

# **On the Realization of Asymmetric High Radix Signed Digital Adder using Neural Network**

A thesis submitted to the Department of Computer Science &  
Engineering of BRAC University, Dhaka, Bangladesh

By

**Tofail Ahammad**  
**Student ID:02201036**

Requirement for the Degree of Bachelor of  
Computer Science & Engineering

Spring 2006



**BRAC University**

## DECLARATION

In accordance with the requirements of the degree of Bachelor of Computer Science and Engineering in the division of Computer Science and Engineering, I present the following thesis entitled 'On the Realization of Asymmetric High Radix Signed Digital Adder using Neural Network'. This work is performed under the supervision of Md. Sumon Shahriar, Lecturer, and Department of Computer Science &Engineering, BRAC University

I hereby declare that the work submitted in this thesis is my own and based on the results found by myself. Materials of work found by other researcher are mentioned by reference. This thesis, neither in whole nor in part, has been previously submitted for any degree.

Signature of Supervisor

Signature of Author

.....

Md. Sumon Shahriar

.....

Tofail Ahammad

## ACKNOWLEDGEMENT

First of all, I would like to express my sincerest gratitude and profound gratitude to my supervisor, Md. Sumon Shahriar, Lecturer, Department of Computer Science and Engineering, BRAC University, for his supervision, persuade guidance. I got his positive response choosing the thesis topic and complete guidance throughout its completion. It is my great opportunity to get chance to work under him for my thesis work. Even though being occupied with busy schedule, he often showed much interest and allocated time to review my mistakes and that help me to improve the work. I gathered knowledge from his comments, revisions and discussions during this period.

I want to give my heartiest thanks to Risat Mahmud Pathan, Lecturer, Department of Computer Science and Engineering, BRAC University, Special thanks to Ex-student of BRAC University, Murtoza Habib who help me to learn Matlab-7 simulation tools. Many thanks to Annajiat Rasel student of BRAC University for his accompany and helping hand in case of necessity. I am also grateful to all of my friends who encourage me to work with this topic and help me several times when it is needed.

Last of all, thanks to the Almighty for helping me in every step of doing as it should be.

## ABSTRACT

This paper proposes an asymmetric high-radix signed-digital (AHSD) adder for addition on the basis of neural network (NN) and shows that by using NN the AHSD number system supports carry-free(CF) addition. Besides, the advantages of the NN are the simple construction in high speed operation. Emphasis is placed on the NN to perform the function of addition based on the novel algorithm in the AHSD number system.

Since the signed-digit number system represent the binary numbers that uses only one redundant digit for any radix  $r \geq 2$ , the high-speed adder in the processor can be realized in the signed-digit system without a delay of the carry propagation. A Novel NN design has been constructed for CF adder based on the AHSD<sub>(4)</sub> number system is also presented. Moreover, if the radix is specified as  $r = 2^m$ , where  $m$  is any positive integer, the binary-to-AHSD<sub>(r)</sub> conversion can be done in constant time regardless of the word-length. Hence, the AHSD-to-binary conversion dominates the performance of an AHSD based arithmetic system.

In order to investigate how NN design based on the AHSD number system achieves its functions, computer simulations for key circuits of conversion from binary to AHSD<sub>(4)</sub> based arithmetic systems are made. The result shows the proposed NN design can perform the operations in higher speed than existing CF addition for AHSD.

# CONTENTS

## CHAPTER 1

Introduction -----	1-2
1.1 Objectives -----	3
1.2 Thesis overview -----	3

## CHAPTER 2

2.1 The Brief History of Neural Networks -----	4
2.1.1 Artificial Neural Networks -----	5
2.2.2 Detailed Description of Neural Network Components -----	6-8
2.2.3 Artificial Neurons and How They Work -----	9-10

## CHAPTER 3

3.1 Signed–Digit Number Representation -----	11-13
3.2 The AHSD Number System -----	13
3.2.1 Binary to AHSD Conversion -----	14
3.2.2 An Algorithm to make Pairs from Binary to AHSD Conversion	14-15
3.2.3 Addition of AHSD <sub>(4)</sub> Number System -----	16
3.2.4 AHSD addition for Radix-5 -----	17
3.2.5 AHSD <sub>(4)</sub> CF Adder -----	18

## CHAPTER 4

4.1 Proposed Design of adder using Neural Network -----	19-20
---	-------

## CHAPTER 5

5.1 Experimental Result and Discussion -----	21-22
5.1.1 Discussion of Simulation Results-1-bit -----	22-25
5.1.2 Discussion of Simulation Results-2-bit -----	25-27
5.1.3 Discussion of Simulation Results-3-bit -----	27-28
5.1.4 Overall Discussion on Experimental Result -----	28

CHAPTER 6	
6.1	Limitation of the Proposed Adder ----- 29
CHAPTER 7	
7.1	Future Development ----- 30
CHAPTER 8	
8.1	Conclusion ----- 31
REFERENCES	----- 32-33
CODING	----- 34-39
APPENDIX	----- 40-73

## LIST OF TABLES

3.1	Rule for selecting carry $c_i$ and intermediate sum $s_i$ based on $\theta_i = x_i + y_i$ and $\theta_{i-1} = x_{i-1} + y_{i-1}$ for radix $r=2a$ .....	12
5.1	NN experimental results for AHSD adder design.....	22
5.2	Summary of 1-bit AHSD number's simulation with different transfer functions using Matlab-7.....	25
5.3	Summary of 2-bit AHSD number's simulation with different transfer functions using matlab-7.....	27
5.4	Summary of 1-bit AHSD number's simulation with logsig transfer function using matlab-7.....	28

## LIST OF FIGURES

2.1	A basic artificial neuron.....	9
4.1	Neural Network prototype for AHSD number system addition.....	19
4.2	Trivial 1-bit AHSD radix-4 adder using NN.....	19
4.3	N-bit adder generalization.....	20
5.1	The Workflow of simulation procedure for AHSD numbers.....	21
5.2	<i>(TF-Tansig)</i> 1-bit AHSD adder simulation and epochs 12.....	23
5.3	<i>(TF-logsig)</i> 1-bit AHSD adder simulation and epochs 100.....	24
5.4	<i>(TF-purelin)</i> 1-bit AHSD adder simulation and epochs 6.....	24
5.5	<i>(TF-Tansig)</i> 2-bit AHSD adder simulation and epochs 100.....	26
5.6	<i>(TF-logsig)</i> 2-bit AHSD adder simulation and epochs 100.....	26
5.7	<i>(TF-logsig)</i> 3-bit AHSD adder simulation and epochs 100.....	28



# CHAPTER 1

## INTRODUCTION

Addition is the most important and frequently used arithmetic operation in computer systems. Generally, a few methods can be used to speed up the addition operation. One is to explicitly shorten the carry-propagation chain by using circuit design techniques, such as detecting the completion of the carry chain as soon as possible, carry look-ahead etc. Another is to convert the operands from the binary number system to a redundant number system, e.g., the signed-digit number system [2, 3] or the residue number system, so that the addition becomes carry-free (CF) [1]. This paper aim is focus on exploring signed-digit (SD) numbers using neural network (NN).

Artificial Neural Networks (ANN's) are relatively crude electronic models based on the neural structure of the brain. Most application use feed forward ANN's and a numerous variant of classical back propagation (BP) algorithm and other training algorithms [5, 9, 15, 21]. ANN's have been used widely in many application areas in recent years. This is an important use issue because there are strong biological and engineering evidences to support that the function i.e. the information processing capability of an ANN is determined by its architecture [11, 12]. This paper introduces a NN design for addition; it is observed that fast addition can be done, at the expense of conversion between the binary number system and the redundant number system.

There are many applications for the SD number representations, most notably in computer and digital signal processing systems. Specifically, the CF adder has been investigated based on the redundant positive-digit numbers and the symmetrical radix-4 SD numbers for high-speed area-efficient multipliers. The symmetrical radix-2 SD number representation has been used in the implementation of RSA cryptographic chips, high-speed VLSI multipliers, FIR filters, IIR filters, dividers, etc. Though arithmetic operations using these number representations can be done carry free, they have common difficulty in conversion to and from the binary number representation. Hence, in the past, many researchers have proposed specific architectures for number system conversion [1].

This paper presents the *asymmetric high-radix signed-digit (AHSD)* number system using NN [4, 5]. The idea of AHSD is not new. A particular AHSD number system was introduced by others research paper “the radix- $r$  stored-borrow” number system. Researchers earlier works have focused on binary stored-borrow number systems, where  $r = 2$ . Instead of proposing a new number representation, this work purposes is to explore the inherent CF property of AHSD by using NN. The CF addition in AHSD based on NN is the basis for our high-speed addition circuits. The conversion of AHSD to and from binary will be discussed in detail. By choosing  $r = 2^m$ , where  $m$  is any positive integer, a binary number can be converted to its canonical AHSD representation in constant time [1].

This paper also presents one simple paring algorithm at the time of converting from binary to  $AHSD_{(r)}$ . Besides, the conversion from binary to AHSD has been considered the bottleneck of AHSD-based arithmetic computation based on NN, the NN design greatly improve the performance of AHSD systems. For illustration, this paper will emphasizes detail discuss on the  $AHSD_{(4)}$ , i.e., the radix-4 AHSD number system [1].

## 1.1 OBJECTIVES

The goal of this work is to design an adder to improve the fast addition of AHSD numbers system using NN. NN have simple construction in high-speed operation that performs high-speed addition in the processor, which can be realized in the signed-digit system without a delay of the carry propagation. It's purpose is to show performance of AHSD adder will be high in operation due to less digit requirement and carry free addition and the better implementation of the adder of redundant number system than radix-2 signed digital adder using NN

## 1.2 THESIS OVERVIEW

In Chapter 2, the brief history of neural networks, definition of neural network, its detailed components, how NN works are discussed in concise manner.

In Chapter 3, it introduces signed-digit number representation, defines AHSD number system and discusses conversion from binary- to- AHSD. Besides, carry-free adder based on AHSD, a general algorithm to make pairs at the time of conversion from binary to AHSD<sub>(r)</sub> and AHSD addition based on radix-5 is also discussed.

In Chapter 4, it shows proposed algorithm for the design of asymmetric high radix signed digital adder using NN.

In Chapter 5, it shows overall work procedure for the simulation and presents simulation results summary of 1, 2, 3-bit addition based on AHSD including figures and tables.

Finally, Chapter 6, 7 and 8 limitations of the proposed adder, directions for future work and conclusion of this work is discussed respectively.

## CHAPTER 2

### **2.1 THE BRIEF HISTORY OF NEURAL NETWORKS**

In the early 1940's scientists came up with the hypothesis that neurons, fundamental, active cells in all animal nervous systems might be regarded as devices for manipulating binary numbers. Thus spawning the use of computers as the traditional replicates of ANNs to be understood is that advancement has been slow. Early on it took a lot of computer power and consequently a lot of money to generate a few hundred neurons. In relation to that consider that an ant's nervous system is composed of over 20,000 neurons and furthermore a human being's nervous system is said to consist of over 100 billion neurons! To say the least replication of the human's neural networks seemed daunting.

The exact workings of the human brain are still a mystery. Yet, some aspects of this amazing processor are known. In particular, the most basic element of the human brain is a specific type of cell, which, unlike the rest of the body, doesn't appear to regenerate. Because this type of cell is the only part of the body that isn't slowly replaced, it is assumed that these cells are what provide us with our abilities to remember, think, and apply previous experiences to our every action. These cells, all 100 billion of them, are known as neurons. Each of these neurons can connect with up to 200,000 other neurons, although 1,000 to 10,000 are typical [16].

The individual neurons are complicated. They have a myriad of parts, sub-systems, and control mechanisms. They convey information via a host of electrochemical pathways. There are over one hundred different classes of neurons, depending on the classification method used. Together these neurons and their connections form a process, which is not binary, not stable, and not synchronous. In short, it is nothing like the currently available electronic computers, or even artificial neural networks. These artificial neural networks try to replicate only the most basic elements of this complicated, versatile, and powerful organism [16].

### 2.2.1 DEFINITION OF ARTIFICIAL NEURAL NETWORKS

There is several views about artificial neural networks, but all of them emphasis on same thing. Here two views are mentioned:

A neural network is a powerful data-modeling tool that is able to capture and represent complex input/output relationships. The motivation for the development of neural network technology stemmed from the desire to develop an artificial system that could perform "intelligent" tasks similar to those performed by the human brain. Neural networks resemble the human brain in the following two ways:

- I. A neural network acquires knowledge through learning.
- II. A neural network's knowledge is stored within inter-neuron connection strengths known as synaptic weights [22].

Neural Networks are a different paradigm for computing: Von Neumann machines are based on the processing/memory abstraction of human information processing. It is based on the parallel architecture of animal brains, a form of multiprocessor computer system, with simple processing elements, a high degree of interconnection, simple scalar messages, and adaptive interaction between elements [9].

## 2.2.2 DETAILED DESCRIPTION OF NEURAL NETWORK COMPONENTS [17, 18]

This section describes the seven major components, which make up an artificial neuron. These components are valid whether the neuron is used for input, output, or is in one of the hidden layers.

### **Component 1. Weighting Factors:**

A neuron usually receives many simultaneous inputs. Each input has its own relative weight, which gives the input the impact that it needs on the processing element's summation function. These weights perform the same type of function, as do the varying synaptic strengths of biological neurons. In both cases, some inputs are made more important than others so that they have a greater effect on the processing element as they combine to produce a neural response.

### **Component 2. Summation Function:**

The first step in a processing element's operation is to compute the weighted sum of all of the inputs. Mathematically, the inputs and the corresponding weights are vectors which can be represented as  $(i_1, i_2 \dots i_n)$  and  $(w_1, w_2 \dots w_n)$ . The total input signal is the dot, or inner, product of these two vectors. This simplistic summation function is found by multiplying each component of the  $i$  vector by the corresponding component of the  $w$  vector and then adding up all the products.  $input_1 = i_1 * w_1$ ,  $input_2 = i_2 * w_2$ , etc., are added as  $input_1 + input_2 + \dots + input_n$ . The result is a single number, not a multi-element vector.

### **Component 3. Transfer Function:**

The result of the summation function, almost always the weighted sum, is transformed to a working output through an algorithmic process known as the transfer function. In the transfer function the summation total can be compared with some threshold to determine the neural output. If the sum is greater than the threshold value, the processing element

generates a signal. If the sum of the input and weight products is less than the threshold, no signal (or some inhibitory signal) is generated. Both types of response are significant. The threshold, or transfer function, is generally non-linear. Linear (straight-line) functions are limited because the output is simply proportional to the input.

