Decoding of Error Correcting codes Using Neural Networks

Atif Haroon

Fahad Hussain

Muhammad Raheel Bajwa

This thesis is presented as part of the Degree of Master of Sciences in Electrical Engineering

Blekinge Institute of Technology 2012

School of Engineering Blekinge Institute of Technology, Sweden

Supervisor: Maria Erman

Examiner: Dr. Sven Joahnsson

Page Left Blank Intentionally

ABSTRACT

Error Correcting codes are used to ensure integrity, accuracy and fault-tolerance in transmitted data. These are categorized as Block Codes and Convolutional Codes. This report primarily focuses on decoding of Block Codes, whereas Convolutional Codes have been discussed and guidelines given for their decoding.

Different techniques have been developed for correction of errors from the received data. Instead of using traditional error correcting techniques, Artificial Neural Networks have been used because of their adaptive learning, self-organization, and real time operation and to project what will most likely happen on the analogy of human brain.

A Back propagation Algorithm for the Artificial Neural Networks has been simulated using Matlab for decoding block codes. The Simulator is trained on all possible code words to detect/correct the errors.

Block Codes have systematic structure whereas Convolutional Codes are produced sequentially in which every coming bit produces some output from the encoder depending upon the input and the past state of the encoder. Therefore, same algorithm cannot be implemented on either. An approach has been developed for decoding Block codes using Artificial Neural Networks and an error vector has been calculated for updating the synaptic weights during the training.

Acknowledgement

IN THE NAME OF ALLAH, MOST BENIFICIENT AND MERCIFUL

We are really thankful to Almighty Allah who has given us with all the resources, so that we have made the thorough use for the welfare and boom of mankind.

We are also honored to have a respectable thesis supervisor Maria Erman and examiner Dr. Sven Johansson for their worthy guidance, response and support all over our thesis. Nothing can be more helpful than the support and guidance of them.

We would also like to thank our friends, school mates and especially our parents who kept backing us in every hard time.

Atif Haroon

Fahad Husssain

Muhammad Raheel Bajwa

Page Left Blank Intentionally

Contents

Chapter1: THESIS INTRODUCTION ………………………………………………..…1

1.1 Introduction ……………………………………………………………………… 1

1.2 Error Correcting Codes ……………………………………………………………. 1

1.2.1 Basic Concept of Error Correcting System …………….…………………………………. 2

1.2.2 Need for Error Correction ………………………………………………………………. 2

1.2.3 Error Correction and Fault-tolerance ………………………………………………… 3

1.3 Motivation ……………………………………………………………………….…... 2

1.4 Artificial Neural Networks ……………………………………………………......… 3

1.5 Goals and Specifications ……………………………………………………..…. 3

1.6 Thesis Layout ………………………………………………………………..…..... 4

Chapter 2: LINEAR BLOCK CODES ……………………………………………………….. 5

2.1 Introduction ……………………………………………………………………….. 5

2.2 Linear Block Codes – Encoding…………………………………………………... 5

2.3 Standard Arrays………………………………………………………………………… 11

2.4 Syndrome Decoding …………………………………………………………………….. 12

2.4.1 Syndrome Tables ……………………………………………………………….... 12

2.5 Weight Distributions and Error Probabilities………...……………………………………13

CHAPTER 3: CONVOLUTIONAL CODES ……………………………………………………… 14

3.1 Introduction …………………………………………………………………………….. 14

3.2 Coding and Decoding with Convolutional Codes …………………………….……….. 14

3.2.1 Code Parameters and Structure of Convolutional Code ………………………………..…... 15

3.3 Coding an Incoming Sequence....…………………………………………………………….. 16

3.3.1 The Encoder Design ……………………………………………………………………… 18

3.3.2 State Diagram …………………………………………………………………………… 20

3.3.3 Tree Diagram …………………………………………………………………………… 22

3.3.4 Trellis Diagram ………………………………………………………………………….. 23

3.4 Decoding using Viterbi Algorithm ...………………………………………………………. 24

CHAPTER 4: ARTIFICIAL NEURAL NETWORKS …………………………………………...25

4.1 Introduction ……………………………………………………………………………….. 25

4.2 Basic Features of Artificial Neural Networks………………………………..…………25

4.3 Learning Process of Human Brain ………………………………………………………. 25

4.4 Basic Neuron Model ……………………………………………………………………. 26

4.5 Backpropagation Algorithm for Multilayer Perceptrons ………………………………… 28

CHAPTER 5: DECODING BLOCK CODES USING NEURAL NETWORKS …………….. ..29

5.1 Introduction ……………………………………………………………………………..... 29

5.2 Backpropagation Network ……………………………………………………………….. 29

5.2.1 Mapping ………………………………………………………………………………….. 29

5.2.2 Layout ……………………………………………………………………………………. 30

5.2.3 Training ………………………………………………………………………………..... 30

5.3 Adjustment of Weights of Connections from a Neuron in the Hidden Layer …………… 31

5.4 Adjustment of Weights of Connections from a Neuron in the Input Layer ……...…… …31

5.5 Adjustments to Threshold Values or Biases ……………………………………………..… 32

5.6 Notation and Equations …………………………………………………………..... 32

5.6.1 Notation ...……………………………………………………………………….....33

5.6.2 Equation ….……………………………………………………………………….34

5.7 Objectives Specified for Backpropagation Simulator ……………………………………..35

5.8 How to Use the Simulator ………………………………………………………….....35

5.8.1 Training Mode …………………………………………………………………......35

5.8.2 Non-training Mode (Test Mode) ………………………………………………………. 35

5.8.3 Operation ………………………………………………………………………… .36

5.9 Summary of Files used in the Backpropagation Simulators …………………………….. 36

5.9.1 Weights.dat …….………………………………………………………………. .... .36

5.9.2 Training.dat ….…………………………………………………………………. ….37

5.9.3 Test.dat ………………………………………………………………………… 37

5.9.4 Output.dat ………………………………………………………………………… 37

CHAPTER 6: NEURAL NETWORK DECODING APPROACH FOR CONVOLUTIONAL CODES …………………………………………………………………………………….. …38

6.1 Introduction ……………………………………………………………………… …38

6.2 Convolutional Codes ……………………………………………………………. … 38

6.3 Design Approach ………………………………………………………………… 39

6.3.1 Neural Network Architecture Selection ………………………………………………. ….39

6.3.2 Updating the Synaptic Weights for Convolutional Decoder…………………………………40

6.3.3 Training Pattern Generation ….……………………………………………………………41

6.3.4 Design Requirements for Convolutional Decoder ….……………………………………. …42

CHAPTER 7: RESULTS AND DISCUSSION ………………………………………………. 43

7.1 Introduction ……………………………………………………………………………..43

7.2 Simulated Codeword and its Input-Output Relationship ………………………………43

7.3 Simulated Artificial Neural Network and its Input-Output Relationship……………..44

7.4 Limitations ………………………………………………………………………...…… 48

7.5 Future Work and Recommendations ……………………………………………….....48

REFERENCES ………………………………………………………………………………50

Page Left Blank Intentionally

Chapter 1

Thesis Introduction

1.1 Introduction

With the advent of modern digital communication system, there have been challenging problems in hand. One of the problems that have been a constant feature of any communication system right from the onset of development of earlier inventions is error correction.

In this chapter error correcting codes including their types, brief history and the reason as to why we carry out error correction are presented. Artificial Neural Networks [1] (ANNs) have also been introduced highlighting their salient features. Finally goals and specifications and thesis layout has been discussed.

1.2 Error Correcting Codes

When binary information is passed from one point to another, there is always some chance that a mistake can be made, a 1 interpreted as a 0, or a 0 taken to be a 1. This can be caused by media defects, electronic noise, component failure, poor connections, deterioration due to age and other factors. When a bit is mistakenly interpreted, a bit error has occurred.

Error correction is the process of detecting bit errors and correcting them [2]. This correction can be done in software or hardware. For high data rates, error correction is done in special-purpose hardware.

The two main types of error correcting codes are convolutional codes [3] and block codes [4]. In each case, strings of bits are divided into blocks of data and each block of data is encoded into a longer block of data.

Convolutional codes tend to operate on smaller blocks of data than block codes and, unlike block codes, the encoding of one block of data depends on the state of the encoder as well as on the data to be encoded. Decoding of convolutional codes is usually done by executing some type of decoding algorithm in a processor.

1.3 Basic concept of error correcting Systems

Each bit can be thought of as one of two letters, 0 or 1. Error correcting systems add extra or redundant bits to computer words. The extra letters (bits) add a certain structure to each word. If that structure is altered by errors, the changes can be detected and corrected.

1.3.1 Need for error correction

Error correction is needed to ensure the accuracy and integrity of data and in some cases, to create fault-tolerance [5]. (where components can fail with no loss of performance or data).

No electronic data transmission or storage system is perfect. Each system makes errors at a certain rate. As data transfer rates and storage densities increase, the error rate also increases. If bit error rate is viewed in terms of media, some electronic systems experience more errors than others. For instance, optical disk has a higher error rate than magnetic disk. Magnetic tape has a higher error rate than magnetic disk. Fiber optic communications cable and semiconductor memory have a low error rate.

1.3.2 Error Correction and Fault-tolerance

If encoded data is written to a storage device in such a way that each byte of the encoded data is recorded on a separate component then it results into creation of fault-tolerance. If one or more components fail, they can only corrupt the data bytes that are written on them.

As long as number of component failures is less than or equal to the correction capability of the error correcting code, the failures will not result in any loss of data or performance.

1.4 Motivation

The basic objective of carrying out this research is the study of ANNs and the error correcting codes. The power of ANNs like self-organization, adaptive learning and the ability to predict as to what is going to happen next on the analogy of human brain has been used for decoding of error correcting codes. This ensures integrity, accuracy and fault tolerance in the transmitted data. Back propagation algorithm for ANNs was simulated using Matlab. It was confirmed during the course of research that if the training parameters are cautiously selected and the network is properly trained, the network can identify the actual code words and hence the desired messages.

1.5 Artificial neural networks

An ANN is a network of artificial neurons [3]. These artificial neurons are specialized computational elements performing simple computational functions. The manners in which these neurons are interconnected define the topology or architecture of the network. Whereas a classical digital computer is programmed, an ANN is trained. Adjusting the strengths of interconnections (weights) among the neurons constitutes training or learning. The concept of memory in a conventional computer corresponds to the concept of weight settings in ANNs.

The processing and storage functions in ANNs are not centralized and distinct, each neuron acts as a processor and the set of weights associated with that neuron act as distributed storage. In a typically ANN one can expect to find hundreds of processors and thousands of storage elements.

Methods based on neural networks have a distinctly different flavor from those based on artificial intelligence (AI) techniques [7]. In classical AI, a symbolic representation of the external world is the starting point and a digital computer is used as a symbol manipulating engine. The symbol string obtained as a solution is converted back into a physical representation for human cognition. ANNs, by virtue of their training, exhibit a more "plastic" behavior. For this reason, ANNs more appropriately belong to a class of methods that are being dubbed as "soft computing".

1.5.1 Goals and Specification

The primary objective is to carryout decoding of block codes using neural networks. The back propagation algorithm for neural networks has been simulated using Matlab. Linear block code (7, 4) has been decoded making use of the simulator. The actual message blocks along with the code words have been used initially to train the neural network simulator in such a way that it has the capability to detect and correct at the most one error per message block during the processing of training. In order to know the efficiency of the simulator it is exposed to the code words in which errors have been embedded and its response is viewed for knowing as to how much responsive neural network is when it has been trained properly.

1.6 Thesis Layout

Chapter 1 gives introduction to thesis. Basic concept of error correction, block codes, convolutional codes and ANNs have been introduced and the motivation to pursue the research has been discussed. In chapter 2 linear block codes with special emphasis on the fact as to how they are encoded and their decoding using standard array and syndrome decoding techniques have been discussed. In chapter 3 encoding of convolutional codes and their decoding using maximum likelihood decoding by utilizing Viterbi algorithm has been discussed in considerable details.

In Chapter 4 neural networks have been discussed starting from basic features and applications of ANNs their analogy to human brain, an engineering approach which includes a simple neuron and the basic neuron model, its architecture and back propagation algorithm. In chapter 5 decoding of block codes using neural networks has been discussed in detail. For this back propagation algorithm has been simulated by using Matlab. The training of neural network has been carried out by specifying various parameters of the network, selecting list of code words and actual massages. In order to test the simulator, codes having errors were given as an input and the response was monitored.

The network has the ability to at least detect and correct one error bit per message block. In chapter 6 an approach for decoding convolutional codes using neural networks has been included for further study. Finally, in chapter 7 an overview of the presented work is discussed, in the shape of results and discussions including achievements and limitations. Source code used for the decoding of linear block codes, the files used for (7, 4) linear block codes results achieved have been attached as appendices.

LINEAR BLOCK CODES

2.1 Introduction

Error-Control coding techniques detect and possibly correct errors that occur when messages are transmitted in a digital communication system. To accomplish this, the encoder transmits not only the information symbols but also extra redundant symbols. The decoder interprets what it receives, using the redundant symbols to detect and possibly correct errors occurred during transmission.

Block coding is a special case of error-control coding [5]. Block coding techniques maps a fixed number of message symbols to a fixed number of code symbols. A block coder treats each block of data independently and is a memory less device.

In this chapter linear block codes have been discussed. Their encoding and decoding techniques have been explained to have a good knowledge of the block codes before considering the decoding of block codes [6] using ANNs.

2.2 Linear Block codes-Encoding

The output of an information source is always a sequence of binary digits “0” or “1”. In block coding, this binary information sequence is segmented into message blocks of fixed length. Each message block, denoted by u, consists of k information digits. There are a total of 2^k distinct messages. The encoder, according to certain rules, transforms each input message u into a binary n-tuple v with n > k. This binary n-tuple v is referred to as the *code word* (or code vector) of message u. Therefore, corresponding to the 2^k possible messages, there are 2^k code words. This set of 2^k code words is called a block code. For a block code to be useful, the 2^k code words must be distinct. Therefore, there should be a one-to-one correspondence between a message u and its code word v [1].

For a block code with 2^k code words and length n, unless it has a certain special structure, the encoding apparatus would be prohibitively complex for large k and n since it has to store the 2^k code words of length n in a dictionary. A desirable structure for a block code to possess is the linearity. With this structure in a block code, the encoding complexity is greatly reduced [2].

A block code of length n and 2^k code words is called a linear (n, k) code if and only if its code words form a k-dimensional subspace of the vector space of all the n-tuples.

A binary block code is linear if and only if the mod-2 sum of two code words is also code word [7]. The (7, 4) linear block code can be express in the form of table as shown in table 2.1. It can be easily checked that sum of any two code words in this code is also a code word.

Since an (n, k) linear code C is a k-dimensional subspace of the vector. Space Vn of all the binary n-tuples, it is possible to find *k* linearly independent code words g_0 , g_1,….., g_{k-1} in C such that every code word v in C is a linear combination of these *k* code words. Mathematically it can be expressed by Equation (2.1)

Messages	Code Words
0 0 0 0	0 0 0 0 0 0 0
1 0 0 0	1 1 0 1 0 0 0
0 1 0 0	0 1 1 0 1 0 0
1 1 0 0	1 0 1 1 1 0 0
0 0 1 0	1 1 1 0 0 1 0
1 0 1 0	0 0 1 1 0 1 0
0 1 1 0	1 0 0 0 1 1 0
1 1 1 0	0 1 0 1 1 1 0
0 0 0 1	1 0 1 0 0 0 1
1 0 0 1	0 1 1 1 0 0 1
0 1 0 1	1 1 0 0 1 0 1
1 1 0 1	0 0 0 1 1 0 1
0 0 1 1	0 1 0 0 0 1 1
1 0 1 1	1 0 0 1 0 1 1
0 1 1 1	0 0 1 0 1 1 1
1 1 1 1	1 1 1 1 1 1 1

TABLE 2.1 LINEAR BLOCK CODE WITH k=4 and n=7

$$\mathbf{v}=u_o\mathbf{g}_o+ u_1\mathbf{g}_1 = \ldots+u_{k-1}\mathbf{g}_{k-1} \qquad (2.1)$$

Where u_i=0 or 1 for 0 < i < k. These *k* linearly independent code words can be arranged as the rows of (k × n) matrix **G**.

$$\mathbf{G}=\begin{bmatrix}\mathbf{g_o}\\ \mathbf{g_1}\\ \mathbf{g_2}\\ \vdots\\ \mathbf{g_{k-1}}\end{bmatrix}=\begin{bmatrix}g_{00} & g_{01} & g_{02} & \cdots & g_{0,n-1}\\ g_{10} & g_{11} & g_{12} & \cdots & g_{1,n-1}\\ \vdots & \vdots & \vdots & \cdots & \vdots\\ g_{k-1,0} & g_{k-1,1} & g_{k-1,2} & \cdots & g_{k-1,n-1}\end{bmatrix} \qquad (2.2)$$

Where g_i **=** **(**g_{i0}, g_{i1}, ….., $g_{i,n-1}$**)** for $0 \le i < k-1$. if **u=** (u_0, u_1, ….., u_{k-1}) is the message to be encoded. The corresponding code word can be expressed by equation (2.3).

v=uG

$$= (u_o\ ,\ u_1, \dots , u_{k-1})\begin{pmatrix}\mathbf{g_0}\\ \mathbf{g_1}\\ \vdots\\ \mathbf{g_{k-1}}\end{pmatrix}$$

$$= u_o\mathbf{g_0}+u_1\mathbf{g_1}+\dots u_{k-1}\mathbf{g_{k-1}} \qquad (2.3)$$

Clearly, the rows of **G** generate (or span) the (n, k) linear code C. For this reason, the matrix **G** is called a generator matrix for C.

