{index1}] = -1$ ;                                     //  $ow_m[ ]$ : output weights
        else
             $ow_m[\text{index1}] = 1$ ;
        end if
    end for
end for

```

Let $r = 0$, now when an input vector is presented to the network, all input neurons except the bias neuron provide the hidden neuron corresponding to the input vector, with a contribution equal to the s value of the vector. And since r is set as zero, the input from the bias neuron is equal to $-s + 1$. Thus the total input to the hidden neuron is 1. All other hidden neurons receive zero or negative input. This ensures that only one hidden neuron fires for each input and so there is no need for restricting any of the inputs.

The examples used in Chapter 4 are repeated here to illustrate the working of the 3C algorithm. Examples 5.3 and 5.4 show how the inputs restricted by the algorithm in the previous chapter are successfully mapped using 3C.

Example 5.1

The inputs and outputs are shown below in Table 5.1. The 3C algorithm can be used to train a network to map these inputs to the outputs. The network architecture is shown in Figure 5.1, and the various network parameters are tabulated in Table 5.2.

Table 5.1: Inputs and outputs for Example 5.1

| Inputs | | Output |
|--------|-------|--------|
| X_1 | X_2 | Y |
| i | i | 0 |
| i | $1+i$ | 1 |
| $1+i$ | i | 1 |
| $1+i$ | $1+i$ | 0 |

As mentioned in Chapter 4, this example is similar to the XOR function taking the input i as the low input and the input $1+i$ as the high input. Each input vector has two elements and so three input neurons are required including the one for the bias neuron. All four training samples are required for the training; so four hidden neurons are required. The

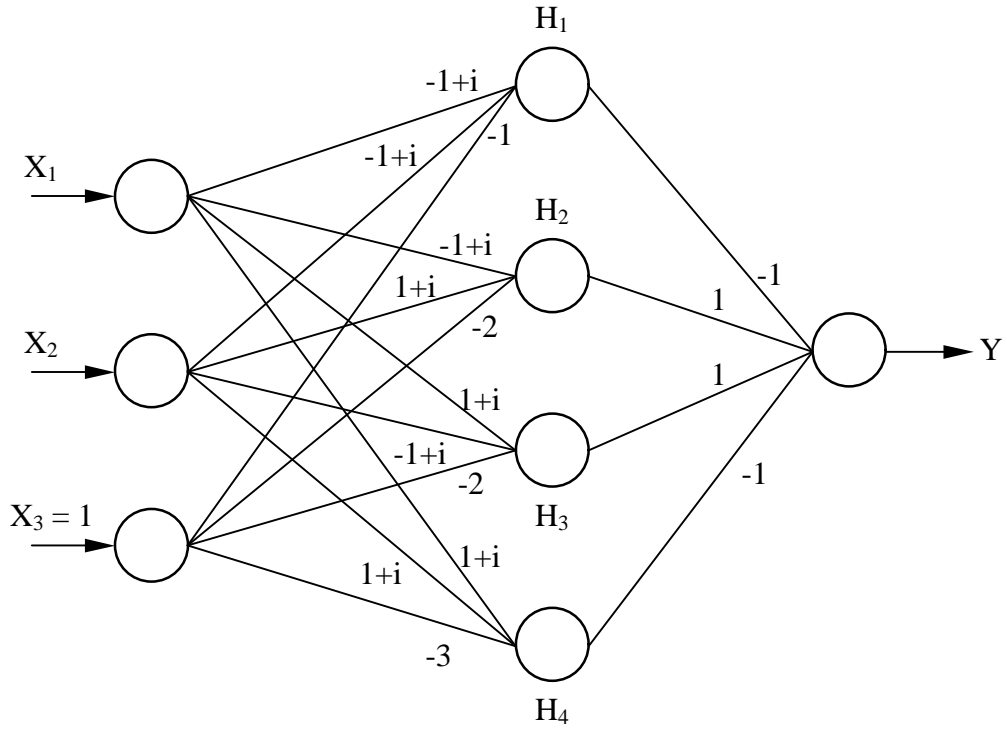


Figure 5.1: The Network Architecture for Example 5.1

Table 5.2: Network Parameters in the input/output mapping of Example 5.1

| | | | | Input to | | | | Output of | | | | Input | Output | | | |
|--------|-------|---|---|----------|--------|------|-------|-----------|-------|-------|-------|-------|--------|-------|------|------|
| Inputs | | s | | Weights | | | H_1 | H_2 | H_3 | H_4 | H_1 | H_2 | H_3 | H_4 | to y | of y |
| i | i | 1 | 2 | $-1+i$ | $-1+i$ | -1 | 1 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | -1 | 0 |
| i | $1+i$ | 1 | 3 | $-1+i$ | $1+i$ | -2 | 0 | 1 | -1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| $1+i$ | i | 1 | 3 | $1+i$ | $-1+i$ | -2 | 0 | -1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| $1+i$ | $1+i$ | 1 | 4 | $1+i$ | $1+i$ | -3 | -1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | -1 | 0 |

inputs need to be mapped to just one output in each case and thus only one output neuron is used here. Since no generalization is required we have $r = 0$. The weights are assigned according to the algorithm and the network is then tested with all inputs. It is seen that all inputs have been successfully mapped to their outputs.

Example 5.2: Network with two output neurons

The 3C algorithm can also be used to train a network with more than one output neuron. The inputs and outputs are shown in Table 5.3. The input vectors have five elements and the corresponding output vectors have two elements.

Table 5.3: Inputs and outputs for Example 5.2

| Inputs | | | | | Outputs | |
|--------|-------|-------|-------|-------|---------|-------|
| X_1 | X_2 | X_3 | X_4 | X_5 | Y_1 | Y_2 |
| 0 | 1+i | 1+i | 0 | i | 1 | 1 |
| 1+i | 0 | 1 | 1+i | 1 | 0 | 1 |
| 1 | 1 | i | 0 | 1 | 1 | 0 |

A total of six input neurons are required including the bias neuron. All three samples need to be used for training and hence three hidden neurons are required. The output layer consists of two neurons. The input and output weights are assigned according to the algorithm as each training sample is presented. No generalization is required so $r = 0$. After the training, the network is tested for all inputs and outputs. Again it can be seen that the mapping is accomplished successfully. The network architecture is shown in Figure 5.2, and the various network parameters obtained during the training are tabulated in Table 5.4.