#### **Component 4. Scaling and Limiting:**

After the processing element's transfer function, the result can pass through additional processes, which scale and limit. This scaling simply multiplies a scale factor times the transfer value, and then adds an offset. Limiting is the mechanism, which insures that the scaled result does not exceed an upper, or lower bound. This limiting is in addition to the hard limits that the original transfer function may have performed. This type of scaling and limiting is mainly used in topologies to test biological neuron models, such as James Anderson's brain-state-in-the -box.

#### **Component 5 Output Function (Competition):**

Each processing element is allowed one output signal, which it may output to hundreds of other neurons. This is just like the biological neuron, where there are many inputs and only one output action. Normally, the output is directly equivalent to the transfer function's result. Some network topologies, however, modify the transfer result to incorporate competition among neighboring processing elements. Neurons are allowed to compete with each other, inhibiting processing elements unless they have great strength. Competition can occur at one or both of two levels. First, competition determines which artificial neuron will be active, or provides an output. Second, competitive inputs help determine which processing element will participate in the learning or adaptation process.

#### **Component 6. Error Function and Back-Propagated Value:**

The back propagation algorithm consists of two phases: the forward phase where the activations are propagated from the input to the output layer, and the backward phase, where the error between the observed actual and the requested nominal value in the output layer is propagated backwards in order to modify the weights and bias values.

In most learning networks, the difference between the current output and the desired output is calculated. This raw error is then transformed by the error function to match particular network architecture. The most basic architectures use this error directly, but some square the error while retaining its sign, some cube the error, and other paradigms modify the raw error to fit their specific purposes. The artificial neuron's error is then typically propagated into the learning function of another processing element. This error term is sometimes called the current error.

The current error is typically propagated backwards to a previous layer. Yet, this back-propagated value can be either the current error, the current error scaled in some manner (often by the derivative of the transfer function), or some other desired output depending on the network type. Normally, this back-propagated value, after being scaled by the learning function, is multiplied against each of the incoming connection weights to modify them before the next learning cycle.

### **Component 7. Learning Function:**

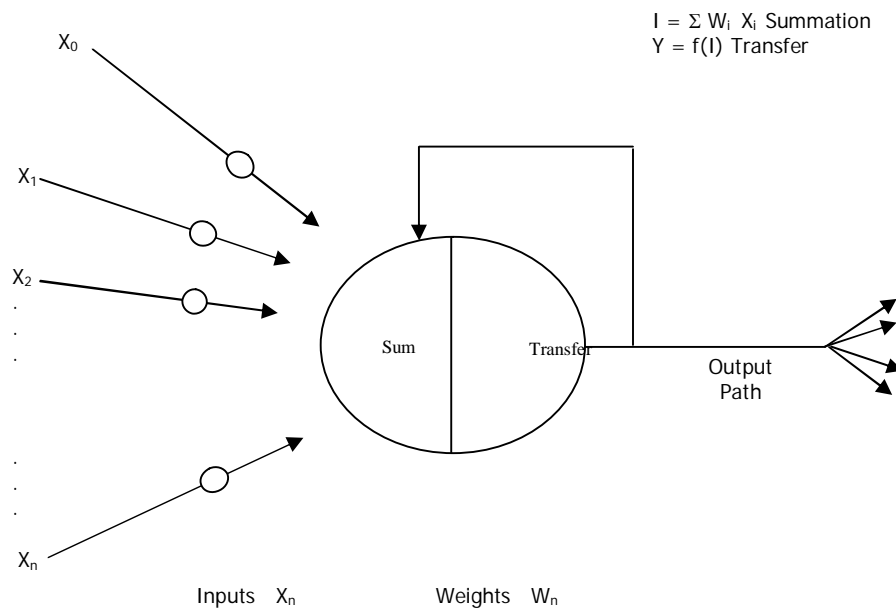
The purpose of the learning function is to modify the variable connection weights on the inputs of each processing element according to some neural based algorithm. This process of changing the weights of the input connections to achieve some desired result could also be called the adoptions function, as well as the learning mode. There are two types of learning: supervised and unsupervised. Supervised learning requires a teacher. The teacher may be a training set of data or an observer who grades the performance of the network results. Either way, having a teacher is learning by reinforcement. When there is no external teacher, the system must organize itself by some internal criteria designed into the network. This is learning by doing.



### 2.2.3 ARTIFICIAL NEURONS AND HOW THEY WORK [13]

The fundamental processing element of a neural network is a neuron. Basically, a biological neuron receives inputs from other sources, combines them in some way, performs a generally nonlinear operation on the result, and then outputs the final result. All natural neurons have the same four basic components. These components are known by their biological names - dendrites, soma, axon, and synapses. Dendrites are hair-like extensions of the soma, which act like input channels. These input channels receive their input through the synapses of other neurons. The soma then processes these incoming signals over time. The soma then turns that processed value into an output, which is sent out to other neurons through the axon and the synapses.

To do this, the basic unit of neural networks, the artificial neurons, simulates the four basic functions of natural neurons. Figure 2.1 shows a fundamental representation of an artificial neuron.



**Figure 2.1:** A basic artificial neuron

In Fig. 2.1 various inputs to the network are represented by the mathematical symbol,  $x(n)$ . Each of these inputs is multiplied by a connection weight. These weights are represented by  $w(n)$ . In the simplest case, these products are simply summed, fed through a transfer function to generate a result, and then output. This electronic implementation is still possible with other network structures, which utilize different summing functions as well as different transfer functions. These networks may utilize the binary properties of ORing and ANDing of inputs. These functions, and many others, can be built into the summation and transfer functions of a network. Other applications might simply sum and compare to a threshold, thereby producing one of two possible outputs, a zero or a one. Other functions scale the outputs to match the application, such as the values minus one and one. Some functions even integrate the input data over time, creating time-dependent networks.

## Chapter 3

### 3.1 SIGNED-DIGIT NUMBER REPRESENTATION [1, 7]

For a given radix  $r$ , each digit  $x_i$  in an SD number system is typically in the range

$$-a \leq x_i \leq +a \quad \text{where } \lceil r-1/2 \rceil \leq a \leq r-1 \quad (1)$$

In such a system, a “carry-free” addition can be performed, where the term “carry-free” in this context means that the carry propagation is limited to a single digit position. In other words, the carry propagation length is fixed irrespective of the word length. The addition consists of two steps. In the first step, an intermediate sum  $s_i$  and a carry  $c_i$  are generated, based on the operand digits  $x_i$  and  $y_i$  at each digit position  $i$ . This is done in parallel for all digit positions. In the second step, the summation  $z_i = s_i + c_{i-1}$  is carried out to produce the final sum digit  $z_i$ . The important point is that it is always possible to select the intermediate sum  $s_i$  and carry  $c_{i-1}$  such that the summation in the second step does not generate a carry. Hence, the second step can also be executed in parallel for all the digit positions, yielding a fixed addition time, independent of the word length.

If the selected value of  $a$  in equation (1) satisfies the condition

$$\lceil r+1/2 \rceil \leq a \leq r-1 \quad (2)$$

then the intermediate sum  $s_i$  and carry  $c_i$  depend only on the input operands in digit position  $i$ , i.e., on  $x_i$  and  $y_i$ . The rules for selecting the intermediate sum and carry are well known in this case. The interim sum is  $s_i = x_i + y_i - r c_i$  where

$$\left\{ \begin{array}{l} 1 \\ \end{array} \right. \quad \text{if } (x_i + y_i) \geq a$$

$$c_i = \begin{cases} -1 & \text{if } (x_i + y_i) \leq -a \\ 0 & \text{if } |x_i + y_i| < a \end{cases} \quad (3)$$

Note that for the most commonly used binary number system (radix  $r=2$ ), condition (2) cannot be satisfied. Carry-free addition according to the rules in (3) therefore cannot be performed with binary operands. However, by examining the input operands in position  $i-1$  together with the operands in digit position  $i$ , it is possible to select a carry  $c_i$  and an interim sum  $s_i$  such that the final summation  $z_i = s_i + c_{i-1}$  never generates a carry. In other words, if one allows the carry  $c_i$  and interim sum  $s_i$  to depend on two digit positions, viz.,  $i$  and  $i-1$ , then condition (2) can be relaxed and  $a = \lfloor r/2 \rfloor$  can also be used to accomplish carry-free addition as explained next.

Let  $x_i, y_i, x_{i-1}$  and  $y_{i-1}$  be the input digits at the  $i$ th and  $(i-1)$ th positions, respectively, and assume that the radix under consideration is  $r = 2a$ . This includes the case where  $r = 2$  and  $a = 1$ . Let  $\theta_i = x_i + y_i$  and  $\theta_{i-1} = x_{i-1} + y_{i-1}$  denote the sums of the input digits at the two positions, respectively. Then, the rules for generating the intermediate sum  $s_i$  and carry  $c_i$  are summarized in Table 3.1. In the table, the symbol “X” indicates a “don’t care”, i.e., the value of  $\theta_{i-1}$  does not matter.

$\theta_i$	$\theta_i \leq -2a$	$-2a < \theta_i < -a$	$\theta_i = -a$		$-a < \theta_i < 0$	$\theta_i = 0$		$0 < \theta_i < a$	$\theta_i \geq 2a$
$\theta_{i-1}$	X	X	$\theta_{i-1} \leq -a$	$\theta_{i-1} > -a$	X	$\theta_{i-1} < a$	$\theta_{i-1} \geq a$	X	X
$c_i$	-1	-1	-1	0	0	0	1	1	1
$s_i$	0	$\theta_i + r$	a	-a	$\theta_i$	a	-a	$\theta_i - r$	0

**Table 3.1:** Rule for selecting carry  $c_i$  and intermediate sum  $s_i$  based on  $\theta_i = x_i + y_i$  and  $\theta_{i-1} = x_{i-1} + y_{i-1}$  for radix  $r=2a$

Classes of signed-digit number representations of symmetrical type, called the ordinary signed-digit (OSD) number system. These number systems are defined for any radix  $r \geq 3$  and for the digit set  $\{-\infty, \dots, -1, 0, 1, \dots, \infty\}$ , where  $\infty$  is an integer such that  $r/2 < \infty < r$ . In an OSD number system, the *redundancy index*, defined as  $\rho = 2\alpha - r + 1$ , ranges from the minimal redundancy  $\lfloor r/2 \rfloor + 1$  to the maximal redundancy  $r - 1$ . The most important

contribution of OSD is to explore the possibility of performing carry-free addition and borrow-free subtraction for fast parallel arithmetic, if enough redundancy is used.

The OSD number system was later extended to the *generalized signed-digit* (GSD) number system. The GSD number system for radix  $r > 1$  has the digit set  $\{-\alpha, \dots, -1, 0, 1, \dots, \beta\}$ , where  $\alpha \geq 0$  and  $\beta \geq 0$ . The redundancy is  $\rho = \alpha + \beta + 1 - r$ . So far, the most important contribution of the works on GSD includes unifying the redundant number representation and sorting the CF addition schemes for the GSD number system according to the radix  $r$  and redundancy index  $\rho$ . However, ideal single-stage CF addition has not been achieved, though two-stage CF addition has been shown to be doable for any GSD system with  $r > 2$  and  $\rho > 2$ , or with  $r > 2$  and  $\rho = 2$  provided that  $\alpha \neq 1$  and  $\beta \neq 1$ . For any GSD system with  $r = 2$  and  $\rho = 1$ , or  $\rho = 2$  and  $\alpha$  or  $\beta$  equals 1, the limited-carry addition must be used.

### 3.2 THE AHSD NUMBER SYSTEM [1]

The radix- $r$  *asymmetric high-radix signed-digit* (AHSD) number system, denoted  $AHSD(r)$ , is a positional weighted number system with the digit set  $S_r = \{-1, 0, \dots, r-1\}$ , where  $r > 1$ . The AHSD number system is a minimally redundant system with only one redundant digit in the digit set. We will explore the inherent carry-free property in AHSD and develop systematic approaches for conversion of AHSD numbers from binary ones. An  $n$ -digit number  $X$  in  $AHSD(r)$  is represented as

$$X = (x_{n-1}, x_{n-2}, \dots, x_0)_r \tag{4}$$

Where  $x_i \in S_r$  for  $i = 0, 1, \dots, n-1$ , and  $S_r = \{-1, 0, 1, \dots, r-1\}$  is the digit set of  $AHSD(r)$ . The value of  $X$  can be represented as

$$X = \sum_{i=0}^{n-1} x_i r^i \tag{5}$$

Clearly, the range of  $X$  is  $\left[\frac{1-r^n}{r-1}, r^n-1\right]$

#### 3.2.1 BINARY TO AHSD CONVERSION [1]

Since the binary number system is the most widely used, the conversion between the AHSD and binary number systems has to be considered. Although the radix  $r$  may be any positive integer, simple binary-to-AHSD conversion can be achieved if  $r = 2^m$  for any positive integer  $m$ . The reason for such simple conversion will be explained later. It has

been assumed  $r = 2^m$  in what follows, unless otherwise specified. Note that there may be more than one  $AHSD_{(r)}$  number representation for a binary number. For instance, the binary number  $(0, 1, 1, 0, 0)_2$  can be converted to two different  $AHSD_{(4)}$  numbers, i.e.,  $(1, -1, 0)_4$  and  $(0, 3, 0)_4$ . Hence, the binary-to- $AHSD(r)$  conversion, being a one-to-many mapping, may be performed by several different methods. The main purpose is to find an efficient and systematic conversion method that takes advantage of the carry-free property by using NN.

### 3.2.2 AN ALGORITHM TO MAKE PAIRS FROM BINARY TO AHSD CONVERSION

Here we follow a general algorithm to make pairs to convert binary to-  $AHSD_{(r)}$ .