A desirable property for a linear block code to possess is the systematic structure of the code words as shown in figure 2.1 where a code word is divided into two parts, the message part and the redundant checking part. The message part consists of *k* unaltered information (or message) digits and the redundant checking part consists of (n-k) parity-check digits, which are linear sums of the information digits. A linear block code with this structure is referred to as a linear systematic block code. The (7, 4) code given in Table 2.1 is a linear systematic block code; the rightmost four digits of each code word are identical to the corresponding information digits.

Redundant Checking part	Missing Part
| ⟵--------- n-k Digits ---------⟶ |	⟵----------- k Digits ----⟶ |

Figure 2.1 Systematic Format of a Code Word

A linear systematic (n,k) code is completely specified by a $k \times n$ matrix **G** shown in equation (2.4)

$$\mathbf{G}=\begin{bmatrix} g_o \\ g_1 \\ g_2 \\ \vdots \\ g_{k-1} \end{bmatrix} = \begin{bmatrix} p_{00} & p_{01} & \cdots & p_{0,n-k-1} & 1 & 0 & 0 & \ldots & 0 \\ p_{10} & p_{11} & \cdots & p_{1,n-k-1} & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} & 0 & 0 & 0 & \ldots & 1 \end{bmatrix} \quad (2.4)$$

Where **P** Matrix $=\begin{bmatrix} p_{00} & p_{01} & \cdots & p_{0,n-k-1} \\ p_{10} & p_{11} & \cdots & p_{1,n-k-1} \\ \vdots & \vdots & \vdots & \vdots \\ p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} \end{bmatrix}$

And K×K Identity Matrix$=\begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{bmatrix}$

Where Pij =0 or 1. Let $\mathbf{I}_k$ denote the k x k identity matrix. Now Matrix **G** can be written as

$\mathbf{G} = (\mathbf{P} \quad \mathbf{I}_k)$ If **u=** $(u_0, u_1, \ldots, u_{k-1})$ be the message to be encoded. The corresponding code word will be

$$\mathbf{v} = (v_0, v_1, v_2, \ldots\ldots v_{n-1})$$

$$= (u_0, u_1, \ldots\ldots u_{k-1}).\,\mathbf{G} \quad (2.5)$$

From equation (2.2) and (2.3) the components of **v** are

$$v_{n-k+i} = u_i \qquad \text{for } 0 \leq i < K \quad (2.6)$$

and

$$v_i = u_o p_{oj} + u_1 p_{1j} + \ldots\ldots + u_{k-1} p_{k-1,j} \quad (2.7)$$

For $0 \leq j < n - k$. Equation (2.6) shows that the rightmost *k* digits of a code word **v** are identical to the information digits $u_0, u_1, \ldots\ldots, u_{k-1}$ to be encoded and equation (2.7) shows that the leftmost (n-k) redundant digits are linear sums of the information digits. The (n-k) equations given by equation (2.7) are called parity-check equations of the code.

For any $k \times n$ Matrix **G** with k linearly independent rows, there exists an (n-k) × n matrix **H** with n-k linearly independent rows such that any vector in the row space of **G** is orthogonal to the rows of **H** and any vector that is orthogonal to the rows of **H** is in the row space of **G.** Hence, (n, k) linear code generated by **G** can be defined in an alternate way as, "an n-tuple **v** is a code word in the code generated by **G** if only if **v** $\boldsymbol{H^T} = \mathbf{0}$**"**. Matrix **H** is called parity-check matrix of the code. The 2^{n-k} linear combinations of the rows of matrix **H** form an (n, n-k) linear code $\mathbf{C}_d$. This code is null space of the (n, k) linear code C generated by matrix **G** (i.e, for any **v** ϵ **C** and any **w** ϵ $\mathbf{C}_d$**, v. w = 0**). $\mathbf{C}_d$ is called dual code of C. Therefore a parity-check matrix for a linear code **C** is a generator matrix for its dual code $\mathbf{C}_d$.

If generator matrix of an (n, k) linear code is in the systematic form of equation (2.2), the parity-check matrix will be

H= $(\boldsymbol{I_{n-k}} \;\; \boldsymbol{P^T})$

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & p_{00} & p_{10} & \cdots & p_{k-1,0} \\ 0 & 1 & 0 & \cdots & 0 & p_{10} & p_{11} & \cdots & p_{k-1,1} \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & p_{0,n-k-1} & p_{1,n-k-1} & \cdots & p_{k-1,n-k-1} \end{bmatrix} \tag{2.8}$$

Where $\boldsymbol{P^T}$ is the transpose of the matrix **P**. If $\mathbf{h}_j$ is the jth row **H**. It can be verified that the inner product of the jth row of given by equation (2.2) and the jth row of **H** given by equation (2.5) is

$$g_{i,hj} = P_{ij} + P_{ij} = 0 \tag{2.9}$$

For 0 ≤ i < k and 0 ≤ j < n-k. This implies that **G.** $\boldsymbol{H^T} = \mathbf{0}$. Also, the (n-k) rows of **H** are linearly independent. Therefore, the **H** matrix of equation (2.8) is a parity-check matrix of the (n, k) linear code generated by the matrix **G** of equation (2.4)

The parity-check equations given by equation (2.7) can also be obtained from the parity-check matrix **H** of equation (2.8). if **u** = (u_0, u_1, ….., u_0, u_0, …, u_{k-1}) are the message to be encoded. In systematic form the corresponding code word would be

$$\mathbf{v} = (v_0, v_1 \ldots\ldots v_{n-k-1}, u_0, u_1, \ldots\ldots u_{k-1}) \tag{2.10}$$

Using the fact that v. H^T = 0, we obtain

$$v_j + u_o p_{oj} + u_1 p_{1j} + u_{k-1} p_{k-1j} = 0 \text{ (For } 0 \leq j < n\text{-k)} \tag{2.11}$$

Rearranging the equations of equation (2.11), the same parity-check equations of equation (2.7) are obtained. Therefore, an (n, k) linear code is completely specified by its parity-check matrix.

Based on the equation (2.6), encoding circuit for an (n,k) linear systematic code can be implemented easily. In the encoding circuit in Figure 2.2. The encoding operation is very simple. The message **u = (** u_0, u_1, ….., u_{k-1}**)** to be encoded is shifted into the message register and simultaneously into the channel.

As soon as the entire message has entered the message register, the (n-k) parity-check digits are formed at the outputs of the (n-k) mod-2 adders. These parity-check digits are then serialized and shifted into the channel. Figure 2.3 shows the encoding circuit for the (7,4) code given in the Table 2.1.

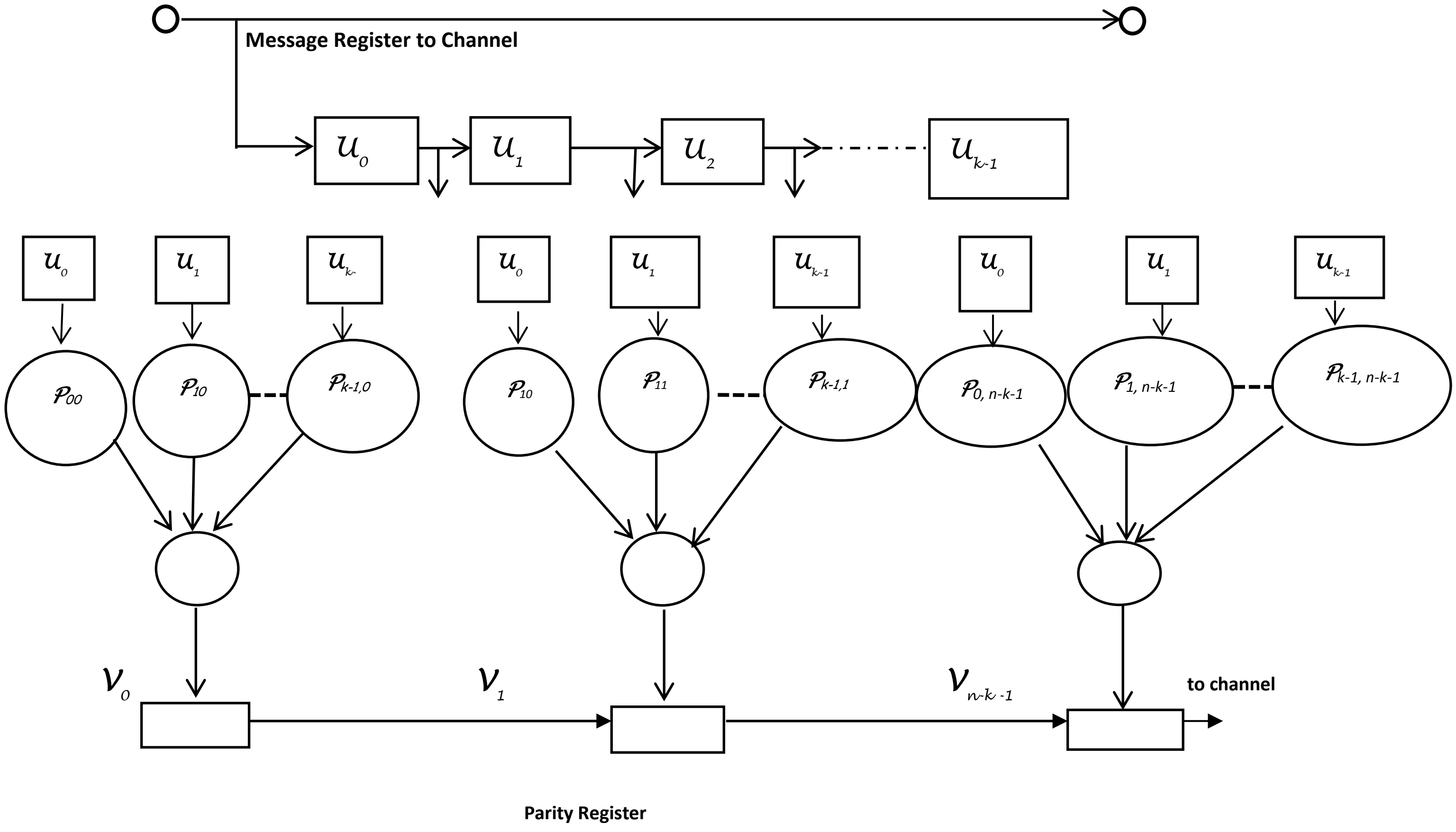


Figure 2.2 Encoding circuit for a Linear system (n,k) code.

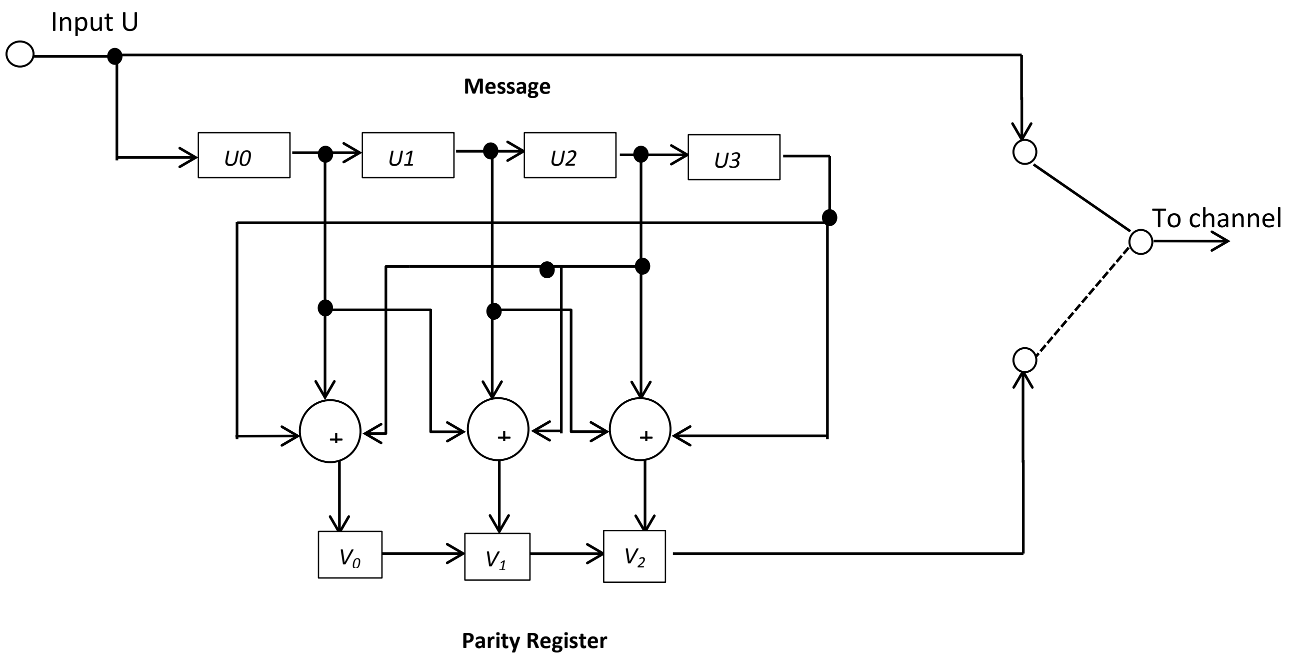


Figure 2.3 Encoding Circuit for (7, 4) Systematic Code

2.3 Standard Arrays

Linear block codes have some properties that simplify the process of encoding [3]. For the **BSC,** lower weight, the sum of the transmitted vectors are more probable that higher weight ones, and maximum likelihood decoding corresponds to minimum distance (or minimum error weight) decoding. By partitioning all the possible received vectors a look-up table is made for minimum distance decoding. This look-up table is called the standard array.

If C is an (n, k) linear block code over GF (q) then C is a subspace of V= GF (q), the vector space of all n-tuples over GF (q). As such it is also a subgroup of <V, +> (under vector addition). A standard array for the code C is generated by removing from **V** the code words in **C** and writing the code words, starting with 0 as the first row of the table. Selecting from the remaining elements of **V** a vector **e** that has minimum weight and writing in the next row the closet **e** + **C,** in the order dictated by the first row and removing the elements of this closet from **V** and repeating the procedure until **V** becomes empty [3].

The received vector ***r*** is decoded by looking it up in the table. The code word **C** appearing at the top of the column in which ***r*** appears is the maximum likelihood code word. The vector ***e*** in the first column of the row in which ***r*** appears is the most likely error pattern. The construction of the array ensures that

e = ***r***– ***c*** has minimum weight.

Since there are 2^{n-k} rows in the standard array, a(n, k) linear block code can correct 2^{n-k} different error patterns.

2.4 Syndrome decoding

The syndrome vector **s** of a received ***r*** is the vector

$$\underline{\mathbf{S}} = \underline{\mathbf{rH}^T} \qquad (2.12)$$

For ***r*** ϵ ***C***, the syndrome is all-zero vector. Syndromes are used to simplify decoding using the standard array.

If **e** is the closet leader of the row **e**+ C, and if ***r*** =***C***+ **e** is some other element of **e**+ **C**. then

$$\begin{aligned} \underline{\mathbf{rH}^T} &= (\underline{\mathbf{c}} + \underline{\mathbf{e}})\,\mathbf{H}^T \\ &= \underline{\mathbf{cH}^T} + \underline{\mathbf{eH}^T} \\ &= \underline{\mathbf{eH}^T} \end{aligned} \qquad (2.13)$$

Hence all element of the row have the same syndrome, which depends only upon the closet leader (or error pattern). Therefore syndromes for single weight error patterns are just the columns of **H**.

2.4.1 Syndrome Table

The storage requirements for decoding may be reduced if instead of storing the whole standard array, a table is made of each closet leader and its associated syndrome. Decoding is carried out by computing the syndrome of the received vector, looking up the associated error pattern (cost leader) in the table, and subtracting this from the received vector.

Using the algebraic structure of linear codes, decoding method can be designed which requires only (n + (n -k)) 2^{n-k} bits of storage (for binary codes, there are 2^{n-k} correctable error patterns of length n, each with a unique n-k bit syndrome) and one vector-matrix multiply, as compared to n^{2n} bits of storage (an n bit code word for each of the 2^n possible received vectors) for the standard array.