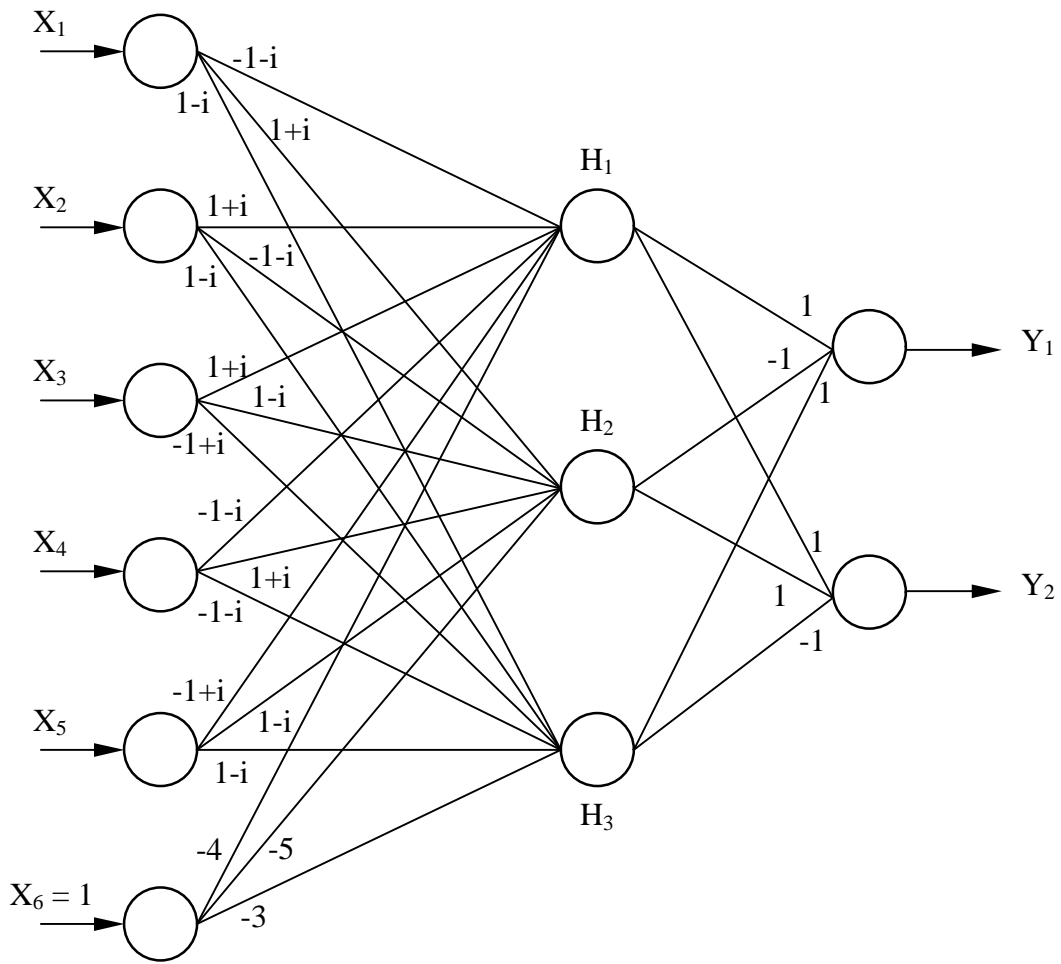


Figure 5.2: The Network Architecture for Example 5.2

Table 5.4: Network Parameters in the input/output mapping of Example 5.2

| | | | | | | | | | | | Input to | | | Output of | | | Output | | | |
|--------|-----|-----|-----|---|---|---------|------|------|------|------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|---|---|
| Inputs | | | | | s | Weights | | | | | H ₁ | H ₂ | H ₃ | H ₁ | H ₂ | H ₃ | y ₁ | y ₂ | | |
| 0 | 1+i | 1+i | 0 | i | 1 | 5 | -1-i | 1+i | 1+i | -1-i | -1+i | -4 | 1 | -8 | -4 | 1 | 0 | 0 | 1 | 1 |
| 1+i | 0 | 1 | 1+i | 1 | 1 | 6 | 1+i | -1-i | 1-i | 1+i | 1-i | -5 | -8 | 1 | -5 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | i | 0 | 1 | 1 | 4 | 1-i | 1-i | -1+i | -1-i | 1-i | -3 | -4 | -5 | 1 | 0 | 0 | 1 | 1 | 0 |

Example 5.3:

The Table 5.5 displays the inputs and their corresponding outputs used in Example 4.3. The 3C algorithm accomplishes the mapping of the second input vector to its output, where the previous algorithm failed. Since the combination of the inputs and their weights results in real inputs to the hidden neurons, the weights of the third vector does not affect the second vector.

Table 5.5: Inputs and outputs for Example 5.3

| Inputs | | | | | Outputs |
|--------|-------|-------|-------|-------|---------|
| X_1 | X_2 | X_3 | X_4 | X_5 | Y |
| 1 | i | 1 | 0 | 1+i | 1 |
| i | 1 | 1+i | 0 | i | 1 |
| i | 1 | 1 | 0 | 0 | 0 |

Figure 5.3 shows the network architecture for the example, and the different network parameters are tabulated in Table 5.6. In this table the cell holding the output of the second vector is shaded to denote the successful classification.

Example 5.4:

This example uses the same table of inputs and outputs as Example 4.4. Table 5.7 holds the inputs and their corresponding outputs. The third input vector in the table is successfully mapped to its output because the weights of the second input vector do not affect it. The network architecture is shown in Figure 5.4, and the different network parameters are tabulated in Table 5.8. The output of the third vector is shaded to denote the successful classification.

From these examples we see that the 3C algorithm has eliminated the need for either of the restrictions mentioned in Chapter 4.

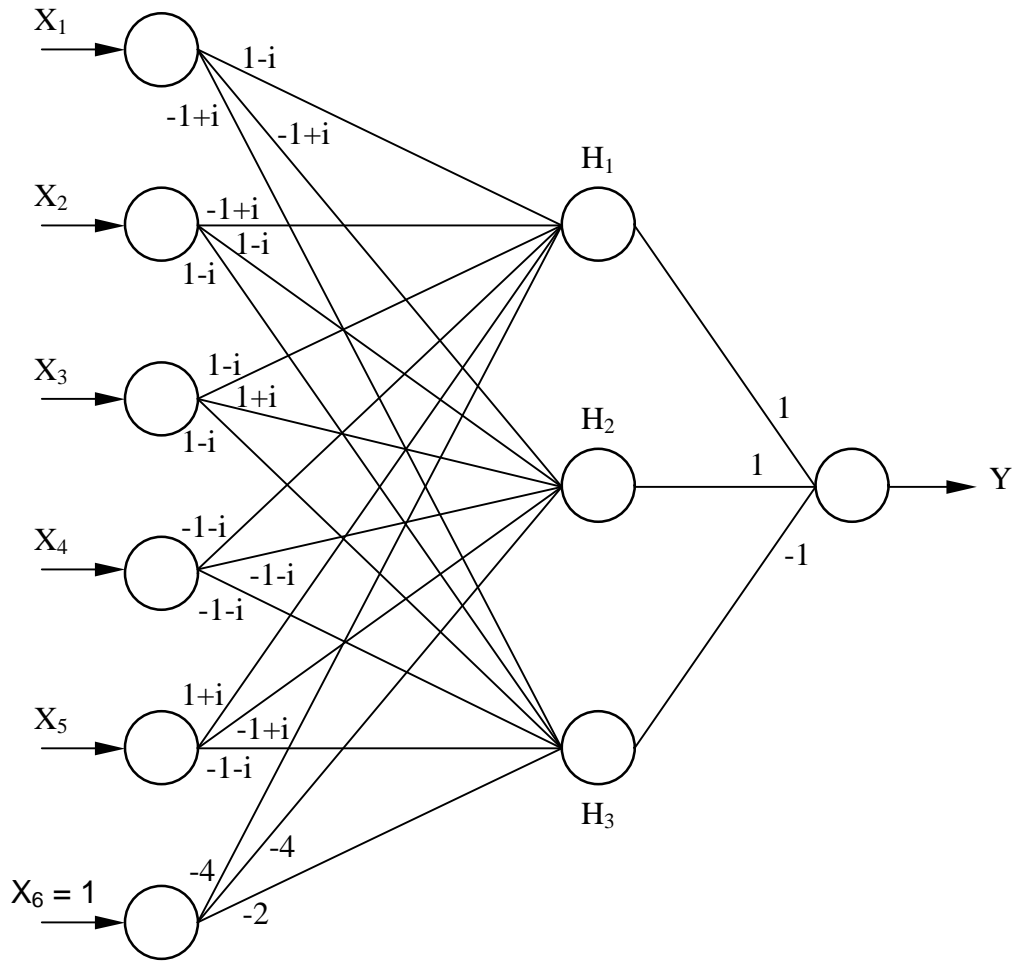


Figure 5.3: The Network Architecture for Example 5.3

Table 5.6: Network Parameters in the input/output mapping of Example 5.3

| | | | | | | | Input to | | | Output of | | | Output | | | | | | |
|--------|---|-----|---|-----|---|---------|----------|------|-----|-----------|------|----|----------------|----------------|----------------|----------------|----------------|----------------|----------|
| Inputs | | | | | s | Weights | | | | | | | H ₁ | H ₂ | H ₃ | H ₁ | H ₂ | H ₃ | Y |
| 1 | i | 1 | 0 | 1+i | 1 | 5 | 1-i | -1+i | 1-i | -1-i | 1+i | -4 | 1 | -5 | -5 | 1 | 0 | 0 | 1 |
| i | 1 | 1+i | 0 | i | 1 | 5 | -1+i | 1-i | 1+i | -1-i | -1+i | -4 | -5 | 1 | -1 | 0 | 1 | 0 | 1 |
| i | 1 | 1 | 0 | 0 | 1 | 3 | -1+i | 1-i | 1-i | -1-i | -1-i | -2 | -5 | -1 | 1 | 0 | 0 | 1 | 0 |

Table 5.7: Inputs and outputs for Example 4.4

| Inputs | | | | | Outputs |
|--------|-------|-------|-------|-------|---------|
| X_1 | X_2 | X_3 | X_4 | X_5 | Y |
| 1 | i | 1 | 0 | $1+i$ | 1 |
| i | 1 | i | 1 | 0 | 0 |
| i | 1 | $1+i$ | $1+i$ | 0 | 1 |

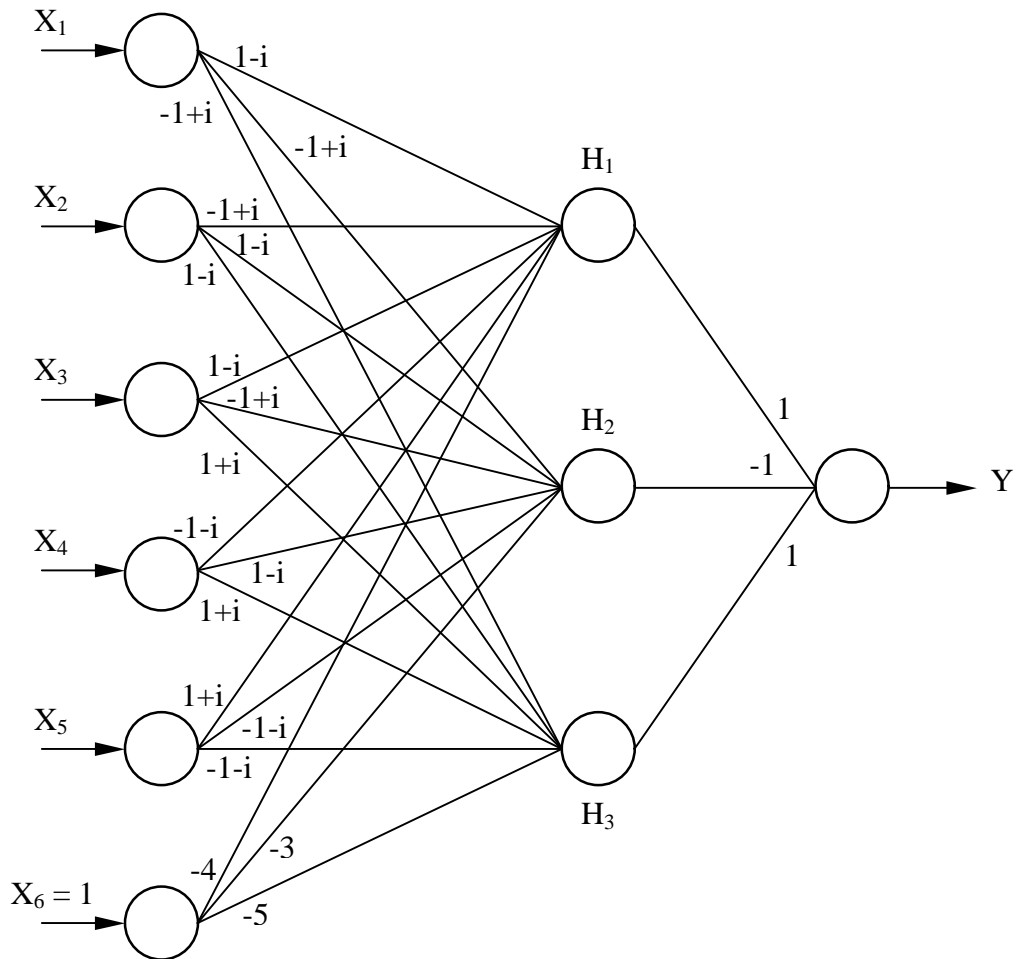


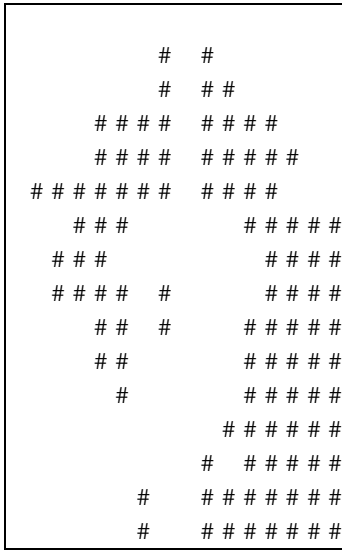
Figure 5.4: The Network Architecture for Example 5.4

Table 5.8: Network Parameters in the input/output mapping of Example 5.4

| | | | | | | | | | | | | Input to | | | Output of | | | Output | |