**Step 1.** Suppose given binary #bits= $n$

**Step 2.** If  $radix = 2^m$ ; where  $m$  is any positive integer

then  $2^p < m < 2^{p+1}$  Where  $p=1,2,3,\dots$

**Step 3.** # Zero (0) will be padded in front of binary bits pattern  $2^{p+1}-n$

**Step 4.** Divide the array by  $m$

**Step 5.** If each sub array is  $=m$

Then stop

**Step 6.** Else, divide each sub array by  $m$

**Proof:**

Recurrence relation for the conversion from binary- to-  $AHSD$  numbers system is:

$$T(n) = \begin{cases} c & \text{if } n=m \text{ where } c \text{ is a constant \& } m > 2 \\ m T(n/m) & \text{if } n > m \end{cases} \quad (6)$$

$$T(n) = m T(n/m) \dots (1)$$

Let,  $n = n/m$

Replace it into equation (1)

$$T(n/2) = m T(n/m^2)$$

$$\text{So, } T(n) = m^2 T(n/m^2)$$

...

$$\dots = m^k T(n/m^k) \text{ where } k=1,2,3..$$

Assume  $n = m^k$

$$T(n) = m^k T(m^k / m^k)$$

$$T(n) = m^k T(1)$$

$$T(n) = m^k$$

$$\text{So, } n = m^k$$

Now from above induction it can be said that

$$\text{Log}_m n = \text{Log}_m m^k = k$$

Complexity of the algorithm to make pairs to convert binary –to-AHSD <sub>(r)</sub>:  $O(\log_m n)$

### 3.2.3 ADDITION OF AHSD <sub>(4)</sub> NUMBER SYSTEM [1]

Here the addition process will be shown for AHSD number system. This addition process can be for 1-bit to n-bit adder design without considering the carry propagation and its delay as well.

**Example:** Here two 4-bit AHSD <sub>(4)</sub> numbers are added.

$$X \rightarrow 11\ 11\ 11\ 11$$

$$Y \rightarrow 11\ 00\ 01\ 10$$

$$X \text{ (AHSD (4))} \rightarrow (3\ 3\ 3\ 3)_4$$

$$Y \text{ (AHSD (4))} \rightarrow (3\ 0\ 1\ 0)_4$$

$$\begin{array}{r} \dots\dots\dots \\ X+Y(Z): \quad \quad 6\ 3\ 4\ 3 \\ \text{Transfer digit: } 1\ 1\ 1\ 1\ 0\ c \\ \text{Interim sum: } \quad 2\ -1\ 0\ 1\ \mu \\ \dots\dots\dots \end{array}$$

$$\text{Final sum S: } (1\ 3\ 0\ 1\ 1)_4$$

$$\text{Result} = (01\ 11\ 00\ 01\ 01)_2 = (453)_{10}$$

The final result is in binary format. The given example illustrates the addition process without carry propagation.

### 3.2.4 AHSD ADDITION FOR RADIX-5

The asymmetric high radix number system considering radix-5 is not well suited for addition. The first conversion from binary–to-AHSD requires pairing bits by 3-bit binary. It will convey the values from 0 to 7 in decimal. But co-efficient of radix-5 will be from 0

to 4 values. Here the values from 0 to 7 in decimal as undetermined values. Hence the addition process will not be possible as well. Thus radix-4 AHSD is best suited for addition that is carry free and fast.

**Example:** Here two 4-bit AHSD<sub>(5)</sub> numbers are added.

$$X \rightarrow 010\ 001\ 011\ 101 = (1117)_{10}$$

$$Y \rightarrow 101\ 010\ 101\ 011 = (2731)_{10}$$

.....

$$X \text{ (AHSD}_{(5)}) \rightarrow (2\ 1\ 3\ 5)_5 = 2*125 + 1*25 + 3*5 + 5*1 = 315$$

$$Y \text{ (AHSD}_{(5)}) \rightarrow (5\ 2\ 5\ 3)_5 = 5*125 + 2*25 + 5*5 + 3*1 = 703$$

.....

$$X+Y(Z): 7\ 3\ 8\ 8$$

.....

$$\text{Transfer digit: } 1\ 1\ 1\ 1\ 0\ C$$

$$\text{Interim sum: } \quad 2\ -2\ 3\ 3\ \mu$$

.....

$$\text{Result} = (1\ 3\ -1\ 4\ 3)_5 = (001\ 011\ -0\ -0\ -1\ 100\ 011)_2 = 5603$$

The final result is in signed-binary format. The given example illustrates the addition process without carry propagation but it produces an incorrect result for the addition.

### 3.2.5 AHSD<sub>(4)</sub> CF ADDER [1]

By using the AHSD representation, fast carry-free (CF) addition can be done. It is well known that the addition and subtraction of signed numbers, using the  $r$ 's-complement representation, can be implemented by the unsigned addition with sign extension and overflow detection. However, even though OSD and GSD provide various CF additions, there can be insufficient redundancy to perform a CF addition for two AHSD<sub>(r)</sub> numbers when  $r > 2$ . In what follows, this thesis work focus only on certain unsigned additions. Suppose addition of numbers  $X = (x_{q-1}, \dots, x_1, x_0)_r$  and  $Y = (y_{q-1}, \dots, y_1, y_0)_r$ , which are two unsigned AHSD<sub>(r)</sub> numbers. Two CF algorithms that realize the addition  $S = X + Y$ .

The radix  $r = 2^m$  should be used in practical AHSD-based implementations to achieve the simplest conversion from binary. The procedure of simulation uses the AHSD<sub>(4)</sub> number

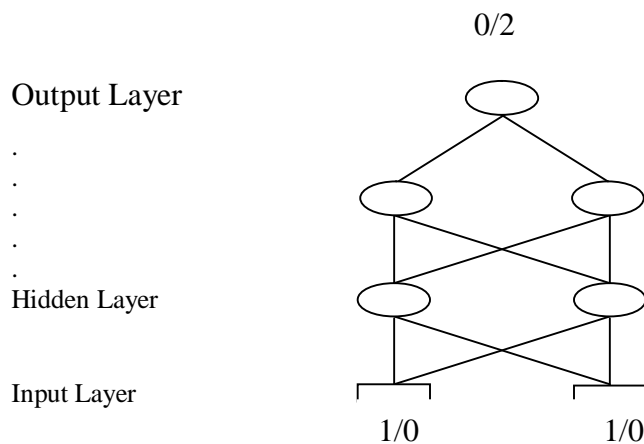


system as an example to show the proposed CF addition by using NN. . Apparently the longest carry propagation chain occurs in this case if an ordinary ripple-carry adder is used. By using the proposed addition algorithm based on NN, it can generate the sum without any carry propagation in the CF adders. An entire AHSD <sub>(4)</sub> CF adder can be implemented using NN.

## Chapter 4

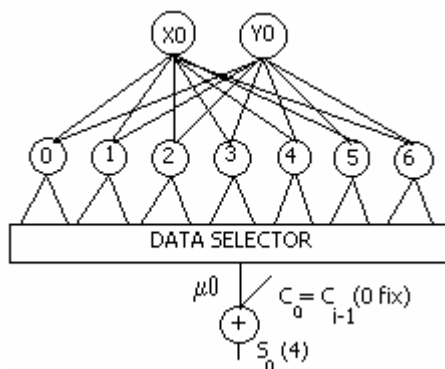
### 4.1 PROPOSED DESIGN OF ADDER USING NEURAL NETWORK

In the following a neural network prototype for AHSD <sub>(4)</sub> number system addition is shown in figure 4.1.



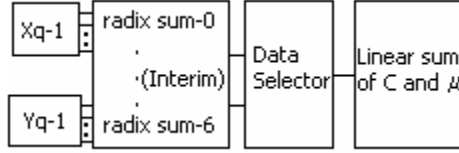
**Figure 4.1:** Neural Network prototype for AHSD number system addition.

The neuron will work on the basis of feed-forward network with parallel processing. This technique can be viewed as doing addition in adder in parallel. The concept of parallel addition using neural network can be shown as the block diagram below.



**Figure 4.2:** Trivial 1-bit AHSD radix-4 adder using NN.

The figure 4.2 shows the atomic view of the adder using Neural Network. When it is made more generalized then the figure will be just like the following.



**Figure 4.3:** N-bit adder generalization.

The N-bit adder generalization is shown in figure 4.3.

**Lemma:** The total number of neurons for generating interim sum and carry of a radix-n asymmetric q-bit adder design is  $q \times 2(n-1)$ .

**Proof:** As n is the radix, so each bit will contain the value of n-1. So the interim sum will be in the range of 0 to  $2(n-1)$ . The total number of neurons for 1-bit adder design will be  $2(n-1)$ . For the design of a q-bit adder, the total number of neurons will be  $q \times 2(n-1)$ .

Here we proposed an algorithm for n-bit adder from binary to AHSD<sub>(4)</sub> using NN.

### ALGORITHM

**Step 1:** Create  $4^n$  input vectors ( $2n$  elements each) that represent all possible input combinations. For example, for 1-bit addition, there will be 4 input vectors (2 elements each): {0, 0}, {0, 1}, {1, 0}, {1, 1}.

**Step 2:** Create  $4^n$  output vectors (ceiling of  $\lceil \frac{n}{2} + 1 \rceil$  elements) that represent the corresponding target combinations. For example, for 1-bit addition, there will be 4 output vectors (2 elements each): {0, 0}, {0, 1}, {0, 1}, {0, 2}.

**Step 3:** Create a feed-forward back propagation neural network

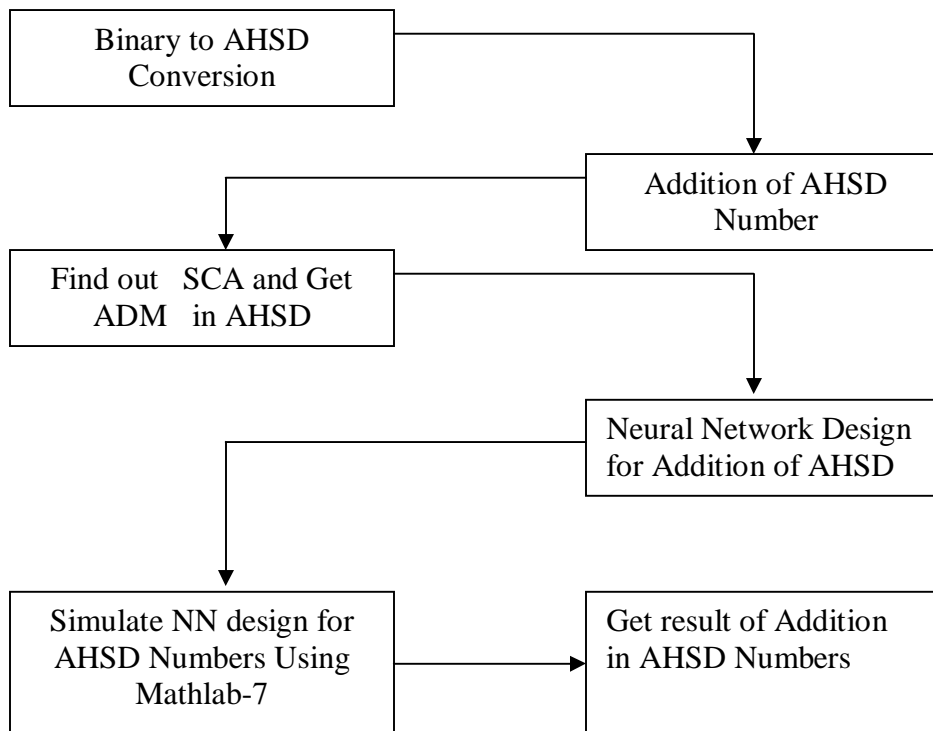
**Step 4:** Train the neural network for the input and output vectors of steps 1 and step 2.

**Step 5:** Simulate the neural network with the input vectors of step 1, getting the sum in AHSD<sub>(4)</sub> as the target.

## Chapter 5

### 5.1 EXPERIMENTAL RESULTS AND DISCUSSION

At first conversion from binary- to- AHSD numbers is done for the manual simulation, then to get self-consistent adjustment (SCA) and adjacent digit modification (ADM) two AHSD numbers is added. Then arithmetic addition of SCA and ADM is done. Now NN design for addition of AHSD is considered. Finally by using Mathlab-7 simulation tool simulation is performing for NN design and getting addition result of AHSD numbers. In the figure 5.1 this workflow procedure is shown.



**Figure5.1:** The Workflow of simulation procedure for AHSD numbers

The proposed approach has been implemented by Matlab-7 simulation tools in a computer with Pentium IV processor. During simulate the proposed NN design 1, 2, 3-bits AHSD numbers are used. Multi-feed forward NN with back propagation training algorithm is used to train the network and to simulate the input that produces results. Satisfactory results is found in the addition of 1, 2, 3-bits AHSD numbers. The table 5.1 summarized numbers of layer and neuron is used for simulation .By see figure5.2, figure5.3, figure5.4, figure5.5 figure5.6 and figure5.7; the scenarios of NN training with epochs and performance can be realized .In the following experimental and simulated results of 1, 2, 3-bits AHSD numbers addition is discussed in detail.

Bit	Layer	Neuron Numbers	Transfer Function
bit-1	2	I=2;O=2	Tansig / Logsig/ Purelin
bit-2	2	I=4;O=2	Tansig/ Logsig
bit-3	2	I=6;O=3	Logsig

**Table 5.1:** NN experimental results for AHSD adder design.

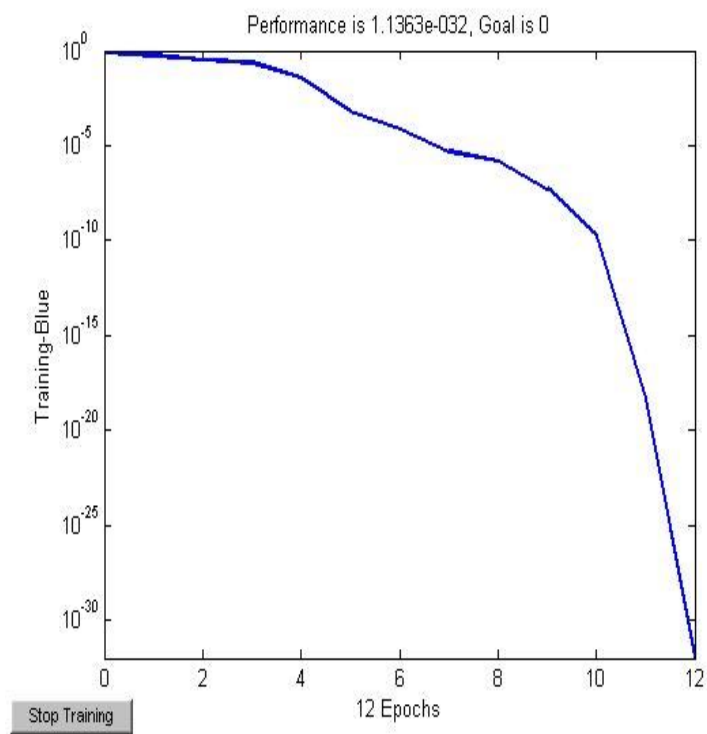
### 5.1.1 DISCUSSION ON SIMULATED RESULTS OF 1-BIT AHSD NUMBERS

Basically, in 1-bit AHSD numbers addition two input and output layers including four (2+2) neuron are used in NN design. Four input and output combination is taking in operation. Here differentiable transfers functions such as *tansig*, *logsig*, or *purelin* are used and got satisfactory result each and ever function. In the below, the summary of experimental result for three transfer function is shown in table 5.1.Besides, figure5.2, figure5.3, figure5.4 show the number of epochs to train the NN with performance.