For (6, 3) systematic linear code generated by

$$\boldsymbol{G} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{2.14}$$

A standard array for this code is shown in Table 2.2. The code words appear in the first row and the correctable error patterns as the first entry row.

Table 2.2 Standard Array for the (6, 3) Code

000000	101001	011010	110011	110100	011101	101110	000111
000001	101000	011011	110010	110101	011100	101111	000110
000010	101011	011000	110001	110110	011111	101100	000101
000100	101101	011110	110111	110000	011001	101010	000011
001000	100001	010010	111011	111100	010101	100110	001111
010000	111001	001010	100011	100100	001101	111110	010111
100000	001001	111010	010011	010110	111101	001110	100111
001100	100101	010110	111111	111000	010001	100010	001011

By using the array of table 2.2 all single bit errors in the received code words can be corrected.

2.5 Weight distribution and error Probabilities

For transmission over a memory less BSC with error probability P, the number of errors in a block of length n are binomially distributed and hence the probability of a decoding error for a t-error correcting (n, k) block code is bounded by [4].

$$\text{P (error)} \leq \sum_{i=t+1}^{n} \begin{bmatrix} n \\ i \end{bmatrix} P^{i} (1-P)^{n-i} \tag{2.15}$$

If the code is linear, and standard array (maximum likelihood) decoding is used, an error occurs if and only if the error pattern is not a closet leader. For a BSC, the probability of a particular error pattern of weight w is given later.

If there are α^i leaders of weight i then the error probability for standard array decoding is given by

$$P(\text{SA Decoding Failure}) = 1 - \sum_{i=0}^{n} \alpha^i P^i (1-P)^{n-i} \qquad (2.16)$$

Standard array decoding is maximum likelihood, and hence any other decoding method is lower bounded by this value.

For a linear block code, the undetectable error patterns are the set of non-zero code words. If there are A^i code words of weight i, then the probability of an undetectable error pattern over a BSC with crossover probability p is given by

$$P(\text{Undetected error}) = 1 - \sum_{i=l}^{n} A^i p^i (1-P)^{n-i} \qquad (2.17)$$

CONVOLUTIONAL CODES

3.1 Introduction

Convolutional codes were first introduced by Elias [5] in 1955 as an alternative to block codes. Shortly thereafter, Wozencraft [6] proposed sequential decoding as an efficient decoding scheme for convolutional codes. In 1963, Massey [7] proposed a less efficient but simpler-to-implement decoding method called threshold decoding. In 1967, Viterbi [8] proposed a maximum likelihood decoding scheme that was relatively easy to implement for codes with small memory orders.

Convolutional coding is a special case of error-control coding. Unlike a block coder, a convolutional coder is not a memory less device. Even though a convolutional coder accepts a fixed number message symbols and produces a fixed number code symbols, its computations depend not only on the current set of input symbols but on some of the previous input symbols.

In this chapter, structure of the convolutional codes has been discussed and as to how a incoming sequence is coded. The method of encoding the convolutional codes by making use of the state diagram, tree and trellis diagrams has been explained in detail. At the end Viterbi decoding has been used in order to decode the given sequence in steps at various time intervals.

3.2 Coding and Decoding With Convolutional Codes

Convolutional codes are commonly specified by three parameters, (n, k, and m).

The quantity k/n called the code rate is a measure of the efficiency of the code. Commonly k and n parameters range from 1 to 8, m from 2 to 10 and the code rate from 1/8 to 7/8.

Often the manufacturers of convolutional code chips specify the code by parameters (n, k and L), the quantity L is called the constraint length of the code and is defined by

$$L = k\,(m-1)$$

3.2.1 Code Parameters and Structure of Convolutional Code

The convolutional code structure is easy to draw from its parameter. First m boxes are drawn which represent the m memory registers. Then n mod II adders are draw to represent the n output bits. Memory registers are connected to the adder using the generator polynomial as shown in the Figure 3.1

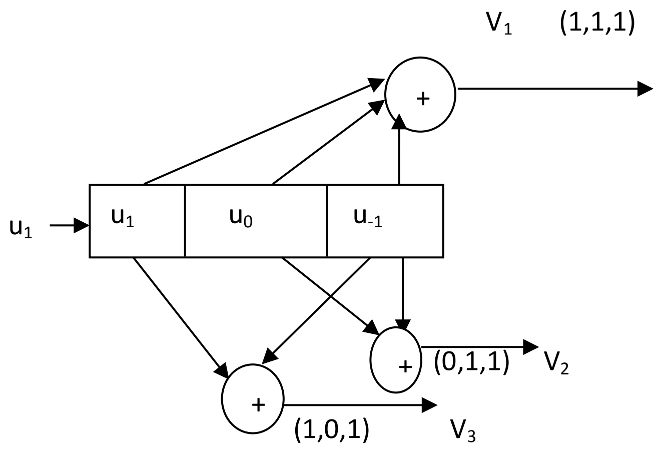


Figure 3.1 Convolutional Code (3,1,3)

Where u_1 is the input. This is a rate 1/3 code. Each input bit is coded into 3 output bits. The constraint length of the code is 2. The 3 output bits are produced by the 3 mod-2 code. Each input bit is coded into 3 output bits. The constraint length of the code is 2. The 3 output bits are produced by the 3 and mod-2 adders by adding up certain bits in the memory registers. The selection of which bits are to be added to produce the output bit called the generator polynomial (G) for that output bit. In this case, the first output bit has a generator polynomial of (1, 1, 1). The second output bit has a generator polynomial of (0,1,1) and the third output bit has a polynomial of (1,0,1). The output bits just the sum of these bits.

$V1= \mathrm{mod2}\ (u_1 + u_0 + u_{-1})$

$V2= \mathrm{mod2}\ (u_1 + u_{-1})$

$V3= \mathrm{mod2}\ (u_1 + u_{-1})$

The polynomials give the code its unique error protection quality. One (3,1,4) code can have completely different properties from an another one depending on the polynomials chosen.

3.3 Coding and Incoming Sequence

The output sequence V, can be computed by convolving the input sequence u with the impulse response g.

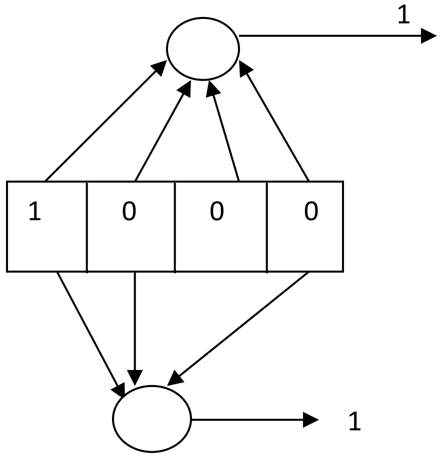
$$V= u*g \tag{3.1}$$

Where * is the convolution

Or in a more generic form

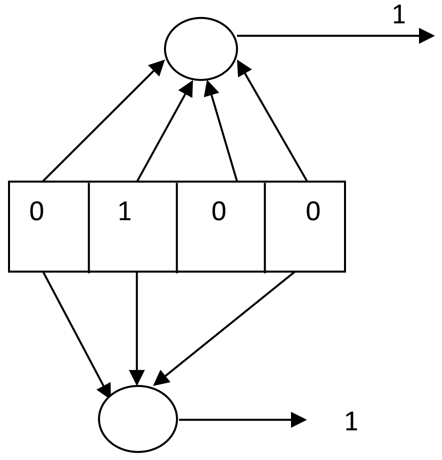
$$v_l^j = \sum_{i=0}^{m} u_{l-i} - g_l^j \tag{3.2}$$

Where v_l^j the output is bit l from the encoder j, and u_{l-i} is the input bit, and g_l^j is the ith term in the polynomial. The process of encoding a solo 1 bit once it is passed through the (2, 1, 4) convolutional encoder showing its output bits at various time intervals will be as shown in figure 3.2



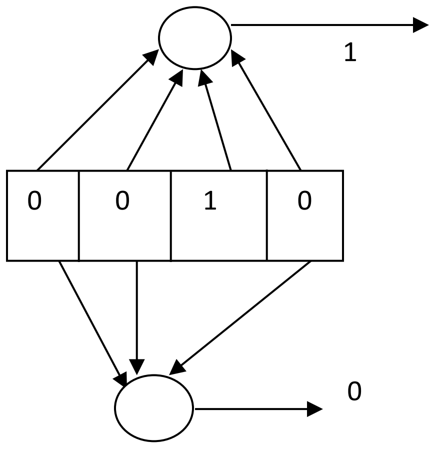
a) t=0, input state=000

Input bit=1, output bits=11



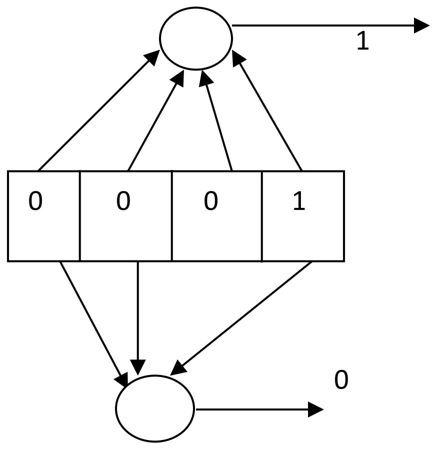
b) t=1, Input state 100

Input bit=0, output bits=11



a) t=2, input state=010

Input bit=1, output bits=10



b) t=3, Input state 001

Input bit=1, output bits=01

Figure 3.2 Sequence of a solo 1 bit passed through the encoder [7].

At t=0, the initial state of the encoder is all zeros. Input bit 1 causes two bits 11 to appear at the output obtained by mod-2 sum of all bits in the registers for the first bit and mod-2 sum of three bits for second output bit per the polynomial coefficients. At t=1, the input bit 1 moves forward one register. The input register is now empty and filled with a flush bit of 0. The encoder is now in state 100. The output bits are now again 11 by the same method.

At t=2 input bit 1 move forward again. Now the encoder state is 010 and another flush bit is moved into the input register. The output bits are now 10.

At t=3, the input bit moves to the last register and the input state is 001. The output bits are now 11. At t=4, the input bit 1 has passed completely through the encoder and the encoder has been flushed to an all zero state, ready for the next sequence.

It can be seen that a single bit has produced an 8-bit output although nominally the code rate is ½. This shows that for small sequence the overhead is much higher than the nominal rate, which only applies to long sequences. If a 0 bit passed through the encoder, an 8 bit all zero sequence will be obtained. The output produced is called the impulse response of the encoder. The 1 bit has a response of 11 11 10 11. The 0-bit similarly has an impulse response of 00 00 00 00.

Convolving the input sequence with the code polynomials produced these two output sequences that are why these codes are called convolutional codes.

3.3.1 The Encoder Design

The encoder for convolutional code uses a table look up to do the encoding. The look up table consists of input bit, Vstate of the encoder, which is one of the 8 possible stated in the case of (2, 1, 4) code and the output bits. For the code (2, 1, 4), since two bits appear at the output, the choices are 00, 01, 10, 11 and the output state which will be the input state for the next bit. Table 3.1 is the look up table for the code (2, 1, 4), which uniquely describes the code. The look up table is different for each code depending on the parameters and the polynomials used.

Table 3.1 Look-up Table for the Encoding of (2, 1, 4) Convolutional Code

Input Bits	**Input State**				**Output Bits**		**Output State**		
I1	S1	S2	S3		O1	O2	S1	S2	S3
0	0	0	0		0	0	0	0	0
1	0	0	0		1	1	1	0	0
0	0	0	1		1	1	0	0	0
1	0	0	1		0	0	1	0	0
0	0	1	0		1	0	0	0	1
1	0	1	0		0	1	1	0	1
0	0	1	1		0	1	0	0	1
1	0	1	1		1	0	1	0	1
0	1	0	0		1	1	0	1	0
1	1	0	0		0	0	1	1	0
0	1	0	1		0	0	0	1	0
1	1	0	1		1	1	1	1	0
0	1	1	0		0	1	0	1	1
1	1	1	0		1	0	1	1	1
0	1	1	1		1	0	0	1	1
1	1	1	1		0	1	1	1	1

Graphically, there are three ways in which we can look at the encoder to gain better understanding of its operation. These are state diagram, tree diagram and trellis diagram.

3.3.2 State Diagram

The state diagram for the (2, 1, 4) code is shown in figure 3.3

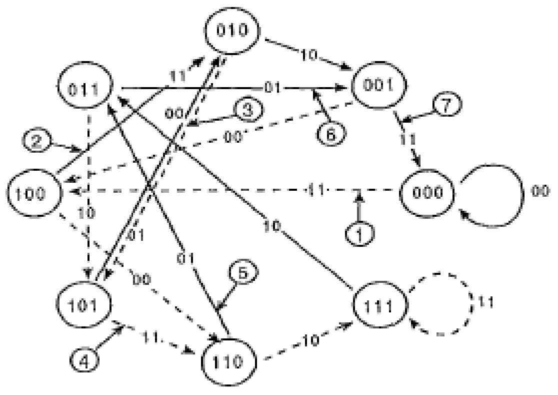


Figure 3.3 state diagram for the (2, 1, 4) code [8].

Each circle in the diagram represents a state. At any one time, the encoder resides in one of these states. The lines to and from it show state transitions that are possible as bits arrive. Only two events can happen at each time, arrival of a 1 bit or arrival of a 0 bit. Each of these two events allows the encoder to jump into a different state. The stage diagram does not have time as a dimension and hence it tends to be not intuitive.

The state diagram contains the same information that is in the table look-up but it is a graphic representation. The solid line indicates the arrival of a 0 and the dashed lines indicate the arrival of a 1. The output bits for each case shown on the line and the arrow indicates the state transition. Some encoder states allow outputs of 11 and 00 and some allow 01 and 10. No state allows all four options.

The encoding of a given sequence 1011 has been depicted in the figure 3.3. At state 000. The arrival of a 1 bit outputs 11 and the output state now becomes 100. The arrival of the next 0 bit outputs 11 and the output state now will be 010. The arrival of the next 1 bit outputs 01 and the output state as 101. The last bit 1 gives output state 110 and outputs 11. So up till now the output bits are 1111 01 11. But this is not the end. The encoder has to be taken to all zero state.

From state 110, by giving input as 0 bit, output state changes to 011 giving 01 as output. From state 011, input bit 0 yields state 001 outputting 01. Last input bit 0 gives state 00 with a final output of 11. The final answer is: **11 11 01 11 01 01 11.**

3.3.3 Tree Diagram

Figure 3.4 shows the tree diagram for the code (2, 1, 4). The tree diagram attempts to show the passage of time as we go deeper into the tree branches. It is somewhat better than a state diagram still not the preferred approach for representing convolutional codes.

Instead of jumping from one state to another, the branches of the tree are followed depending on whether a 1 or 0 is received.

The first branch in figure 3.4 indicates the arrival of a 0 or a 1 bit. The starting state is assumed to be 000. If a 0 is received, branch goes up and if a 1 is received, it goes downward. In figure 3.4, the solid lines show the arrival of a 0 bit and the shaded lines the arrival of a 1 bit. The first two bits show the output bits and the number inside the parenthesis is the output state.

