Hardware Implementation of Stochastic Neural Networks

A Thesis Submitted to the Faculty of Graduate Studies in Partial Fulfillment of the Requirements for the Degree of Master of Science

Scott J. Peters

Department of Electrical & Computer Engineering
University of Manitoba

© 2004 Scott J. Peters

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION PAGE

Hardware Implementation of Stochastic Neural Networks

BY

Scott J. Peters

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree

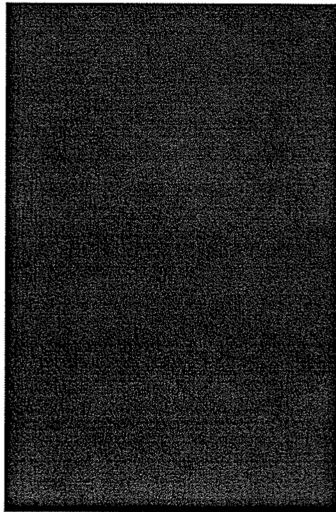
of

MASTER OF SCIENCE

SCOTT J. PETERS ©2004

Permission has been granted to the Library of The University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilm Inc. to publish an abstract of this thesis/practicum.

The author reserves other publication rights, and neither this thesis/practicum nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.



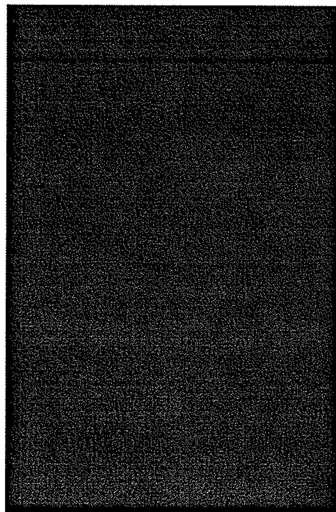
Abstract

This thesis examines the performance of artificial neural networks (ANNs) developed using stochastic arithmetic hardware. The attractiveness of stochastic arithmetic for hardware based neural computation stems from the simplicity of stochastic computation units. This allows the implementation of a fully parallel stochastic artificial neural network (SANN) that is much smaller in size than a binary radix integer (BRI) ANN.

A stochastic architecture offers several advantages over conventional arithmetic: resistance to noise, low circuit area, and variable computational accuracy without hardware changes. The trade off associated with this fully parallel architecture is an increase in communication time, due to the serial nature of data in the network. There is also variance in estimating data calculated by the network due to the random signals used to represent information. It is hoped that the advantages outweigh the disadvantages for certain applications however.

In this thesis an examination of stochastic arithmetic units and random number generators is performed. The underlying theory of neural networks is then discussed. This thesis also details modifications that are made to the conventional network model for a specialized application. Stochastic neural hardware circuits are then described and implemented in a field programmable gate array (FPGA).

This system is then tested by processing simulated visual inputs that may be encountered by a mobile robot. The results from the stochastic system are compared with a BRI implementation of an ANN. The stochastic network shows inferior performance to the BRI network, due primarily to the variance of the outputs during the training process. If the SANN is initialized with weight vectors generated by the BRI network however, the SANN performs reasonably as a classifier. The network is also tested with noise applied to the inputs, again with adequate results.



Acknowledgements

I would like to thank my advisor Dr. Dean McNeill, for his guidance in completing this thesis. He has taught me many things both in and out of school.

I would like to express my gratitude to the examining committee for their time spent evaluating this work.

Finally, I would like to thank my family and friends for their support in the good times and bad. My experiences at this university have been most memorable.

Financial support for this study was provided by the Natural Sciences and Engineering Research Council (NSERC). Computing resources were provided by the Canadian Microelectronics Corporation (CMC).

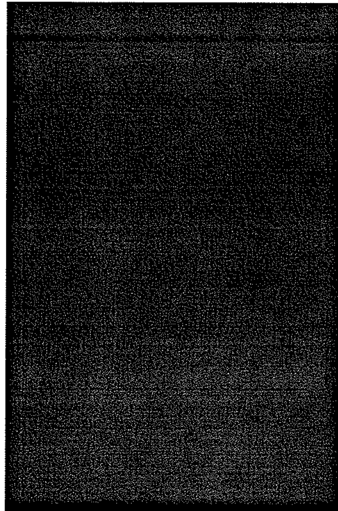


Table of Contents

Abstract ii

Acknowledgements iv

Table of Contents v

List of Figures viii

List of Tables xi

List of Acronyms xii

1 Introduction 1

2 Number Representation 3

2.1 Precision of Stochastic Signals 5

3 Stochastic Arithmetic Logic 9

3.1 Multiplication 9

3.1.1 Multiplication by -1 11

3.1.2 Squaring Operation 11

3.2 Stochastic Summation 12

3.2.1 A Problem With Multiplexors 14
3.3 Stochastic Computation Using Markov Chains 16
3.3.1 Exponential and Gaussian Functions 17
3.3.2 Hyperbolic Tangent Function 21
3.3.3 Division/Square Root Extraction 21
3.4 Stochastic Polynomials and Functions Based on Taylor Series Approximations 28

4 Random Number Generation 31
4.1 Random Number Generation Using an Analog Source 32
4.2 Random Number Generation Using Digital Sources 32
4.2.1 Linear Feedback Shift Registers 33
4.2.2 Cellular Automata 33
4.2.3 CA30 Example 35
4.2.4 CA38490 37
4.3 Variable Probability Generation 40

5 Neural Network Theory 42
5.1 Radial Basis Function Network Theory 44
5.2 RBF First Layer Network Training 47
5.2.1 Hard Competitive Learning 48
5.2.2 Frequency Sensitive Competitive Learning 49
5.2.3 Soft Competitive Learning 50
5.3 Second Layer Training 52

6 Neural Networks Based on Stochastic Hardware 53
6.1 Development Platform 53
6.2 RBF Network Model Modifications 55
6.3 Neural Network Implementation 57
6.3.1 Input Stage 57
6.3.2 Cellular Automata 58
6.3.3 'Weight' Module 59
6.3.4 Sumsquare Module 61
6.3.5 Exponential Function Module 61
6.3.6 RISM Summer Module 62
6.3.7 Stepped Velocity Divider Module 62
6.3.8 Output Integrators 63

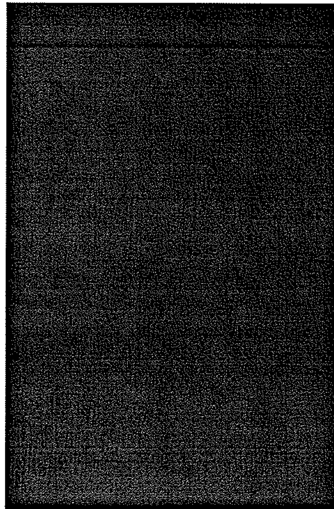
6.4 System Sequencing 63
6.4.1 Controller Software and Signals 64
6.5 Hardware Allocation and Circuit Area 65
6.5.1 Circuit Area of Stochastic Elements 65
6.6 Processing Time 67

7 Testing and Results 69
7.1 Test Problem 69
7.1.1 Network Error 69
7.1.2 Network Training Results 71
7.2 A Simulated Visual Environment 76
7.2.1 Input Generation 76
7.2.2 Results 78

8 Conclusions and Future Work 85

9 References 87

10 VHDL Source Code 90
A.1 sExp.vhd 90
A.2 svdivcounter.vhd 91
A.3 outcount_wsel.vhd 92



List of Figures

Figure 2.1: Contrasting number representations in computational machines. (a) BRI representation, (b) Stochastic representation. 4

Figure 2.2: Binomial distributions for different values of P_i. 6

Figure 2.3: Plot of *CV* vs. P_i for given values of n. 7

Figure 2.4: *n* vs. P_i for given values of *CV*. 7

Figure 3.1: Stochastic multiplication circuits. 11

Figure 3.2: Stochastic Inverter. 11

Figure 3.3: Stochastic Squaring Circuit. 12

Figure 3.4: (a) Intra-count summer (b) RISM summer. 15

Figure 3.5: Auto-correlation of the stochastic sum of five inputs. 15

Figure 3.6: Auto-correlation of the stochastic sum of eight inputs. 16

Figure 3.7: General Markov Chain State Diagram. 17

Figure 3.8: (a) State diagram approximating *exp(-2Gx)* (b) State diagram approximating $exp(-x^2)$. 18

Figure 3.9: Output of exponential function *exp(-2Gx)*. 19

Figure 3.10: Output of exponential function when applied with radial inputs of the form $||Pi-w||$. $w = 0$. 19

Figure 3.11: Output of 2-D Gaussian function. *m* = [0,0]. G = 1500. 20

Figure 3.12: Output of 2-D Gaussian function. *m* = [0.6, 0.6]. *G* = 1500. . 20

Figure 3.13: State diagram of stochastic *tanh(Nx)* function. 21

Figure 3.14: Comparison of stochastic *tanh(Nx)* and theoretical *tanh(Nx/2)*. *N = 16*. 22
Figure 3.15: Comparison of stochastic *tanh(Nx)* and theoretical *tanh(Nx/2)*. *N = 2048*. 22
Figure 3.16: (a) unipolar divider, (b) bi-polar divider, (c) square root extractor. 24
Figure 3.17: Quotient estimate convergence of stochastic unipolar divider (software simulation). 26
Figure 3.18: Quotient estimate convergence of stochastic unipolar dividers (software simulation). 26
Figure 3.19: Output of hardware unipolar divider circuit with stepped velocity. 27
Figure 3.20: Theoretical saturating division operation. 27
Figure 3.21: Stochastic polynomial circuits. (a) Bipolar, (b) Unipolar. . . . 28
Figure 3.22: Output of stochastic approximation to *Sin(x)/3*. 29
Figure 3.23: Output of stochastic approximation to *Cos(x)/3*. 30
Figure 3.24: Output of stochastic approximation to *Exp(x)/3*. 30
Figure 4.1: Stochastic bit stream source based on an analog noise diode. 32
Figure 4.2: Schematic diagram of an 8-bit maximal length LFSR. 33
Figure 4.3: Waterfall plot of 64-bit LFSR. 34
Figure 4.4: Schematic of a neighbourhood of 3 CA. 35
Figure 4.5: Truth table for Rule 30 CA, abbreviated CA30. 36
Figure 4.6: Waterfall plot of the 64-bit CA30. 37
Figure 4.7: Time plot of 64-bit CA38490. 38
Figure 4.8: Cross-correlation plot of adjacent bit streams. 39
Figure 4.9: Auto-correlation plot of bit streams. 39
Figure 4.10: 8-bit variable probability generating circuit. 41
Figure 4.11: Output from Variable Probability Generator. 41
Figure 5.1: Decision regions formed by a) an MLP network and (b) an RBF network. 43
Figure 5.2: RBF Neural Network diagram. 45
Figure 5.3: Isolation in an HCL training scheme. 49
Figure 5.4: Simplified diagram of mean vector convergence in SCL example. 51

Figure 6.1: Hardware component block diagram. 54

Figure 6.2: Nios Development Board with Stratix FPGA.
Photo ©Altera Corp. 55

Figure 6.3: Hardware component block diagram of SANN. 56

Figure 6.4: Block Diagram of 64-bit CA. 58

Figure 6.5: Weight module block diagram. 59

Figure 6.6: Sumsquare module block diagram. 61

Figure 6.7: 'Svdiv' module block diagram. 62

Figure 7.1: Sample 2-D Input Space. 70

Figure 7.2: Software simulation training results. 2-D inputs,
2 data clusters. 71

Figure 7.3: Software simulation. Network error vs. Epoch. 72

Figure 7.4: Hardware stochastic network training results. 2-D inputs,
2 data clusters. 73

Figure 7.5: Hardware stochastic system. Network error vs. Epoch. 73

Figure 7.6: Hardware Stochastic System. Neuron response from test set. 74

Figure 7.7: Training results for 3 input clusters. 75

Figure 7.8: Stochastic system response from test vectors.
3 Data clusters. 75

Figure 7.9: Sensor and light input grid used for network training. Sensors
are denoted by the red dots, training light sources are green,
test light sources are blue. 77

Figure 7.10: Output from each neuron after training. 78

Figure 7.11: Output from conventional arithmetic ANN with
additive noise. 79

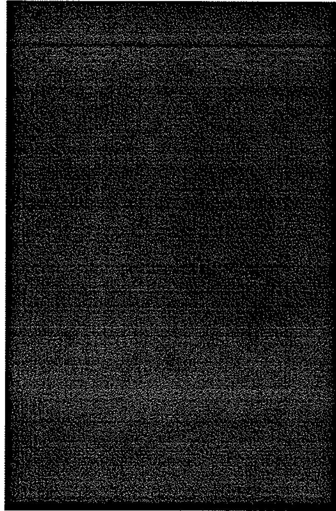
Figure 7.12: Error corresponding to training process of simulated
BRI network. 80

Figure 7.13: Output from trained stochastic network. 81

Figure 7.14: Error plot from training the stochastic neural network. 82

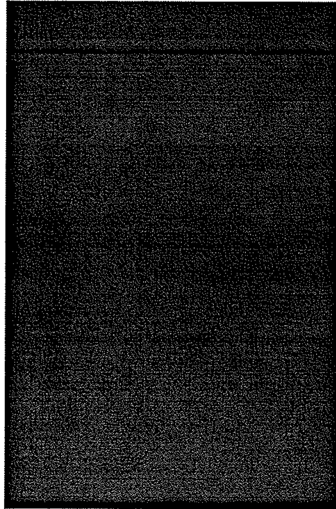
Figure 7.15: Output from pre-trained stochastic network, noise
variance is $\sigma^2 = 0.05$. 83

Figure 7.16: Output from pre-trained stochastic network, noise
variance is $\sigma^2 = 0.10$. 83