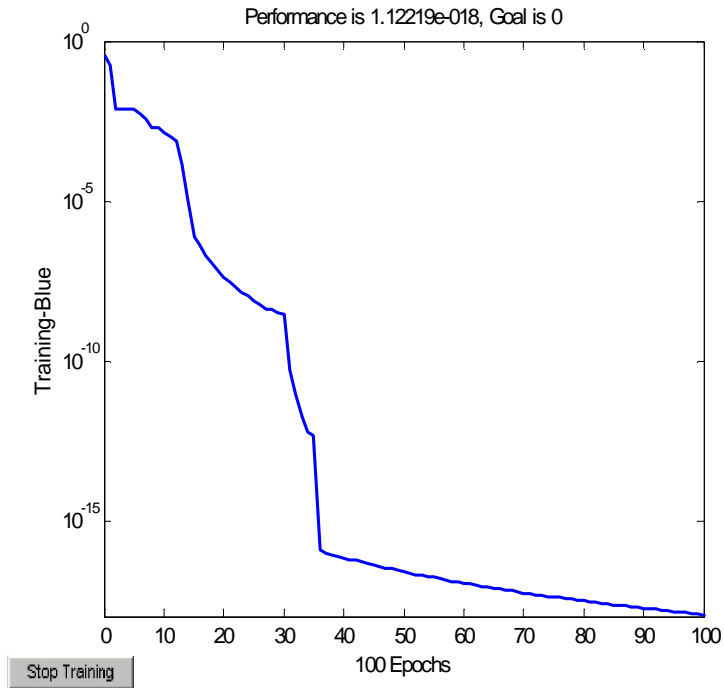
In appendix simulation is included. Even though high epochs to train accurate result it can be said that from table5.1 the performance of *logsig* transfer function is the best for the addition of 1-bit AHSD numbers .In the below performance and epochs for training NN can be shown in equalities.

Performance: Tansig < Purelin < Logsig ..... (7)

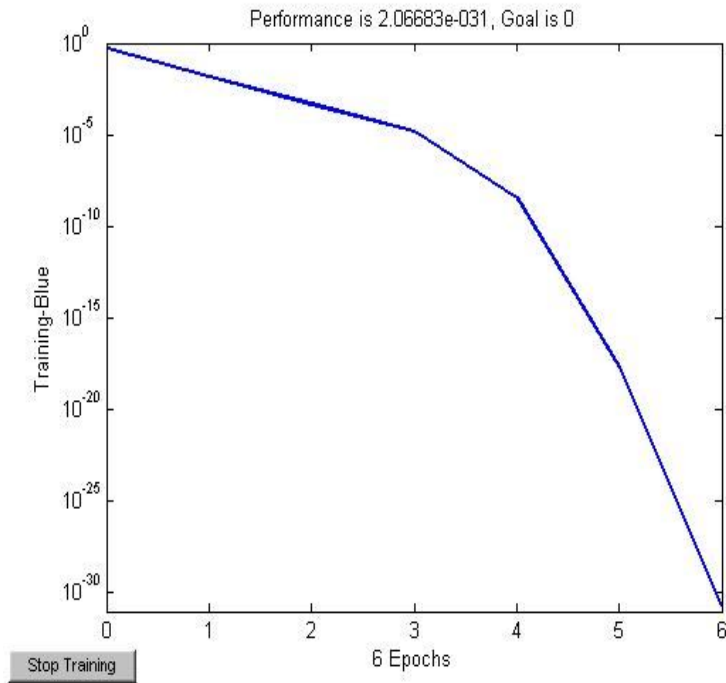
Epochs : Logsig < Tansig < Purelin .....(8)



**Figure 5.2:** (*TF-Tansig*) 1-bit AHSD adder simulation and epochs 12.



**Figure5.3:** (*TF-Logsig*) 1-bit AHSD adder simulation and epochs 100.

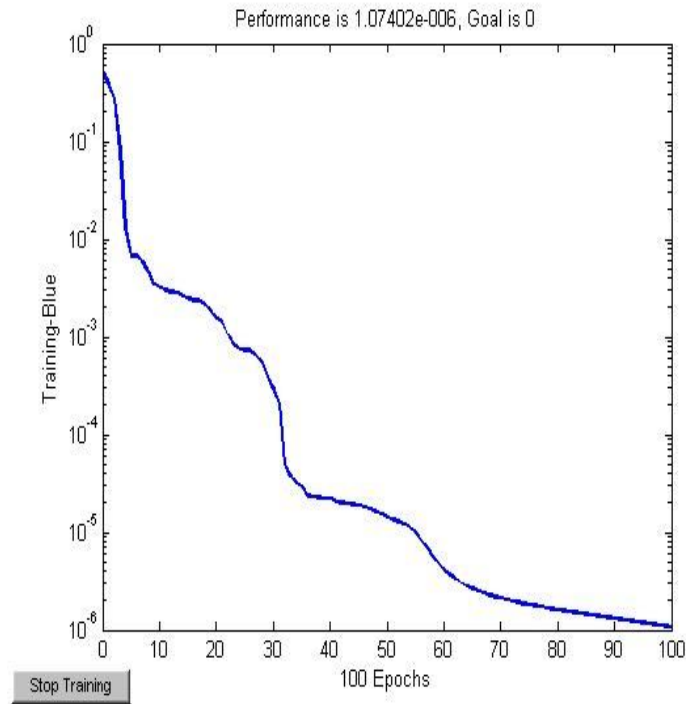


**Figure5.4:** (*TF-Purelin*) 1-bit AHSD adder simulation and epochs 6.

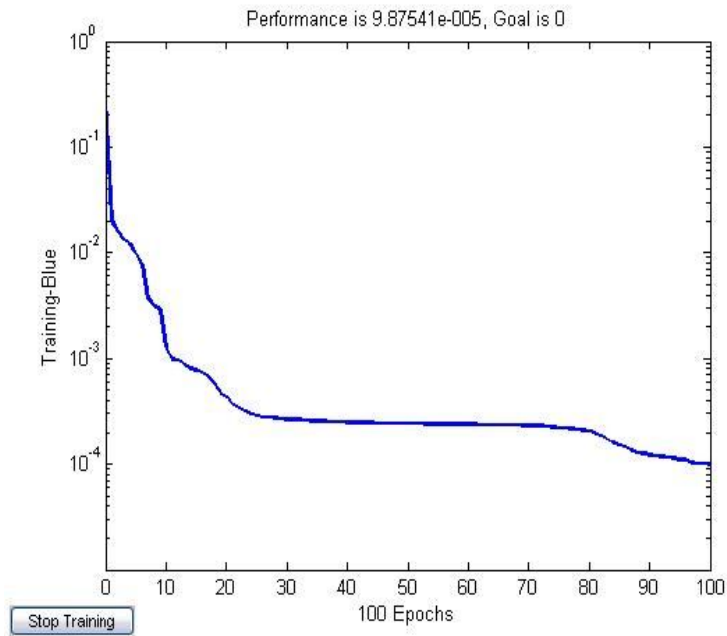
<b>T.F/ TRAINLIM</b>	<b>TANSIG</b>	<b>LOGSIG</b>	<b>PURELIN</b>
Epochs	12/100	100/100	6/100
MSE	1.1363e-032/0	1.12219e-018/0	2.06683e-031/0
Gradient	6.27215e-016/1e-010	5.27372e-010/1e-010	5.49675e-016/1e-010
Performance	1.1363e-032	1.12219e-018	2.06683e-031

**Table 5.2:** Summary of 1-bit AHSD number's simulation with different transfer functions using Matlab-7.

In 2-bit AHSD numbers addition two input and output layers including seven (4+2) neuron are used in NN design. During simulation  $4^2$  input and output combination is taking in operation. Except one *TF purelin*, satisfactory result is achieved by *tansig* & *logsig TF*. Table5.2 shows summary of experimental result of two TF. Besides, performance (with number of epochs to train the NN perform by each *TF*) is shown by in figure5 and figure5.6. It can be said that from the table5.2, the performance of *logsig TF* is the best for the addition of 2-bit AHSD numbers and simulation part is also illustrated in appendix.



**Figure5.5:** (*TF-Tansig*) 2-bit AHSD adder for simulation and 100 epochs.



**Figure5.6:** (*TF-Logsig*) 2-bit AHSD adder for simulation and 100 epochs.

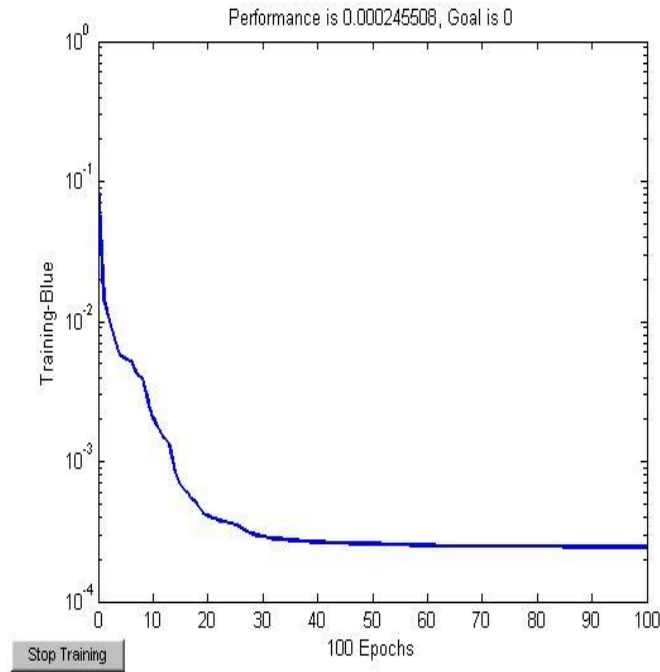


<b>T.F/TRAINLIM</b>	<b>TANSIG</b>	<b>LOGSIG</b>
Epochs	100/100	100/100
MSE	9.91532e-010/0	9.87541e-005/0
Gradient	4.65299e-005/1e-010	0.000500187/1e-010
Performance	9.91523e-010	9.87541e-005

**Table 5.3:** Summary of 2-bit AHSD number's simulation with different transfer functions using matlab-7.

### 5.1.3 DISCUSSION ON SIMULATED RESULT OF 3-BIT AHSD NUMBERS

In 3-bit AHSD numbers addition two input, two output layers including 9 (6+3) neuron are used in NN design.  $4^3$  input and output, combination is taking in simulation. Only *logsig TF* satisfies the result. The summary of experimental result for *logsig TF* is shown in table5.3. Besides, figure5.7 shows the number of epochs to train the NN and performance and appendix shows simulation part of 3-bit AHSD numbers.



**Figure5.7:** (*TF-Logsig*) 3-bit adder simulation and epochs 100.

<b>TRAINLM /T.F</b>	<b>Epoch</b>	<b>MSE</b>	<b>Gradient</b>	<b>Performance</b>
Logsig	100/100	0.000245508/0	0.00118849/1e-010	.000245508

**Table 5.4:** Summary of 1-bit AHSD number's simulation with logsig transfer function using matlab-7.

## 5.2 OVERALL DISCUSSION ON SIMULATED RESULT OF AHSD NUMBERS

It can be summarized from table5.2, table5.3and table5.4 satisfactory performance is found using *TF logsig* in the design of AHSD radix-4 using NN. The performance shows the better implementation of the adder of redundant number system than radix-2 signed digital adder using NN [5].

## CHAPTER 6

### LIMITATION OF THE PROPOSED ADDER

Due to limitation of time, unavailable resources, equipments and technologies some features can't be implemented and especially hardware implementation is not implemented because of duration of thesis work. A complete investigation on the existing process and comparison with the other adder finds out the following limitations.

**Infeasible to Implementation:** It has been investigated during simulation on radix-5/6/7 for the addition of AHSD numbers system. To represent decimal value from 5 to 7 three binary bits are required that makes a pair at the time of conversion from binary to AHSD numbers system. These 3 bits represents maximum seven and minimum zero numerical value in decimal those are used as co-efficient for radix-5/6/7. Even though presentation of decimal value 5, 6, 7 required 3 binary bits but a problem will exist that is co-efficient of radix-5/6/7 can't be equal or more than that radix.

**Difficulty on Circuit or Architecture:** Induction shows that radix-8 is possible with less difficulty but implementation of the circuit or architecture for the proposed digital adder will become infeasible as radix goes high. Moreover, radix-4 is best suited for addition of AHSD numbers system

## **CHAPTER 7**

### **FUTURE DEVELOPMENT**

For the future implementation this thesis work target is to overcome the limitations of the existing adder. Then focus is given on adding new features that will ensure the service of the adder more sophisticated and user friendly. This will enhance and improve the usability and reliability of the adder to a higher degree. Keeping this in mind some directions for future work is give below:

- I. In future proposed NN design can be realized by using hardware.
- II. It can be realized current mode CMOS or voltage mode CMOS.
- III. Simulation for hardware can be done using SPICE simulation tool.

## CHAPTER 8

### CONCLUSION

In this paper one CF digital adder for AHSD<sub>(4)</sub> number system based on NN has been proposed. Additionally, if  $r = 2^m$  for any positive integer  $m$ , the interface between AHSD and the binary number system can be realized and it can be easily implement able. To make pairs at the time of conversion from binary to-AHSD, an algorithm has also been proposed. Besides, NN design with Matlab-7 simulation tool is used to achieve high speed for addition on AHSD. Even though hardware implementation is not done here but realization shows, the proposed NN design can be flexible. Since both the binary-to-AHSD converter and CF adder operate in a constant time, we can conclude that the AHSD-to-binary converter dominates the performance of the entire AHSD based on NN design system. The time complexity of the entire AHSD CF adder is  $O(\log_m n)$ . Finally, it is appealed that additional studies is necessary including not only the implementation of CF adder with different design skills, but also the applications of various CF adders to design fast arithmetic. Hopefully, in future this paper can be extended for the digital adder of AHSD number system based on NN and which can be realized by using hardware, using current mode CMOS or voltage mode CMOS and using SPICE simulation tool for hardware realization.