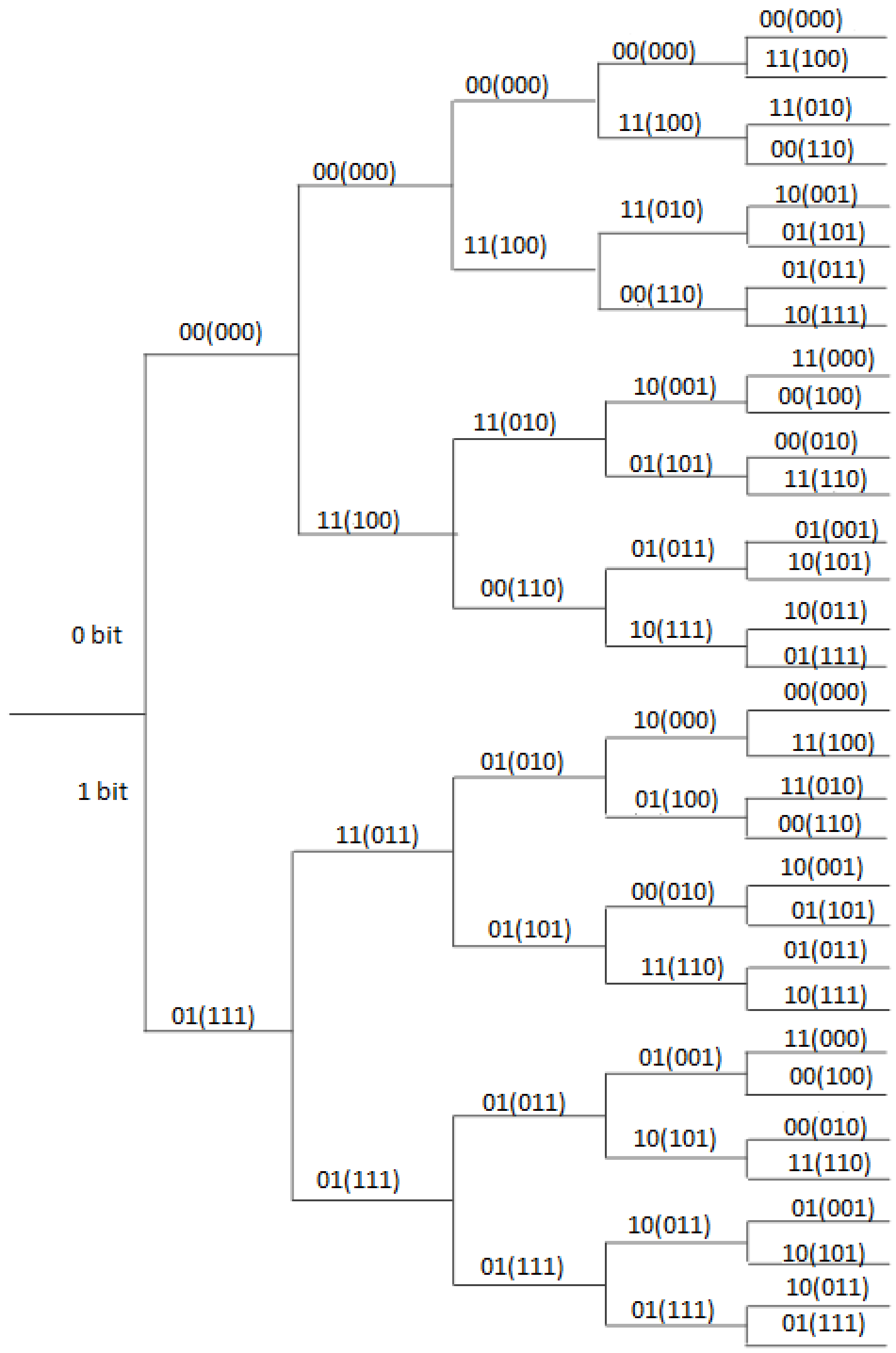


Figure 3.4 Tree Diagram of (2, 1, 4) Code

For the same sequence 1011 as before. At branch 1, going down gives output 11 and the state changes to 100. Now with 0 bit, going up, the output bits become 11 and the state is now 011. With next incoming bit 1 the output changes to 01 and now the output state is 101.

The next incoming bit 1 outputs bits 11. From this point, in response to a 0 bit input, an output of 01 is obtained and an output state of 011. The encoder needs to be flushed by 0 bits so the sequence actually becomes 1011000, with the last 3 bits being the flush bits. The output of the complete of sequence becomes 11 11 01 11 01 01 11. The answer remains the same which was obtained while decoding the sequence using state diagram.

3.3.4 Trellis Diagram

Trellis diagrams are messy but generally preferred over both the tree and the state diagrams because they represent linear time sequencing of events. Trellis diagram for (2, 1, 4) convolutional code is shown in figure 3.5

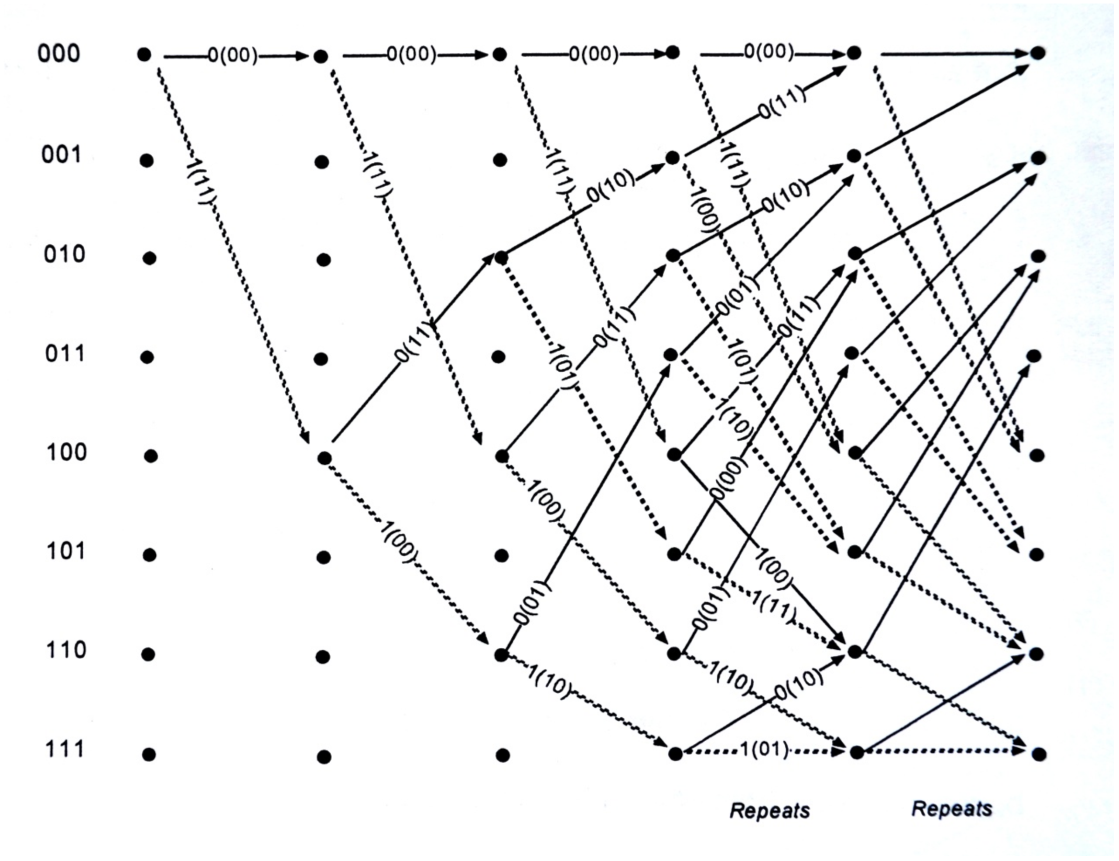


Figure 3.5 Trellis Diagram of (2, 1, 4) code [6]

The x-axis is discrete time and all possible states are shown on the y-axis. With the passage of time the trellis is traversed horizontally. Each transition means new bits have arrived.

The trellis diagram is drawn by lining up all the possible states (2^L) in the vertical axis. Each state is connected to the next state by the allowable code words for that state. There are only two choices possible at each other state, which are determined by the arrival of either a 0 or a 1 bit. The arrows show the input bit and the output bits are shown in parentheses. The arrows going upwards represent a 0 bit and going downwards represent a 1 bit. The trellis diagram is unique to each code, same as both the state and tree diagrams are with starting point as state 000, the trellis expands and in L bits becomes fully populated such that all transitions are possible. The transitions then repeat from this point on.

3.4 Decoding using Viterbi algorithm

Viterbi decoding is the best-known implementation of the maximum likely hood decoding [9]. Here the options are narrowed systematically at each time tick. The principal used to reduce the choices is that the errors occur infrequently and the probability of error is small. Also the probability of two errors in a row is much smaller than a single error that is the errors are distributed randomly.

The Viterbi decoder examines an entire received sequence of a given length. The decoder computes a metric for each path and makes a decision based on this metric. All paths are followed until two paths converge on one node. Then the path with the higher metric is kept and the one with lower metric is discarded. The paths selected are called the survivors.

For an N bit sequence, total numbers of possible received sequences are 2^N. Of these only 2^{kL} are valid paths. Where L is the number of states and K is the sequence. The Viterbi algorithm applies the maximum likelihood principles to limit the comparison to 2 to the power of kL surviving paths of instead of checking all paths.

The most common metric used is the Hamming distance metric[8] shown in Table 3.2 this is just dot product between the received code word and the allowable code word.

If the received sequence is **01 11 01 11 01 01 11,** it can be decoded using Viterbi algorithm and finally the trellis looks like as shown in Figure 3.6

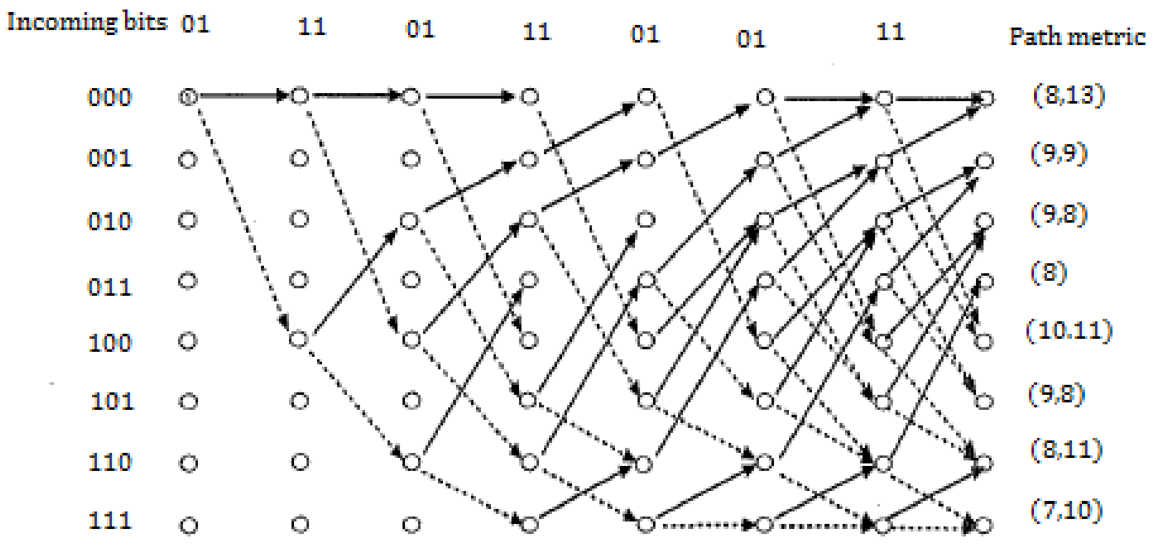


Figure 3.6 Final Trellis Diagram [7]

The paths with the highest metric are the winner paths. The path traced by states 000, 100, 010, 110, 011, 001, 000 corresponding to bits 1011000 is the decoded sequence.

The length of the trellis in this case is 4 bits +m bits. Ideally this should be equal to the length of the message, but by a truncation process, storage requirements can be reduced and decoding need not be delayed until the end of the transmitted sequence.

Chapter 4

ARTIFICIAL NEURAL NETWORKS

4.1 Introduction

This chapter has been included to give the reader an insight of as to what are Artificial Neural Networks ANNs. The feature and applications of Artificial Neural Netwroks ANNs have been highlighted. Similarities amongst human brain and Artificial Neural Netwroks ANNs have been investigated. In an engineering approach a simple neuron and McCulloch –Pits model of artificial neurons has been discussed to have an idea as to how an ANN looks like [9].

4.2 Basic Features of ANNs

An artificial neural network is an abstract simulation of real nervous system. ANNs are computing systems made up of a number of highly interconnected information processing units i.e. neurons [10] [11].

Neural networks are trained and not programmed. Neutrons of ANNs are so highly interconnected that a state of neuron affects the potential of the large number of other neurons to which it is connected according to the weights. In ANNs connection weights are adaptive and processing units have not linear activation functions.

The network may use imprecise and unreliable elements but because of highly redundant distributed structure, it is highly robust to noisy input and neuron failure. An ANN can create its own organization or representation of the information it receives during learning time.

4.3 Learning Process of a Human Brain

Much is still unknown about how the brain trains itself to process information. In the human brain, a typical neuron collects signals from others through a host of fine structures called dendrites [8].

The neuron sends out spikes of electricity activity through a long, thin strand known as an axon, which splits into thousands of branches. At the end of each branch, a structure called a synapse [10], converts the activity from the axon into electrical effects that inhibit or excite activity in the connected neurons.

When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes.

4.4 Basic Neuron Model

The McCulloch-Pits model [8] of artificial neuron is a formal neuron model and it is a multi-input non-linear model with interconnection weights W_{ij} which are also called synaptic weights or strengths. A non linear limiting or threshold function Ψ (u_j) is represented as cell body. The simplest model of an artificial neuron sums the weighted inputs and passes the result through a nonlinear threshold function as shown in the equation.

$$y_j = \boldsymbol{\Psi}\,[\textstyle\sum_{j=1}^{n} w_{ij} + \Theta_j] \qquad (4.1)$$

Where

Ψ is a limiting or threshold function called an activation function.

Θ_j is the external threshold called an offset or bias.

w_{ij} are synaptic weights or strengths.

x_i are the inputs (i= 1,2,….., n)

n is the number of inputs.

y_j represents the output.

Sometimes θ_j is taken as weight w_{j0} therefore additional input will be equal to +1. So eqn (4.1) becomes

$$y_j = \boldsymbol{\Psi}[\,\textstyle\sum_{i=1}^{n} w_{ij}\, x_i] \qquad (4.2)$$

A weighted sum of all inputs x_j is compared with threshold value θ_j if sum is greater θ_j neuron's output is equal to logic 1 otherwise logic will be 0.

The simple model is shown in the Figure

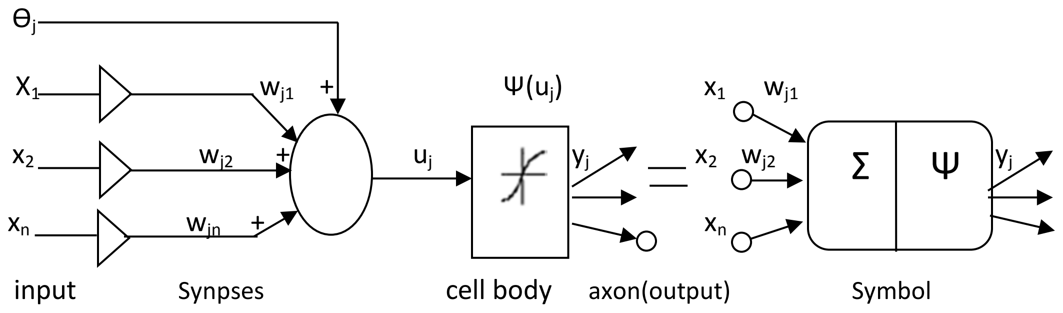


Figure 4.1 Basic Neuron cell and its symbols

The activation level or the state of neuron is measured by the output signal. For example, yj=1 if the neuron is active and y_j=0 if the neuron is inactive. In the basic neuron model the output signal is usually determined by a monotonically increasing sigmoid function of a weighted sum of the input signals. Such a sigmoid function can mathematically be described as [8]

$$y_{j=} \tanh(\Upsilon u) = \frac{1-e^{-2\Upsilon u_j}}{1+e^{-2\Upsilon u_j}} \qquad (4.3)$$

For a symmetrical bipolar representation or [9]

$$y_j = \frac{1}{1+e^{-2\Upsilon u_j}} \qquad (4.4)$$

Where γ is a positive constant variable which controls the “steepness”

Slope of the sigmoid function. The basic artificial neuron is characterized by its nonlinearity and the threshold. McCulloch pits model of the neuron uses only the binary (hard limiting) function.

4.5 Backprpogation algorithm for multilayer Perceptrons

The approach of the adaptive learning (updating) of synaptic weights can be extended to multiplayer perceptron. For simplicity a multilayer perceptron of three layers is considered. First hidden layer consists n0 inputs and n1 units, the second layer with n2 units and the output layer with n3 units.

The behavior of the network is determined on the basis of inputs and outputs pairs. The input output pairs are expressed as stable states of neurons which are usually represented by +1 (ON) and -1 (OFF). The learning of multiplayer perceptrons for a specific task is equivalent to finding the values of all synaptic weights such that the desired output is generated for a given input.

Therefore learning of a multiplayer perceptrons consists in adjusting all weights such that error measure between the desired output signal d_{jp} and the actual output signal y_{jp} averaged overall learning examples p will be minimum (possibly zero). The standard back propagation uses the steepest decent gradient approach [9] to minimize the mean square error function. Such a local error function for any pth learning example can be written as

$$E_p = \frac{1}{2}\sum_{j=1}^{n}(d_{jp} - y_{jp})^2 = \frac{1}{2}\sum_{j=1}^{n} e_{jp}^2 \quad (4.5)$$

Now the global error function may be written as

$$E_p = \sum_p E_p = \frac{1}{2}\sum_p^n \sum_j (d_{jp} - y_{jp})^2 \quad (4.6)$$

Where d_{jp} and y_{jp} are desired and the actual output signals of the jth output neuron for the pth pattern.

Architecture of the standard back propagation algorithm of a three layer perceptron is shown in the figure 4.4 [9].