List of Tables

Table 6.1: Command Character Summary for Stochastic Control Software 64
Table 6.2: Summary of Hardware Requirements for Stochastic Logic Elements 66
Table 7.1: Averaged Response of Trained BRI Neurons. $\sigma^2 = 0.1$. 80
Table 7.2: Averaged Response of Trained Stochastic Network. $\sigma^2 = 0.1$. 84



List of Acronyms

ANN - Artificial Neural Network

BRI - Binary Radix Integer

CA - Cellular Automata

DFF - D-type Flip Flop

FSCL - Frequency Sensitive Competitive Learning

HCL - Hard Competitive Learning

LFSR - Linear Feedback Shift Register

MLP - Multi-Layer Perceptron

PRNG - Pseudo Random Number Generator

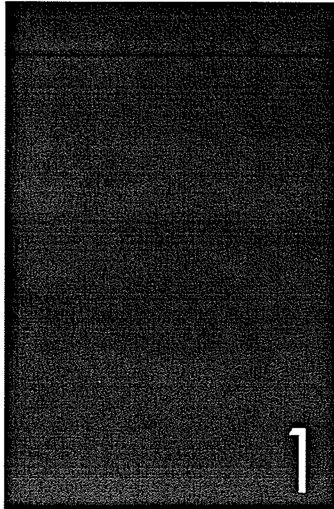
RBF - Radial Basis Function

RNG - Random Number Generator

SA - Stochastic Arithmetic

SCL - Soft Competitive Learning

VPG - Variable Probability Generator



Introduction

Stochastic computers are computers that operate on variables which are probabilistic in nature[1]. This type of variable representation allows the use of computing elements which are very simple and inexpensive to implement in a VLSI device. Stochastic computing has gained attention in recent years as an efficient tool in implementing parallel computation systems. Stochastic arithmetic allows many simple processing units to work in parallel in order to process data. Advantages of this approach are low power consumption, resistance to noise, small circuit area, and simplicity of computational elements. While the proliferation of low-cost microprocessors may make this irrelevant for some applications, there is a class of computational system which would greatly benefit from such a massively parallelizable architecture. These are artificial neural networks (ANNs).

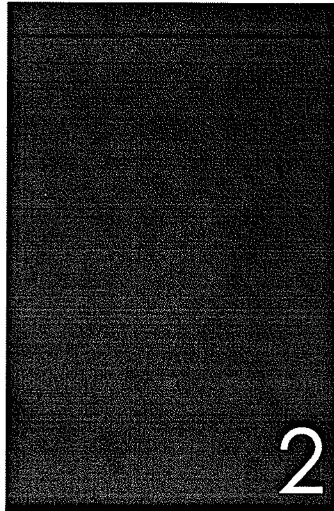
Artificial neural networks require many parallel processing units, which in a conventional computational environment are extremely expensive in terms of silicon area. In particular, the multiplication operation requires a very large circuit using conventional binary radix integer (BRI) architectures. BRI multipliers require thousands of gates, while stochastic arithmetic (SA) multipliers require only a single AND gate. In ANN applications there are a large number of multiplication operations that need to be performed, which creates bottleneck issues in a system that has a single multiplier. Using stochastic arithmetic will allow more multipliers to be implemented on a single device.

Stochastic processing also has the advantage of being able to vary the accuracy of a computation without changing hardware. Conversely, processing time may be increased to obtain more accurate results. This means that if less accurate results are acceptable, the system can produce these results at a higher rate. Conventional BRI systems have a set precision based on processor implementation (governed by the width of the data bus), and varying this is a complicated process that requires arithmetic and logical units to be completely redesigned.

Stochastic signals require just a single connection to transmit data, as opposed to multiple line buses of conventional computers. While it does take more time to transmit a data value serially (directly dependent on the precision desired in the system), the circuit area required to transmit data is greatly reduced.

The main disadvantage of SA systems is the variance inherent in estimating a random signal which will degrade results.

Previous work in this area has yielded various results. Different schemes of number representation have been studied ([1], [9], [13]), detailed analysis of multiplexing schemes have been performed [16], and stochastic systems have been simulated in software [5]. Systems have also been implemented in hardware using various VLSI configurations ([17], [18], [22], [25]). Different training methods have also been implemented and tested ([5], [10], [11]). Also the effects of multiple processing steps on stochastic bit streams have been examined ([15], [21]). A key element in all of the implementation studies is that all of the systems are easily expandable to accommodate any size of neural network. This related work is surveyed in greater detail in the following sections.



Number Representation

The uniqueness of stochastic computation is founded on the way numbers are represented. Conventional computing systems use the BRI format to represent numeric data. This BRI format allows a high data density, where each bit has a different significance or weight in the total value of the number. This is where a stochastic system differs.

Data in a stochastic computer is encoded as a serial stream of bits, where each bit has the same significance in the total value of the number. These bits are generated by a pseudo-random number generator (PRNG). Each bit thus has a probability that it will be a '1' or a '0'. This probability is termed the *generating probability.* The bit streams are used to encode the generating probability.

Figure 2.1 shows the value 6 represented as a conventional 4 bit BRI number, and equivalent representations in the stochastic format. Note that there are multiple ways to represent the same number using the stochastic format. In the example, the value '6' is represented by the stochastic bit stream due to the fact that there are six '1's in the stream of 16 bits. Of course, as with any random process there will be some error with respect to the exact value generated. The number represented in a stochastic stream is interpreted as a probability value, as opposed to an integer value.

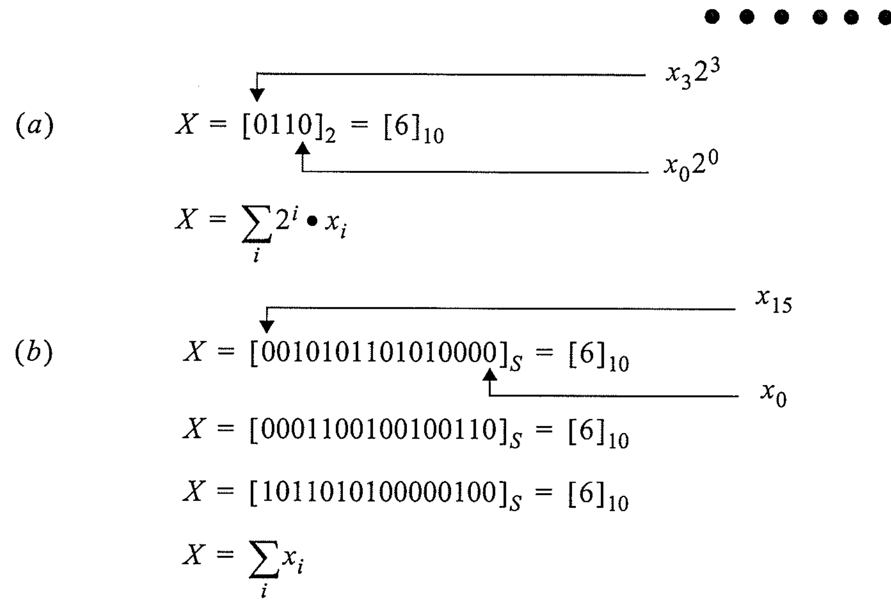


Figure 2.1: Contrasting number representations in computational machines. (a) BRI representation, (b) Stochastic representation.

A Bernoulli process is a single random event that has a binary outcome (e.g. flipping a coin). Each bit is generated as a Bernoulli process, with probability P_i that it will be '1'. Consider a single bit as a binary random variable x_i [12]. The formula for the Bernoulli process is:

$$p(x_i) = P_i^{x_i}(1 - P_i)^{1 - x_i} \tag{2.1}$$

If this is one element in a vector of n uncorrelated random variables then the probability of the complete vector $\boldsymbol{x}$ is given by:

$$p(x) = \prod_{i=1}^{n} P_i^{x_i}(1 - P_i)^{1 - x_i} \tag{2.2}$$

The length of the bit stream is referred to as the *integration interval*, because this is the interval over which the bit stream is integrated to form an estimate of its value. In most cases, many more than 64 bits are used to estimate the value P_i. When the integration interval is long enough and the Bernoulli probability is assumed to be constant for the entire stream, then the signal can be estimated by:

$$P_i \approx \frac{k}{N} \tag{2.3}$$

Here k represents the number of events ('1's or pulses in this case), and N is the total number of bit cells (Bernoulli trials) in the integration interval.

There are different ways to interpret the count of pulses in a stream. If the values are viewed as strictly representing values from $(0 \rightarrow 1)$, then the representation is termed *unipolar.* The magnitude of this vector is encoded as $S_i^U = P_i$.

Though useful in and of itself, the interpretation can be extended to represent values in the range $(-1 \rightarrow 1)$ by using a simple transformation:

$$S_i^B = 2S_i^U - 1 \cong 2P_i - 1 \cong 2\frac{k}{n} - 1 \tag{2.4}$$

This is termed *bipolar representation.*

2.1 Precision of Stochastic Signals

The probability of k pulses in n bit cells is a binomial distribution given by:

$$p_k(k) = C(n, k)P_i^k(1 - P_i)^{n-k} \tag{2.5}$$

where $C(n, k)$ is the number of possible combinations of k elements from a set of n elements:

$$C(n, k) = \frac{n!}{k!(n-k)!} \tag{2.6}$$

The mean and variance of this distribution are:

$$\mu_k = \sum_{k=0}^{n} p_k(k)k = nP_i \tag{2.7}$$

$$\sigma_k^2 = \sum_{k=0}^{n} p_k(k)(k - \mu_k)^2 = nP_i(1 - P_i) \tag{2.8}$$

Figure 2.2 shows binomial distributions for various values of P_i. The variance in the estimate of P_i (the *primary statistic*) can be clearly seen.

The coefficient of variation measures the precision of the estimate of P_i [12]. The coefficient of variation is defined as:

$$CV = \frac{\sigma_k}{\mu_k} = \sqrt{\frac{1-P_i}{nP_i}} \tag{2.9}$$

This equation is solved for n to determine the required count interval for a desired precision. For the unipolar case:

$$n = \frac{1-P_i}{P_i(CV)^2} = \frac{1-S_i^U}{S_i^U}\left(\frac{S_i^U}{n_i^U}\right)^2 \tag{2.10}$$

and for the bipolar case:

$$n = \frac{P_i(1-P_i)}{(P_i-0.5)^2}\left(\frac{S_i^B}{n_i^B}\right)^2 = \frac{(1+S_i^B)(1-S_i^B)}{(S_i^B)^2}\left(\frac{S_i^B}{n_i^B}\right)^2 \tag{2.11}$$

Note that for a lower value of p_i a higher value of n is required for the same accuracy. Figure 2.3 and Figure 2.4 show the relationship between the coefficient of variation and P_i, and the required integration interval for a desired accuracy for the unipolar case.

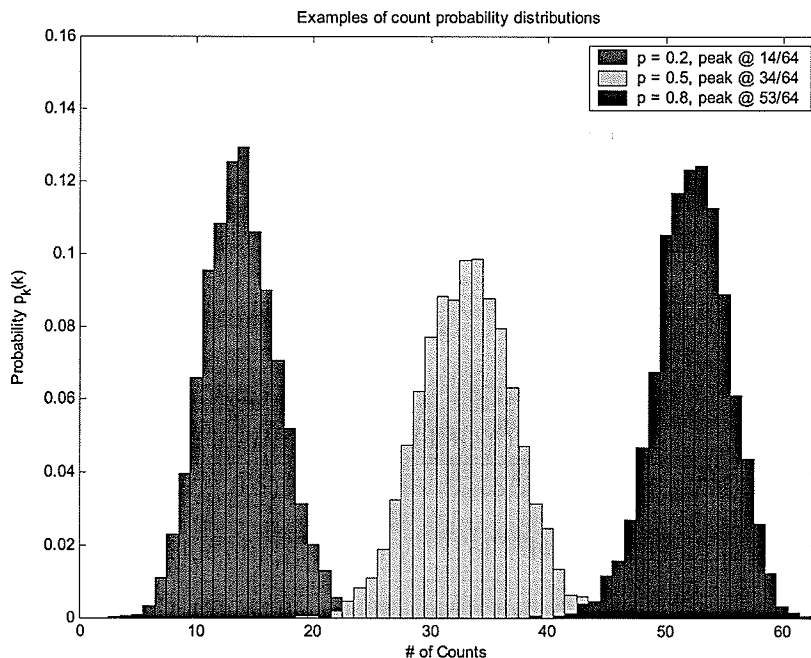


Figure 2.2: Binomial distributions for different values of P_i.

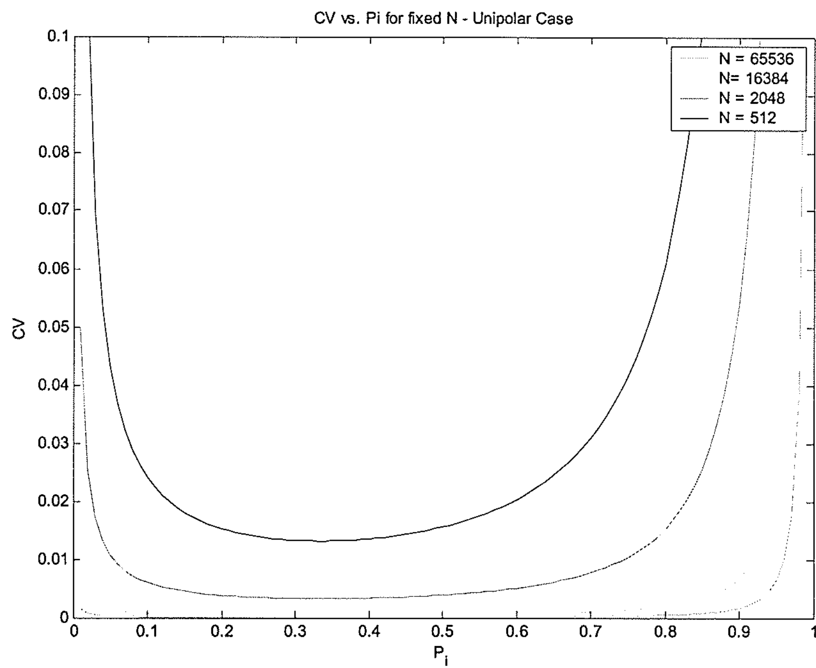


Figure 2.3: Plot of CV vs. P_i for given values of n.