## REFERENCES

- [1] S.H. Sheih, and C.W. Wu, "Asymmetric high-radix signed-digit number systems for carry-free addition," *Journal of information science and engineering* 19, 2003, pp.1015-1039.
- [2] B. Parhami, "Generalized signed-digit number systems: a unifying framework for redundant number representations," *IEEE Transactions on Computers*, vol. 39, 1990, pp.89-98.
- [3] S.H. Sheih, and C.W. Wu, "Carry-free adder design using asymmetric high-radix signed-digit number system," in the *Proceedings of 11th VLSI Design/CAD Symposium*, 2000, pp. 183-186.
- [4] M. Sakamoto, D. Hamano and M. Morisue, "A study of a radix-2 signed-digit al fuzzy processor using the logic oriented neural networks," *IEEE International Systems Conference Proceedings*, 1999, pp. 304- 308.
- [5] T. Kamio, H. Fujisaka, and M. Morisue, "Back propagation algorithm for logic oriented neural networks with quantized weights and multilevel threshold neurons," *IEICE Trans. Fundamentals*, vol. E84-A, no.3, 2001.
- [6] A. Moushizuki, and T. Hanyu, "Low-power multiple-valued current-mode logic using substrate bias control," *IEICE Trans. Electron.*, vol. E87-C, no. 4, pp. 582-588, 2004 .
- [7] D. Phatak, "Hybrid signed-digit number systems: a unified framework for redundant number representations with Bounded carry propagation chains," *IEEE Transaction on Computers*. vol. 43, no. 8, pp. 880-891, 2004.
- [8] [http://engr.smu.edu/~mitch/ftp\\_dir/pubs/cmpeleng01.ps](http://engr.smu.edu/~mitch/ftp_dir/pubs/cmpeleng01.ps).
- [9] <http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html>

[10] [http://iee.uow.edu.au/~daniel/software/libneural/BPN\\_tutorial/BPN\\_English/BPN\\_English/node7.html](http://iee.uow.edu.au/~daniel/software/libneural/BPN_tutorial/BPN_English/BPN_English/node7.html)

[11] <http://www.dacs.dtic.mil/techs/neural/neural1.html>

[12] <http://www.dacs.dtic.mil/techs/neural/neural2.html>

[13] <http://www.dacs.dtic.mil/techs/neural/neural2.html#RTFToC5>

[14] <http://www.dacs.dtic.mil/techs/neural/neural2.html#RTFToC6>

[15] <http://www.dacs.dtic.mil/techs/neural/neural3.html>

[16] <http://www.dacs.dtic.mil/techs/neural/neural4.html>

[17] <http://www.dacs.dtic.mil/techs/neural/neural5.html>

[18] <http://www.dacs.dtic.mil/techs/neural/neural6.html#RTFToC22>

[19] <http://www.benbest.com/computer/nn.html>

[20] [http://en.wikipedia.org/wiki/Neural\\_network](http://en.wikipedia.org/wiki/Neural_network)

[21] <http://www.seattlerobotics.org/encoder/nov98/neural.html>

[22] <http://www.nd.com/welcome/whatisnn.htm>

## CODING

### ***For 1-bit AHSD Number***

P1= [0; 0]

P2= [0; 1]

P3= [1; 0]

P4= [1; 1]

o1= [0; 0]

o2= [0; 1]

o3= [0; 1]

o4= [0; 2]

I= [P1 P2 P3 P4]

O= [o1 o2 o3 o4]

PR=minmax(I)

S1=2

S2=2

net=newff(PR,[S1 S2],{'tansig','tansig'}) / net=newff(PR,[S1 S2],{'logsig' 'logsig'})

/net=newff(PR,[S1 S2],{'purelin' 'purelin'})

[net,tr]=train (net,I,O)

P=sim (net, I)

P=round (P)



### ***For 2-bit AHSD Number***

x1= [0;0;0;0];

x 2= [0;0;0;1];

x 3= [0;0;1;0];

x 4= [0;0;1;1];

x 5= [0;1;0;0];

x 6= [0;1;0;1];

x 7= [0;1;1;0];

x 8= [0;1;1;1];

x 9= [1;0;0;0];

x 10= [1;0;0;1];

x 11= [1;0;1;0];

x12= [1;0;1;1];

x13= [1;1;0;0];

x14= [1;1;0;1];

x15= [1;1;1;0];

x16= [1;1;1;1];

n1=[0;0];

n2=[0;.1];

n3=[0;.2];

n4=[0;.3];

n5=[0;.1];

n6=[0;.2];

n7=[0;.3];

n8=[.1;0];

n9=[0;.2];

n10=[0;.3];

n11=[.1;0];

n12=[.1;.1];

n13=[0;.3];

```

n14=[.1;0];
n15=[.1;.1];
n16=[.1;.2];
I=[x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16]
O=[n1 n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16]
PR=minmax(I)
S1=4;
S2=2;
net=newff(PR,[S1 S2]) / net=newff(PR,[S1 S2],{'logsig' 'logsig' 'logsig' 'logsig' })
[net,tr]=train (net,I,O)
M=sim(net, I)
M=round (M*10)

```

### ***For 3-bit AHSD Number***

```

x1= [0;0;0;0;0;0];
x2= [0;0;0;0;0;1];
x3= [0;0;0;0;1;0];
x4= [0;0;0;0;1;1];
x5= [0;0;0;1;0;0];
x6= [0;0;0;1;0;1];
x7= [0;0;0;1;1;0];
x8= [0;0;0;1;1;1];
x9= [0;0;1;0;0;0];
x10= [0;0;1;0;0;1];
x11= [0;0;1;0;1;0];
x12= [0;0;1;0;1;1];
x13= [0;0;1;1;0;0];
x14= [0;0;1;1;0;1];
x15= [0;0;1;1;1;0];
x16= [0;0;1;1;1;1];

```

x17= [0;1;0;0;0;0];  
x18= [0;1;0;0;0;1];  
x19= [0;1;0;0;1;0];  
x20= [0;1;0;0;1;1];  
x21= [0;1;0;1;0;0];  
x22= [0;1;0;1;0;1];  
x23= [0;1;0;1;1;0];  
x24= [0;1;0;1;1;1];  
x25= [0;1;1;0;0;0];  
x26= [0;1;1;0;0;1];  
x27= [0;1;1;0;1;0];  
x28= [0;1;1;0;1;1];  
x29= [0;1;1;1;0;0];  
x30= [0;1;1;1;0;1];  
x31= [0;1;1;1;1;0];  
x32= [0;1;1;1;1;1];  
x33= [1;0;0;0;0;0];  
x34= [1;0;0;0;0;1];  
x35= [1;0;0;0;1;0];  
x36= [1;0;0;0;1;1];  
x37= [1;0;0;1;0;0];  
x38= [1;0;0;1;0;1];  
x39= [1;0;0;1;1;0];  
x40= [1;0;0;1;1;1];  
x41= [1;0;1;0;0;0];  
x42= [1;0;1;0;0;1];  
x43= [1;0;1;0;1;0];  
x44= [1;0;1;0;1;1];  
x45= [1;0;1;1;0;0];  
x46= [1;0;1;1;0;1];  
x47= [1;0;1;1;1;0];

x48= [1;0;1;1;1;1];  
x49= [1;1;0;0;0;0];  
x50= [1;1;0;0;0;1];  
x51= [1;1;0;0;1;0];  
x52= [1;1;0;0;1;1];  
x53= [1;1;0;1;0;0];  
x54= [1;1;0;1;0;1];  
x55= [1;1;0;1;1;0];  
x56= [1;1;0;1;1;1];  
x57= [1;1;1;0;0;0];  
x58= [1;1;1;0;0;1];  
x59= [1;1;1;0;1;0];  
x60= [1;1;1;0;1;1];  
x61= [1;1;1;1;0;0];  
x62= [1;1;1;1;0;1];  
x63= [1;1;1;1;1;0];  
x64= [1;1;1;1;1;1];  
t1= [0; 0; 0];  
t2= [0; 0; .1];  
t3= [0; 0; .2];  
t4= [0; 0; .3];  
t5= [0; .1; 0];  
t6= [0; .1; .1];  
t7= [0; .3; 0];  
t8= [0;.1;.3];  
t9= [0;0;.1];  
t10= [0;0;.2];  
t11= [0;0;.3];  
t12= [0;.1;0];  
t13= [0;.1;.1];  
t14= [0;.1;.2];

t15= [0;.1;.3];  
t16= [0;.2;0];  
t17= [0;0;.2];  
t18= [0;0;.3];  
t19= [0;.1;0];  
t20= [0;.1;.1];  
t21= [0;.1;.2];  
t22= [0;.1;.3];  
t23= [0;.2;0];  
t24= [0;.2;.1];  
t25= [0;0;.3];  
t26= [0;.1;0];  
t27= [0;.1;.1];  
t28= [0;.1;.2];  
t29= [0;.2;.3];  
t30= [0;0;.3];  
t31= [0;.2;.1];  
t32= [0;.2;.2];  
t33= [0;.1;0];  
t34= [0;.1;.1];  
t35= [0;.1;.2];  
t36= [0;.1;.3];  
t37= [0;.2;0];  
t38= [0;.2;.1];  
t39= [0;.3;.2];  
t40= [0;.2;.3];  
t41= [0;.1;.1];  
t42= [0;.1;.2];  
t43= [0;.1;.3];  
t44= [0;.2;0];  
t45= [0;.3;.1];

t46= [0;.2;.2];

t47= [0;.2;.3];

t48= [0;.3;0];

t49= [0;.1;.2];

t50= [0;.1;.3];

t51= [0;.2;0];

t52= [0;.2;.1];

t53= [0;.2;.2];

t54= [0;.2;.3];

t55= [0;.3;0];

t56= [0;.3;.1];

t57= [0;.1;.3];

t58= [0;.2;0];

t59= [0;.2;.1];

t60= [0;.2;.2];

t61= [0;.2;.3];

t62= [0;.3;0];

t63= [0;.3;.1];

t64= [0;.3;.2];

I=[x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21 x22  
x23 x24 x25 x26 x27 x28 x29 x30 x31 x32 x33 x34 x35 x36 x37 x38 x39 x40 x41 x42  
x43 x44 x45 x46 x47 x48 x49 x50 x51 x52 x53 x54 x55 x56 x57 x58 x59 x60 x61 x62  
x63 x64]

O= [t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t15 t16 t17 t18 t19 t20 t21 t22 t23 t24 t25  
t26 t27 t28 t29 t30 t31 t32 t33 t34 t35 t36 t37 t38 t39 t40 t41 t42 t43 t44 t45 t46 t47 t48  
t49 t50 t51 t52 t53 t54 t55 t56 t57 t58 t59 t60 t61 t62 t63 t64]

PR=minmax(I)

S1=6

S2=3

net=newff(PR,[S1 S2],{'logsig' 'logsig' 'logsig' 'logsig' 'logsig' 'logsig'});

[net,tr]=train(net,I,O)

M=sim(net, I)

M=round (10\*M)

## APPENDIX

### SIMULATION USING NEURAL NETWORK BY MATLAB-7

#### *1-Bit AHSD Numbers*

#### For TANSIG Transfer Function

P1=[0;0]

P2=[0;1]

P3=[1;0]

P4=[1;1]

o1=[0;0]

o2=[0;.1]

o3=[0;.1]

o4=[0;.2]

I= [P1 P2 P3 P4]

O= [o1 o2 o3 o4]

PR=minmax(I)

S1=2

S2=2

P1 =

0

0

P2 =

0

1

P3 =

1

0

P4 =

1

1

o1 =

0

0

o2 =

0

0.1000

o3 =

0

0.1000

```
o4 =
0
0.2000
I =
0 0 1 1
0 1 0 1
O =
0 0 0 0
0 0.1000 0.1000 0.2000
```

```
PR =
```

```
0 1
```

```
0 1
```

```
S1 =
```

```
2
```

```
S2 =
```

```
2
```

```
net=newff(PR,[S1 S2])
```

```
net = Neural Network object:
```

```
architecture:
```

```
numInputs: 1
```

```
numLayers: 2
```

```
biasConnect: [1; 1]
```

```
inputConnect: [1; 0]
```

```
layerConnect: [0 0; 1 0]
```

```
outputConnect: [0 1]
```

```
targetConnect: [0 1]
```

```
numOutputs: 1 (read-only)
```

```
numTargets: 1 (read-only)
```

```
numInputDelays: 0 (read-only)
```

```
numLayerDelays: 0 (read-only)
```

```
subobject structures:
```

```
inputs: {1x1 cell} of inputs
```

```
layers: {2x1 cell} of layers
```

```
outputs: {1x2 cell} containing 1 output
```

```
targets: {1x2 cell} containing 1 target
```

```
biases: {2x1 cell} containing 2 biases
```

```
inputWeights: {2x1 cell} containing 1 input weight
```

```
layerWeights: {2x2 cell} containing 1 layer weight
```

```
functions:
```

```
adaptFcn: 'trains'
```

```
initFcn: 'initlay'
```

```
performFcn: 'mse'
```

```
trainFcn: 'trainlm'
```

```
parameters:
```

```
adaptParam: .passes
```

```
initParam: (none)
```

```
performParam: (none)
```

```
trainParam: .epochs, .goal, .max_fail, .mem_reduc,
```

```
.min_grad, .mu, .mu_dec, .mu_inc,
```

```
.mu_max, .show, .time
```

```
weight and bias values:
```

```
IW: {2x1 cell} containing 1 input weight matrix
```

```
LW: {2x2 cell} containing 1 layer weight matrix
```

```
b: {2x1 cell} containing 2 bias vectors
```

```
other:
```



```

userdata: (user stuff)
[net,tr]=train(net,I,O)
TRAINLM, Epoch 0/100, MSE 0.754762/0, Gradient 1.71559/1e-010
TRAINLM, Epoch 12/100, MSE 1.1363e-032/0, Gradient 6.27215e-016/1e-010
TRAINLM, Minimum gradient reached, performance goal was not met.
net =
Neural Network object:
architecture:
numInputs: 1
numLayers: 2
biasConnect: [1; 1]
inputConnect: [1; 0]
layerConnect: [0 0; 1 0]
outputConnect: [0 1]
targetConnect: [0 1]
numOutputs: 1 (read-only)
numTargets: 1 (read-only)
numInputDelays: 0 (read-only)
numLayerDelays: 0 (read-only)
subobject structures:
inputs: {1x1 cell} of inputs
layers: {2x1 cell} of layers
outputs: {1x2 cell} containing 1 output
targets: {1x2 cell} containing 1 target
biases: {2x1 cell} containing 2 biases
inputWeights: {2x1 cell} containing 1 input weight
layerWeights: {2x2 cell} containing 1 layer weight
functions:
adaptFcn: 'trains'
initFcn: 'initlay'
performFcn: 'mse'
trainFcn: 'trainlm'
parameters:
adaptParam: .passes
initParam: (none)
performParam: (none)
trainParam: .epochs, .goal, .max_fail, .mem_reduc,
.min_grad, .mu, .mu_dec, .mu_inc,
.mu_max, .show, .time
weight and bias values:
IW: {2x1 cell} containing 1 input weight matrix
LW: {2x2 cell} containing 1 layer weight matrix
b: {2x1 cell} containing 2 bias vectors
other:
userdata: (user stuff)
tr =
epoch: [0 1 2 3 4 5 6 7 8 9 10 11 12]
perf: [1x13 double]
vperf: [1x13 double]
tperf: [1x13 double]
mu: [1x13 double]
P=sim(net, I)
P =
0 0 0 0.0000
0 0.1000 0.1000 0.2000
P=round (P*10)