|--------|---|-----|-----|-----|---|---|---------|------|------|------|------|----------|----------------|----------------|----------------|----------------|----------------|----------------|----------|
| Inputs | | | | | | s | Weights | | | | | | H ₁ | H ₂ | H ₃ | H ₁ | H ₂ | H ₃ | Y |
| 1 | i | 1 | 0 | 1+i | 1 | 5 | 1-i | -1+i | 1-i | -1-i | 1+i | -4 | 1 | -8 | -8 | 1 | 0 | 0 | 1 |
| i | 1 | i | 1 | 0 | 1 | 4 | -1+i | 1-i | -1+i | 1-i | -1-i | -3 | -8 | 1 | 0 | 0 | 1 | 0 | 0 |
| i | 1 | 1+i | 1+i | 0 | 1 | 6 | -1+i | 1-i | 1+i | 1+i | -1-i | -5 | -8 | 0 | 1 | 0 | 0 | 1 | 1 |

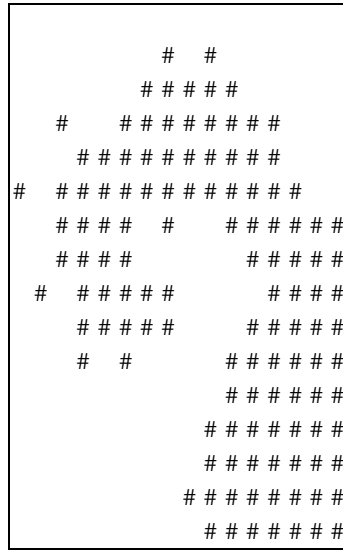
To analyze the generalization capability of the 3C algorithm, the pattern classification experiments explained in Chapter 4 are repeated here. The classification of the spiral pattern is done first. The points in the 16 by 16 pattern area are represented by their row and column coordinates and both the encoding schemes used in the previous chapter are used here. In each case, the same 75 training samples from each pattern used earlier are used here too. Thus all networks are similar to the ones used in Chapter 4, with the same number of neurons in each of the layers. Figures 5.5 (c) to (f) and 5.6 (a) to (d), show the results of the spiral pattern classification using the two encoding schemes when the value of r is varied from 1 to 4. It is interesting to note that Figure 5.5 is identical to Figure 2.4. However this is expected as the same samples are presented for learning. The results of the classification of the second pattern are presented in Figures 5.7 (c) to (f) and 5.8 (a) to (b). In all cases we see that the classification is very good. As expected for higher values of r the black region increases in size since the density of samples from the black region is greater than the density of samples from the white region. Thus the 3C algorithm accomplishes good classification while being capable of handling complex valued inputs without any restrictions. A summary of the experiments is presented in Tables 5.9 and 5.10. These tables contain the number of points classified and misclassified in the two experiments when the different encoding schemes are used.

r = 1



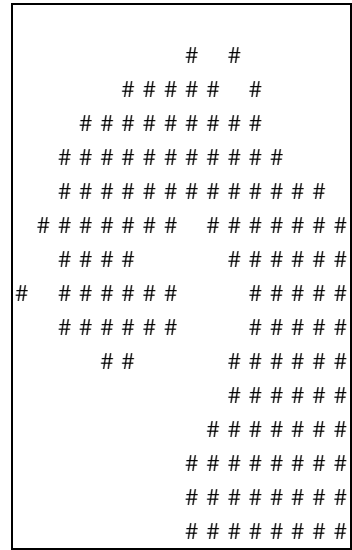
(a)

r = 2



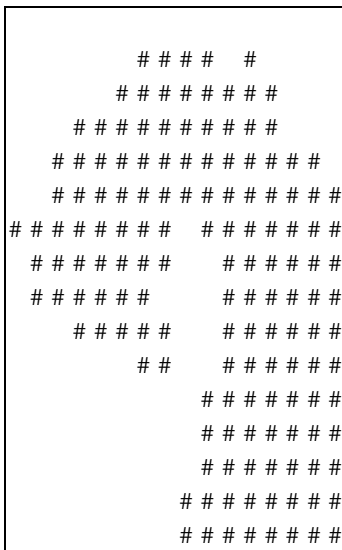
(b)

r = 3



(c)

r = 4



(d)

Figure 5.6: Results of spiral pattern classification using the quaternary encoding

Table 5.9: No. of points classified/misclassified in the spiral pattern

| | | No. of points | | | |
|----------------------------|---------------|---------------|---------|---------|---------|
| | | $r = 1$ | $r = 2$ | $r = 3$ | $r = 4$ |
| First Encoding Scheme | Classified | 236 | 234 | 227 | 207 |
| | Misclassified | 20 | 22 | 29 | 49 |
| Quaternary Encoding Scheme | Classified | 230 | 232 | 233 | 220 |
| | Misclassified | 26 | 24 | 23 | 36 |

Table 5.10: No. of points classified/misclassified in the second pattern

| | | No. of points | | | |
|----------------------------|---------------|---------------|---------|---------|---------|
| | | $r = 1$ | $r = 2$ | $r = 3$ | $r = 4$ |
| First Encoding Scheme | Classified | 229 | 225 | 207 | 185 |
| | Misclassified | 27 | 31 | 49 | 71 |
| Quaternary Encoding Scheme | Classified | 229 | 223 | 212 | 200 |
| | Misclassified | 27 | 33 | 44 | 56 |

When we compare the above tables with Table 4.9 and 4.10 we see that the 3C algorithm has classified many more points accurately regardless of the encoding scheme. Also on an average the quaternary scheme classifies a few more points accurately than the first encoding scheme.