There are two basic approaches to find the minimum of global error function E. The first technique [9] is the online or as per example learning in which the training patterns are presented sequentially usually in random order. The second technique [10] is batch learning in which the total error function E is minimized in such a way that the weight changes are accumulated using over all learning examples before the weights are actually changed.

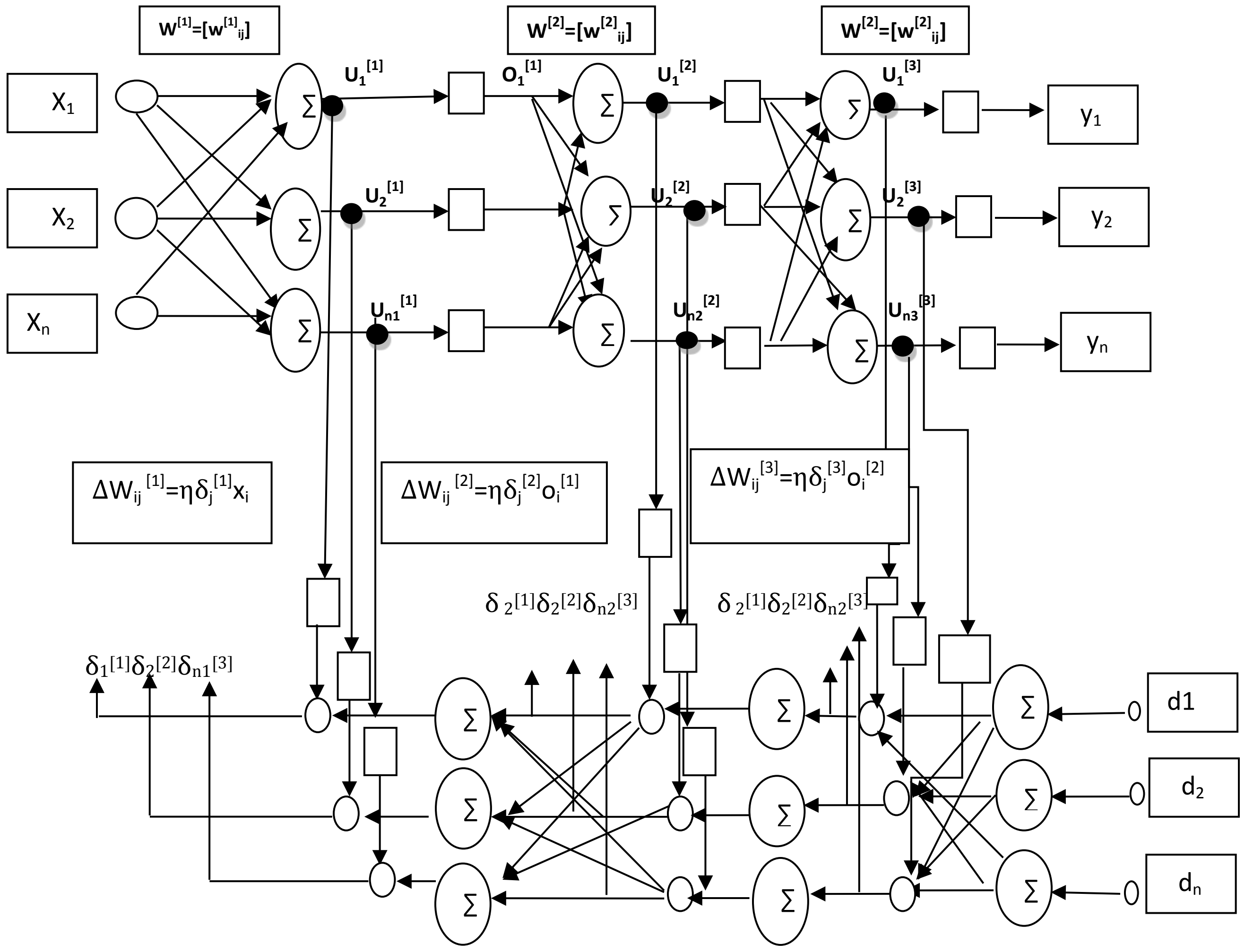

$W^{[1]}=[w^{[1]}_{ij}]$
$W^{[2]}=[w^{[2]}_{ij}]$
$W^{[2]}=[w^{[2]}_{ij}]$
X_1
X_2
X_n
$U_1^{[1]}$
$U_2^{[1]}$
$U_{n1}^{[1]}$
$O_1^{[1]}$
$U_1^{[2]}$
$U_2^{[2]}$
$U_{n2}^{[2]}$
$U_1^{[3]}$
$U_2^{[3]}$
$U_{n3}^{[3]}$
y_1
y_2
y_n
$\Delta W_{ij}^{[1]}=\eta\delta_j^{[1]}x_i$
$\Delta W_{ij}^{[2]}=\eta\delta_j^{[2]}o_i^{[1]}$
$\Delta W_{ij}^{[3]}=\eta\delta_j^{[3]}o_i^{[2]}$
$\delta_1^{[1]}\delta_2^{[2]}\delta_{n1}^{[3]}$
$\delta_2^{[1]}\delta_2^{[2]}\delta_{n2}^{[3]}$
$\delta_2^{[1]}\delta_2^{[2]}\delta_{n2}^{[3]}$
d1
d_2
d_n

Figure 4.2 Network Architecture for the standard Back propagation Algorithm of a three layer perceptron

In the online learning approach the gradient search in the synaptic weight space is carried out on the basis of a local error function Ep. First determine an updating formula for the synaptic weights Wsji (s=3) of the output layer. Using the chain rule

$$\Delta \mathrm{w}_{ji}^{[3]} = -\eta \frac{\partial E_p}{\partial \mathrm{w}_{ji}^{[3]}} = -\eta \frac{\partial E_p}{\partial \mathrm{u}_{ji}^{[3]}} \frac{\partial \mathrm{u}_{ji}^{[3]}}{\partial \mathrm{w}_{ji}^{[3]}} \tag{4.7}$$

Here

$$u_j^{[3]} = \sum_{i=1}^{n3} w_{ji}^{[3]} x_i^3 = \sum_{i=1}^{n3} e_{ji}^{[3]} o_i^2 \tag{4.8}$$

And the local error called delta is defined as

$$\delta_j^{[3]} = \frac{\partial E_p}{\partial \mathrm{w}_j^{[3]}} = \frac{\partial E_p}{\partial e_{jp}} \frac{\partial e_{jp}}{\partial u_j^{[3]}} = - \quad e_{jp} \frac{\partial \Psi_j^{[3]}}{\partial u_j^{[3]}} \tag{4.9}$$

A general formula for updating the weights in the output layer will be

$$\Delta \mathrm{w}_{ji}^{[3]} = -\eta \delta_j^{[3]} \mathrm{X}_i^{[3]} = -\eta \delta_j^{[3]} \mathrm{o}_i^{[2]} \tag{4.10}$$

Where

$$\delta_j^{[3]} = \; e_{jp} ({\Psi_j^{[3]}}^{i})(d_{ji} - y_{jp}) \frac{\partial \Psi_j^{[3]}}{\partial u_j^{[3]}} \tag{4.11}$$

Updating the synaptic weights in the hidden layers is a little more complicated.

For the second hidden layer it can be shown as

$$\Delta \mathrm{w}_{ji}^{[2]} = -\eta \frac{\partial E_p}{\partial \mathrm{w}_{ji}^{[2]}} = -\eta \frac{\partial E_p}{\partial \mathrm{u}_j^{[2]}} \frac{\partial \mathrm{u}_j^{[2]}}{\partial \mathrm{w}_{ji}^{[2]}} = \eta \delta_j^{[2]} \mathrm{x}_i^{[2]} \tag{4.12}$$

Where the local error for the second hidden layer is defined as

$$\delta_{ji}^{[2]} = - \frac{\partial E_p}{\partial \mathrm{w}_j^{[2]}} \qquad (j= 1, 2, 3 \ldots n2) \qquad (4.13)$$

However this local error cannot be directly evaluated as is done for the local errors in the output layer. Using the chain rule we can write

$$\delta_{ji}^{[2]} = - \frac{\partial E_p}{\partial \mathrm{u}_j^{[2]}} = - \frac{\partial E_p}{\partial \mathrm{o}_j^{[2]}} \frac{\partial \mathrm{o}_j^{[2]}}{\partial \mathrm{u}_j^{[2]}} \qquad (4.14)$$

δ_j can be expressed as

$$\delta_j^{[2]} = \Psi_j^{[2]} \mathrm{u}_j^{[2]} \qquad (4.15)$$

Therefore

$$\delta_j^{[2]} = - \frac{\partial E_p}{\partial \mathrm{o}_j^{[2]}} = - \frac{\partial \Psi^{[2]}}{\partial \mathrm{u}_j^{[2]}} \qquad (4.16)$$

The factor can be evaluated

$$\frac{\partial E_p}{\partial \mathrm{o}_j^{[2]}} = - \sum_{i=1}^{n_3} \frac{\partial E_p}{\partial \mathrm{u}_i^{[3]}} \frac{\partial \mathrm{u}_i^{[3]} j}{\partial \mathrm{o}_j^{[2]}} = \sum_{i=1}^{n_3} \frac{\partial E_p}{\partial \mathrm{u}_i^{[3]}} \frac{\partial}{\partial \mathrm{o}_j^{[2]}} [\sum_{i=1}^{n3} w_{ik}^{[3]} x_k^{[3]}]$$

$$= \sum_{i=1}^{n_3} \delta_j^{[3]} \ \delta_j^{[3]} \frac{\partial}{\partial \mathrm{o}_j^{[2]}} \sum_{i=1}^{n3} w_{ik}^{[3]} x_k^{[3]} \qquad (4.17)$$

The local error in the second layer can be evaluated as

$$\delta_i^{[2]} = \frac{\partial \Psi^{[2]}}{\partial \mathrm{u}_j^{[2]}} \left[\sum_{i=1}^{n_3} \delta_j^{[3]} w_{ij}^{[3]} \right] \qquad (4.18)$$

Analogously

$$\Delta w_{ji}^{[1]} = \eta\delta_j^{[1]} x_i^{[1]} = \eta\delta_j^{[1]} o_i^{[0]} \quad (4.19)$$

Generally the local error of the hidden layers is determined on the basis of the local errors at an upper layer. Starting with the highest output layers δ_j is computed, the errors δ_j can be propagated backward to the lower layers. Figure 4.8 shows the functional schemes of the back propagation algorithm. The major difference of the learning rules for the output layer is the evaluation of the local errors δ_j (s = 1,2, 3). In the output layer the error is the function of the desired and the actual output and the derivative of the sigmoid activation function. For the hidden layers the local errors are evaluated on the basis of the local errors in the upper layers.

The basic back propagation algorithm can be performed by initialize all synaptic weights W_{ij} to small random values. Inputs in forms of learning examples are presented and the actual outputs are calculated for all neurons using the present value W_{ij} and the patterns. The desired output are specified and local errors δ_j for all layers are calculated. The synaptic weights are adjusted according to the synaptic formula.

$$\Delta w_{ji}^{[1]} = \eta\delta_j^{[x]} x_i^{[x]} \quad (4.20)$$

Another input pattern is presented corresponding to the next learning example and the complete process is repeated. All the learning examples are presented cyclically until the weights are stabilized i.e until the error of the entire set is acceptably low and the network converges. After training the multilayer perceptron usually has the feature of generalization i.e it has the ability for proper response to input patterns not presented during the learning process. Such a generalization is a important feature of multilayer perceptron.

DECODING BLOCK CODES USING NEURAL NETWORKS

5.1 Introduction

In this chapter neural networks has been utilized to decode the block codes [10]. Back propagation algorithm is used in order to train the neural network having one hidden layer to decode various block codes. The complete procedure for implementation and the training, testing and operation of the neural network simulator used has been discussed in detail.

5.2 Back propagation Network

Back propagation network is a very popular model in neural networks. It does not have a feedback connection, but errors are back-propagated during training. Least mean square error is used to correct synaptic weights. Errors in the output determine measures of hidden layer output errors, which are used as a basis for adjustment of connection weights between the input and hidden layers. Adjusting the two sets of weights between the pairs of layers and recalculating the outputs is an iterative process that continues until the errors fall below a tolerance level. Learning rate parameters scale the adjustments to weights. A momentum parameter is used for scaling the adjustments from a previous iteration and adding to the adjustments in the current iteration.

5.2.1 Mapping

The back propagation network maps input vectors. Pairs of input and output vectors are chosen to train the network. Once training is complete, the weights are set and the network can be used to find outputs or new inputs. The number of neurons in the input layer determines the dimension of the inputs, and the number of neurons in the output layer determines the dimension of outputs. If there are k neurons in the input layer and m neurons in the output layer, then this network can make a mapping of the k-dimensional space to an m-dimensional space. Of course, what that mapping depends on what pair of patterns or vectors are used as exemplars to train the network. Once the network is trained, the network gives the image of a new input vector under this mapping. Knowing what mapping is required, the back propagation network to be trained, tells the dimensions of the input space and the output space, in this way the number of neurons in the input and output layers can be determined.

5.2.2 Layout

The architecture of a back propagation network used for decoding block codes is shown in Figure 5.1 While there can be many hidden layers, back propagation network with only one hidden layer has been used for explanation. Also, the number of neurons in the input layer and that in the output layer are determined by the dimensions of the input and output patterns, respectively. As for the hidden layer, it is not easy to determine how many neurons are needed. In order to avoid cluttering the figure, the layout in Figure 5.1 has five input neurons, three neurons in the hidden layer, and four output neurons with a few representative connections. Therefore there are three fields, one for input neurons, and one for hidden processing elements, and one for the output neurons. There are feed forward connections from every neuron in fields A to every neuron in field B, and in turn, from every neuron in field B to every neuron in field C. Thus there are two sets of weights; those figuring in the activations of hidden layer neurons and those that help determine the output neuron activations. In training, all of these weights are adjusted by considering cost function in terms of the error in the computed output pattern and the desired output pattern.

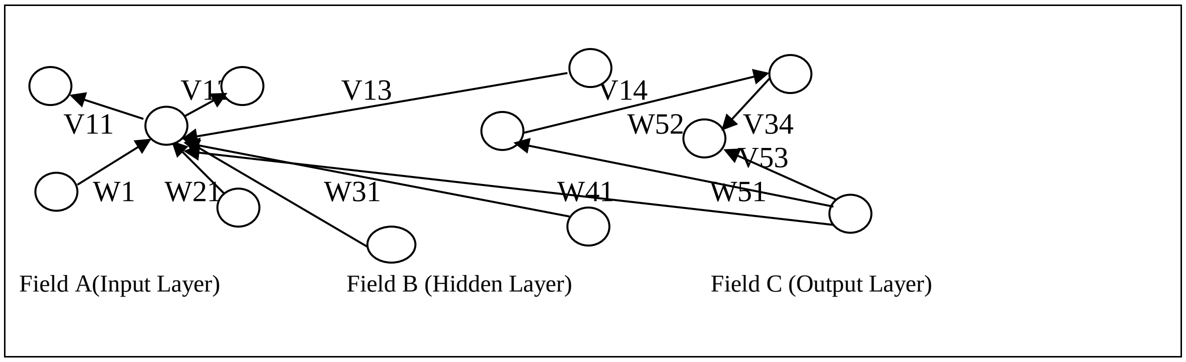


Figure 5.1 Layout of a Back propagation Algorithm

5.2.3 Training

The back propagation network undergoes supervised training, with a finite number of pattern pairs consisting of an input pattern and a desired or target output pattern. An input pattern is presented at the input layer. The neurons here pass the pattern digits to the next layer neurons, which are in a hidden layer. The outputs of the hidden layer neurons are obtained by using a bias, and also a threshold function with the activations determined by the weights and the inputs. These hidden layer outputs become inputs to the output neurons, which also process using possibly a bias and a threshold function with their activations to determine the final output from the network. The computed pattern and the input pattern are compared, a function of this error for each component of the pattern is determined, and adjustment to weights of connections between the hidden layer and the output layer is computed. A similar computation, still based on the error in the output, is made for the connection weights between the input and hidden layers. The process is then repeated as many times as needed until the error within a prescribed tolerance is achieved. This procedure is repeated with each pattern pair assigned for training the network.