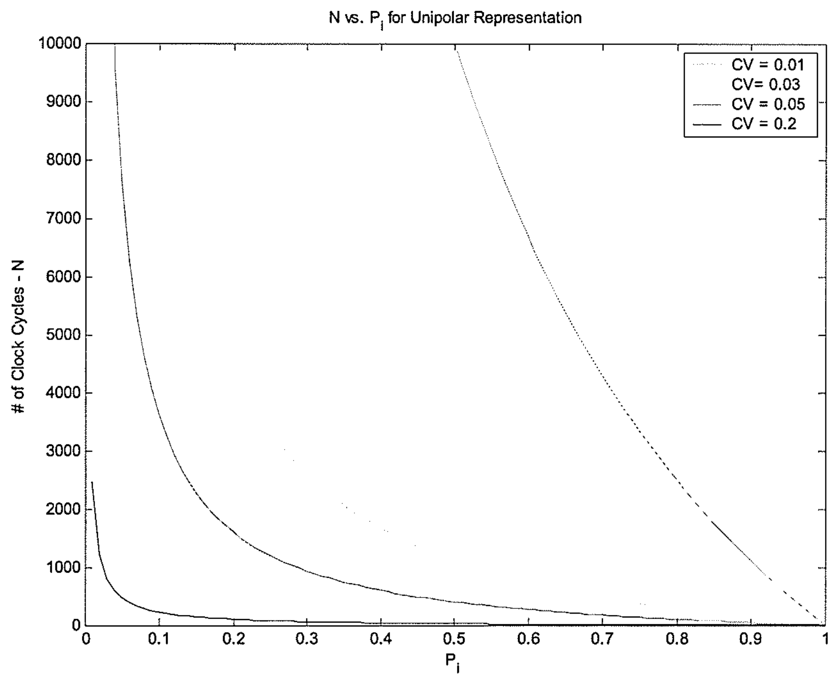


Figure 2.4: n vs. P_i for given values of CV.

Since each bit in the BRI format has a different significance, the maximum error that can occur for a single bit error is $\pm 2^{N-1}$ in an N-bit number. In the stochastic case, the maximum error in the total value for a single flipped bit is 1 where N is the number of bits in the stream. This property makes stochastic computation an attractive option in a noisy environment. This is noise is acceptable in an ANN application due to the fact that the benefit of a quickly computed approximate answer can yield higher performance than a very accurate answer arrived at slowly.

Stochastic Arithmetic Logic

The majority of stochastic arithmetic operations can be performed by using simple logic gates and counters, as opposed to complicated digital circuits such as multipliers which can require thousands of logic gates. Using circuits of such large size in a neural network application is prohibitive in terms of silicon real estate. Considering this, we will begin our discussion of stochastic arithmetic hardware at the simplest level in terms of circuit complexity, the multiplier.

3.1 Multiplication

The basic function of an ANN is to computer a weighted sum of an input vector with a parameters of the network. In the worst case, ANNs require $O(n^2)$ multiplication operations, where n is the number of neurons in the network. BRI multipliers are very complex circuits, hence there are very few of them in a general purpose processor (usually only one).

Using a stochastic architecture, multiplication is the simplest operation to perform due to the probabilistic nature of the number format. A simple AND gate is all that is required to perform unipolar multiplication, and a simple XNOR gate can perform full four quadrant (signed) multiplication for bipolar values [1]. Consider two stochastic bit streams, A and B. The generating probability for each stream is:

$$P(A = 1) = A \tag{3.1}$$
$$P(B = 1) = B \tag{3.2}$$

meaning that bits in stream A have probability A of being a '1' and similarly for stream B. For the unipolar case:

$$P(Y = 1) = Y = AB \tag{3.3}$$

and for the bipolar case:

$$Y = AB + (1 - A)(1 - B) \tag{3.4}$$

Since in the bipolar case:

$$P(A = 1) = \frac{[A + 1]}{2} \tag{3.5}$$

$$P(B = 1) = \frac{[B + 1]}{2} \tag{3.6}$$

we have:

$$Y = 2P(Y = 1) - 1 = \frac{(AB + 1)}{2} \tag{3.7}$$

where Y is the output in both cases. In the unipolar case, the output of the AND gate is '1' if and only if both of the input values are '1'. These input values are '1' with a probability of A and B respectively. If these input streams are independent, then the output probability if the product of the two input probabilities.

If both inputs to the multiplier are Bernoulli sequences, the output of a multiplier is still a Bernoulli sequence. However if one of the inputs is not a Bernoulli sequence, the output will still have the correct primary statistic but will not be a Bernoulli sequence [5].

It is important to determine the statistics of the resulting signal from an arithmetic operation such as multiplication. The expected value of each input is given by the expression for the mean of a binomial process, which says that the average number of events (in this case, the number of '1's) is equal to the probability of generating a '1' times the number of trials:

$$E(X_i) = np_i \tag{3.8}$$

The expected value of two multiplied streams of bits can be modelled as a binomial process with the value of n being the same but with the value of p_i being the product of the two input probabilities $p_o = p_1 p_2$ [1]. The expected value of the output is then:

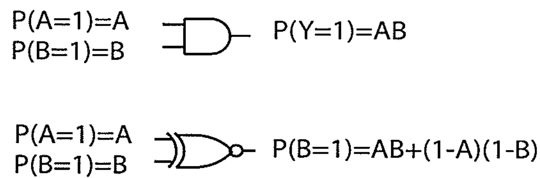


Figure 3.1: Stochastic multiplication circuits.

$$E(X_o) = np_1p_2 \tag{3.9}$$

Circuit diagrams of stochastic multipliers are shown in Figure 3.1

3.1.1 Multiplication by -1

To multiply by -1, a NOT gate is required. This is obviously only applicable in the bipolar representation, because there are no negative numbers in the unipolar interpretation. This is best shown with a numerical example.

For instance if an input stream has:

$$p_i = 0.6^B \rightarrow 0.8^U \tag{3.10}$$

then the output stream will be given as:

$$p_o = 1 - p_i^U = 0.2^U \rightarrow -0.6^B \tag{3.11}$$

The variance of the output stream is the same as the variance of the input stream. Figure 3.2 shows a circuit diagram of a stochastic inverter.

3.1.2 Squaring Operation

While it is a simple operation to multiply two numbers together, multiplying a bit stream by itself is slightly more difficult. In the case of unipolar arithmetic, applying the same input stream to both ports of an AND gate will give an output stream that is identical to the

Figure 3.2: Stochastic Inverter.

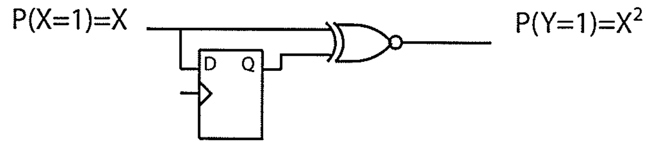


Figure 3.3: Stochastic Squaring Circuit.

input stream. In the case of bipolar arithmetic, using the same stream as input to both ports of an XNOR gate will result in an output stream of '1's. This is why it is important for input streams to be independent for the arithmetic units to function properly. To produce independent bit streams, a stochastic delay element is used. This is accomplished by using a D-type flip flop. A bipolar squaring circuit is shown in Figure 3.3. The XNOR gate is substituted for an AND gate to square unipolar bit streams.

3.2 Stochastic Summation

There are multiple techniques that can be used to perform the summing operation ([1], [5]). The simplest method is to use an OR gate to perform an approximate addition:

$$Y = A(1-B)+B(1-A)+AB \tag{3.12}$$

$$Y = A+B-AB \tag{3.13}$$

The approximation is accurate for small values of A or B, but deteriorates if both are large.

A more accurate way to perform addition is to use a multiplexor (MUX) with stochastic selector signals. Since probability values are the quantities represented, the output stream probability cannot be greater than 1. This gives stochastic addition an inherent averaging property:

$$y = \frac{1}{N}\sum_i p_i \tag{3.14}$$

where N is the number of inputs.

Stochastic summations can be viewed as a Poisson process. Poisson processes model independent events at a certain location (i.e. a stream of independent bits arriving at the input of a MUX). Assume that a stream of pulses arrives at a MUX input with rate λ in a Pois-

son process. The pulses are collected over a time period T, and the probability of k pulses is given by:

$$p(k) = \frac{(\lambda T)^k e^{-\lambda T}}{k!} \tag{3.15}$$

with the expected number of pulses in an interval given by $E(k) = \lambda T$. The Poisson distribution is a valid approximation to the binomial distribution of a bit stream provided that the integration interval is large [16]. When m of these independent streams are summed (with rates $\lambda_1, \lambda_2, \ldots, \lambda_m$) the total rate is given by:

$$\lambda_{total} = \sum_{i=1}^{m} \lambda_i \tag{3.16}$$

Since these streams are stochastically multiplexed, the average rate is observed at the output of the MUX.

There are two different approaches that can be used to combine stochastic bit streams that have been defined by Card [16]: *inter-count multiplexing* and *intra-count multiplexing*. Both of these methods can be implemented in hardware using a multiplexor, driven by $\log_2 m$ select lines.

In the case of inter-count multiplexing, each stream is counted for an equal proportion of the counting interval, then the counted stream is switched. Thus, all streams are guaranteed to contribute equally to the final sum. In the case of intra-count multiplexing, the counted stream is switched randomly after each clock cycle. Provided the integration interval is long enough, all inputs contribute equally to the total rate of the output stream.

The main difference between these techniques is that in the case of inter-count multiplexing, the output rate is different depending on which stream is selected at the current time, giving a different pulse rate on the output. Another drawback of the inter-count method is the increased variance due to the doubly stochastic process, though the primary statistic is preserved.

In the case of intra-count multiplexing, the probability of a certain stream being selected, and a pulse being present on that stream on the same clock cycle, are independent. On average, the rate of output pulses is the same, given by the average of the input probabili-

ties. The main drawback of the intracount method is that an extended count interval is required due to the fact that all input permutations are required (though in a high speed computational system this is not a significant factor). The result of this fundamental difference is that the output of an intra-count multiplexor is still a Poisson process, while the output of an inter-count multiplexor is not.

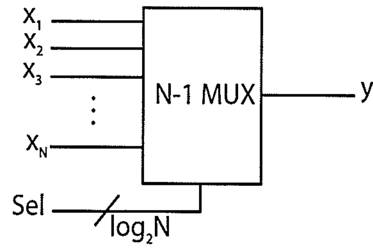
3.2.1 A Problem With Multiplexors

A proposed way of adding numbers in a stochastic system is to present all of the input streams to a multiplexor and choose a different binary stream at each clock cycle. But what happens when the number of input streams are not present in powers of 2?

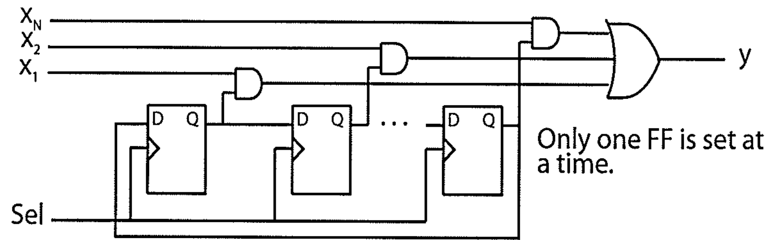
There are two potential solutions to this problem. First, zero value streams could be applied to MUX inputs that aren't needed (a constant '0' for unipolar, a $p = 0.5$ stream for bipolar). This will reduce the total value of the output rate due to the averaging process, but will approximate the result with a purely combinational circuit.

A better solution [5] is to use an N-to-1 switch where N is the number of inputs to be summed. This will be termed the *Random Incremental Selection Method* (RISM).The switch controlling the output stream is selected by a single random signal which would change the selected input if it is a '1' (selecting the next stream sequentially), or not changing it if it is a '0'. If the selector stream has a generating probability of $p = 0.5$ it is observed through software simulation that the channel selection distribution is uniform (which is desired) for large integration intervals. A diagram of an N-1 MUX and a RISM summer are shown in Figure 3.4.

Software simulation has also shown that the error in the approximated sum is the lowest for RISM adders. Auto-correlation for the intercount summer output is the highest of the three methods, meaning that the output stream is less random in nature compared to the other two count methods. The results from summing 5 streams are shown in Figure 3.5. Results from summing eight streams are seen in Figure 3.6. The significance of these two plots is to show the effects of using a MUX type summer when inputs are not present in a power of 2.



(*a*)



(*b*)

Figure 3.4: (a) Intra-count summer (b) RISM summer.