Chapter 6

Time Series Prediction

The Mackey-Glass time series, originally developed to model white blood cell production as presented by Azoff [1], is commonly used to test the performance of neural networks.

The series is a chaotic time series making it an ideal representation of the nonlinear oscillations of many physiological processes. The discrete time representation of the series was used by Tang [19] to test the performance of the CC4 algorithm. The same can be used here to test the performances of the two algorithms of chapters 4 and 5. The discrete time representation of the Mackey-Glass equation is given below: -

$$x(k+1) - x(k) = \alpha x(k-\tau) / \{1 + x^\gamma(k-\tau)\} - \beta x(k)$$

The values of the different parameters in the equation are assigned as follows: -

$$\alpha = 3, \beta = 1.0005, \gamma = 6, \tau = 3$$

Since $\tau = 3$, four samples are required to obtain a new point. Thus the series is started with four arbitrary samples:

$$x(1) = 1.5, \quad x(2) = 0.65, \quad x(3) = -0.5, \quad x(4) = -0.7$$

Using these samples a series of 200 points is generated and it oscillates within the range -2 to +2. Of these 200 points, about nine tenths are fed to the networks designed by the two algorithms for training. Then the networks are tested using the remaining points. In the training and the testing, four consecutive points in the series are given as input and the next point is used as the output. Thus a sliding window of size four is used at each and every step. So if nine tenths of the points are to be used for training, the total number

of sliding windows available is 175, where the first window consists of points 1 to 4 with the 5th as the output, and the last window consists of points 175 to 178 with the 179th point as the output.

The range of the series is divided into 16 equal regions and a point in each region can be represented by the index of the region. These indices ranging from 1 to 16 can be represented using the quaternary encoding scheme discussed in Chapter 3. Since four points are required in each training or testing, the 5 character codewords for each of the four inputs are concatenated together. Thus each input vector has 21 elements, where the last element in the vector represents the bias. Unlike the inputs, output points are binary encoded using four bits. This is done to avoid the possibility of generating invalid output vectors that would not belong to the class of expected vectors of the quaternary encoding scheme. Hence 21 neurons are required in the input layer, 175 in the hidden layer (one for each sliding window), and 4 in the output layer.