5.3 Adjustment of Weights of connections from a neuron in hidden layer

The activation of a neuron in a layer other than the input layer is the sum of products of its inputs and the weights corresponding to the connections that bring in those inputs. Considering the jth neuron in the hidden layer and taking j= 2. If the input pattern is (1.1, 2.4, 3.2, 5.1, 3.9) and the target output pattern is (0.52, 0.75, 0.97). If the weights given for the second hidden layer neuron are given by the vector (-0.33, 0.07, -0.45, 0.13, 0.37). The activation will be (- 0.33. 1.1) + (0.07. 2.4) + (- 0.45 . 3.2) + (0. 13. 5.1) + (0.37 . 3.9) =0.471. Now adding to this an optional bias, or threshold value e.g. 0.679 to give 1.15 and using the sigmoid function given by 1 I (1 + exp (- x)), with x = 1.15, the output of this hidden layer neuron will be 0.7595. If the computed output pattern also turns out to be (0.61, 0.41, 0.57, 0.53), while the desired pattern is (0.52, 0.25, 0.75, 0.97). Obviously, there is a discrepancy between what is desired and what is computed. The component wise differences are given in the vector (- 0.09, - 0.16, 0.18, 0.44). This vector is used to form another vector where each component is a product of the error component, corresponding computed pattern component and the complement of the later with respect to 1. For the first component, error is - 0.09, computed pattern component is 0.61, and its complement is 0.39. Multiplying these together, we get - 0.02. Calculating the other components similarly, the vector comes out to be (- 0.02, - 0.04, 0.04, 0.11). Now the weights on the connections are needed between the second neuron in the hidden layer and the different output neurons. If these weights are given by the vector (0.85, 0.62, - 0.10, 0.21). The error of the second neuron in the hidden layer can be calculated using its output. Error = 0. 7595 . (1 - 0.7595) . (0.85 . -0.02) + (0.62 . - 0.04) + (-0.045) (-0.10 . 0.4) + (0.21 . 0.11)) = -0.0041. Next, learning rate parameter for this layer is needed. It is now set as 0.2. Multiplying this output of the second neuron with learning rate parameter of 0.2 in the hidden layer, to get 0.1519. Each of the components of the vector (- 0.02, - 0.04, 0.04, 0.11) is multiplied now by 0.1519, which is the latest computation attained. The result is a vector that gives the adjustments to the weights on the connections that go from the second neuron in the hidden layer to the output neurons. These values are given in the vector (- 0.003, - 0.006, 0.006, 0.017). After these adjustments are added, the weights to be used in the next cycle on the connections between the second neuron in the hidden layer and the output neurons become those in the vector (0.847, 0.614, ;... 0.094, 0.227).

5.4 Adjustment of weights of connections from a neuron in the input layer

To calculate the adjustments for the weights on connections going from the ith neuron in the input layer to neurons in the hidden layer and assuming I = 3.
To determine the adjustments for the weights on connections between the input and hidden layer, the errors determined for the outputs of hidden layer neurons, a learning rate parameter, and the activations of the input neurons, which are just the inputs, are needed. If the learning rate parameter is 0.15. Then the weight adjustments for the connections from the third input neuron to the hidden layer neurons are obtained by multiplying the particular hidden layer neuron's output error by the learning rate parameter and by the input component from the input neuron. The adjustment for the weight on the connection from the third input neuron to the second hidden layer neuron is

0.15 • (- 0.0041) = - 0.002.

If the weight on this connection is, - 0.045, then adding the adjustment of - 0.002, the modified weight of - 0.452 is achieved, to be used in the next iteration of the network operation. Similar calculations are made to modify all other weights as well.

5.5 Adjustments of threshold values or biases

The bias or the threshold value which were added to the activation, before applying the threshold function to get the output of neuron, will also be adjusted based on the error being propagated back. The adjustment for the threshold value of a neuron in the output layer is obtained by multiplying the calculated error (not just the difference) in the output at the output neuron and the learning rate parameter used in the adjustment calculation for weights at this layer. In the example, we had the learning rate parameter as 0.2, and the error vector as (- 0.02, - 0.04, 0.04, 0.11), so the adjustments to the threshold values of the four output neurons are given by the vector (- 0.004, - 0.008, 0.008, 0.022). These adjustments are added to the current levels of threshold values at the output neurons. The adjustment to the threshold value of a neuron in the hidden layer is obtained similarly by multiplying the learning rate with the computed error in the output of the hidden layer neuron. Therefore for the second neuron in the hidden layer, the adjustment to its threshold value is calculated as 0.15 (- 0.0041), which is - 0.0006. Adding to this to the current threshold value of 0.679 to get 0.6784, which is to be used for this neuron in the next cycle of operation of the neural network.

5.6 Notations and equations

In order to explain the mathematical equations which have been used during the course of implementation of back propagation algorithm, various notation used are required to be explained so that the equations used to specify the output of various neurons in different layers and adjustment of weights of neurons is completely understood.

5.6.1 Notation

Two matrices have been specified, in order to understand the equations used, whose elements are the weights on connections. One matrix refers to the interface between the input and hidden layer, and the second refers to that between the hidden layer and the output layer. Since connections exist from each neuron in one layer to every neuron in the next layer, there is a vector of weights on the connections going out from any one neuron. Putting this vector into a row of the matrix, there will as many rows as there are neurons from which connections are established.

If M1and M2 be these matrices of weights, then M1 [i] [j] represents the weight on the connection between the ith input neuron to the jth neuron in the hidden layer. Similarly, M2 [i] [J] denotes the weight on the connection between the ith neuron in the hidden layer and the jth output neuron.

Notations x, y and z are used for the outputs of neurons in the input layer, hidden layer, and output layer, respectively, with a subscript attached to denote which neuron in a given layer is being referred. If P denote the desired output pattern, with p; as the components and m be the number of input neurons, so that according to our notation (X_1, X_2….., X_m) will denote the input pattern. Let's say if P has r components, the output layer needs r neurons. λ is the learning rate parameter for the hidden layer, and μ is for the output layer, also θ with the appropriate subscript represents the threshold value or bias for a hidden layer neuron, and τ with an appropriate subscript refers to the threshold value of an output neuron. The errors in output at the output layer are denoted by $e_{j's}$ and those at the hidden layer by $t_{i's}$. The /', prefix to any parameter means change in or adjustment to that parameter. Also the threshold function used is the sigmoid function, f(X) = 1/ (1 + exp (-x)).

5.6.2 Equations

Output of j_{th} hidden layer neuron:

$$Y_j = f\left(\sum_i x_i \; M_1[i][j] + \theta_j\right) \tag{5.1}$$

Output of j_{th} output layer neuron:

$$Z_j = f\left(\sum_i y_i \; M_2[i][j] + \tau_j\right) \tag{5.2}$$

ith component of vector of output differences: desired value-computed value = P_i- Z_i,
ith component of output error at the output layer:

$$e_i = z_j(1- z_j)(p_i - z_j) \tag{5.3}$$

ith component of output error at the hidden layer:

$$t_i = Y_i(1- Y_i) \left(\sum_j M2[i][j] \; e_i \right) \tag{5.4}$$

Adjustment for weight between ith neuron in hidden layer and jth output neuron:

$$\Delta M2[i][j] \;\; \mu Y_i \; e_j \tag{5.5}$$

Adjustment for weight between ith input neuron and jth neuron in hidden layer:

$$M1[i][j] = \lambda t_i X_i \tag{5.6}$$

Adjustment to the threshold value or bias for the jth output neuron:

$$\Delta t_i = \mu \, e_j \quad (5.7)$$

Adjustment to the threshold value or bias for the jth hidden layer neuron:

$$\Delta\theta_j = \lambda \, e_j \quad (5.8)$$

Use of momentum parameter ϒ instead of Eqn (5. 7) and (5.8) results into

$$\Delta M2[i][j](t) = \mu Yi \; e_j + \lambda \Delta M2[i][j](t-1) \quad (5.9)$$

and

$$\Delta M1[i][j](t) = \mu Xi \; e_j + \lambda \Delta M1[i][j](t-1) \quad (5.10)$$

5.7 Objectives Specified for a back propagation simulator

The simulator that is used allows the user to specify the number and size of all layers. More than one hidden layers can be used depending upon the requirement. The state of the network can be saved and restored for subsequent use and it has the ability to run from an arbitrarily large training data set or test data set. The user is asked to provide key network and simulation parameters. The key information is displayed at the end of the simulation.

5.8 How to use a simulator

There are two modes of operation in the simulator. The user should know which mode of operation is desired. These modes are Training mode and Non -training mode (Test mode).

5.8. 1 Training Mode

In this mode training file prepared to train the simulator is given to the current directory of the simulator called training.dat. This file contains input and corresponding output patterns on which the simulator is given necessary training. Each input value is separated by one or more spaces. As a convention, a few extra spaces are used to separate the inputs from the outputs. An example of a training.dat file that contains two patterns can be expressed as:-

```
0.4     0.5   0.89
0.23    0.8   -0.3

-0.4          -0.8
 0.6          0.34
```

The first pattern has inputs 0.4, 0.5, and 0.89 with an expected output of -0.4 and - 0.8. The second pattern has inputs of 0.23, .08 and - 0.03 and outputs of 0.06 and 0.34. Since there are three inputs and two outputs, the input layer size for the network must be three neurons and the output layer size must be two neurons. Another file that is used in training is the weights file. Once the simulator reaches the error tolerance that was specified by the user or the maximum number of iterations, the simulator saves the state of the network, by saving all of its weights in a file called weights.dat. This file can then be used subsequently in another run of the simulator in non-training mode. The total and average error is presented at the end of the simulation to have an idea of the trained network. In addition, the output generated by the network for the last pattern vector is stored in an output file called output.dat.

5.8.2 Non-training Mode (Test Mode)

In this mode, test data is given to the simulator in a file called test.dat. This file contains only input patterns. When this file is applied to an already trained network, an output.dat file is generated, which contains the outputs from the network for all of the input patterns. The network goes through one cycle of operation in this mode, covering all the patterns in the test.dat file. To start up the network, the weights file, weights.dat is read to initialize the state of the network and the network size parameters are to be specified which were used during training of the network.

5.8.3 Operation

The simulator is trained for a chosen architecture. Keeping in mind the fact that input and output layer sizes are dictated by the input patterns presented to the network and the outputs expected form the network. Once architecture has been decided, the training data is prepared and saved in the training.dat file.

For training the network mode 1 is selected and the values for the error tolerance and the learning rate parameter, lambda or beta are specified. The maximum number of cycles, or passes through the training data required to train the network are also given. The number of layers (between three and five, three implies one hidden layer whereas five implies three hidden layers) and the size for each layer, from the input to the output is also specified.

The simulator then begins training and reports the current cycle number and the average error for each cycle. Error must decrease with time. If it is not then the simulator is restarted with a brand new set of random weights to get better solution. Once the simulation is over the information about the number of cycles, patterns used and the average error that resulted can be viewed. The weights are saved in the weights.dat file. The same can be renamed to subsequently use this particular state of the network later. The size and number of layers can be obtained from the information contained in this file. To get a full blown accounting of each pattern and the match to that pattern, by copying the training file to the test file and deleting the output information from it and running the simulator in the test mode to get a full list of all the input stimuli and responses in the output.dat file.

5.9 Summary of the files used in back propagation simulation

In order to carry out the training of the simulator and then using the same values of the network while giving the test data and monitoring the output various file are used to store the training and test data and also the weight of the trained network.

5.9.1 Weights.dat

This file is used to store the weights for the network. It shows the layer number followed by the weights that feed into the layer. The first layer, or input layer, layer zero does not have any weights associated with it. An example of the weights.dat file is shown for a network with three layers of sizes 3, 5 and 2. The row width for layer n matches the column length for layer n+ 1:

```
1 -0.199660 -0.859660 -0.339660   -0.259660   0.520340
1 0.293860 -0.487140   0.212860   -0.967140   -0.427140
1 0.542106 -0.177894   0.322106   -0.977894   0.562106
2 -0.175350 -0.835350
2 -0.330167 -0.250167
2 0.503317 0.283317
2 -0.477158 0.222842
2.-0.928322 -0.388322
```

In this file the row width for layer 1 is 5, corresponding to the output of that (middle) layer. The input for the layer is the column length, which is 3, just as specified. For layer 2, the output size is the row width, which is 2, and the input size is the column length, 5, which is the same as the output for the middle layer.

5.9.2 Training.dat

This file contains the input patterns for training. Large training file can be made without degrading the performance of the simulator. The simulator caches data in memory for processing. This is to improve the speed of the simulation since disk accesses are expensive in time. A data buffer, which has a maximum size specified, define statement in the program, is filled with data from the training.dat file whenever data is needed.

5.9.3 Test.dat

The test.dat file is just like the training.dat file but without expected outputs. This file is given to a trained neural network in test mode to see what responses it gives for untrained data.

5.9.4 Output.dat

The output.dat file contains the results of the simulation. In Test mode, the input and output vectors are shown for all pattern vectors. In the training mode, the expected output is also shown, but only the last vector in the training set is presented, since the training set is usually quite large. Output file in training mode is shown as

```
 For input vector:

0.400000    -0.400000

Output vector is:

0.880095

Expected output vector is:

0.900000
```

DECODING CONVOLUTIONAL CODES USING NEURAL NETWORKS

6.1 Introduction

In 1967, Viterbi proposed a maximum likelihood decoding scheme that was relatively easy to implement convolutional codes with small memory orders. Since Viterbi algorithm has been implemented in sequential and parallel processing environments to speed up the decoding process [11].

In this Chapter an approach of decoding the convolutional codes using neural networks has been discussed. The neural network architecture selected consists of one hidden layer. Viterbi algorithm has been recommended for generating the training patterns for the neural network decoder. A detailed mathematical treatment of how to update the synaptic weights is carried out for the neural network has been explained by first calculating the error vector. Flow chart for the neural network training programmed has also been shown to be considered for implementation.

6.2 Convolutional Codes

A convolutional encoder is designated by (n, k, J), where n is the number of encoder outputs, k is the number of inputs, and J is known as the constraint length [10], [11].

In this chapter during the course of discussion the convolutional encoder (2, 1, 3) shown in Figure 6.1 will be used. Figure 6.2 shows its trellis diagram. In general, the trellis structure is repeated after trellis depth J is reached.

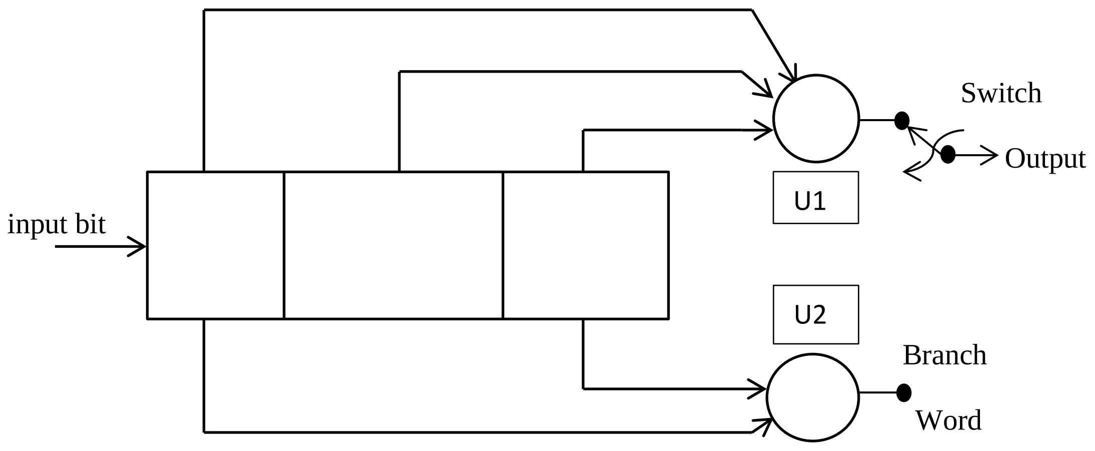


Figure 6.1 Convolutional Encoder (2,1,3)

6.3 Design Approach

Since one of the goals to be achieved is to improve the speed of the decoding process of convolutional code, it is necessary to find a neural network with an optimum structure to accomplish this task [8] [9].