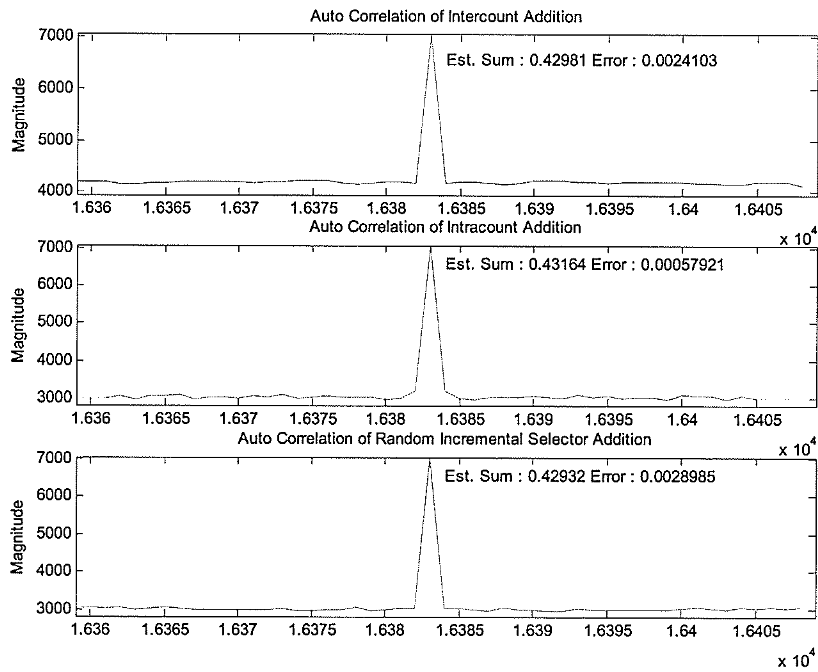


Figure 3.5: Auto-correlation of the stochastic sum of five inputs.

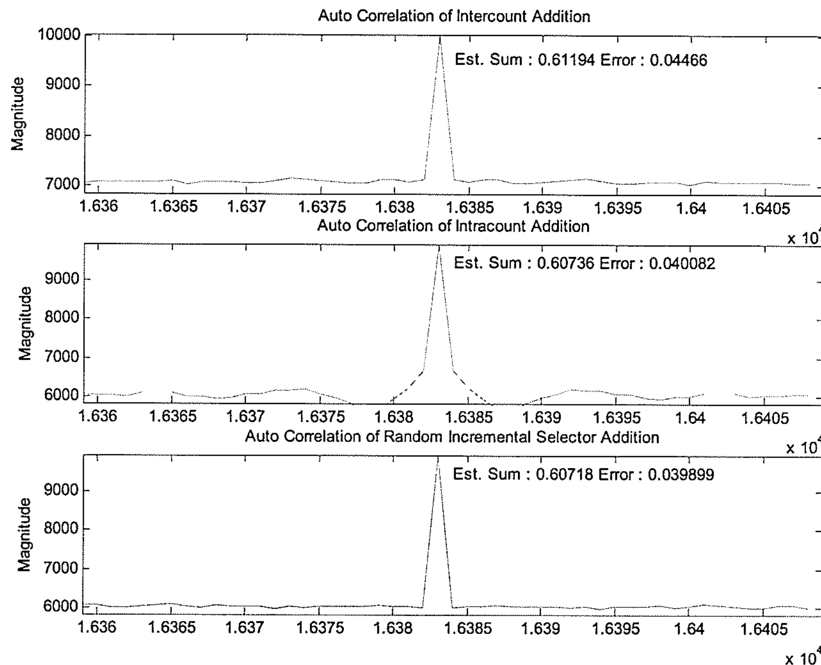


Figure 3.6: Auto-correlation of the stochastic sum of eight inputs.

This data was produced using a software simulation to randomly generate the operands, and stochastically estimate the sum. The error value indicated is the difference between the estimated sum and the calculated sum.

In the final implementation of this system, the intra-count multiplexing system was used where there are two inputs to be summed. This is due to the simplicity of a 2-1 MUX circuit. Where more than 2 inputs are required the RISM selector is used. The RISM technique also has the advantage of requiring a single select line, although the circuit is more complex.

3.3 Stochastic Computation Using Markov Chains

Aside from simple arithmetic operations, more complex mathematical functions can also be computed stochastically using sequential computation units. The most common building block for these units is the stochastic counter, which is a digital counter that saturates at either end of its count cycle. This means that if the maximum count is reached, the counter does not overflow to zero on the next increment cycle, similarly there is no under-

flow when decrementing from a zero state. These counters can be used to implement a Markov chain or an error counter that have interesting applications in stochastic computation.

The state transition diagram of a generic Markov chain is detailed in Figure 3.7. The machine is initialized to some state, and traverses the state space based on a random input with probability p. On each clock cycle, if the input is '1' the state will move up the chain, or if the input is '0' the state will move down the chain. The value of p can be a function of several variables that vary the state transition probabilities depending on the current state of the machine. Different mathematical functions can be realized by using simple combinational logic on the outputs of the counter, or by changing the probability with which the counter will increment or decrement.

3.3.1 Exponential and Gaussian Functions

The implementation of a radial basis function (RBF, defined in chapter 5) ANN requires a function that varies based on the 'distance' (usually Euclidean) of the input to a certain target parameter. While there are many functions that can accomplish this task, the most commonly used is the Gaussian function [2]. There are two potential implementations of this in a stochastic system. Firstly, a Gaussian function can be directly approximated [20]. Secondly, an exponential function can be implemented and supplied with a distance varying input [5].

These methods both use a stochastic counter combined with output logic. The precision of the approximation of each function can be increased by increasing the number of states in

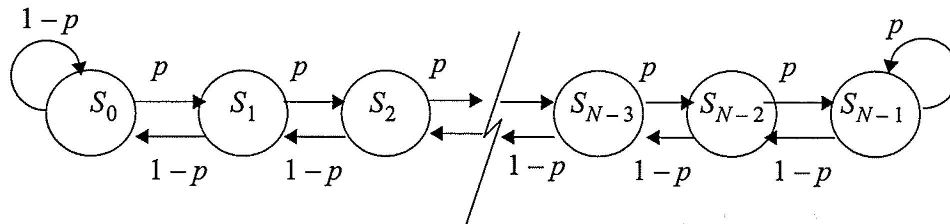


Figure 3.7: General Markov Chain State Diagram.

the counter, although this comes at the cost of requiring more clock cycles for the counter to converge to a steady state [19]. The exponential function has a software variable gain parameter, while the Gaussian function does not (it is fixed in hardware). Also, the approximation of the exponential function requires that the number of states N be significantly greater than the gain parameter G [5]. The state machine of each function is shown in Figure 3.8.

The input for both functions must be encoded as a bipolar signal, and the output is encoded as a unipolar signal.

Figure 3.9 shows the output of the exponential function with $N = 2048$ for various values of G. Figure 3.10 shows the same exponential unit with the input representing a distance measure between two variables. The function saturates at low values of G, resulting in a wide variance. Figure 3.11 and Figure 3.12 are exponential functions implemented as 2-D Gaussian functions. The mean of the first function is at [0,0] and the second function is

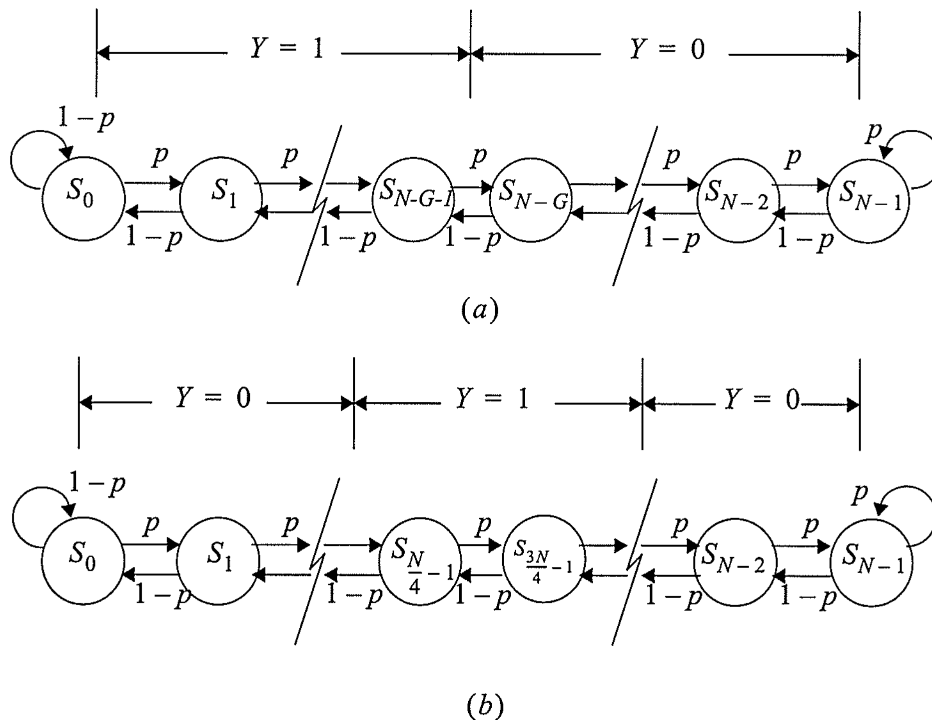


Figure 3.8: (a) State diagram approximating exp(-2Gx) (b) State diagram approximating exp($-x^2$).

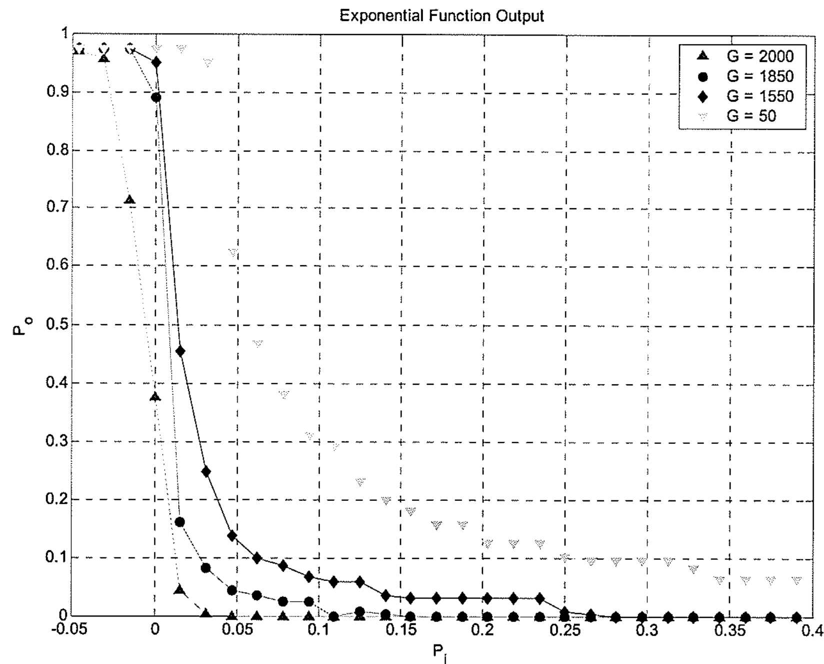


Figure 3.9: Output of exponential function exp(-2Gx).

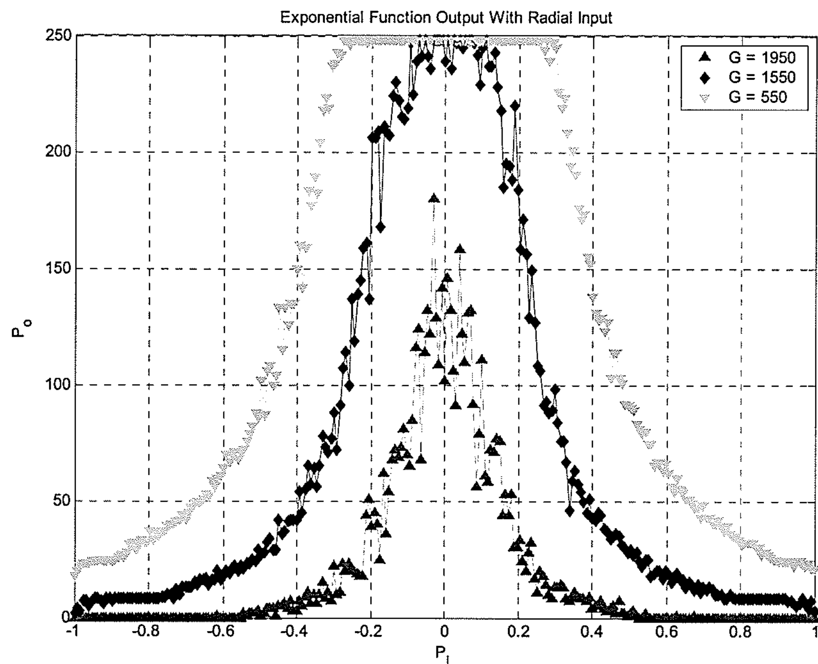


Figure 3.10: Output of exponential function when applied with radial inputs of the form $||P_I\text{-}w||$. $w = 0$.

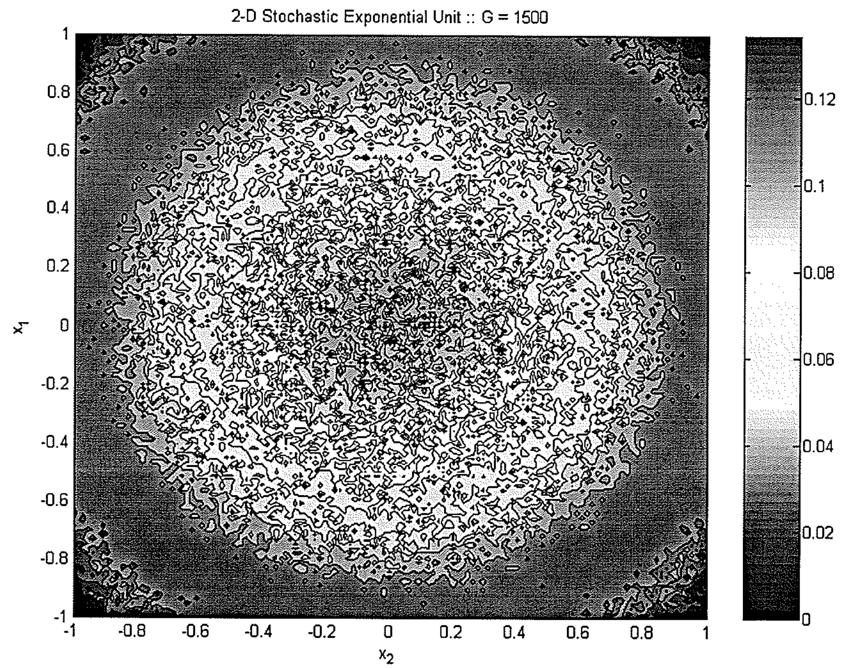


Figure 3.11: Output of 2-D Gaussian function. $\mu = [0,0]$. $G = 1500$.