After the training, the two networks are tested using the same 175 windows to check their learning ability. Then the rest of the windows are presented to the networks to predict future values. The inputs are always points from the original series calculated by the Mackey-Glass equation to avoid an error buildup. The outputs of the network are compared against the expected values in the series. The performance of the first algorithm for r varying as 4, 5, 6 and 7 is shown in Figures 6.1, 6.2, 6.3 and 6.4 respectively. Similarly the performance of the 3C algorithm for different values of r is presented in the Figures 6.5, 6.6, 6.7 and 6.8. The values of r are again varied as 4, 5, 6 and 7 respectively.

In each of the figures only points 160 to 200 are shown for readability. The solid

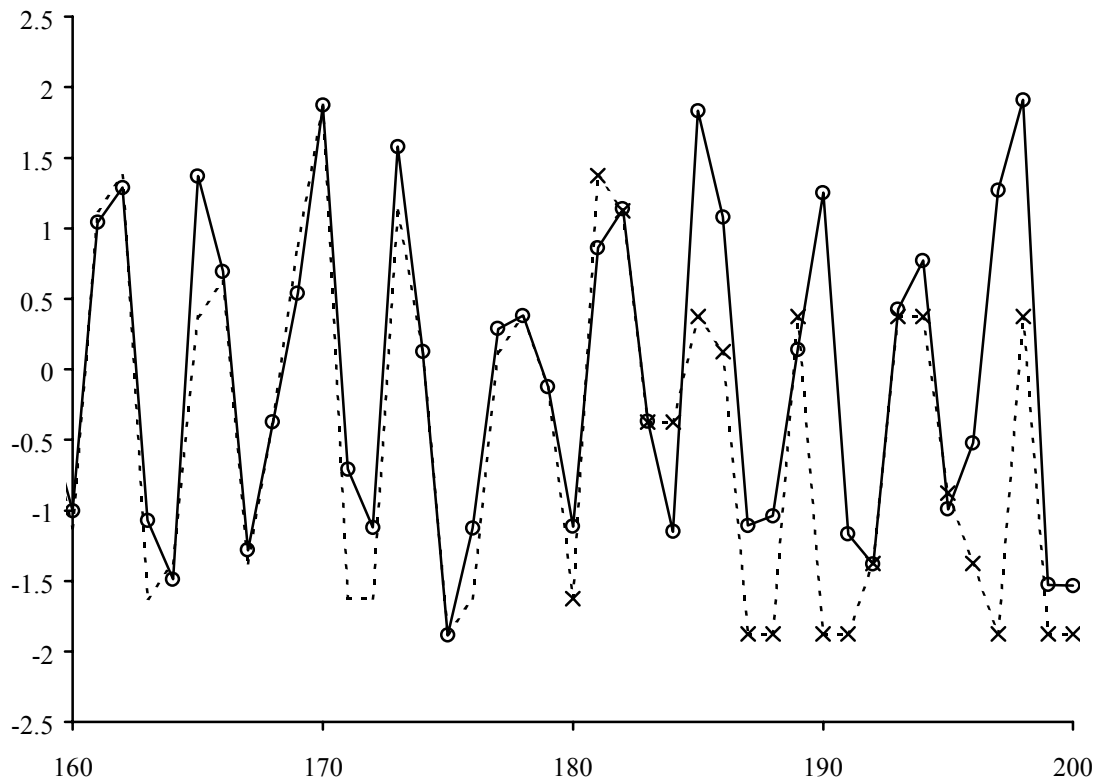


Figure 6.1: Mackey-Glass time series prediction using the first algorithm, $r = 4$
 Dotted line till point 180 – training samples
 “o” – Actual data, “x” – predicted data

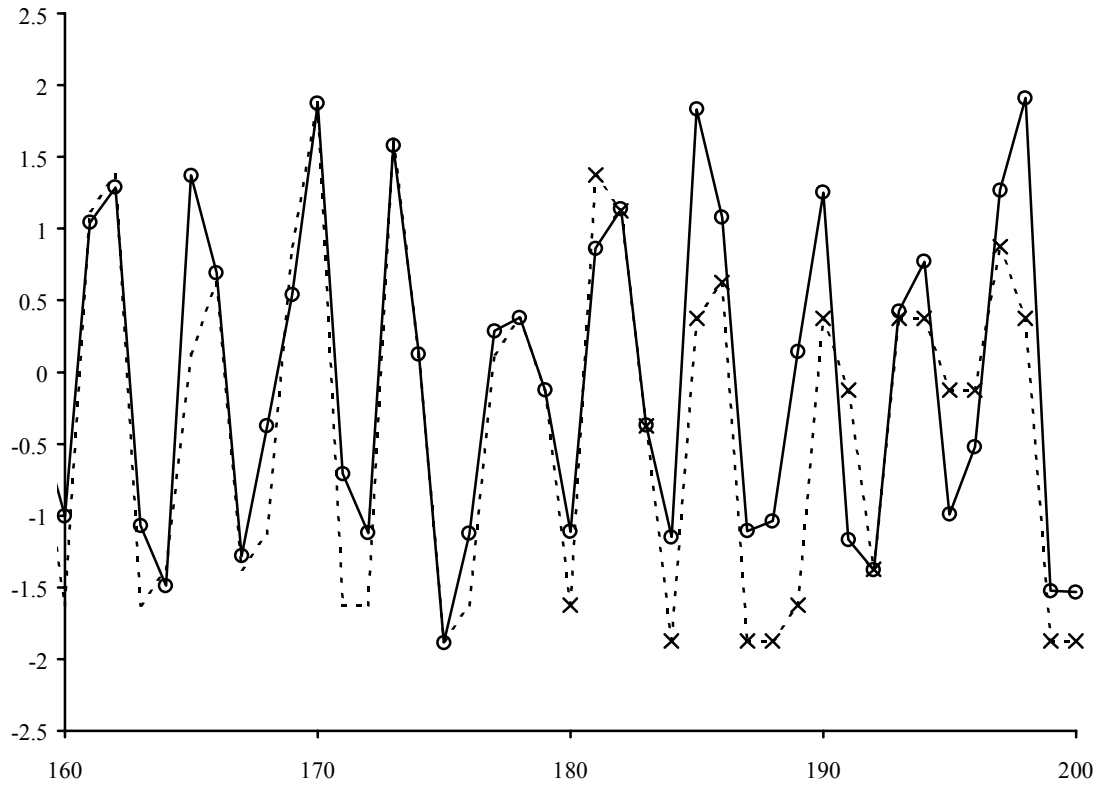


Figure 6.2: Mackey-Glass time series prediction using the first algorithm, $r = 5$
 Dotted line till point 180 – training samples
 “o” – Actual data, “x” – predicted data

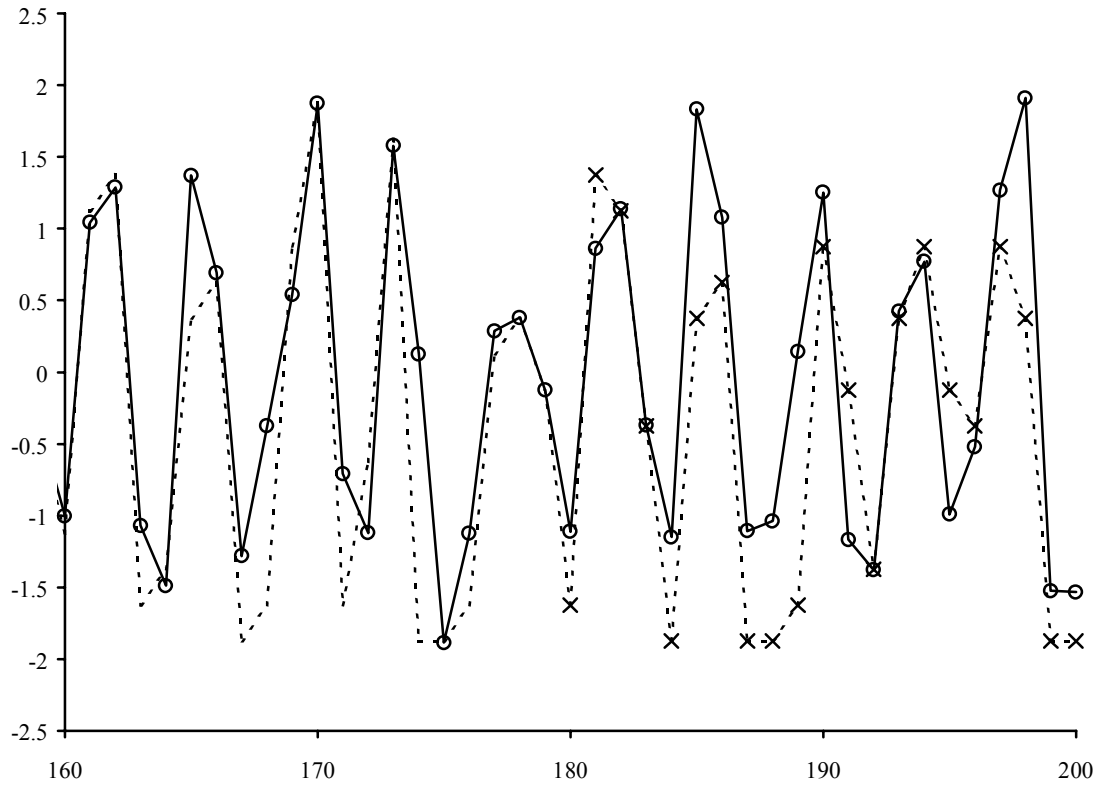


Figure 6.3: Mackey-Glass time series prediction using the first algorithm, $r = 6$
 Dotted line till point 180 – training samples
 “o” – Actual data, “x” – predicted data

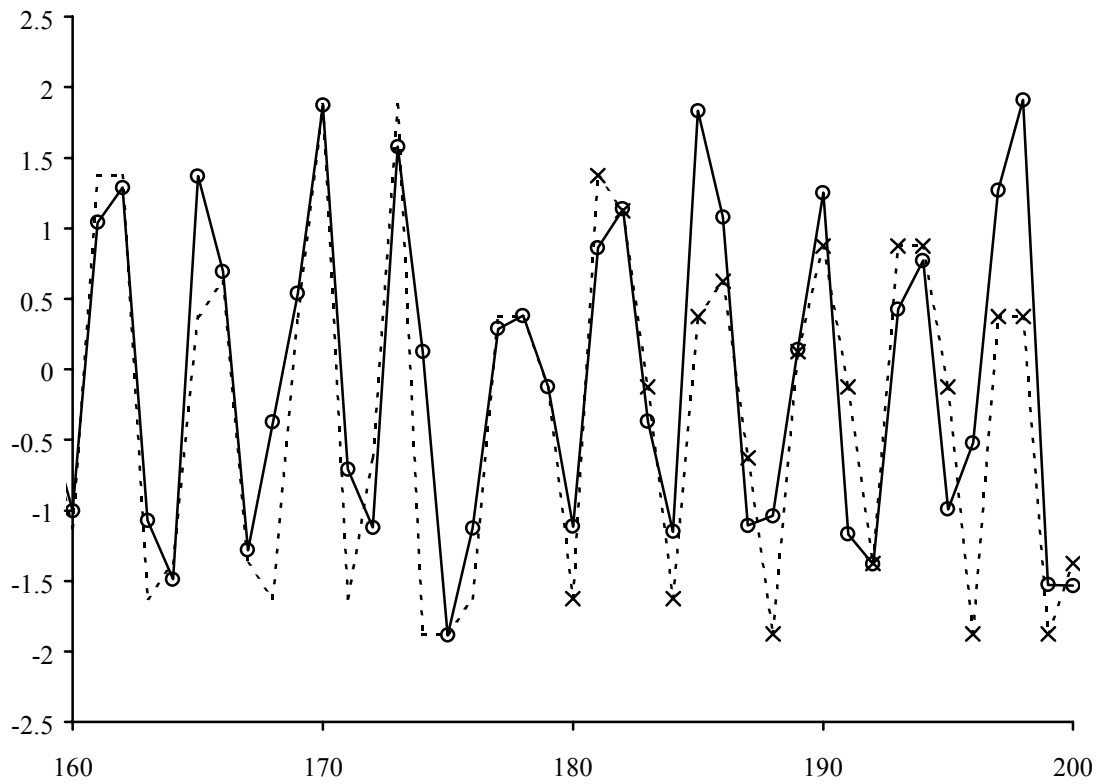


Figure 6.4: Mackey-Glass time series prediction using the first algorithm, $r = 7$
 Dotted line till point 180 – training samples
 “o” – Actual data, “x” – predicted data

line represents the original series and the lighter line represents the outputs of the networks designed by the two algorithms. The lighter line from point 160 to 179 shows how well both the networks have learnt the samples for different values of r . The points that are predicted by the networks are represented by a “×” on the lighter line. The actual points generated by the Mackey-Glass equation are represented by a “o” on the solid line. The first point that is predicted is the point number 180 using the original series points 176, 177, 178 and 179. The next point that is predicted is 181 using the points 177, 178, 179 and 180. The point number 180, which is used as input here is the original point in the series generated by the Mackey-Glass equation and not the point predicted by the network. Similarly the last point to be predicted is the point number 200 using the actual points 196 to 199 from the series. The networks always predict one point ahead of time and both the networks predict the turning points in the series efficiently. Thus both the networks are capable of learning the quasi-periodic property of the series. This ability is of great importance in financial applications, where predicting the turning point of the price movement is more important than predicting the day to day values.