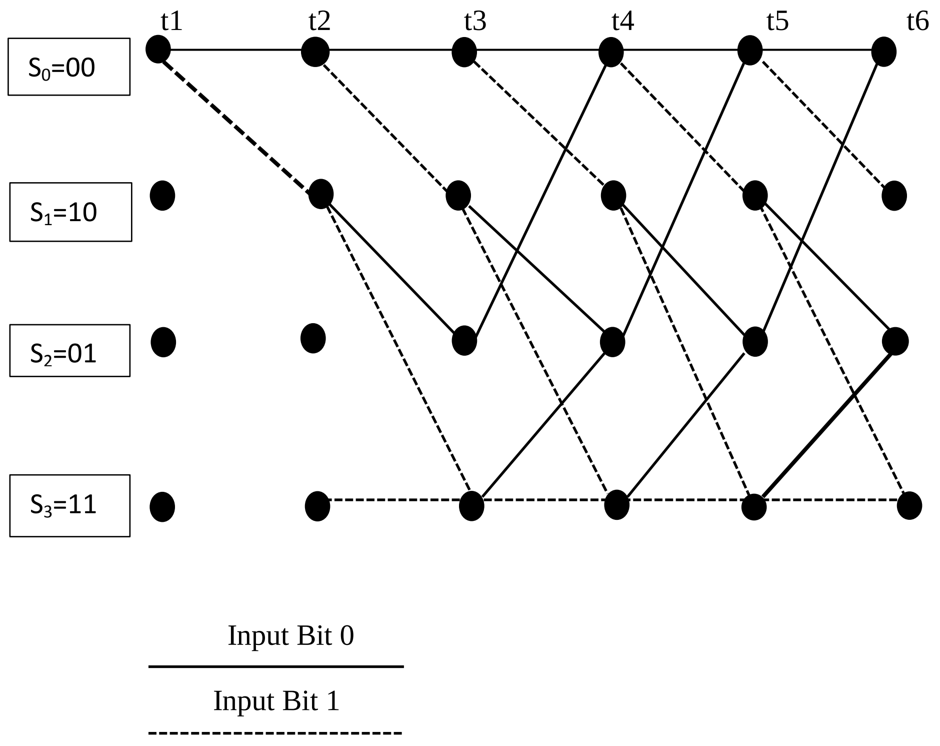


Figure 6.2 Trellis Diagram of the (2, 1, 3) Encoder shown in Figure 6.1

6.3.1 Neural Network Architecture Selection

A simple perception with one hidden layer is chosen as the basic structure to keep the network as simple as possible and the dynamic node creation method is used in training the network to assure that the optimum network with the minimum number of neurons in the hidden layer is created.

Because the decoding of convolutional code is dependent on historical data, the input to the neural network decoder must contain previously received code words. The output of the neural network should be the decoded result of the given input it should also be able to keep track of previously received code words. In order to achieve this, the neural network maps the most recently received code words, excluding the least recently received code word (the obsolete one), to the output of the network and feeds this information back to the input of the network for decoding the next code word as shown in Figure 6.3.

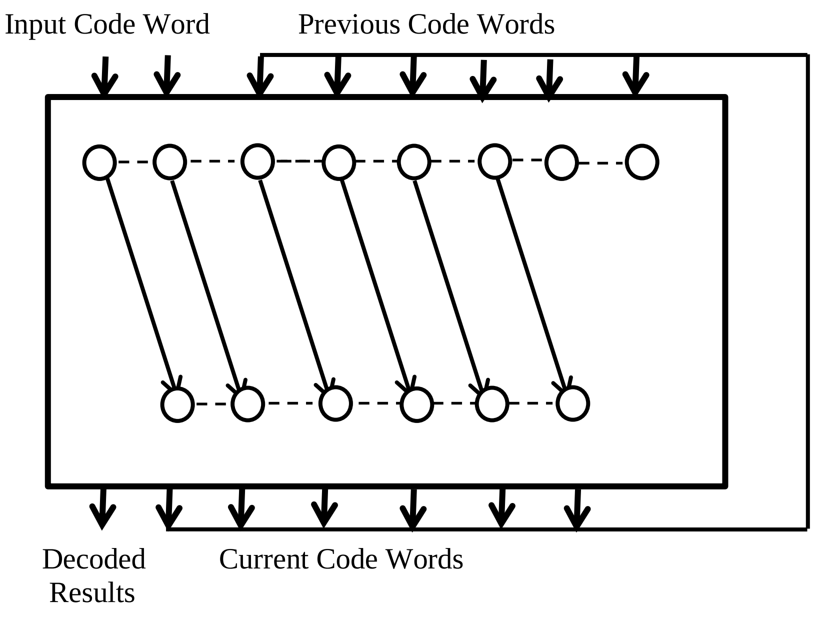


Figure 6.3 Neural Network convolutional decoder.

6.3.2 Updating the Synaptic Weights for Convolutional Decoder

If d_i is the desired output and y_i is the actual output of the decoder, then

$$E = \sum_{k=1}^{n} |e_k|^2 \qquad (6.1)$$

Where n = number of code words in the training file
In neural network convolutional decoder of Figure 6.3 there is an offset of 6 bits.
Present code word +2 x (Past code words) = 6 bits offset

$$E = \sum_{\mathrm{k}=1}^{\mathrm{n}} |e_k|^2$$

$$E = \sum_{\mathrm{k}=1}^{\mathrm{n}} (d_k - (\tanh(\mathrm{yk}))^2 \qquad 6.2)$$

$$\frac{\partial \mathrm{E}}{\partial \mathrm{w_l}} = \sum_{\mathrm{k}=1}^{\mathrm{n}} \frac{\partial}{\partial w_l} (d_k - \mathrm{tahnhy}(\sum_{\mathrm{i}=1}^{\mathrm{n_w}} w_i x_i))^2$$

Where k= $\sum_{\mathrm{i}=1}^{\mathrm{n_w}} w_i x_i$

$$\frac{\partial \mathrm{E}}{\partial \mathrm{w_l}} = \sum_{\mathrm{k}=1}^{\mathrm{n}} 2(d_k - \mathrm{tahnhy}(\sum_{\mathrm{i}=1}^{\mathrm{n_w}} w_i x_i)) \quad (-\frac{\partial}{\partial \mathrm{w_l}} \mathrm{tahnhy}(\sum_{\mathrm{i}=1}^{\mathrm{n_w}} w_i x_i))$$

$$\frac{\partial \mathrm{E}}{\partial \mathrm{w_l}} = -2\sum_{k=1}^{n}(d_k - \mathrm{tanhy}(\textstyle\sum_{i=1}^{n_w} w_i x_i)) \quad (\frac{\partial}{\partial \mathrm{w_l}}\mathrm{tanhy}(\textstyle\sum_{i=1}^{n_w} w_i x_i) \quad) \qquad (6.3)$$

$$\frac{\partial \mathrm{E}}{\partial \mathrm{w_l}} = -2\sum_{k=1}^{n}(d_k - \mathrm{tanh}\Upsilon(\textstyle\sum_{i=1}^{n_w} w_i x_i))\Upsilon(1-\mathrm{tanh}\Upsilon(\textstyle\sum_{i=1}^{n_w} w_i x_i))^2))((\textstyle\sum \frac{\partial \mathrm{w_i}}{\partial \mathrm{w_l}} x_i))$$

$$\frac{\partial \mathrm{E}}{\partial \mathrm{w_l}} = -2\sum_{k=1}^{n}(d_k - \mathrm{tanh}\Upsilon(\textstyle\sum_{i=1}^{n_w} w_i x_i))\Upsilon(1-\mathrm{tanh}^2\,\Upsilon(\textstyle\sum_{i=1}^{n_w} w_i x_i)) \qquad (6.4)$$

Where $\left(\sum \frac{\partial \mathrm{w_i}}{\partial \mathrm{w_l}} \mathrm{x_i}\right) = \sum_{\mathrm{i}} \delta_{il} x_i = x_l$ (6.5)

$$\frac{\partial \mathrm{E}}{\partial \mathrm{w_l}} = -2\sum_{k=1}^{n}(d_k - \mathrm{tanh}\Upsilon(\textstyle\sum_{i=1}^{n_w} w_i x_i))\Upsilon(1-\mathrm{tahn}\Upsilon(\textstyle\sum_{i=1}^{n_w} w_i x_i))^2))((x_l)) \qquad (6.6)$$

$$(\mathrm{w_l})_{\mathrm{new}} = (\mathrm{w_l})_{\mathrm{old}} - \eta\frac{\partial \mathrm{E}}{\partial \mathrm{w_l}} \qquad (6.7)$$

6.3.3 Training Pattern Generation

It is essential to train the network with a set of patterns which result in the correct decoding of any sequence of code words. The simplest approach to this task is to utilize the conventional encoding and decoding procedures in the pattern generating process. For a decoder which decodes Q-1 previously received code words along with the current code word there are 2 Q possible combinations of data sequences to decode. Training patterns can be generated by passing all possible combinations of data sequences through the encoder to generate the code word sequences which should be received by the decoder.

However, these 2Q patterns do not include those code sequences which contain transmission errors. Therefore, it is necessary to append the patterns of those sequences which contain transmission errors to assure error-correcting capabilities of the neural network decoder. The target patterns can be generated by decoding the erroneous code word sequences with conventional decoding, methods such as the Viterbi algorithm.

The training pattern generation program utilizes the encoding object and the Viterbi decoding object. The encoding object is used to generate those at terms which do not contain transmission error and the Viterbi decoding object is used to generate those which do contain transmission errors. The network training program, with flow chart shown in Figure 6.4, does the construction, training, and storage of the weight matrices file of a network based on the training pattern file. The network is trained by adjusting weight matrices after each pass of the training pattern file. Convergence is achieved when the maximum error at the output neurons becomes less than the specified amount given in the training pattern file. The network decoder program simulates the network described by the weight matrices file which decodes on a given input code word data file. The outputs of the network decoder program can then be used to compare the performance of a neural network decoder with that of a conventional decoder.

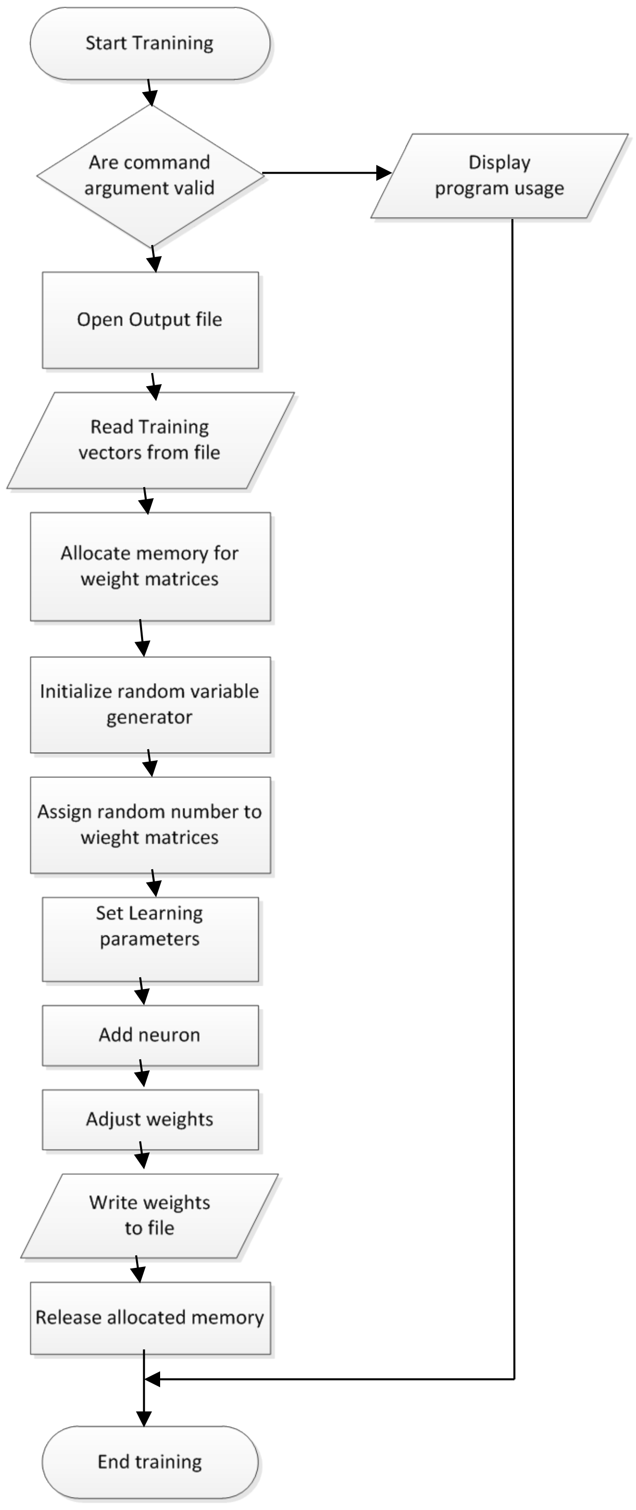
6.3.4 Design Requirements for Convolutional Decoder

For the convolutional decoder there is a requirement of writing three programs for simulation of the network if Matlab is used.

A training pattern generates program by utilizing the encoding object and Viterbi decoding object. A network training program for which the flow chart is shown in Figure 6.4. The network is trained by adjusting weight matrices after each pass of the training pattern file.

Convergence is achieved when the maximum error at the output neurons becomes less than the specified amount given in the training pattern file. The network decoder program to simulate the network described by the weight matrices file which decodes a given input code word data file.

FLOW CHART


Start Tranining
Are command argument valid
Display program usage
Open Output file
Read Training vectors from file
Allocate memory for weight matrices
Initialize random variable generator
Assign random number to wieght matrices
Set Learning parameters
Add neuron
Adjust weights
Write weights to file
Release allocated memory
End training

Chapter 7

RESULTS AND DISCUSSION

7.1 Introduction

The original data in the form of blocks can although be reconstructed by various techniques as discussed in considerable detail in chapter 2 and chapter 3, but instead of using conventional techniques neural network has been used to do the same job. In this technique a three layer neural network was used. Back propagation algorithm was utilized for its training and least mean squared error to adjust the weights between input layer and hidden layer and hidden layer and output layer. The network does not have feedback connections, but errors are back propagated during training. The network thus used was trained before the actual data (output of the simulated channel) was provided as input. The whole procedure comprising source code and the deliberations on the results thus achieved can be viewed in the sequence as presented in the succeeding paragraphs.

7.2 Simulated Codeword and Its Input-Output Relationship

Linear block code (7,4) was used to train the ANN simulator by copying the code into training.dat file. The Matlab source code was used to train the network keeping in view the procedure explained in chapter 5.

7.3 Simulated ANN and Its Input-Output Relationship

The utilized neural network was Multilayer Perceptron (MLP). In this network back-propagation algorithm was used during training mode to adjust the weights of the connections existing between input layer and hidden layer and also between hidden layer and output layer. In this proposed network all the nodes were connected to all the nodes in the adjacent layer through unidirectional links.

First of all the most important part of this implementation was training the network. Initially all the synaptic weights were given small random values. Then the message blocks along with their code words were provided as input to the network. For each input, network provided a relevant output. This output was compared with the actual source symbol. The difference between this output and the desired output was calculated as explained in chapter 5. This constituted the error function which was propagated back through the network and the weights were adjusted till the time the error was reduced as per the specified error function which can be varied according to the requirement and the application.

The lesser is the intended error, the more time it takes for the network to be trained. Since the nature of the error space cannot be known a-priori, neural network analysis often requires a large number of individual runs to determine the best solution. Most learning rules have built-in mathematical terms to assist in this process which control the 'speed' (Beta-coefficient) and the 'momentum' of the learning. A trained neural network can be used as an analytical tool on other data. We have calculated and draw this learning curve in figure 7.1.

In order to test the network a test.dat file in which the errors were appended was given to the simulator. To test the simulator, there is no need to specify any training runs and instead the network works in forward propagation mode. New inputs are presented to the input pattern where they filter into and are processed by the middle layers as though training was taking place, however, at this point the output is retained and no back propagation occurs. The output of a forward propagation run is the predicted model for the data, where the input vector and also the output vector achieved after processing the data through the trained neural network has been shown. The output.dat file also contains the rounded output vector as well.

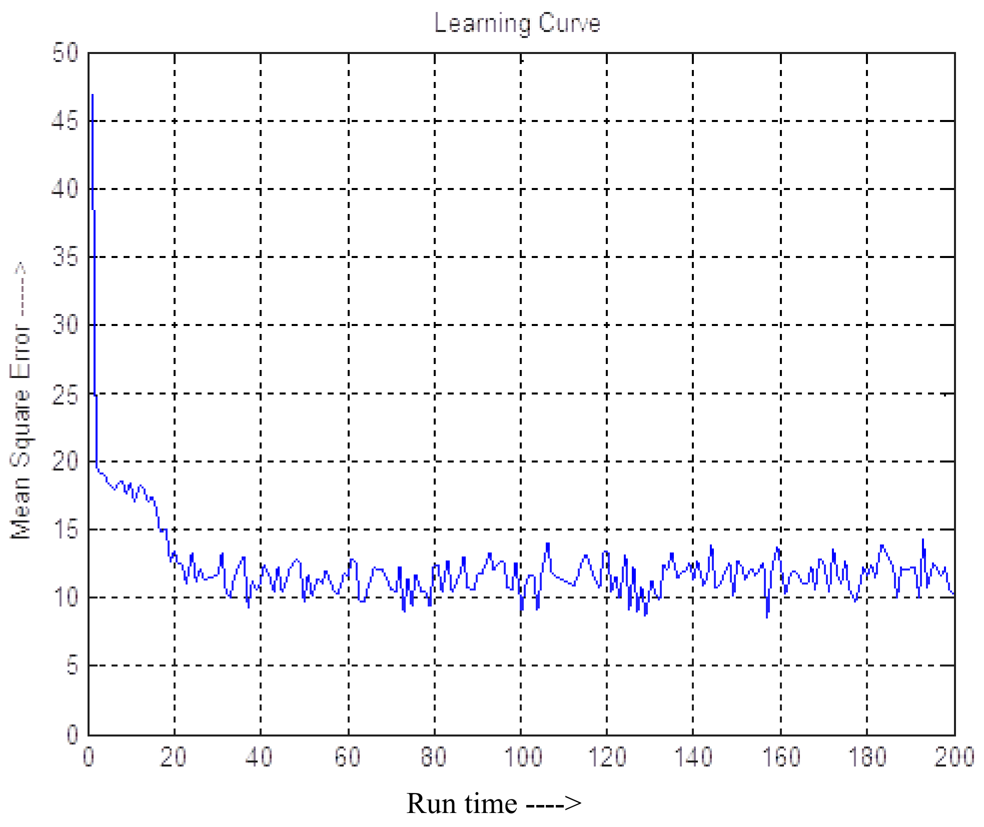


Figure 7.1: Learning curve of training mode [11]

The above figure shows the learning of neural network. The x-axis shows the number of times we make it run to achieve the threshold. Y-axis shows the mean square error.