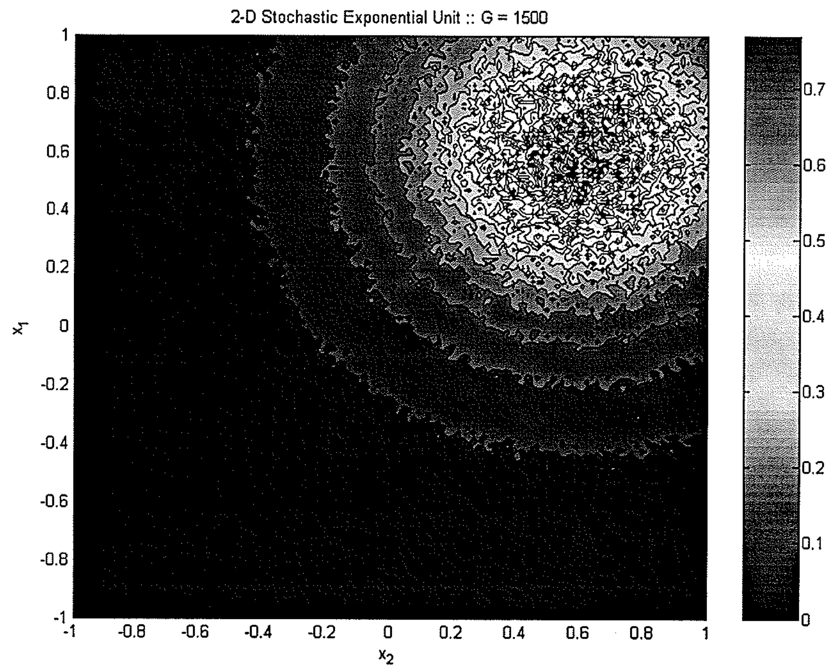


Figure 3.12: Output of 2-D Gaussian function. $\mu = [0.6, 0.6]$. $G = 1500$.

[1,1]. Output from the Gaussian function is not shown because it is not used in the final implemented system. Data for these plots was collected from a hardware implementation of this module, not software simulation.

3.3.2 Hyperbolic Tangent Function

A hyperbolic tangent function can be approximated in a similar fashion to the exponential function. Both the input and output of this function are encoded as bipolar stochastic signals. The approximation is valid for smaller values of N (< 256), and is poorer for high values. The state diagram for a hyperbolic tangential function can be seen in Figure 3.13. The output of a tangential function with a gain parameter of $N = 16$ is shown in Figure 3.14. Figure 3.15 shows the output when the value of N is increased. The results shown here are for a tangential function simulated in software, although the hardware circuits were also implemented and the results are comparable. The tangential function is primarily used for multi-layer perceptron neural networks that are beyond the scope of this project.

3.3.3 Division/Square Root Extraction

Division is a difficult operation to approximate accurately using stochastic arithmetic. This is due to the fact that the range of values is very limited, and division by a sufficiently small number will easily produce a result outside the allowable range.

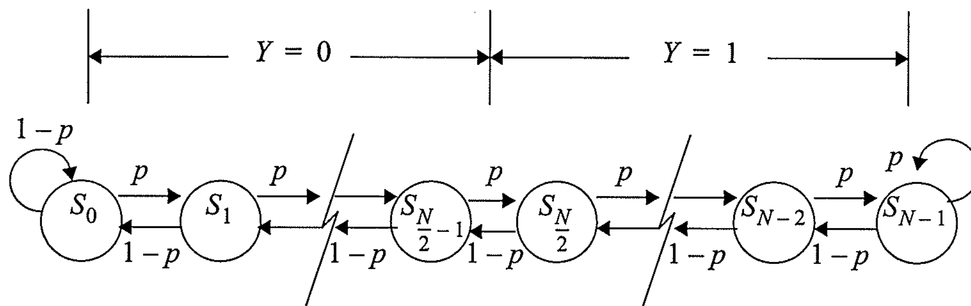


Figure 3.13: State diagram of stochastic tanh(Nx) function.

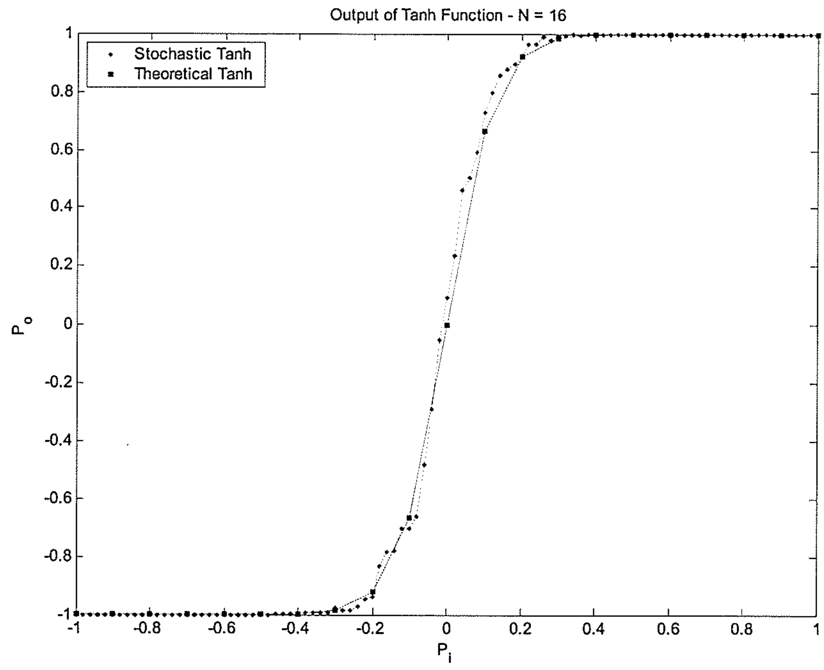


Figure 3.14: Comparison of stochastic tanh(Nx) and theoretical tanh($Nx/2$). $N = 16$.

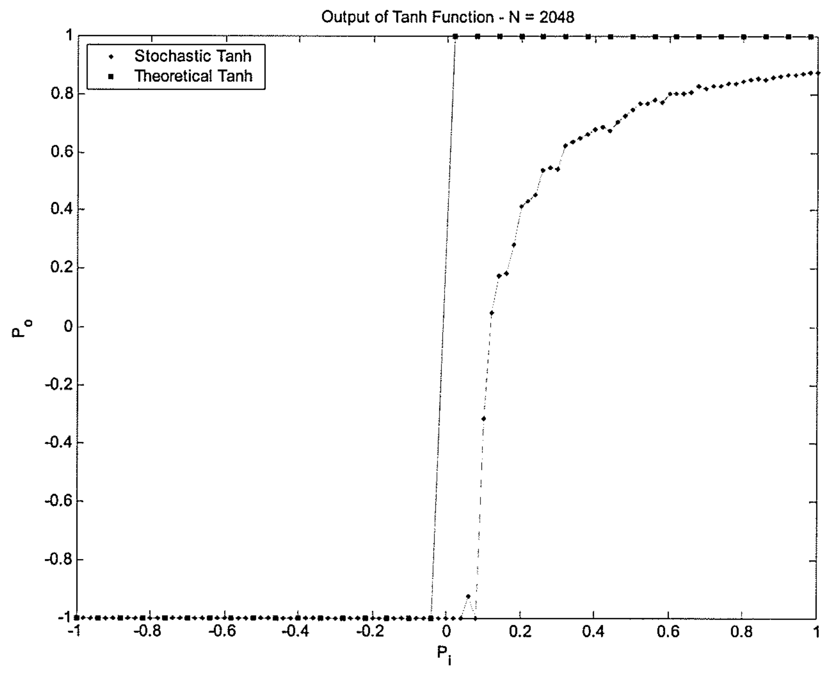


Figure 3.15: Comparison of stochastic tanh(Nx) and theoretical tanh($Nx/2$). $N = 2048$.

Other techniques of number representation such as the ratio method ([1], [13]) can represent values from $(-\infty, \infty)$. This number format allows division to be implemented with an inverter and a multiplier. However, this format also makes other arithmetic circuits more complicated to implement. A higher precision of generating probability is also required for this method due to the fact that the range of numbers is much greater.

The basic technique for stochastic division in this coding scheme is to use a stochastic counter to represent an error between the estimation of the quotient and the real quotient [1]. Suppose that the output of a unipolar stochastic counter is assumed to be p_q, which is an approximation to p_x/p_y. This means that $p_q p_y$ should be equal to p_x. We can then define an error function:

$$e = p_y p_q - p_x \tag{3.17}$$

If the error function is squared, then it is always positive and bounded below by 0. If p_q changes such that its derivative is negative, e will eventually be forced to zero. Taking the derivative of the squared error function with respect to time (or equivalently p_q which is the only stochastic process that varies with time) we arrive at the following expression:

$$\frac{d(e^2)}{dt^2} = 2e\dot{p}_q p_y \tag{3.18}$$

The derivative of the square of the error function must always be less than zero for the error to reduce. Since p_y is always positive (unipolar divider), e and $\dot{p}_o$ must be chosen to have opposite signs. This is realized as follows:

$$\dot{p}_o = -\alpha e = -\alpha(p_2 p_o - p_1) \tag{3.19}$$

where α is a constant. If the product of $p_q p_y$ is formed and applied to the decrement line of the counter, then this relation will hold. Eventually the system will reach a steady state of:

$$p_y p_q = p_x \Rightarrow p_q = p_x/p_y \tag{3.20}$$

The same logic can be applied to derive circuits for a bipolar divider and a square root extractor. However, since these circuits are not used in this application, they will not be examined further.

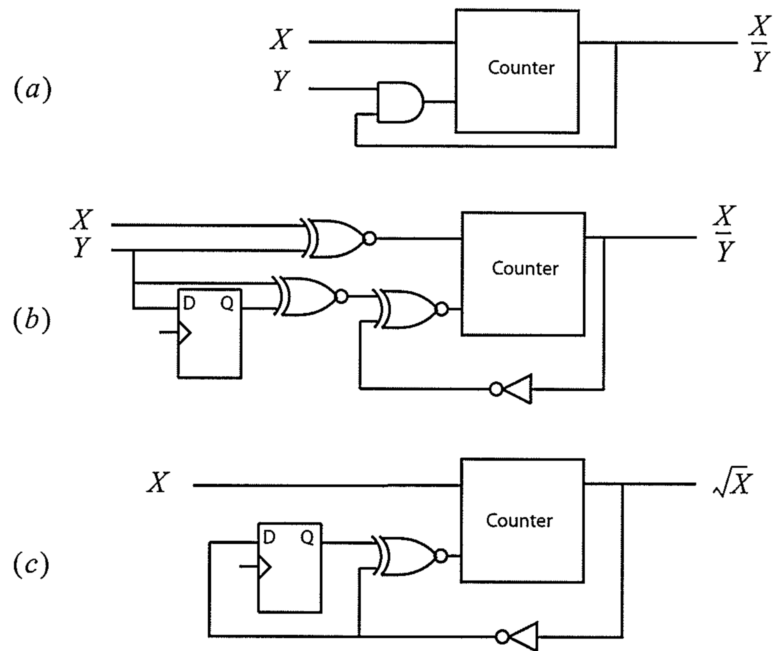


Figure 3.16: (a) unipolar divider, (b) bi-polar divider, (c) square root extractor.

The division operation can be extremely slow to converge if the divisor is small (this is the input that determines if the counter will decrement). This is because many clock cycles can occur before a state transition occurs. The convergence time will be greatly increased, and the error, when estimating the quotient, will be high (because the integration interval is fixed).

A solution to this problem has been proposed and simulated in software by Brown [5]. Further efforts have produced a hardware version of the circuit that is used in this thesis research. Block diagrams of divider circuits and a square root extractor are shown in Figure 3.16.

The proposed technique uses what Brown refers to as *stepped velocity division*. The major feature of this technique is that instead of incrementing or decrementing the counter by 1, the state of the counter varies by a dynamic modulus. The algorithm is as follows:

1.The main count register 'CNT' is initialized to its maximum value (MAX), a register containing the variable 'SCHEDULE' is initialized to MAX/8, and a third register called C_INT initialized to '0'.

2.After each clock cycle, if the increment line is '1' and the decrement line is '0' then SCHEDULE is added to 'CNT'. If the increment line is '0' and the decrement line is '1' then SCHEDULE is subtracted from CNT. If both the increment and decrement lines are '1' or '0' then CNT remains unchanged.

3.If the increment line is '1' and the decrement line is '0' then SCHEDULE is also added to C_INT.

4.If C_INT = 0 (overflow) then SCHEDULE is divided by two. This process is repeated until SCHEDULE becomes '0'.

The main counter covers the same range of values at each stage (value of the SCHEDULE register) due to the fact that C_INT takes longer to overflow as SCHEDULE is reduced.

The total quotient is the integral of the stochastic stream with $P_i(t) = CNT(t)$. Since this technique converges much more quickly than the conventional method, the error of the quotient is much lower. Evidence of this can be seen in Figure 3.17 and Figure 3.18. Where two different quotients are evaluated for 16384 clock cycles. The error of the stepped velocity technique is observed to be significantly lower than that of the conventional (fixed velocity) method due to the decreased convergence time.