If we compare the performance of the two networks, we see that though the first algorithm’s network predicts the turn of the series almost as accurately as the 3C network, it fails to predict many individual points successfully. Again the restrictions on the first algorithm’s inputs affect the prediction. The 3C network, on the other hand, predicts most of the points with high accuracy.

Stability of the networks is another important feature in deciding the network performance and is governed by the consistency of the outputs when network parameters are changed. The value of r is repeatedly changed in both the algorithms and from the

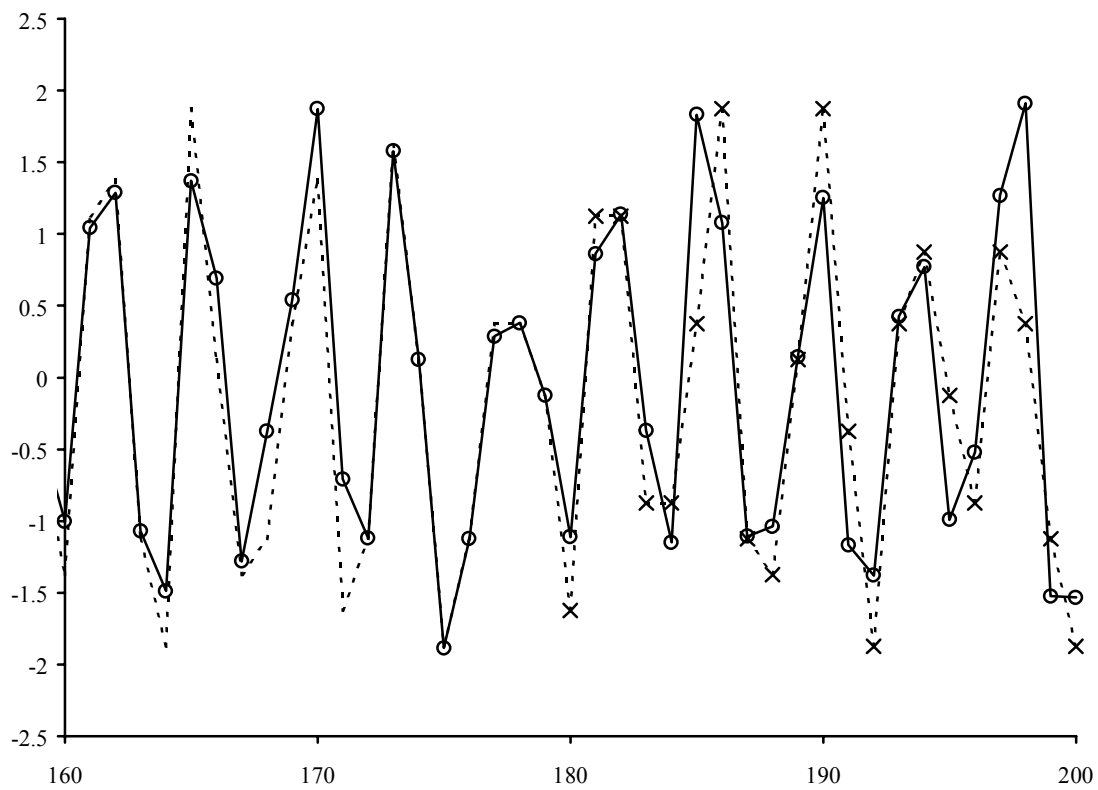


Figure 6.5: Mackey-Glass time series prediction using 3C, $r = 4$
 Dotted line till point 180 – training samples
 “o” – Actual data, “x” – predicted data

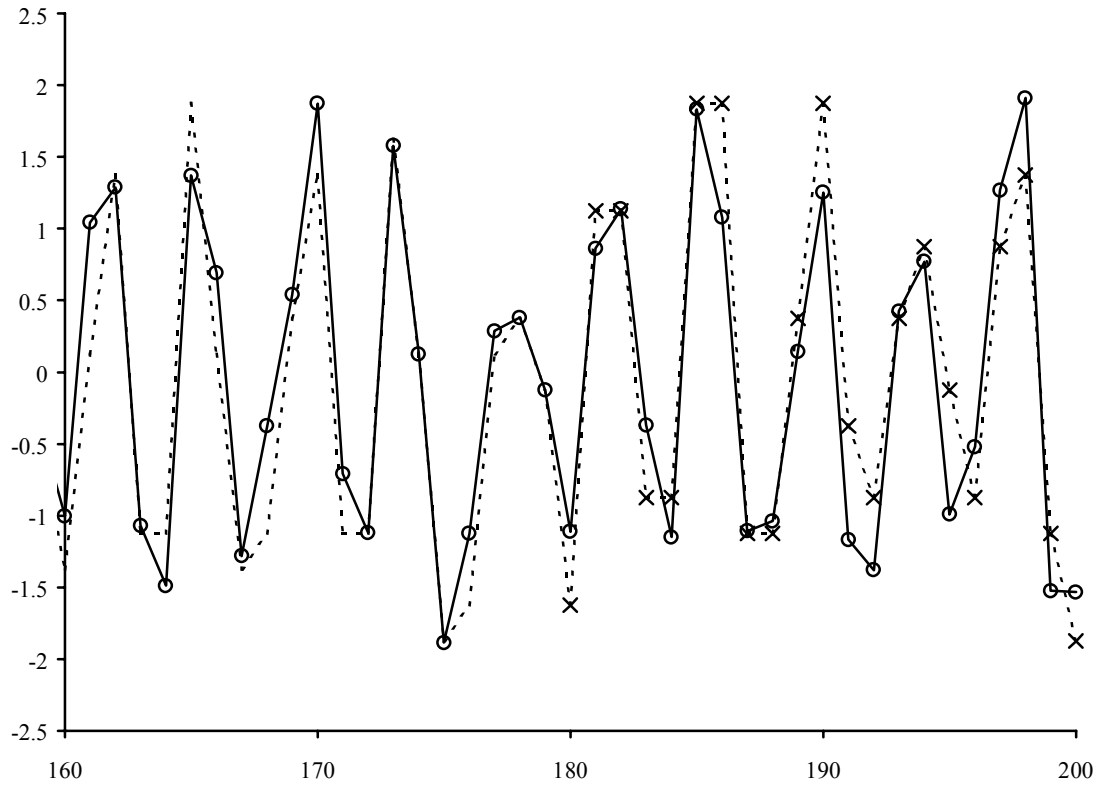


Figure 6.6: Mackey-Glass time series prediction using 3C, $r = 5$
 Dotted line till point 180 – training samples
 “o” – Actual data, “x” – predicted data

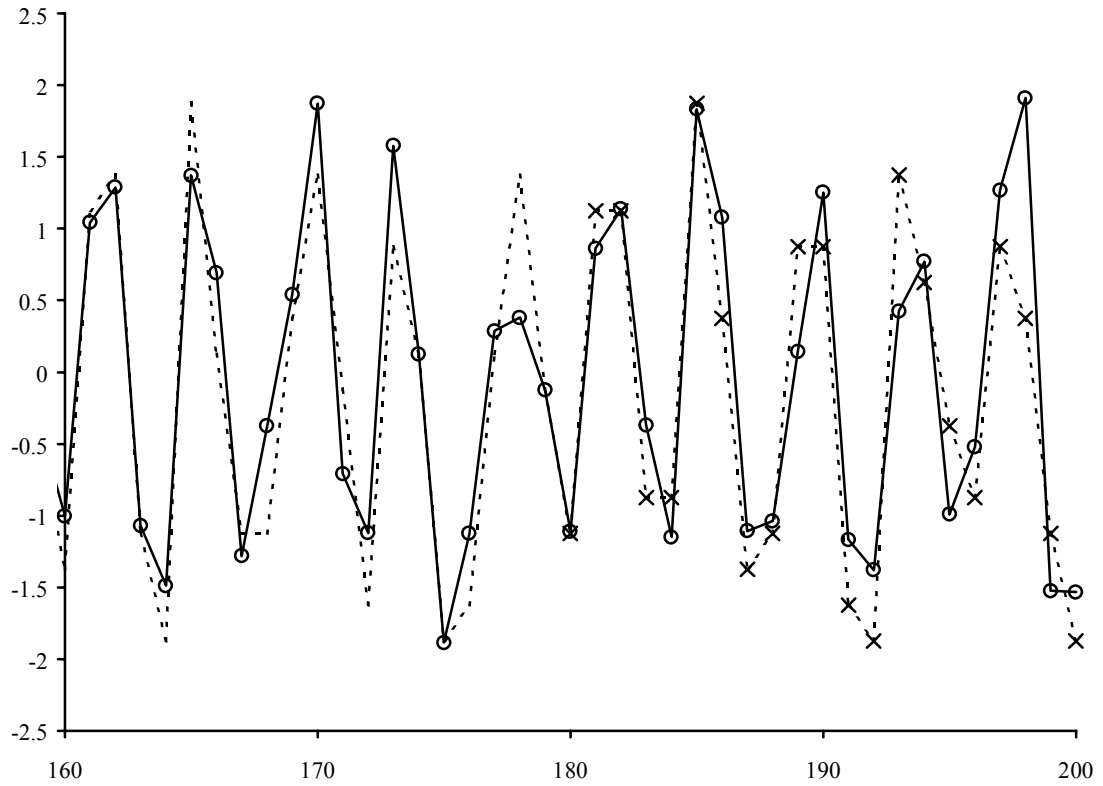


Figure 6.7: Mackey-Glass time series prediction using 3C, $r = 6$
 Dotted line till point 180 – training samples
 “o” – Actual data, “x” – predicted data

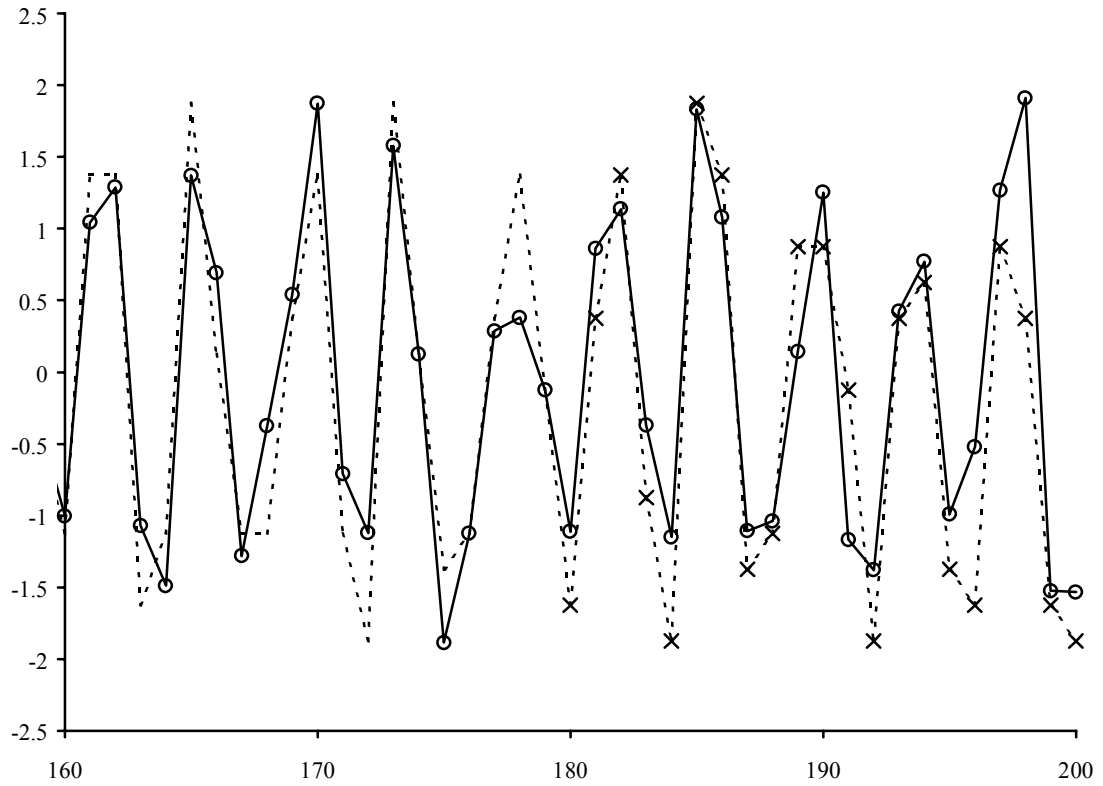


Figure 6.8: Mackey-Glass time series prediction using 3C, $r = 7$
 Dotted line till point 180 – training samples
 “o” – Actual data, “x” – predicted data

figures it is clear that the networks perform appreciably well. Further if the value of r is increased to 10 the network performance is still quite good. The results for both the algorithms when $r=10$, are presented in Figures 6.9 and 6.10. Thus we see that the stability of the networks designed by either of the algorithm is not sensitive to the value of the radius of generalization r .

The normalized mean square error of the points predicted by both algorithms for each value of r is shown in Table 6.1. It is clear that the 3C algorithm has predicted the points with much greater accuracy than the first algorithm.

Table 6.1: Normalized mean square error for both algorithms

| | Normalized Mean Square Error of predicted points for varying r | | | | |
|---------------------|------------------------------------------------------------------|---------|---------|---------|----------|
| | $r = 4$ | $r = 5$ | $r = 6$ | $r = 7$ | $r = 10$ |
| The first Algorithm | 0.0904 | 0.04 | 0.0365 | 0.0331 | 0.0307 |
| The 3C Algorithm | 0.0238 | 0.0112 | 0.0193 | 0.0221 | 0.0275 |

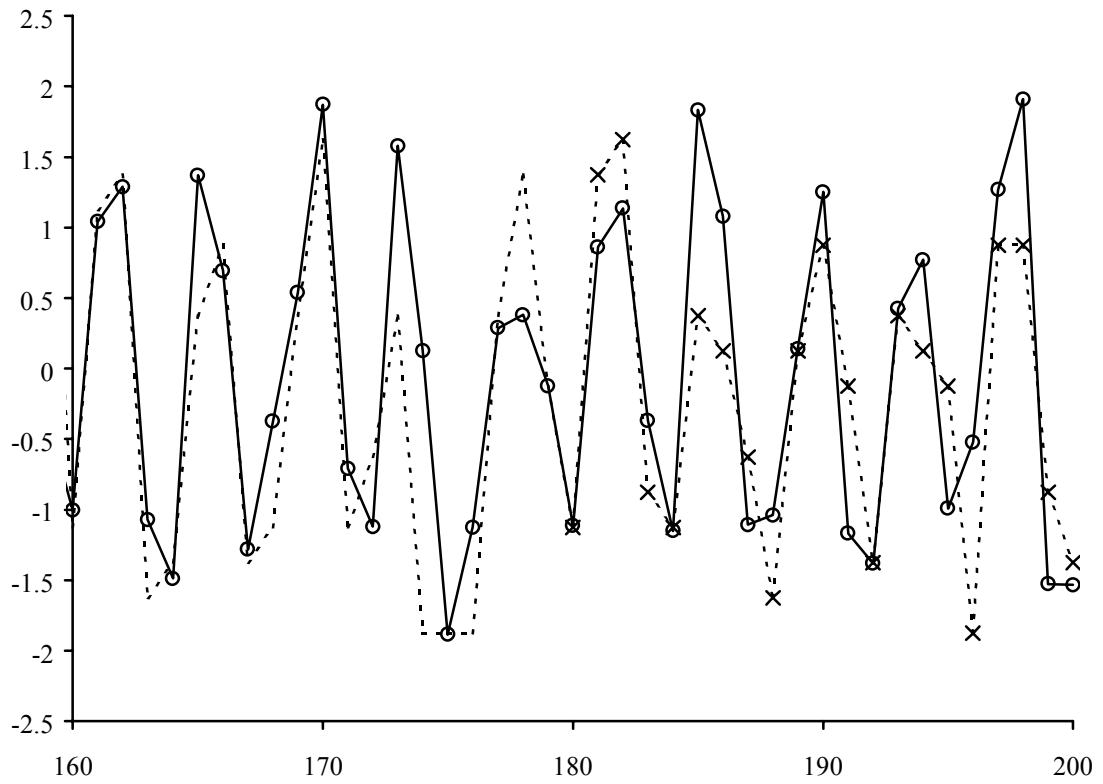


Figure 6.9: Mackey-Glass time series prediction using the first algorithm, $r = 10$

Dotted line till point 180 – training samples

“o” – Actual data, “x” – predicted data

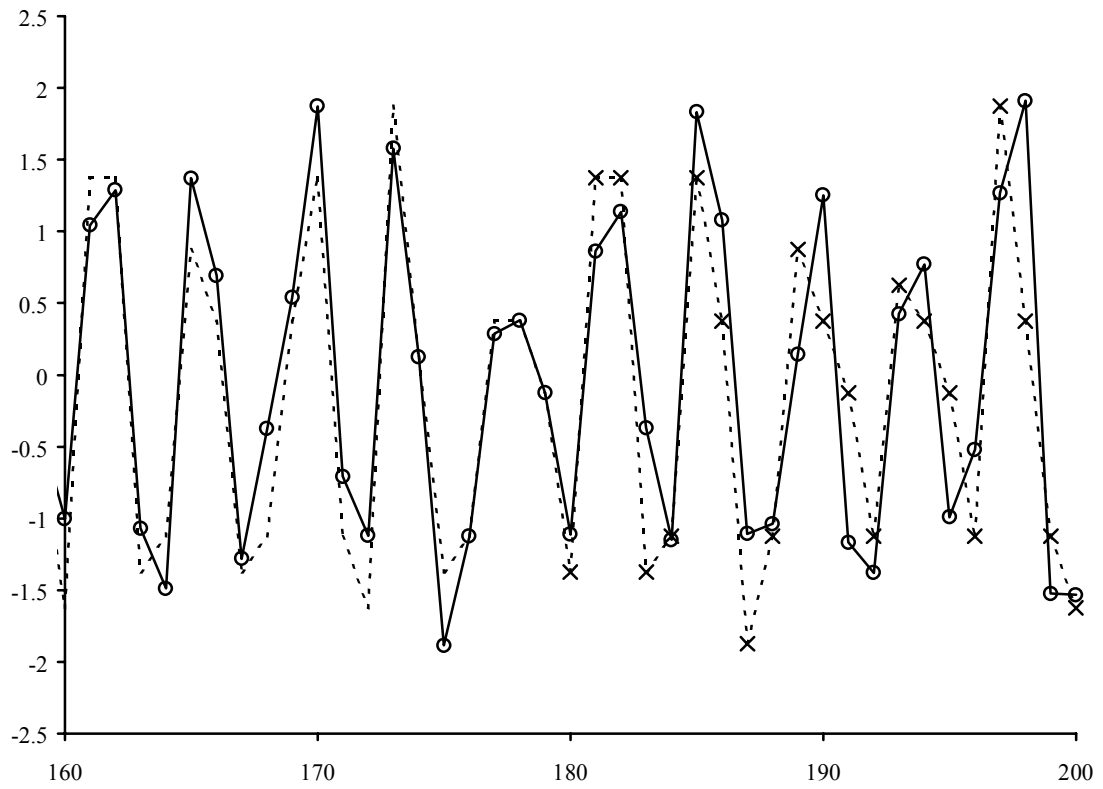


Figure 6.10: Mackey-Glass time series prediction using 3C, $r = 10$
 Dotted line till point 180 – training samples
 “o” – Actual data, “x” – predicted data

Chapter 7

Conclusion