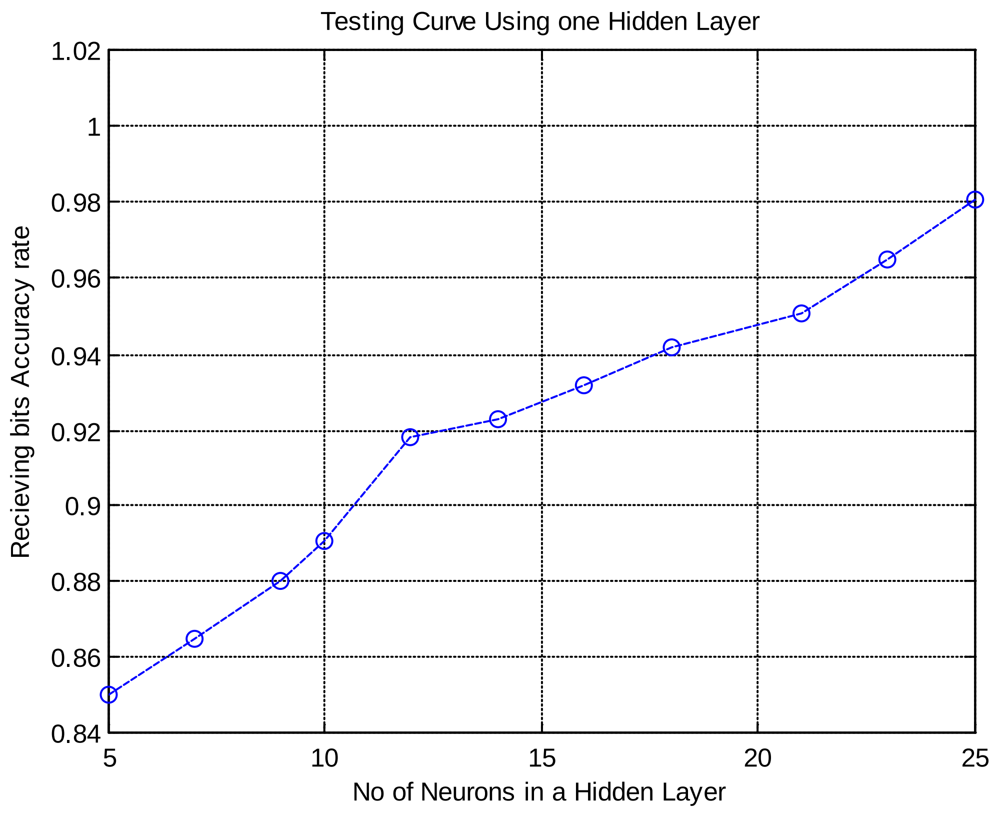


Figure7.2: Plot showing the decoded bits using single hidden layer

Figure 7.2 shows the testing curve with the number of hidden layers is 1 and the percentage of the accuracy in bits received.
This curve shows that the maximum accuracy that we can achieve is 98% in the case of one hidden layer.

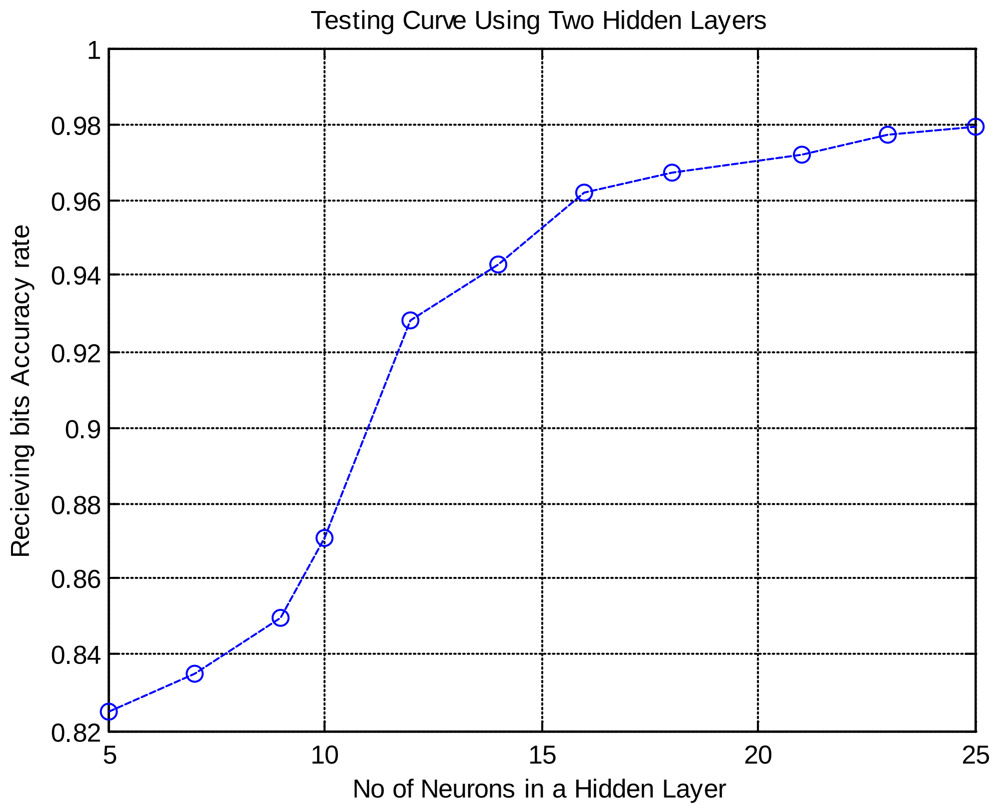


Figure7.3: The plot of the decoded bits using 2 hidden layers

Figure 7.3 shows the decoded bits after the error correction when we use 2 hidden layers. The result is almost the same as in figure 7.2 with just little difference.

The number of neurons we use in all the cases is the same.

Figure 7.4 shows the plot for 3 hidden layers but with the same number of neurons. According to the calculations and results the number of hidden layers may change the time to reach the accuracy, but the level of accuracy is almost the same there maybe the difference of 1 or 2 percent.

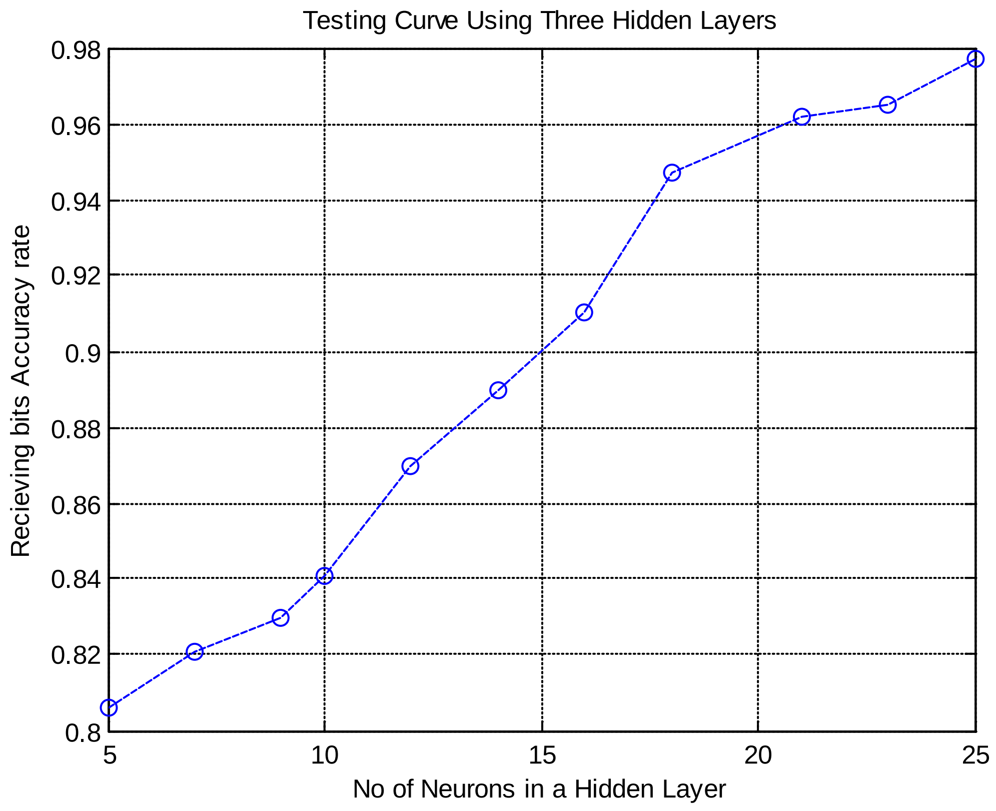


Figure7.4: The plot of the decoded bits using 3 hidden layers

So after looking at the curves in the above figures 7.2 and 7.3 we can conclude that the bits that are received after the error correction are decoded 97% correctly.

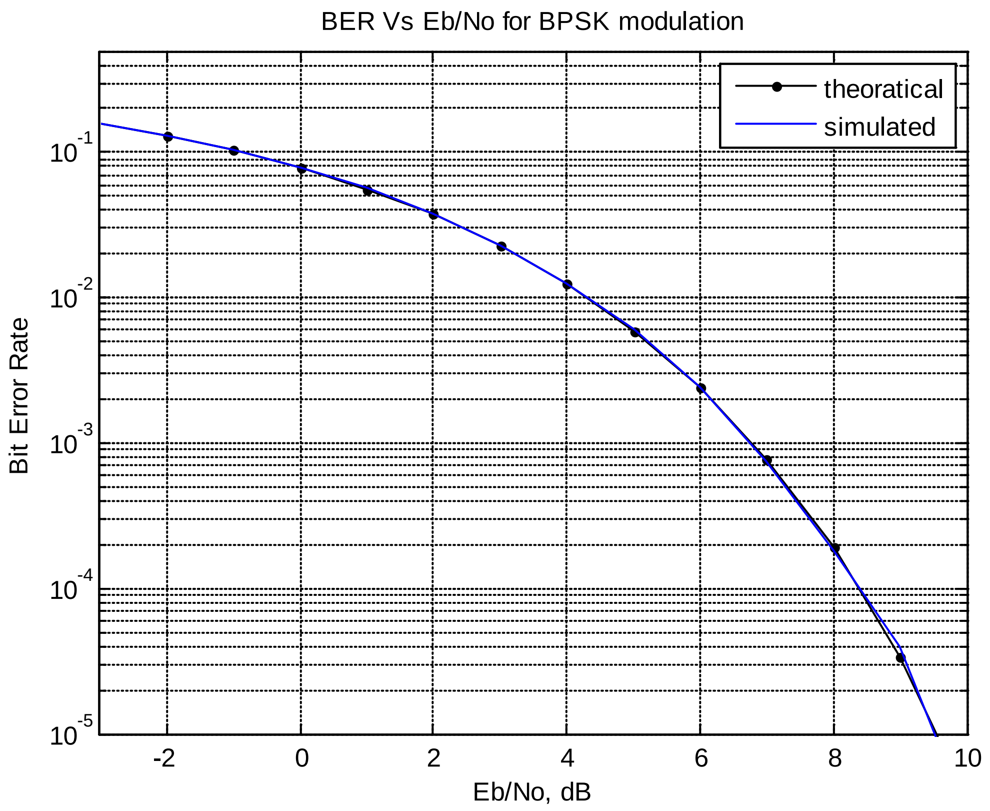


Figure7.4: Plot of the decoded bits using BPSK

If we take a case of BPSK the Bit Error Rate we calculate is almost 1 in 100000, but this is the ideal case which does not happen and if we include the factors like AWGN, forward error correction, frame synchronization it will obviously be changed to around 10 %.

7.4 Summary and Conclusions

We used different data for learning and testing. Although the simulated network provided the output which corresponds to the original data but there is a limitation associated with it. That is the desired output was known a priori and the network was trained according to that. However in many applications the desired output is not known a priori. In that particular case the network makes the decision as to what it was in the absence of any reference information. However this network cannot be expected to provide the correct output in the absence of a reference.

During training the network on various codes it has been observed that the number of neurons in the input and output layers can be exactly specified because they depend upon the message bits in the code word respectively. But, it is difficult to specify the number of neurons in the hidden layer. Therefore, more time is consumed for selecting different values of hidden layer neurons, learning rate parameter and also the number of cycles required to train the network so that the output should converge for the specified value of error tolerance.

The network was given an input of 200 bits generated using the algorithm and was asked to predict the next bit. This algorithm was used to generate a set of 2000 training facts to train a back propagation network. The first data sets generated were generated with sets of correlated data that is 10 sets of 200 bits were generated using the algorithm.

We generated a database of 1000 sets of 200 bits for a test on the trained network. The network got 982 out of 1000 correct which is 98.2% using a threshold of 0.3 to determine correct responses. This was a result with a very high degree of probability was better than random. Now we introduced 5% noise into the training data and then it was trained on that until it could correctly predict all of the 1000 training patterns.

To make the problem more difficult, a network was next trained on a set where each of the 1000 training sets was generated with a different 31 bit seed. These were used first without noise and produced a net which got 427 out of 500 correct. Corrupting the training set with 10% random noise produced a network which produces 402 correct out of 500 noisy test sets. The noisy data coupled with the fact that we trained this net also using uncorrelated data (the 31 bit seeds were unrelated for each set) greatly lengthened the training time. It took 43 hours and 28 minutes for the network to learn the training set in this worst case training set (i.e. noise and uncorrelated data sets).

If we sum up the results and conclusion for the tests we have done, we can conclude that the threshold level that we achieve is almost the same in all the cases whether its single, double or 3 layer that is 98% , 97 % so it's not much difference. But the time that takes to reach that level is little different and is the least in case of single layer.

7.5 Future Work and Recommendations

In order to make use of ANN's for decoding the convolutional codes, an approach which can be followed for further study has been discussed in considerable detail in chapter 6 which includes the design architecture to be used and basing upon the same mathematical calculations have been made for updating the synaptic weights for convolutional decoder. For training pattern generation a flow chart for neural network training program has also been included. The same may be used to design a simulator for decoding convolutional codes.

References:

[1] Shih-Chi Huang and Yih-Fang Huang, “ Bounds on the Number of Neurons in Multilayer Perceptions”. IEEE Trans. Neural Networks, Vol. 2, no. 1, January 1991.

[2] Gomez, J.M Lopez, O. Montes, M. Bota, SA. Juvells, I. Herms, A. Fac, de Fascia, “Implementation and design of new model of neural network with application on typographical character recognition”. IEEE international conference on image processing 1996.

[3] Ciocoiu, I.B “Analaog decoding using gradient-type Neural Network ,” IEEE Trans. Communications, Vol. 7, pp. 1034-1038, July, 1996.

[4] Data & Analysis center for Software, “Artificial Neural Networks Technology”, 1992.(http:// www.dasc.dtic.mil/techs/neural/neural.title,html) printed November 1998.

[5] Haykin S, Neural Networks, 2nd ed, prentice Hall, 1999.

[6] Hamalainen, A. , “Convolutional Decoding using recurrent neural networks” IEEE Trans. Optical Networks, Vol. 5, pp. 3323-3327, 1999.

[7] Wells and Richard B, Applied Coding and Information Theory for Engineers, Upper Saddle River, NJ: Printice- Hall 1999.

[8] Lin and Ming-Bo, “New Path History Management Circuits for Viterbi Decoders,” IEEE Trans. Communications, Vol. 48, pp. 1605-1608, October, 2000.

[9] **Secker**, P.J., “A Generalized framework for convolutional decoding using a recurrent neural network” IEEE Trans. Communications, Vol. 3, pp. 1502-1506, December, 2003

[10] **Berber**, S.M, “Convolutional decoders based on artificial neural networks” IEEE Trans. Communications, Vol. 2, pp. 1551-1556, December, 2004

[11] Rajbhandari, S., “The performance of PPM using Neural Network and Symbol Decoding for Diffused Indoor Optical Wireless Links, Vol. 3, pp. 161-164, July, 2007