Figure 3.19 shows the output of a unipolar stepped velocity divider implemented in hardware (the processing time is 16384 clock cycles). For reference, a plot of the theoretical results for this function can be seen in Figure 3.20.

The stepped velocity technique can also be applied to square root extraction, and any other computational element involving error minimization.

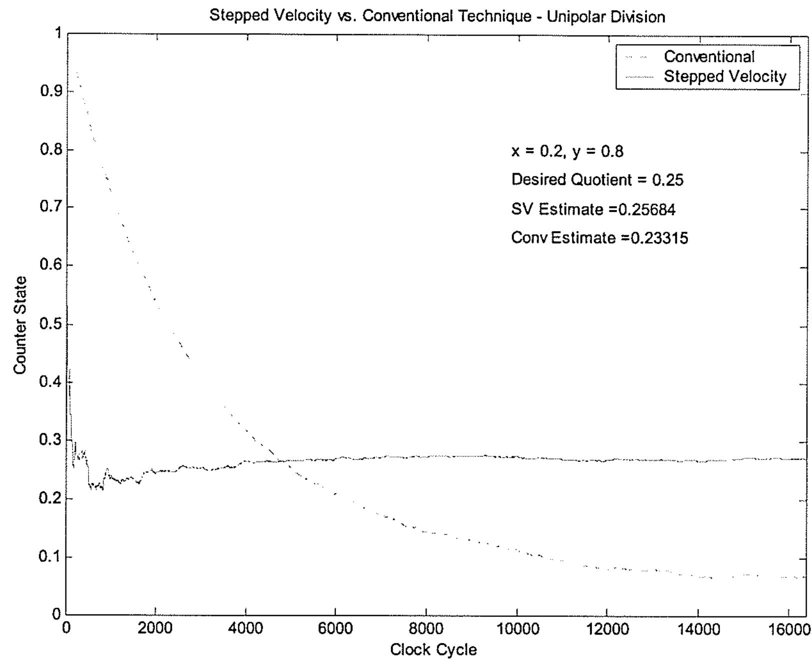


Figure 3.17: Quotient estimate convergence of stochastic unipolar divider (software simulation).

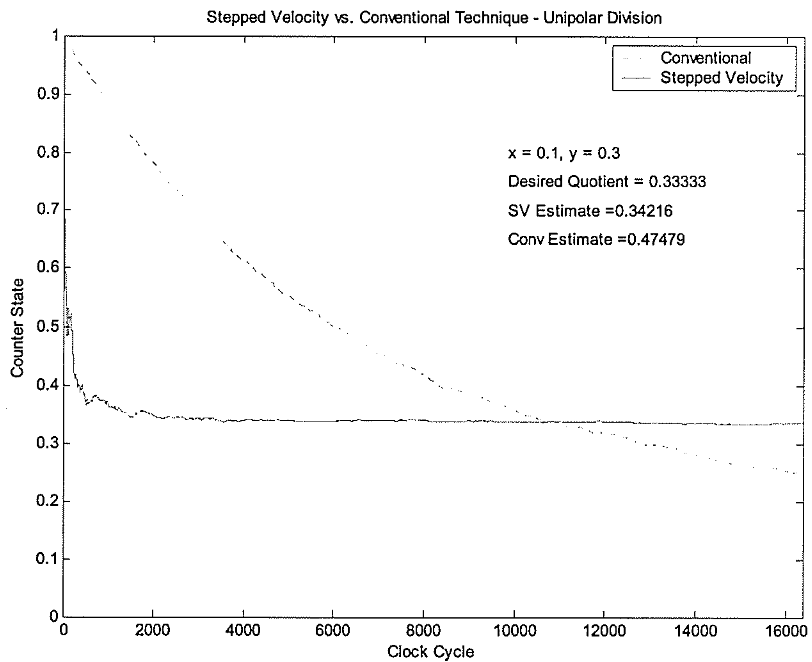


Figure 3.18: Quotient estimate convergence of stochastic unipolar dividers (software simulation).

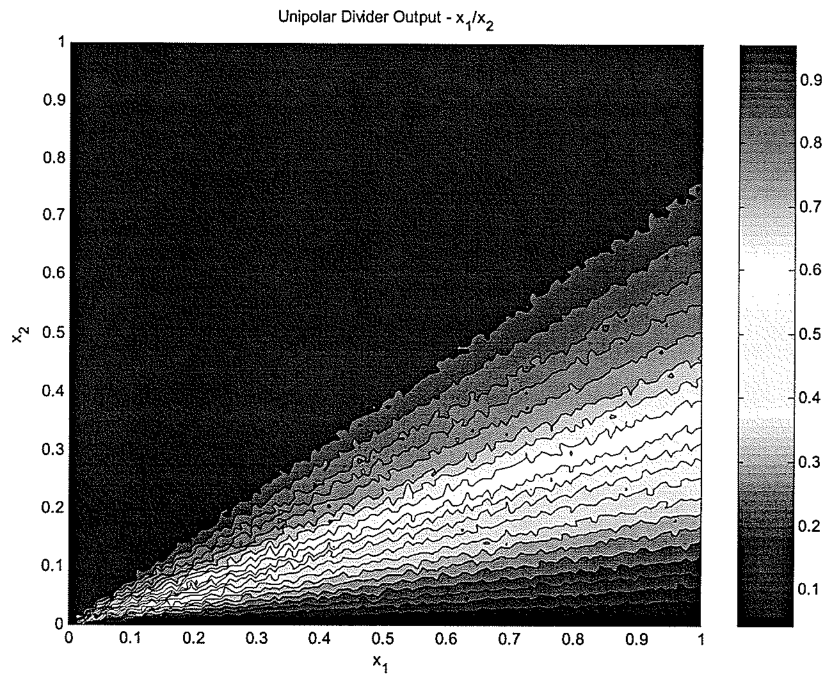


Figure 3.19: Output of hardware unipolar divider circuit with stepped velocity.

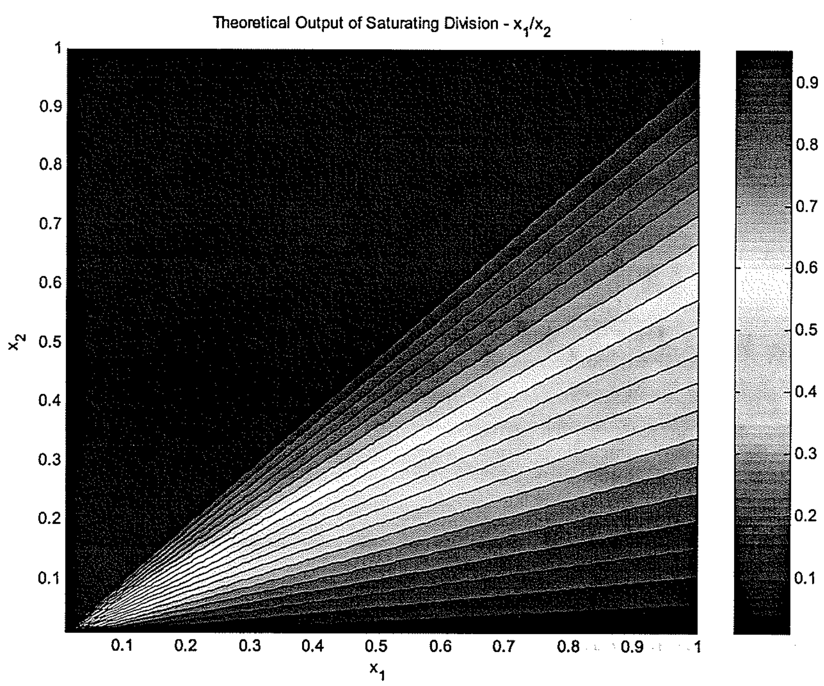


Figure 3.20: Theoretical saturating division operation.

3.4 Stochastic Polynomials and Functions Based on Taylor Series Approximations

It is possible to approximate any non-linear function using a Taylor series expansion [31]. Taylor series expansions approximate functions using a sum of polynomial functions with multiplicative coefficients:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n \tag{3.21}$$

When $x_0 = 0$, a Taylor series is also commonly referred to as a McLaurin series.

In theory one could construct a circuit to approximate a Taylor series expansion of a non-linear function. It is possible to represent an nth order polynomial function using $(n-1)$ stochastic delay elements [14] and an n-input AND gate or $n-1$ XNOR gates for the cases of unipolar or bipolar respectively. These circuits are shown in Figure 3.21.

For an input range limited of $(-1, 1)$ a third order approximation of a Taylor series is sufficient to describe a sine or cosine function. This is convenient when limited to 8-bit preci-

Figure 3.21: Stochastic polynomial circuits. (a) Bipolar, (b) Unipolar.

sion stochastic generators because it is impossible to represent a multiplicative factor less than 1/255.

However, a problem arises when such an approximation is implemented in a stochastic architecture. When n terms are summed there is an inherent division by n. Regardless of this averaging, the stochastic McLaurin series gives a reasonable approximation to the sine/3 and cosine/3 functions over the allowable input range. The input is encoded as a bipolar signal in both cases, and the output is bipolar as well. The outputs of the sine and cosine function can be seen in Figure 3.22 and Figure 3.23 respectively. The exponential function is also approximated, though not as well. The output of the exponential function can be seen in Figure 3.24.

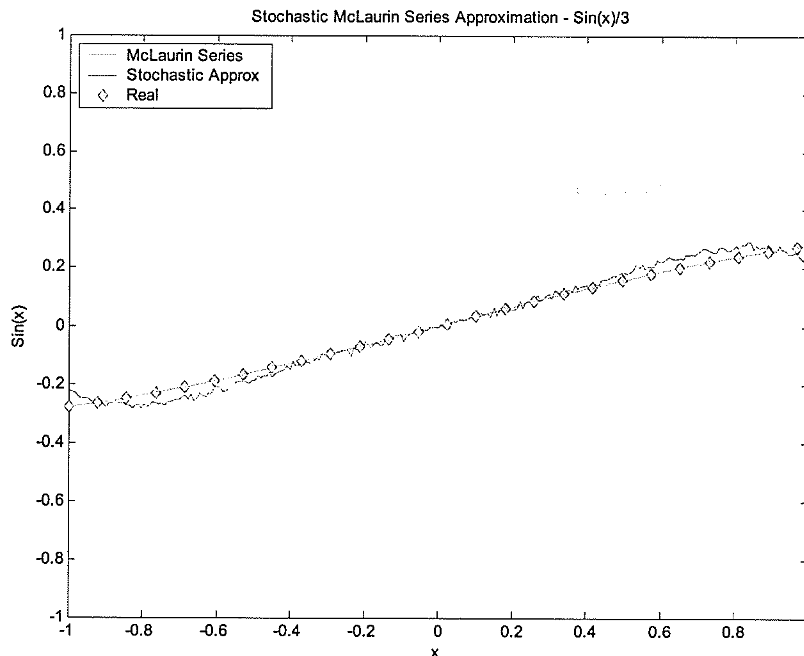


Figure 3.22: Output of stochastic approximation to Sin(x)/3.

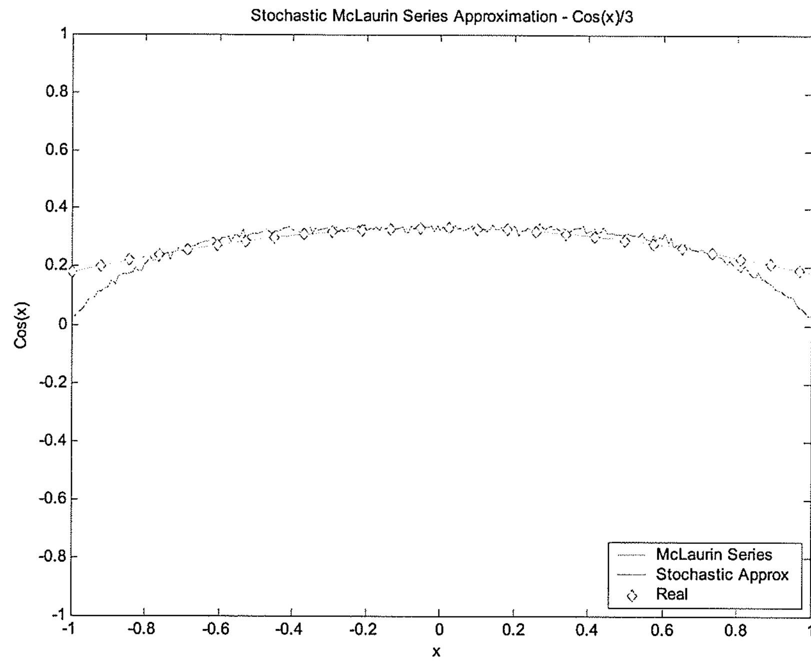


Figure 3.23: Output of stochastic approximation to Cos(x)/3.