Previously, training of complex input neural networks was done using techniques like the backpropagation and perceptron learning rules. These techniques require lot of time and resources to complete the training. This thesis presents two algorithms for instantaneously trained complex neural networks based on the corner classification approach. The first algorithm, a straightforward generalization of the CC4, performs well only when certain inputs are restricted. The performance of this algorithm was tested using two pattern classification experiments and the results were not satisfactory. Our second algorithm, the 3C algorithm, does not place any restrictions on the inputs. Its performance was tested using the same pattern classification experiments and its generalization capability was found to be quite satisfactory.

Encoding techniques for the input data were discussed and a new encoding technique called quaternary encoding was introduced. This technique makes some modifications to unary encoding so as to accommodate all four characters of the input alphabet. Using this technique, the 3C algorithm performs very well in the pattern classification and time series experiments. An added benefit of this encoding is that the number of input neurons required is greatly reduced when compared to the networks designed using CC4 for the same problems.

Like the CC4, the two algorithms have their limitations. The first algorithm places restrictions on the inputs making it unsuitable for practical purposes. The 3C

algorithm while being very efficient can handle only four inputs. Also, as with the CC4, a network of the size required by the 3C algorithm poses a problem with respect to hardware implementation. However its high suitability of software implementation due to low requirement of computational power and its instantaneous training makes up for the limitations.

In the future the 3C algorithm should be modified and adapted to handle non-binary complex inputs. This would remove the need for encoding the inputs in many cases and greatly increase the number of areas of application. The 3C algorithm can be applied in the following scenarios:

Financial Analysis – In most financial applications it is not enough to predict the future price of an equity or commodity; it is more useful to predict exactly when the directionality of price values will change. One could define two modes of behavior, namely the *up* and *down* trends and represent them by the imaginary 0 and 1. Within a trend, the peak and trough could be represented by the real 0 and 1. This gives us four states 0, 1, i and $1+i$, which can be presented as inputs to the 3C algorithm.

Communications and Signal Processing – Since the inputs handled by the 3C are complex, the inputs of many passband modulation schemes could be directly applied to the 3C feedforward network.

Bibliography

- [1] E. M. Azoff, *Neural network time series forecasting of financial markets*, John Wiley and Sons, New York 1994.
- [2] N. Benvenuto and F. Piazza, *On the complex back propagation algorithm*, IEEE Trans. Signal Process, 40, 967-969, 1992.
- [3] G. M. Georgiou, *Parallel Distributed Processing in the Complex Domain*, Ph.D. Dissertation, Department of Computer Science, Tulane University, April 1992.