```

P =  
0 0 0 0  
0 1 1 2

### **For LOGSIG Transfer Function**

P1=[0;0]  
P2=[0;1]  
P3=[1;0]  
P4=[1;1]  
o1=[0;0]  
o2=[0;.1]  
o3=[0;.1]  
o4=[0;.2]  
I= [P1 P2 P3 P4]  
O= [o1 o2 o3 o4]  
PR=minmax(I)  
S1=2  
S2=2  
P1 =  
0  
0  
P2 =  
0  
1  
P3 =  
1  
0  
P4 =  
1  
1  
o1 =  
0  
0  
o2 =  
0  
0.1000  
o3 =  
0  
0.1000  
o4 =  
0  
0.2000  
I =  
0 0 1 1  
0 1 0 1  
  
O =  
0 0 0 0  
0 0.1000 0.1000 0.2000  
PR =  
0 1  
0 1  
S1 =  
2

```

S2 =
2
net=newff(PR,[S1 S2],{'logsig' 'logsig'})
net =
Neural Network object:
architecture:
numInputs: 1
numLayers: 2
biasConnect: [1; 1]
inputConnect: [1; 0]
layerConnect: [0 0; 1 0]
outputConnect: [0 1]
targetConnect: [0 1]
numOutputs: 1 (read-only)
numTargets: 1 (read-only)
numInputDelays: 0 (read-only)
numLayerDelays: 0 (read-only)
subobject structures:
inputs: {1x1 cell} of inputs
layers: {2x1 cell} of layers
outputs: {1x2 cell} containing 1 output
targets: {1x2 cell} containing 1 target
biases: {2x1 cell} containing 2 biases
inputWeights: {2x1 cell} containing 1 input weight
layerWeights: {2x2 cell} containing 1 layer weight
functions:
adaptFcn: 'trains'
initFcn: 'initlay'
performFcn: 'mse'
trainFcn: 'trainlm'
parameters:
adaptParam: .passes
initParam: (none)
performParam: (none)
trainParam: .epochs, .goal, .max_fail, .mem_reduc,
.min_grad, .mu, .mu_dec, .mu_inc,
.mu_max, .show, .time
weight and bias values:
IW: {2x1 cell} containing 1 input weight matrix
LW: {2x2 cell} containing 1 layer weight matrix
b: {2x1 cell} containing 2 bias vectors
other:
userdata: (user stuff)
[net,tr]=train(net,I,O)
TRAINLM, Epoch 0/100, MSE 0.356467/0, Gradient 0.367335/1e-010
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 1.469039e-016.
In trainlm at 318
In network.train at 278
TRAINLM, Epoch 25/100, MSE 7.63838e-009/0, Gradient 4.32674e-005/1e-010
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 1.468044e-017.
In trainlm at 318
In network.train at 278
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 1.466953e-018.

```

In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.475246e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.470280e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.443012e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.463436e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.466850e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474613e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473830e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474718e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.467657e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474565e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.464296e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.472131e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473622e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.405666e-017.

In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.472902e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.407667e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474201e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 8.459617e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473498e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.425045e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474609e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.397570e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.472373e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.212445e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473069e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.403337e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.454547e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.385101e-017.

In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 3.830809e-021.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.403797e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474653e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.352293e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 7.499080e-020.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.418361e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473579e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.390825e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.376381e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 8.467089e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473997e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.398684e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.472176e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.255075e-017.

In trainlm at 318  
In network.train at 278  
TRAINLM, Epoch 50/100, MSE 2.44675e-017/0, Gradient 2.45823e-009/1e-010  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.451460e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.387854e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473524e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474579e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474616e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 8.451983e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 2.775657e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.380239e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.374168e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.382319e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.416152e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.375329e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474196e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.

Results may be inaccurate. RCOND = 1.234620e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.302878e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474761e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473588e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.392567e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473137e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.392277e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.267869e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.367548e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 2.393547e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.356621e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.377161e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.255631e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.388704e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.



Results may be inaccurate. RCOND = 1.377106e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474462e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 8.125362e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.397043e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474585e-017.  
> In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.472549e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.393926e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.466244e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.385446e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.465896e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.381296e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.410190e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.379379e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.407685e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.

Results may be inaccurate. RCOND = 4.237177e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473589e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.232026e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 3.019857e-020.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.385112e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.172551e-019.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474679e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.474379e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.386011e-017.  
In trainlm at 318  
In network.train at 278  
TRAINLM, Epoch 75/100, MSE 4.13152e-018/0, Gradient 1.01253e-009/1e-010  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.435807e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 8.386067e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.338887e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.201722e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.050186e-019.  
In trainlm at 318  
In network.train at 278

Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.403624e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.473393e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.383014e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.451383e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.393587e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 6.210125e-020.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.359811e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.318832e-021.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.394854e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.467578e-020.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 8.118802e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.471337e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.377725e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.359138e-018.  
In trainlm at 318  
In network.train at 278

Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.381083e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.454444e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 8.317996e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.401809e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.406485e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.444427e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.393227e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.330493e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.366809e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.439675e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.388025e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.399658e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.358406e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.270091e-018.  
In trainlm at 318  
In network.train at 278

Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.256444e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.437202e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.257325e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.471018e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.257358e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.472414e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.248379e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.462136e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.409976e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.428080e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.269389e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.405253e-018.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.372129e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.410334e-018.  
In trainlm at 318  
In network.train at 278

Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.363691e-017.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.800134e-022.  
In trainlm at 318  
In network.train at 278  
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 1.402411e-017.  
In trainlm at 318  
In network.train at 278  
TRAINLM, Epoch 100/100, MSE 1.12219e-018/0, Gradient 5.27372e-010/1e-010  
TRAINLM, Maximum epoch reached, performance goal was not met.  
net =  
Neural Network object:  
architecture:  
numInputs: 1  
numLayers: 2  
biasConnect: [1; 1]  
inputConnect: [1; 0]  
layerConnect: [0 0; 1 0]  
outputConnect: [0 1]  
targetConnect: [0 1]  
numOutputs: 1 (read-only)  
numTargets: 1 (read-only)  
numInputDelays: 0 (read-only)  
numLayerDelays: 0 (read-only)  
subobject structures:  
inputs: {1x1 cell} of inputs  
layers: {2x1 cell} of layers  
outputs: {1x2 cell} containing 1 output  
targets: {1x2 cell} containing 1 target  
biases: {2x1 cell} containing 2 biases  
inputWeights: {2x1 cell} containing 1 input weight  
layerWeights: {2x2 cell} containing 1 layer weight  
functions:  
adaptFcn: 'trains'  
initFcn: 'initlay'  
performFcn: 'mse'  
trainFcn: 'trainlm'  
parameters:  
adaptParam: .passes  
initParam: (none)  
performParam: (none)  
trainParam: .epochs, .goal, .max\_fail, .mem\_reduc,  
.min\_grad, .mu, .mu\_dec, .mu\_inc,  
.mu\_max, .show, .time  
weight and bias values:  
IW: {2x1 cell} containing 1 input weight matrix  
LW: {2x2 cell} containing 1 layer weight matrix  
b: {2x1 cell} containing 2 bias vectors  
other:  
userdata: (user stuff)  
tr =  
epoch: [1x101 double]

```

perf: [1x101 double]
vperf: [1x101 double]
tperf: [1x101 double]
mu: [1x101 double]
P=sim(net, I)
P =
0.0000 0.0000 0.0000 0.0000
0.0000 0.1000 0.1000 0.2000
P=round(P*10)
P =
0 0 0 0
0 1 1 2

```

FOR PURELIN TRANSFER FUNCTION

```

P1=[0;0]
P2=[0;1]
P3=[1;0]
P4=[1;1]
o1=[0;0]
o2=[0;.1]
o3=[0;.1]
o4=[0;.2]
I= [P1 P2 P3 P4]
O= [o1 o2 o3 o4]
PR=minmax(I)
S1=2
S2=2
P1 =
0
0
P2 =
0
1
P3 =
1
0
P4 =
1
1
o1 =
0
0
o2 =
0
0.1000
o3 =
0
0.1000

o4 =
0
0.2000
I =
0 0 1 1
0 1 0 1

O =

```

```

0 0 0 0
0 0.1000 0.1000 0.2000
PR =
0 1
0 1
S1 =
2
S2 =
2
net=newff(PR,[S1 S2],{'purelin' 'purelin'})
net=newff(PR,[S1 S2],{'purelin' 'purelin'})
net =
Neural Network object:
architecture:
numInputs: 1
numLayers: 2
biasConnect: [1; 1]
inputConnect: [1; 0]
layerConnect: [0 0; 1 0]
outputConnect: [0 1]
targetConnect: [0 1]
numOutputs: 1 (read-only)
numTargets: 1 (read-only)
numInputDelays: 0 (read-only)
numLayerDelays: 0 (read-only)
subobject structures:
inputs: {1x1 cell} of inputs
layers: {2x1 cell} of layers
outputs: {1x2 cell} containing 1 output
targets: {1x2 cell} containing 1 target
biases: {2x1 cell} containing 2 biases
inputWeights: {2x1 cell} containing 1 input weight
layerWeights: {2x2 cell} containing 1 layer weight
functions:
adaptFcn: 'trains'
initFcn: 'initlay'
performFcn: 'mse'
trainFcn: 'trainlm'
parameters:
adaptParam: .passes
initParam: (none)
performParam: (none)
trainParam: .epochs, .goal, .max_fail, .mem_reduc,
.min_grad, .mu, .mu_dec, .mu_inc,
.mu_max, .show, .time
weight and bias values:
IW: {2x1 cell} containing 1 input weight matrix
LW: {2x2 cell} containing 1 layer weight matrix
b: {2x1 cell} containing 2 bias vectors
other:
userdata: (user stuff)
[net,tr]=train(net,I,O)
TRAINLM, Epoch 0/100, MSE 0.520096/0, Gradient 7.9388/1e-010
TRAINLM, Epoch 6/100, MSE 2.06683e-031/0, Gradient 5.49675e-016/1e-010
TRAINLM, Minimum gradient reached, performance goal was not met.
net =

```



```

Neural Network object:
numInputs: 1
numLayers: 2
biasConnect: [1; 1]
inputConnect: [1; 0]
layerConnect: [0 0; 1 0]
outputConnect: [0 1]
targetConnect: [0 1]
numOutputs: 1 (read-only)
numTargets: 1 (read-only)
numInputDelays: 0 (read-only)
numLayerDelays: 0 (read-only)
subobject structures:
inputs: {1x1 cell} of inputs
layers: {2x1 cell} of layers
outputs: {1x2 cell} containing 1 output
targets: {1x2 cell} containing 1 target
biases: {2x1 cell} containing 2 biases
inputWeights: {2x1 cell} containing 1 input weight
layerWeights: {2x2 cell} containing 1 layer weight
functions:
adaptFcn: 'trains'
initFcn: 'initlay'
performFcn: 'mse'
trainFcn: 'trainlm'
parameters:
adaptParam: .passes
initParam: (none)
performParam: (none)
trainParam: .epochs, .goal, .max_fail, .mem_reduc,
.min_grad, .mu, .mu_dec, .mu_inc,
.mu_max, .show, .time
weight and bias values:
IW: {2x1 cell} containing 1 input weight matrix
LW: {2x2 cell} containing 1 layer weight matrix
b: {2x1 cell} containing 2 bias vectors
other:
userdata: (user stuff)
tr =
epoch: [0 1 2 3 4 5 6]
perf: [1x7 double]
vperf: [NaN NaN NaN NaN NaN NaN NaN]
tperf: [NaN NaN NaN NaN NaN NaN NaN]
mu: [1x7 double]
P=sim(net, I)
P =
0.0000 -0.0000 0.0000 -0.0000
-0.0000 0.1000 0.1000 0.2000
P=round (P*10)
P =
0 0 0 0
0 1 1 2

```

## ***2-Bit AHSD Numbers***

### **For TANSIG Transfer Function**

x1= [0;0;0;0];  
x2= [0;0;0;1];  
x3= [0;0;1;0];  
x4= [0;0;1;1];  
x5= [0;1;0;0];  
x6= [0;1;0;1];  
x7= [0;1;1;0];  
x8= [0;1;1;1];  
x9= [1;0;0;0];  
x10= [1;0;0;1];  
x11= [1;0;1;0];  
x12= [1;0;1;1];  
x13= [1;1;0;0];  
x14= [1;1;0;1];  
x15= [1;1;1;0];  
x16= [1;1;1;1];  
n1=[0;0];  
n2=[0;.1];  
n3=[0;.2];  
n4=[0;.3];  
n5=[0;.1];  
n6=[0;.2];  
n7=[0;.3];  
n8=[.1;0];  
n9=[0;.2];  
n10=[0;.3];  
n11=[.1;0];  
n12=[.1;.1];  
n13=[0;.3];  
n14=[.1;0];  
n15=[.1;.1];  
n16=[.1;.2];  
I=[x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16]  
I =  
Columns 1 through 12  
0 0 0 0 0 0 0 0 1 1 1 1  
0 0 0 0 1 1 1 1 0 0 0 0  
0 0 1 1 0 0 1 1 0 0 1 1  
0 1 0 1 0 1 0 1 0 1 0 1  
Columns 13 through 16  
1 1 1 1  
1 1 1 1  
0 0 1 1  
0 1 0 1  
O=[n1 n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16]