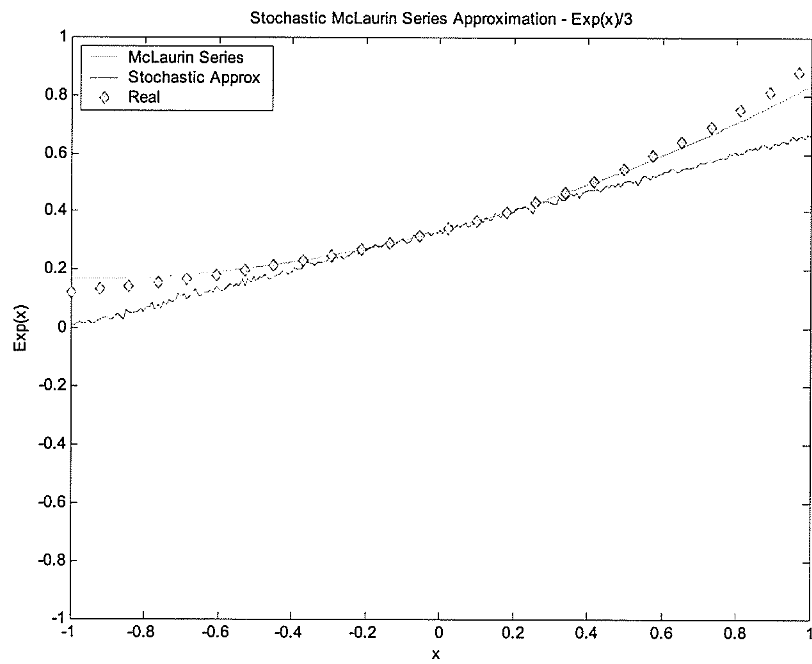
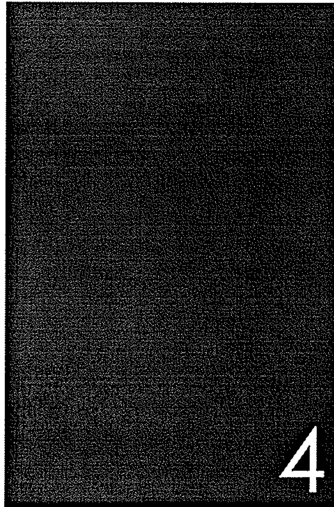


Figure 3.24: Output of stochastic approximation to Exp(x)/3.



Random Number Generation

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."
- John von Neumann

To generate stochastic bit streams a source of random bits is required. There are several possible approaches. It is important to state that it is impossible to produce truly random numbers with a digital circuit, they are instead pseudo random.

It is also desirable to have a 'simple' configuration from the perspective of the layout of a hardware circuit. Each bit stream produced should have a consistently minimal auto-correlation value as well as a minimal cross correlation value (when cross-correlated with any other stream output from the RNG). It is also desirable that each bit stream should have a probability $p(x = 1) = 0.5$ of each bit being a '1' or '0'. The reason for these requirements will become clear in the following discussion.

There are several properties that a suitable PRNG should have for application to stochastic signal generation. A set of statistical 'randomness' tests have been developed by Marsaglia[1]. It is beyond the scope of this work to explain each test in detail, suffice it to say that there are seventeen tests in this suite which examine various aspects of the output of a

1. Available at http://stat.fsu.edu/~geo/diehard.html

PRNG, both bit-stream-wise and as a whole. Passing this stringent suite of tests is a reasonable requirement for an RNG in this application.

4.1 Random Number Generation Using an Analog Source

One potential method of random number generation is to use a noisy diode whose output is compared to a normalized input voltage. This comparison is converted to a binary value and clocked into a flip flop to produce a digital bit stream [1]. A circuit diagram for an analog noise source is shown in Figure 4.1. Diode sources are sensitive to temperature, so open loop operation will yield sub-optimal results. Closed loop implementations are readily available, however this choice is still less than optimal in a purely digital environment.

4.2 Random Number Generation Using Digital Sources

A more suitable method for VLSI implementation of a PRNG is the use of a digital circuit. The configuration of such devices is limited to two major formats, the linear feedback shift register (LFSR) and the cellular automata (CA). Other methods have also been examined using VLSI-friendly techniques, such as tunable ring oscillators [23].

For the purposes of this discussion, the terms 'cell', 'bit', and 'flip-flop' will be used interchangeably. The term 'tap' will refer to the output of single flip-flop from within a register.

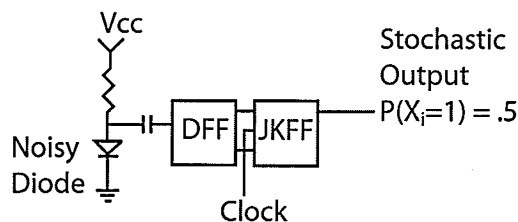


Figure 4.1: Stochastic bit stream source based on an analog noise diode.

4.2.1 Linear Feedback Shift Registers

LFSRs are a very hardware efficient way of producing long sequences of random bits. There are many configurations of such registers that will produce a maximal length sequence, meaning that the RNG will visit all possible states before repeating a state. This is a desirable property from the perspective of implementing an RNG. A block diagram of an 8-bit LFSR configured to generate a maximal length sequence is given in Figure 4.2.

The output from a 64-bit LFSR can be seen in Figure 4.3. The plot is a time-value plot of the bits in the LFSR. Time increases up the vertical axis, showing the state of each bit in the register for each time step. White squares represent ones, black squares represent zeros.

This RNG does not pass the suite of tests designed to ensure adequate randomness described by Hortensius et al [7]. This is due to the fact that the output pattern is highly regular, even though the total value of the register is random in sequence. Such temporal and spatial correlation is also undesirable in the application of stochastic computing.

4.2.2 Cellular Automata

CA research was pioneered by Stephen Wolfram who used CAs as a tool to model various complex processes (e.g. turbulent fluid flow, biological growth). Similar to LFSRs, CAs are also constructed using a register and feedback. CAs are similar in operation to LFSRs but the feedback is implemented slightly differently.

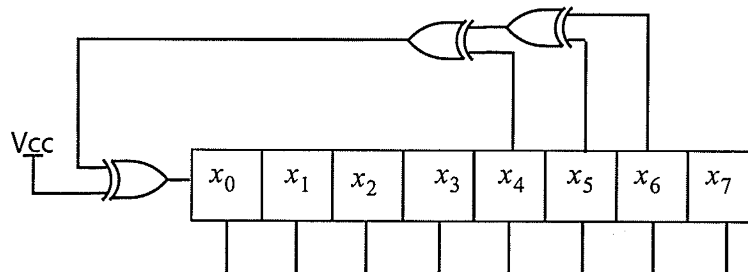


Figure 4.2: Schematic diagram of an 8-bit maximal length LFSR.

Wolfram has classified 1-Dimensional CA's into four distinct groups. Class 1 CAs evolve to homogeneous global states, Class 2 CAs evolve to periodic structures, Class 3 CAs remain in a chaotic state, and Class 4 CAs generate complex localized and propagating structures. The only suitable candidate for random number generation is Class 3.

There is also a distinction that can be drawn based on the starting condition of a CA. CAs that produce random streams from a fixed initial condition are termed *autoplectic*, while CAs that have a pseudo random initial condition are termed *homoplectic*. The most interesting CAs for this application are Class 3 autoplectic CAs, which offer the greatest simplicity to a stochastic processing system [27].

CAs have a regular structure that does not change depending on their width. CAs also employ local routing, which is attractive from a circuit layout perspective. The boundary conditions of a CA can be cyclic or open, meaning that the ends either wrap around or are fixed, respectively. In this application, cyclic boundary conditions are chosen.

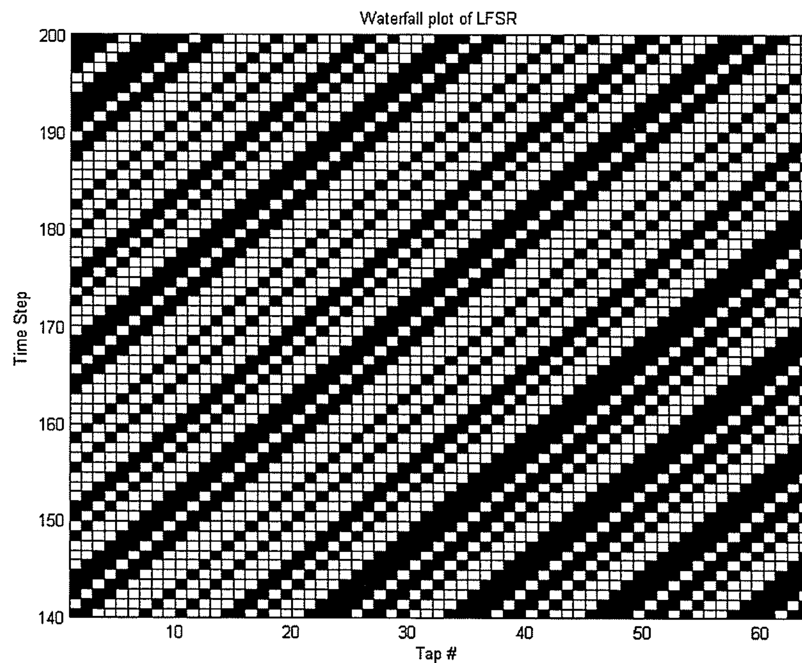


Figure 4.3: Waterfall plot of 64-bit LFSR.

CAs can also be connected in a two or three dimensional fashion, allowing for much more complex output patterns, however, the overhead in terms of a VLSI implementation as compared to a simple 1-D case make these a less attractive option [8].

4.2.3 CA30 Example

The behaviour of a CA is determined by the *rule* of the CA and the feedback interconnection scheme. Both of these features will be described in the following example.

Perhaps the simplest CA is the "Rule 30 nearest neighbour" CA, seen in Figure 4.4.

The three nearest neighbours are fed back through a logic function, which determines the next state of each bit in the register. The feedback taps are abbreviated with the notation $\{i-1, i, i+1\}$, meaning that the cells on the immediate left and right as well as the ith cell are fed into the logic function.

Wolfram has developed a naming system for CAs that specifies the feedback function.Since there are three feedback signals, there are 8 conditions to which any cell can respond. This means that there are 2^{2^3} possible implementations of this feedback connection. Figure 4.5 shows how the rule system for naming CAs is specified. The output for the truth table is represented as a binary number, which is then converted to its decimal equivalent. This decimal number is the *Rule Number* of the CA.

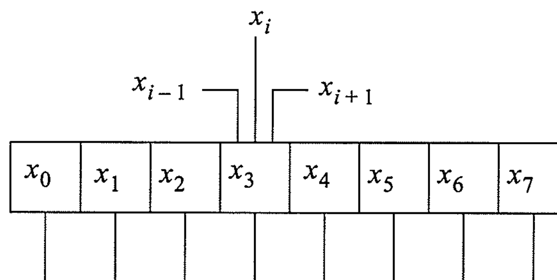


Figure 4.4: Schematic of a neighbourhood of 3 CA.

Note that the rule number is independent of the feedback scheme so there are:

$$N = \frac{n!}{k!(n-k)!} \tag{4.1}$$

possible feedback line configurations, where n is the number of bits in the CA and k is the neighbourhood size of feedback connections. This gives the total number of possible configurations of CAs consisting of n bits, each exhibiting different properties.

Figure 4.6 shows the output of a 64-bit CA30 with nearest-neighbour connections and cyclic boundary conditions. The CA is initialized in position 1 (white squares are '1's, black squares are '0's). The regular pattern of triangles can be observed, even though this is classified as a class 3 CA [7]. The first 60 time steps are shown, progressing from the initialization state at the bottom of the chart. While there is a reasonably low cross-correlation between adjacent streams of bits as well as a low auto-correlation value, this RNG does not pass the suite of DIEHARD tests due to the semi-regularity of the output as a whole [8].

CA30 Truth Table

x_{i-1}	x_i	x_{i+1}	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

$[00011110]_2 \rightarrow [30]_{10} \Rightarrow CA30$

Figure 4.5: Truth table for Rule 30 CA, abbreviated CA30.

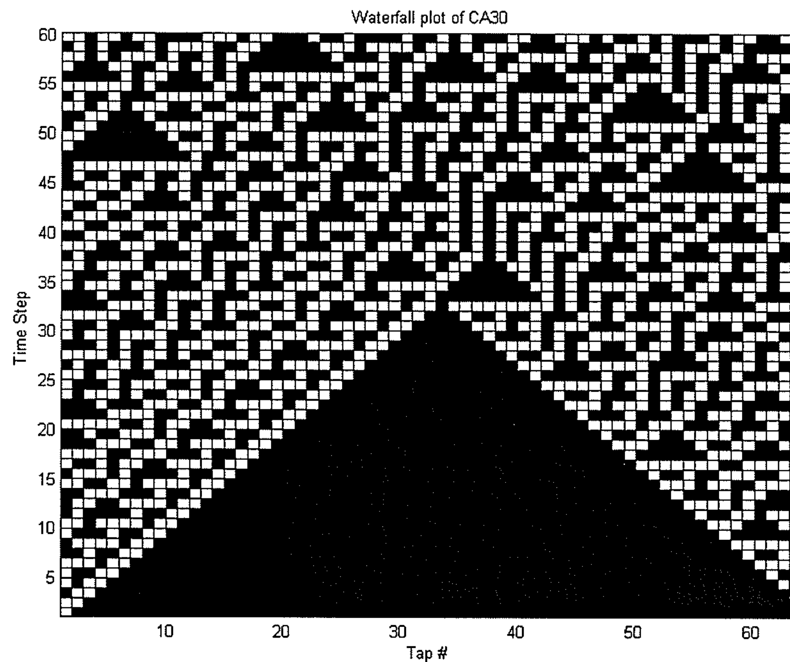


Figure 4.6: Waterfall plot of the 64-bit CA30.

4.2.4 CA38490

The poor overall randomness properties of CA30 make it undesirable in this application. Another CA that has been examined in the literature that passes the DIEHARD suite of tests is CA38490 [8].

This particular implementation uses a "neighbourhood-of-four" configuration with offsets of $\{i-3, i, i+4, i+9\}$ for the feedback paths. The neighbourhood-of-four configuration is chosen because most FPGAs use a 4-input look up table to implement logic functions (more on this later). Due to the fact that the feedback taps are well apart, the CA38490 has consistently low cross-correlation between adjacent bit streams.

The CA38490 is the CA chosen for this application, because of the properties discussed above. In the final system implementation six 64-bit CAs are used, each with different autoplectic starting conditions.