- [4] M. T. Hagan, H. B. Demuth and M. Beale, *Neural Network Design*, PWS Publishing Company, Massachusetts, 1996.
- [5] S. C. Kak, *New training algorithm in feedforward neural networks*, First International Conference on Fuzzy Theory and Technology, Durham, N. C., October 1992. Also in Wang, P.P. (Editor), *Advance in fuzzy theory and technologies*, Durham, N. C. Bookwright Press, 1993.
- [6] S. C. Kak, *On training feedforward neural networks*, Pramana J. Physics, 40, 35-42, 1993.
- [7] S. C. Kak, *New algorithms for training feedforward neural networks*, Pattern Recognition Letters, 15, 295-298, 1994.
- [8] S. C. Kak, *On generalization by neural networks*, First International Conference on Computational Intelligence and Neuroscience, North Carolina, 1995.
- [9] S. C. Kak and J. Pastor, *Neural networks and methods for training neural networks*, U. S. Patent No. 5,426,721, June 20, 1995.
- [10] S. C. Kak, *A class of instantaneously trained neural networks*, Information Sciences, 148, 97-102, 2002.
- [11] M. S. Kim and C. C. Guest, *Modification of Backpropagation for complex-valued signal processing in frequency domain*, International Joint Conference on Neural Networks, (San Diego, CA), pp. III-27 – III-31, June 1990.
- [12] H. Leung and S. Haykin, *The complex back propagation algorithm*, IEEE Trans. Signal Process, 39, 2101-2104, 1991.

- [13] C. Li, X. Liao and J. Yu, *Complex-Valued Recurrent Neural Network with IIR Neuron Model: Training and Applications*, *Circuits Systems Signal Processing*, 21, 461-471, 2002.
- [14] G. R. Little, S. C. Gustafson and R. A. Senn, *Generalization of the back propagation neural network learning algorithms to permit complex weights*, *Applied Optics*, 29, 1591-1592, 1990.
- [15] A. J. Noest, *Phasor Neural Networks*, *Neural Information Processing Systems* (D.Z. Anderson, ed.), (Denver 1987), pp.584-591, American Institute of Physics, New York, 1988.
- [16] A. J. Noest, *Associative memory in sparse phasor neural networks*, *Europhysics Letters*, 6, 469-474, 1988.
- [17] A. J. Noest, *Discrete-state phasor neural nets*, *Physical Review A*, 38, 2196-2199, 1988.
- [18] J. G. Sutherland, *A holographic model of memory, learning and expression*, *International Journal of Neural Systems*, 1, 259-267, 1990.
- [19] K. W. Tang, *CC4: A new Corner Classification approach to neural network training*, Master's Thesis, Department of Electrical and Computer Engineering, Louisiana State University, May 1997.
- [20] K. W. Tang and S. C. Kak, *Fast Classification Networks for Signal Processing*, *Circuits Systems Signal Processing*, 21, 207-224, 2002.

Vita

Pritam Rajagopal received his bachelor of engineering degree in Electronics and Communications Engineering from the University of Madras, Chennai, India in 2000. He decided to pursue higher studies and was accepted by the Department of Electrical and Computer Engineering at Louisiana State University. He began the graduate program in the fall of 2000. He expects to receive the degree of Master of Science in Electrical Engineering in May 2003.