O =

Columns 1 through 7

```
0 0 0 0 0 0 0
0 0.1000 0.2000 0.3000 0.1000 0.2000 0.3000
```

Columns 8 through 14

```
0.1000 0 0 0.1000 0.1000 0 0.1000
0 0.2000 0PR=minmax(I).3000 0 0.1000 0.3000 0
```

Columns 15 through 16

```
0.1000 0.1000
```

```
0.1000 0.2000
```

```
PR=minmax(I)
```

```
PR =
```

```
0 1
```

```
0 1
```

```
0 1
```

```
0 1
```

```
S1=4;
```

```
S2=2;
```

```
net=newff(PR,[S1 S2])
```

```
net =
```

```
Neural Network object:
```

```
architecture
```

```
numInputs: 1
```

```
numLayers: 2
```

```
biasConnect: [1; 1]
```

```
inputConnect: [1; 0]
```

```
layerConnect: [0 0; 1 0]
```

```
outputConnect: [0 1]
```

```
targetConnect: [0 1]
```

```
numOutputs: 1 (read-only)
```

```
numTargets: 1 (read-only)
```

```
numInputDelays: 0 (read-only)
```

```
numLayerDelays: 0 (read-only)
```

```
subobject structures:
```

```
inputs: {1x1 cell} of inputs
```

```
layers: {2x1 cell} of layers
```

```
outputs: {1x2 cell} containing 1 output
```

```
targets: {1x2 cell} containing 1 target
```

```
biases: {2x1 cell} containing 2 biases
```

```
inputWeights: {2x1 cell} containing 1 input weight
```

```
layerWeights: {2x2 cell} containing 1 layer weight
```

```
functions:
```

```
adaptFcn: 'trains'
```

```
initFcn: 'initlay'
```

```
performFcn: 'mse'
```

```
trainFcn: 'trainlm'
```

```
parameters:
```

```
adaptParam: .passes
```

```
initParam: (none)
```

```
performParam: (none)
```

```
trainParam: .epochs, .goal, .max_fail, .mem_reduc,
```

```
.min_grad, .mu, .mu_dec, .mu_inc,
.mu_max, .show, .time
weight and bias values:
IW: {2x1 cell} containing 1 input weight matrix
LW: {2x2 cell} containing 1 layer weight matrix
b: {2x1 cell} containing 2 bias vectors
other:
userdata: (user stuff)
[net,tr]=train(net,I,O)
TRAINLM, Epoch 0/100, MSE 0.521624/0, Gradient 2.82224/1e-010
TRAINLM, Epoch 25/100, MSE 0.000731269/0, Gradient 0.0629324/1e-010
TRAINLM, Epoch 50/100, MSE 1.44424e-005/0, Gradient 0.0230807/1e-010
TRAINLM, Epoch 75/100, MSE 1.81926e-006/0, Gradient 0.0060433/1e-010
TRAINLM, Epoch 100/100, MSE 1.07402e-006/0, Gradient 0.00171766/1e-010
TRAINLM, Maximum epoch reached, performance goal was not met.
net =
Neural Network object:
architecture:
numInputs: 1
numLayers: 2
biasConnect: [1; 1]
inputConnect: [1; 0]
layerConnect: [0 0; 1 0]
outputConnect: [0 1]
targetConnect: [0 1]
numOutputs: 1 (read-only)
numTargets: 1 (read-only)
numInputDelays: 0 (read-only)
numLayerDelays: 0 (read-only)
subobject structures:
inputs: {1x1 cell} of inputs
layers: {2x1 cell} of layers
outputs: {1x2 cell} containing 1 output
targets: {1x2 cell} containing 1 target
biases: {2x1 cell} containing 2 biases
inputWeights: {2x1 cell} containing 1 input weight
layerWeights: {2x2 cell} containing 1 layer weight
functions:
adaptFcn: 'trains'
initFcn: 'initlay'
performFcn: 'mse'
trainFcn: 'trainlm'
parameters:
adaptParam: .passes
initParam: (none)
performParam: (none)
trainParam: .epochs, .goal, .max_fail, .mem_reduc,
.min_grad, .mu, .mu_dec, .mu_inc,
.mu_max, .show, .time
weight and bias values:
IW: {2x1 cell} containing 1 input weight matrix
LW: {2x2 cell} containing 1 layer weight matrix
b: {2x1 cell} containing 2 bias vectors
other:
userdata: (user stuff)
tr =
```

```

epoch: [1x101 double]
perf: [1x101 double]
vperf: [1x101 double]
tperf: [1x101 double]
mu: [1x101 double]
M=sim(net, I)
M =
Columns 1 through 7
0.0002 -0.0031 0.0019 -0.0002 -0.0000 -0.0005 0.0003
0.0001 0.0996 0.2007 0.2995 0.1000 0.2004 0.2999
Columns 8 through 14
0.1020 0.0019 -0.0002 0.0987 0.0993 0.0003 0.1020
0.0001 0.2007 0.2995 -0.0005 0.1005 0.2999 0.0001
Columns 15 through 16
0.0981 0.0995
0.0997 0.2003
M=round(M*10)
M =
Columns 1 through 9
0 0 0 0 0 0 0 1 0
0 1 2 3 1 2 3 0 2
Columns 10 through 16
0 1 1 0 1 1 1
3 0 1 3 0 1 2

```

### **For LOGSIG Transfer Function**

```

x1= [0;0;0;0];
x2= [0;0;0;1];
x3= [0;0;1;0];
x4= [0;0;1;1];
x5= [0;1;0;0];
x6= [0;1;0;1];
x7= [0;1;1;0];
x8= [0;1;1;1];
x9= [1;0;0;0];
x10= [1;0;0;1];
x11= [1;0;1;0];
x12= [1;0;1;1];
x13= [1;1;0;0];
x14= [1;1;0;1];
x15= [1;1;1;0];
x16= [1;1;1;1];
n1=[0;0];
n2=[0;.1];
n3=[0;.2];
n4=[0;.3];
n5=[0;.1];
n6=[0;.2];
n7=[0;.3];
n8=[.1;0];
n9=[0;.2];
n10=[0;.3];
n11=[.1;0];
n12=[.1;.1];
n13=[0;.3];
n14=[.1;0];

```

```

n15=[.1;.1];
n16=[.1;.2];
I=[x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16]
O=[n1 n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16]
PR=minmax(I)
S1=4;
S2=2;
I =
Columns 1 through 9
0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 0
0 0 1 1 0 0 1 1 0
0 1 0 1 0 1 0 1 0
Columns 10 through 16
1 1 1 1 1 1 1
0 0 0 1 1 1 1
0 1 1 0 0 1 1
1 0 1 0 1 0 1
O =
Columns 1 through 5
0 0 0 0 0
0 0.1000 0.2000 0.3000 0.1000
Columns 6 through 10
0 0 0.1000 0 0
0.2000 0.3000 0 0.2000 0.3000
Columns 11 through 15
0.1000 0.1000 0 0.1000 0.1000
0 0.1000 0.3000 0 0.1000
Column 16

0.1000
0.2000
PR =
0 1
0 1
0 1
0 1
net=newff(PR,[S1 S2],{'logsig' 'logsig' 'logsig' 'logsig' })
net =
Neural Network object:
architecture:
numInputs: 1
numLayers: 2
biasConnect: [1; 1]
inputConnect: [1; 0]
layerConnect: [0 0; 1 0]
outputConnect: [0 1]
targetConnect: [0 1]
numOutputs: 1 (read-only)
numTargets: 1 (read-only)
numInputDelays: 0 (read-only)
numLayerDelays: 0 (read-only)
subobject structures:
inputs: {1x1 cell} of inputs
layers: {2x1 cell} of layers
outputs: {1x2 cell} containing 1 output

```

targets: {1x2 cell} containing 1 target  
biases: {2x1 cell} containing 2 biases  
inputWeights: {2x1 cell} containing 1 input weight  
layerWeights: {2x2 cell} containing 1 layer weight  
functions:  
adaptFcn: 'trains'  
initFcn: 'initlay'  
performFcn: 'mse'  
trainFcn: 'trainlm'  
parameters:  
adaptParam: .passes  
initParam: (none)  
performParam: (none)  
trainParam: .epochs, .goal, .max\_fail, .mem\_reduc,  
.min\_grad, .mu, .mu\_dec, .mu\_inc,  
.mu\_max, .show, .time  
weight and bias values:  
IW: {2x1 cell} containing 1 input weight matrix  
LW: {2x2 cell} containing 1 layer weight matrix  
b: {2x1 cell} containing 2 bias vectors  
other:  
userdata: (user stuff)  
[net,tr]=train(net,I,O)  
TRAINLM, Epoch 0/100, MSE 0.280293/0, Gradient 0.76117/1e-010  
TRAINLM, Epoch 25/100, MSE 0.000287684/0, Gradient 0.00369984/1e-010  
TRAINLM, Epoch 50/100, MSE 0.000241957/0, Gradient 7.20442e-005/1e-010  
TRAINLM, Epoch 75/100, MSE 0.000220763/0, Gradient 0.000599438/1e-010  
TRAINLM, Epoch 100/100, MSE 9.87541e-005/0, Gradient 0.000500187/1e-010  
TRAINLM, Maximum epoch reached, performance goal was not met.  
net =  
Neural Network object:  
architecture:  
numInputs: 1  
numLayers: 2  
biasConnect: [1; 1]  
inputConnect: [1; 0]  
layerConnect: [0 0; 1 0]  
outputConnect: [0 1]  
targetConnect: [0 1]  
numOutputs: 1 (read-only)  
numTargets: 1 (read-only)  
numInputDelays: 0 (read-only)  
numLayerDelays: 0 (read-only)  
subobject structures:  
inputs: {1x1 cell} of inputs  
layers: {2x1 cell} of layers  
outputs: {1x2 cell} containing 1 output  
targets: {1x2 cell} containing 1 target  
biases: {2x1 cell} containing 2 biases  
inputWeights: {2x1 cell} containing 1 input weight  
layerWeights: {2x2 cell} containing 1 layer weight  
functions:  
adaptFcn: 'trains'  
initFcn: 'initlay'  
performFcn: 'mse'  
trainFcn: 'trainlm'

```

parameters:
adaptParam: .passes
initParam: (none)
performParam: (none)
trainParam: .epochs, .goal, .max_fail, .mem_reduc,
.min_grad, .mu, .mu_dec, .mu_inc,
.mu_max, .show, .time
weight and bias values:
IW: {2x1 cell} containing 1 input weight matrix
LW: {2x2 cell} containing 1 layer weight matrix
b: {2x1 cell} containing 2 bias vectors
other:
userdata: (user stuff)
tr =
epoch: [1x101 double]
perf: [1x101 double]
vperf: [1x101 double]
tperf: [1x101 double]
mu: [1x101 double]
M=sim(net, I)
M=round(M*10)
M =
Columns 1 through 5
0.0014  0.0014  0.0014  0.0015  0.0013
0.0048  0.1036  0.1986  0.2992  0.0997
Columns 6 through 10
0.0015  0.0013  0.0995  0.0089  0.0014
0.1993  0.3013  0.0038  0.1928  0.2998
Columns 11 through 15

0.1055  0.0961  0.0078  0.0997  0.0922
0.0429  0.1002  0.3065  0.0003  0.0701
Column 16
0.1042
0.1998
M =
Columns 1 through 9
0  0  0  0  0  0  0  1  0
0  1  2  3  1  2  3  0  2
Columns 10 through 16
0  1  1  0  1  1  1
3  0  1  3  0  1  2

```



### *3-Bit AHSD Numbers*

#### **For LOGSIG Transfer Function**

x1= [0;0;0;0;0;0];  
x2= [0;0;0;0;0;1];  
x3= [0;0;0;0;1;0];  
x4= [0;0;0;0;1;1];  
x5= [0;0;0;1;0;0];  
x6= [0;0;0;1;0;1];  
x7= [0;0;0;1;1;0];  
x8= [0;0;0;1;1;1];  
x9= [0;0;1;0;0;0];  
x10= [0;0;1;0;0;1];  
x11= [0;0;1;0;1;0];  
x12= [0;0;1;0;1;1];  
x13= [0;0;1;1;0;0];  
x14= [0;0;1;1;0;1];  
x15= [0;0;1;1;1;0];  
x16= [0;0;1;1;1;1];  
x17= [0;1;0;0;0;0];  
x18= [0;1;0;0;0;1];  
x19= [0;1;0;0;1;0];  
x20= [0;1;0;0;1;1];  
x21= [0;1;0;1;0;0];  
x22= [0;1;0;1;0;1];  
x23= [0;1;0;1;1;0];  
x24= [0;1;0;1;1;1];  
x25= [0;1;1;0;0;0];  
x26= [0;1;1;0;0;1];  
x27= [0;1;1;0;1;0];  
x28= [0;1;1;0;1;1];  
x29= [0;1;1;1;0;0];  
x30= [0;1;1;1;0;1];  
x31= [0;1;1;1;1;0];  
x32= [0;1;1;1;1;1];  
x33= [1;0;0;0;0;0];  
x34= [1;0;0;0;0;1];  
x35= [1;0;0;0;1;0];  
x36= [1;0;0;0;1;1];  
x37= [1;0;0;1;0;0];  
x38= [1;0;0;1;0;1];  
x39= [1;0;0;1;1;0];  
x40= [1;0;0;1;1;1];  
x41= [1;0;1;0;0;0];  
x42= [1;0;1;0;0;1];  
x43= [1;0;1;0;1;0];  
x44= [1;0;1;0;1;1];  
x45= [1;0;1;1;0;0];  
x46= [1;0;1;1;0;1];  
x47= [1;0;1;1;1;0];  
x48= [1;0;1;1;1;1];

x49= [1;1;0;0;0;0];  
 x50= [1;1;0;0;0;1];  
 x51= [1;1;0;0;1;0];  
 x52= [1;1;0;0;1;1];  
 x53= [1;1;0;1;0;0];  
 x54= [1;1;0;1;0;1];  
 x55= [1;1;0;1;1;0];  
 x56= [1;1;0;1;1;1];  
 x57= [1;1;1;0;0;0];  
 x58= [1;1;1;0;0;1];  
 x59= [1;1;1;0;1;0];  
 x60= [1;1;1;0;1;1];  
 x61= [1;1;1;1;0;0];  
 x62= [1;1;1;1;0;1];  
 x63= [1;1;1;1;1;0];  
 x64= [1;1;1;1;1;1];

I=[x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21 x22 x23 x24 x25 x26  
 x27 x28 x29 x30 x31 x32 x33 x34 x35 x36 x37 x38 x39 x40 x41 x42 x43 x44 x45 x46 x47 x48 x49 x50  
 x51 x52 x53 x54 x55 x56 x57 x58 x59 x60 x61 x62 x63 x64]

I =

Columns 1 through 9

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1
0	0	0	0	1	1	1	1	0
0	0	1	1	0	0	1	1	0
0	1	0	1	0	1	0	1	0

Columns 10 through 18

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1
1	1	1	1	1	1	1	0	0
0	0	0	1	1	1	1	0	0
0	1	1	0	0	1	1	0	0
1	0	1	0	1	0	1	0	1

Columns 19 through 27

0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1
0	0	1	1	1	1	0	0	0
1	1	0	0	1	1	0	0	1
0	1	0	1	0	1	0	1	0

Columns 28 through 36

0	0	0	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0
1	1	1	1	1	0	0	0	0
0	1	1	1	1	0	0	0	0
1	0	0	1	1	0	0	1	1
1	0	1	0	1	0	1	0	1

Columns 37 through 45

1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1
1	1	1	1	0	0	0	0	1
0	0	1	1	0	0	1	1	0
0	1	0	1	0	1	0	1	0

Columns 46 through 54

```
1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1
1 1 1 0 0 0 0 0 0
1 1 1 0 0 0 0 1 1
0 1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0 1
```

Columns 55 through 63

```
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1
1 1 0 0 0 0 1 1 1
1 1 0 0 1 1 0 0 1
0 1 0 1 0 1 0 1 0
```

Column 64

```
1
1
1
1
1
1
```

```
t1= [0;0;0];
t2= [0;0;.1];
t3= [0;0;.2];
t4= [0;0;.3];
t5= [0;.1;0];
t6= [0;.1;.1];
t7= [0;.3;0];
t8= [0;.1;.3];
t9= [0;0;.1];
t10= [0;0;.2];
t11= [0;0;.3];
t12=[0;.1;0];
t13=[0;.1;.1];
t14=[0;.1;.2];
t15=[0;.1;.3];
t16=[0;.2;0];
t17=[0;0;.2];
t18=[0;0;.3];
t19=[0;.1;0];
t20=[0;.1;.1];
t21=[0;.1;.2];
t22=[0;.1;.3];
t23=[0;.2;0];
t24=[0;.2;.1];
t25=[0;0;.3];
t26=[0;.1;0];
t27=[0;.1;.1];
t28=[0;.1;.2];
t29=[0;.2;.3];
t30=[0;0;.3];
t31=[0;.2;.1];
t32=[0;.2;.2];
t33=[0;.1;0];
t34=[0;.1;.1];
t35=[0;.1;.2];
```

t36=[0;.1;.3];  
 t37=[0;.2;.0];  
 t38=[0;.2;.1];  
 t39=[0;.3;.2];  
 t40=[0;.2;.3];  
 t41=[0;.1;.1];  
 t42=[0;.1;.2];  
 t43=[0;.1;.3];  
 t44=[0;.2;.0];  
 t45=[0;.3;.1];  
 t46=[0;.2;.2];  
 t47=[0;.2;.3];  
 t48=[0;.3;.0];  
 t49=[0;.1;.2];  
 t50=[0;.1;.3];  
 t51=[0;.2;.0];  
 t52=[0;.2;.1];  
 t53=[0;.2;.2];  
 t54=[0;.2;.3];  
 t55=[0;.3;.0];  
 t56=[0;.3;.1];  
 t57=[0;.1;.3];  
 t58=[0;.2;.0];  
 t59=[0;.2;.1];  
 t60=[0;.2;.2];  
 t61=[0;.2;.3];  
 t62=[0;.3;.0];  
 t63=[0;.3;.1];  
 t64=[0;.3;.2];  
 O=[t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t15 t16 t17 t18 t19 t20 t21 t22 t23 t24 t25 t26 t27 t28 t29 t30  
 t31 t32 t33 t34 t35 t36 t37 t38 t39 t40 t41 t42 t43 t44 t45 t46 t47 t48 t49 t50 t51 t52 t53 t54 t55 t56 t57 t58  
 t59 t60 t61 t62 t63 t64]

O =  
 Columns 1 through 5  
 0 0 0 0 0  
 0 0 0 0 0.1000  
 0 0.1000 0.2000 0.3000 0  
 Columns 6 through 10  
 0 0 0 0 0  
 0.1000 0.3000 0.1000 0 0  
 0.1000 0 0.3000 0.1000 0.2000  
 Columns 11 through 15  
 0 0 0 0 0  
 0 0.1000 0.1000 0.1000 0.1000  
 0.3000 0 0.1000 0.2000 0.3000  
 Columns 16 through 20  
 0 0 0 0 0  
 0.2000 0 0 0.1000 0.1000  
 0 0.2000 0.3000 0 0.1000  
 Columns 21 through 25  
 0 0 0 0 0  
 0.1000 0.1000 0.2000 0.2000 0  
 0.2000 0.3000 0 0.1000 0.3000  
 Columns 26 through 30  
 0 0 0 0 0  
 0.1000 0.1000 0.1000 0.2000 0

```

0 0.1000 0.2000 0.3000 0.3000
Columns 31 through 35
0 0 0 0 0
0.2000 0.2000 0.1000 0.1000 0.1000
0.1000 0.2000 0 0.1000 0.2000
Columns 36 through 40
0 0 0 0 0
0.1000 0.2000 0.2000 0.3000 0.2000
0.3000 0 0.1000 0.2000 0.3000
Columns 41 through 45
0 0 0 0 0
0.1000 0.1000 0.1000 0.2000 0.3000
0.1000 0.2000 0.3000 0 0.1000
Columns 46 through 50
0 0 0 0 0
0.2000 0.2000 0.3000 0.1000 0.1000
0.2000 0.3000 0 0.2000 0.3000
Columns 51 through 55
0 0 0 0 0
0.2000 0.2000 0.2000 0.2000 0.3000
0 0.1000 0.2000 0.3000 0
Columns 56 through 60
0 0 0 0 0
0.3000 0.1000 0.2000 0.2000 0.2000
0.1000 0.3000 0 0.1000 0.2000
Columns 61 through 64
0 0 0 0
0.2000 0.3000 0.3000 0.3000
0.3000 0 0.1000 0.2000
PR=minmax(I)
PR =
0 1
0 1
0 1
0 1
0 1
0 1
0 1
S1=6
S1 =
6
S2=
3
S2 =
3
net=newff(PR,[S1 S2],{'logsig' 'logsig' 'logsig' 'logsig' 'logsig' 'logsig'});
net =
Neural Network object:
architecture:
numInputs: 1
numLayers: 2
biasConnect: [1; 1]
inputConnect: [1; 0]
layerConnect: [0 0; 1 0]
outputConnect: [0 1]
targetConnect: [0 1]
numOutputs: 1 (read-only)

```

numTargets: 1 (read-only)  
numInputDelays: 0 (read-only)  
numLayerDelays: 0 (read-only)  
subobject structures:  
inputs: {1x1 cell} of inputs  
layers: {2x1 cell} of layers  
outputs: {1x2 cell} containing 1 output  
targets: {1x2 cell} containing 1 target  
biases: {2x1 cell} containing 2 biases  
inputWeights: {2x1 cell} containing 1 input weight  
layerWeights: {2x2 cell} containing 1 layer weight  
functions:  
adaptFcn: 'trains'  
initFcn: 'initlay'  
performFcn: 'mse'  
trainFcn: 'trainlm'  
parameters:  
adaptParam: .passes  
initParam: (none)  
performParam: (none)  
trainParam: .epochs, .goal, .max\_fail, .mem\_reduc,  
.min\_grad, .mu, .mu\_dec, .mu\_inc,  
.mu\_max, .show, .time  
weight and bias values:  
IW: {2x1 cell} containing 1 input weight matrix  
LW: {2x2 cell} containing 1 layer weight matrix  
b: {2x1 cell} containing 2 bias vectors  
other:  
userdata: (user stuff)  
[net,tr]=train(net,I,O)  
TRAINLM, Epoch 0/100, MSE 0.103566/0, Gradient 5.30693/1e-010  
TRAINLM, Epoch 25/100, MSE 0.00035158/0, Gradient 0.0102603/1e-010  
TRAINLM, Epoch 50/100, MSE 0.000257346/0, Gradient 0.00151319/1e-010  
TRAINLM, Epoch 75/100, MSE 0.000247143/0, Gradient 0.000124151/1e-010  
TRAINLM, Epoch 100/100, MSE 0.000245508/0, Gradient 0.00118849/1e-010  
TRAINLM, Maximum epoch reached, performance goal was not met.  
net =  
Neural Network object:  
architecture:  
numInputs: 1  
numLayers: 2  
biasConnect: [1; 1]  
inputConnect: [1; 0]  
layerConnect: [0 0; 1 0]  
outputConnect: [0 1]  
targetConnect: [0 1]  
numOutputs: 1 (read-only)  
numTargets: 1 (read-only)  
numInputDelays: 0 (read-only)  
numLayerDelays: 0 (read-only)  
subobject structures:  
inputs: {1x1 cell} of inputs  
layers: {2x1 cell} of layers  
outputs: {1x2 cell} containing 1 output  
targets: {1x2 cell} containing 1 target  
biases: {2x1 cell} containing 2 biases

```

inputWeights: {2x1 cell} containing 1 input weight
layerWeights: {2x2 cell} containing 1 layer weight
functions:
adaptFcn: 'trains'
initFcn: 'initlay'
performFcn: 'mse'
trainFcn: 'trainlm'
parameters:
adaptParam: .passes
initParam: (none)
performParam: (none)
trainParam: .epochs, .goal, .max_fail, .mem_reduc,
.min_grad, .mu, .mu_dec, .mu_inc,
.mu_max, .show, .time
weight and bias values:
IW: {2x1 cell} containing 1 input weight matrix
LW: {2x2 cell} containing 1 layer weight matrix
b: {2x1 cell} containing 2 bias vectors
other:
userdata: (user stuff)
tr =
epoch: [1x101 double]
perf: [1x101 double]
vperf: [1x101 double]
tperf: [1x101 double]
mu: [1x101 double]
M=sim(net, I)
M =
Columns 1 through 5
0.0000 0.0000 0.0000 0.0000 0.0000
0.0039 0.0057 0.0161 0.0283 0.0764
0.0124 0.1029 0.1964 0.3001 0.0123
Columns 6 through 10
0.0000 0.0001 0.0000 0.0000 0.0000
0.1043 0.2950 0.0934 0.0060 0.0110
0.1033 0.0062 0.3005 0.0890 0.2009
Columns 11 through 15
0.0000 0.0001 0.0001 0.0000 0.0000
0.0315 0.0893 0.1088 0.0978 0.1145
0.3045 0.0084 0.0890 0.2009 0.3048
Columns 16 through 20
0.0000 0.0000 0.0000 0.0001 0.0001
0.1907 0.0080 0.0143 0.0688 0.1007
0.0085 0.2005 0.2989 0.0186 0.0967
Columns 21 through 25
0.0001 0.0000 0.0000 0.0000 0.0000
0.1290 0.1028 0.1899 0.1803 0.0160
0.1999 0.2992 0.0188 0.0966 0.3007
Columns 26 through 30
0.0001 0.0001 0.0000 0.0001 0.0000
0.0488 0.1080 0.0998 0.1840 0.1011
0.0085 0.1016 0.1985 0.3009 0.3047
Columns 31 through 35
0.0000 0.0000 0.0000 0.0000 0.0000
0.1961 0.1794 0.1208 0.0592 0.1003
0.0879 0.1954 0.0123 0.0960 0.2020

```

Columns 36 through 40  
0.0000 0.0000 0.0000 0.0000 0.0000  
0.1175 0.2174 0.2048 0.2291 0.2173  
0.3005 0.0123 0.1035 0.1870 0.3013

Columns 41 through 45  
0.0000 0.0000 0.0000 0.0000 0.0000  
0.0960 0.0833 0.1275 0.2236 0.2095  
0.0886 0.1999 0.3033 0.0083 0.0886

Columns 46 through 50  
0.0000 0.0000 0.0000 0.0000 0.0000  
0.2152 0.2318 0.3314 0.1047 0.0887  
0.2003 0.3040 0.0084 0.2008 0.2992

Columns 51 through 55  
0.0000 0.0000 0.0000 0.0000 0.0000  
0.2074 0.2051 0.2049 0.2017 0.3269  
0.0186 0.1024 0.2015 0.3000 0.0187

Columns 56 through 60  
0.0000 0.0000 0.0000 0.0000 0.0000  
0.2924 0.0970 0.1850 0.1938 0.1982  
0.0965 0.2994 0.0081 0.0984 0.1978

Columns 61 through 64  
0.0000 0.0000 0.0000 0.0000  
0.2160 0.3044 0.3066 0.2567  
0.3001 0.0137 0.1019 0.1992

M=round(10\*M)

M =

Columns 1 through 9  
0 0 0 0 0 0 0 0 0  
0 0 0 0 1 1 3 1 0  
0 1 2 3 0 1 0 3 1

Columns 10 through 18  
0 0 0 0 0 0 0 0 0  
0 0 1 1 1 1 2 0 0  
2 3 0 1 2 3 0 2 3

Columns 19 through 27  
0 0 0 0 0 0 0 0 0  
1 1 1 1 2 2 0 0 1  
0 1 2 3 0 1 3 0 1

Columns 28 through 36  
0 0 0 0 0 0 0 0 0  
1 2 1 2 2 1 1 1 1  
2 3 3 1 2 0 1 2 3

Columns 37 through 45  
0 0 0 0 0 0 0 0 0  
2 2 2 2 1 1 1 2 2  
0 1 2 3 1 2 3 0 1

Columns 46 through 54  
0 0 0 0 0 0 0 0 0  
2 2 3 1 1 2 2 2 2  
2 3 0 2 3 0 1 2 3

Columns 55 through 63  
0 0 0 0 0 0 0 0 0  
3 3 1 2 2 2 2 3 3  
0 1 3 0 1 2 3 0 1

Column 64

0



3  
2