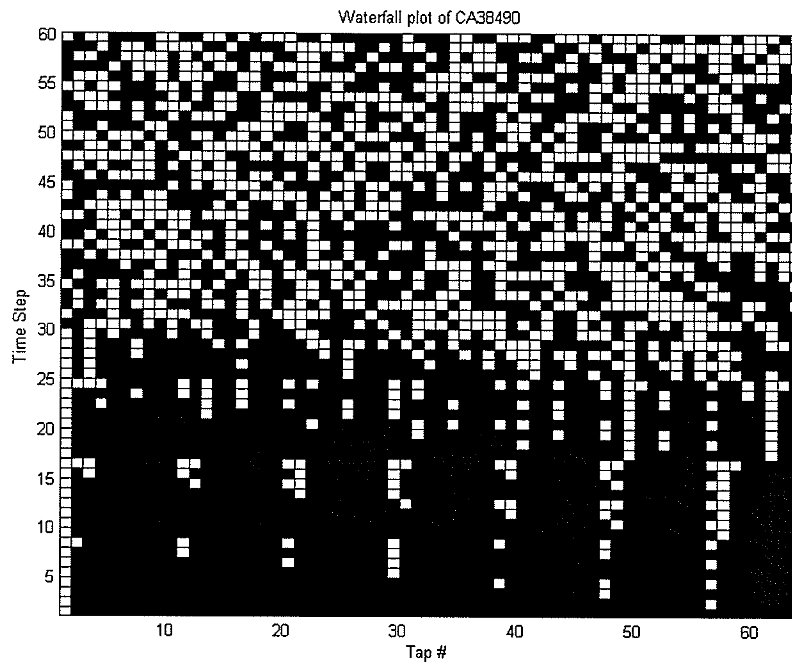


Figure 4.7: Time plot of 64-bit CA38490.

Figure 4.7 shows the time plot of the output of CA38490, initialized at position 1. After approximately 30 time steps, no regular patterns can be seen in the output, contrasting with the output of the CA30.

Figure 4.8 and Figure 4.9 show the correlation between the output streams of all three RNGs discussed. The LFSR has the poorest auto-correlation sequence, while both CA implementations are comparable. The cross correlation of the CA38490 is more consistent than that of CA30.

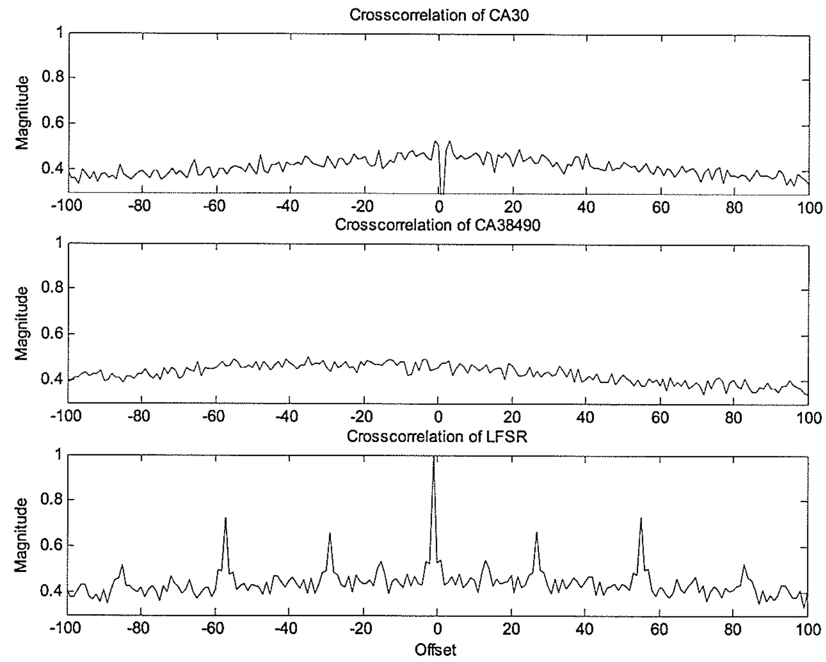


Figure 4.8: Cross-correlation plot of adjacent bit streams.

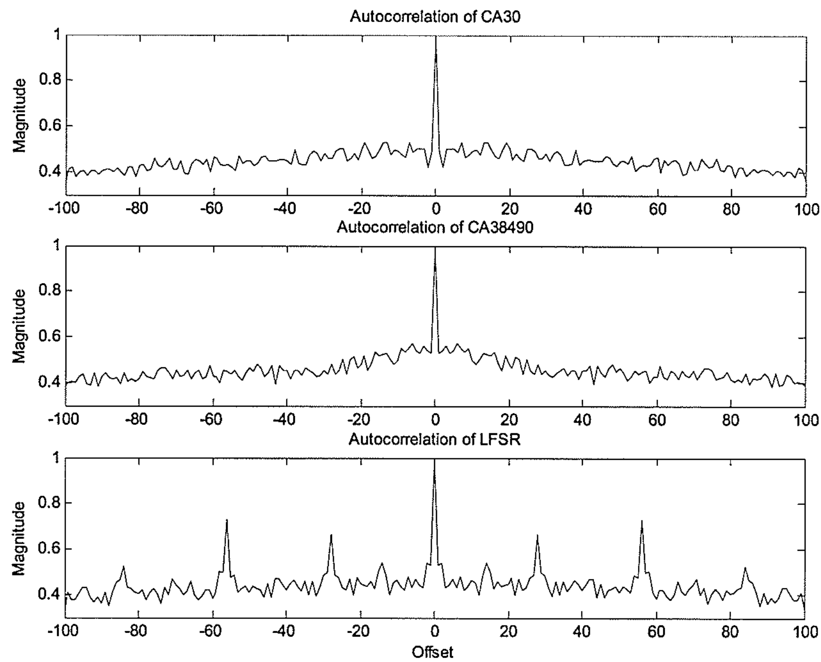


Figure 4.9: Auto-correlation plot of bit streams.

4.3 Variable Probability Generation

Now that a suitable PRNG has been selected, it is desirable to be able to produce a stochastic bit stream with any given generating probability value in the range of $(0 \rightarrow 1)$. Let A_i, $i = 1, 2, 3\ldots$ be a set of independent sequence of binary random variables each with a generating probability of $p_i = 0.5$ [1]. Consider Boolean functions of these variables:

$$B_1 = A_1 \tag{4.2}$$

$$B_2 = \overline{A_1}A_2 \tag{4.3}$$

$$B_3 = \overline{A_1}\overline{A_2}A_3 \tag{4.4}$$

$$B_n = \overline{A_1}\overline{A_2}\ldots A_n \tag{4.5}$$

where n is the amount of precision desired. These functions are such that no more than one of the B_i may be equal to '1' at a given time, i.e. $B_iB_j = 0, \forall i \neq j$. The generating probability of each of the $B_i s$ is B^{-i} (binary weighted). Since the $B_i s$ are independent the probability of each B_i sums to 1 when fed through an OR gate. So, if we wish to generate a stream with a probability represented by a binary vector $\boldsymbol{x} = X_1X_2X_3\ldots$ then:

$$X = X_1B_1 + X_2B_2 + X_3B_3 + \ldots \tag{4.6}$$

has a generating probability equal to $\boldsymbol{x}$.

A circuit to realize this function can be readily constructed using simple combinational logic, as shown in Figure 4.10 for $n = 8$. The probability generator has a repeating structure that can easily be modified to accommodate any desired precision.

The circuit in Figure 4.10 is capable of generating stochastic streams with bit probabilities of all values in the required range. Figure 4.11 shows the integration of each value of the resulting stochastic stream for 16384 clock cycles. The difference between the expected value of the count and the actual value of the count is shown in the lower plot. The maximum error is less than one bit in the estimation of the generating probability of the stream.

All of the necessary computational units required to implement a stochastic artificial neural network in hardware have been specified.

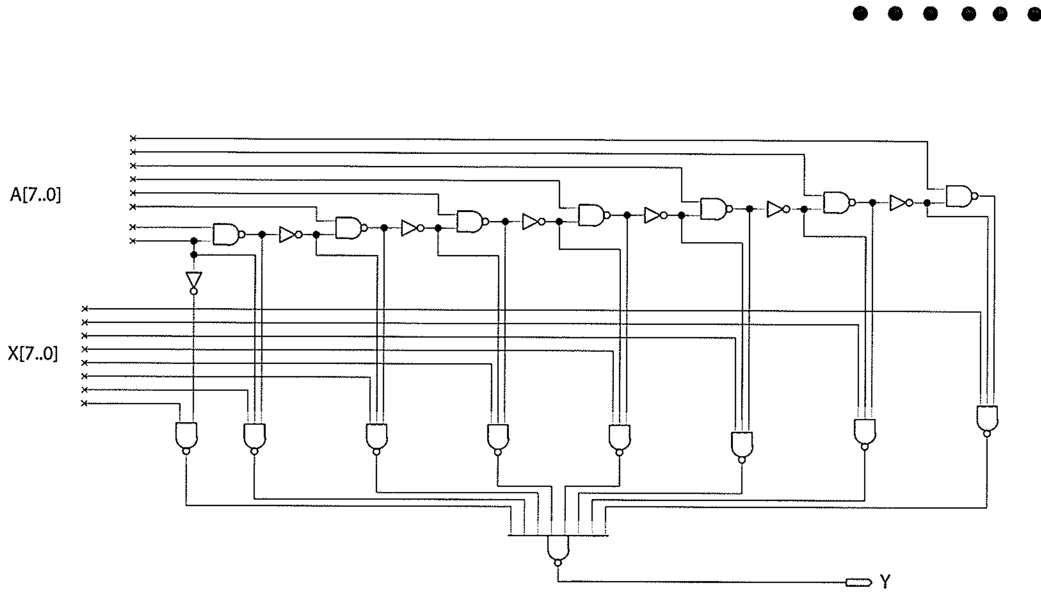


Figure 4.10: 8-bit variable probability generating circuit.

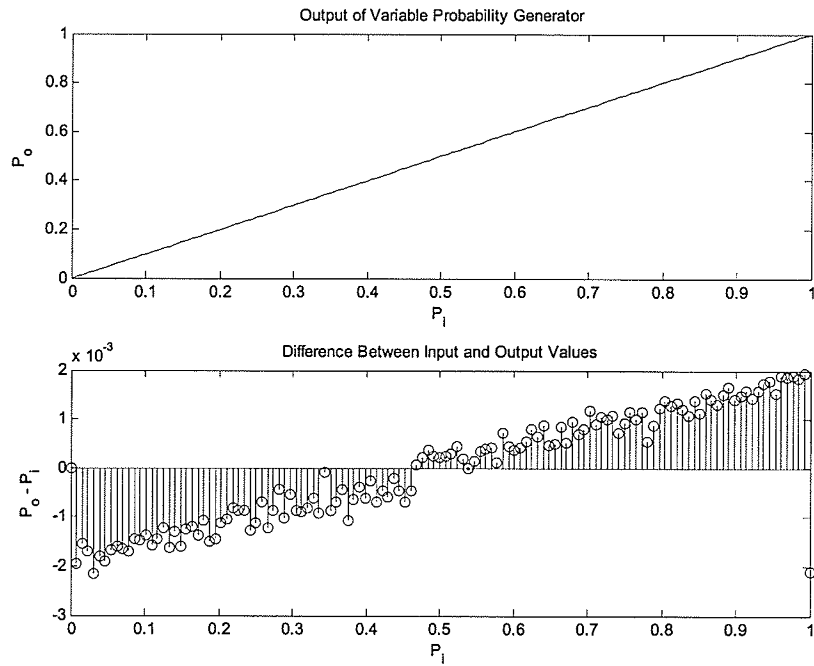


Figure 4.11: Output from Variable Probability Generator.

5

Neural Network Theory

Neural computation is a method of processing data that attempts to model the way a biological brain computes information. The basic computation module in a neural computer is the *neuron*. This is a simple processing unit that is capable of receiving multiple inputs, performing a weighted sum and non-linear transformation, then outputting the result. Neurons are interconnected with *synapses* and can have a high degree of fan-in and fan-out. It is estimated that the human brain contains on the order of 10 billion neurons, with over 60 trillion synaptic connections between them [26]. Due to the massively parallel structure of the human brain, it is possible to obtain a high data processing rate while having a relatively low 'clock frequency'.

Neural networks created by humans are referred to as artificial neural networks (ANNs). Due to the extremely parallel nature of neural networks, it is generally inefficient to implement them in general hardware using a standard computer architecture. Software can be written to model neural computation, but a large amount of memory and an extremely fast processor are required to approach real-time operation for even simple networks.

The majority of applications of neural computers involve pattern recognition and classification. One of the main features of this type of computer is the ability to learn statistical properties of a set of data. This process is much like a child learning about its environment by repetition and error correction. Input vectors are applied, and the network attempts to

build classification categories from the data provided. The network is then able to generalize this information and attempt to classify input data that is not part of the training set.

Neural networks are trained using techniques that are classified into two major categories: *supervised* training and *unsupervised* training. Supervised training requires input data along with target network output values for each data point. Unsupervised learning does not require these data labels.

Two of the most common types of ANNs are the multi-layer perceptron network (MLP) and the radial basis function network (RBF). Both network types compare a d-dimensional input vector to a set of desired patterns (also called prototype vectors, weight vectors, or mean vectors in the case of RBF networks), and have several outputs which indicate which prototype pattern is most closely matched with the input. Each network produces a distinct *decision region* defined by the weight vectors. Figure 5.1 shows a diagram of the decision region formed by each type of network for a 2-D input space.

The main difference between the MLP network and the RBF network is the way that patterns are classified into different regions of the input space. The MLP network forms regions separated by linear boundaries that are determined by the weight vectors. The RBF network forms regions based on the mean vectors associated with each neuron. MLP networks compute a non-linear function of a scalar product of synapse weights and input

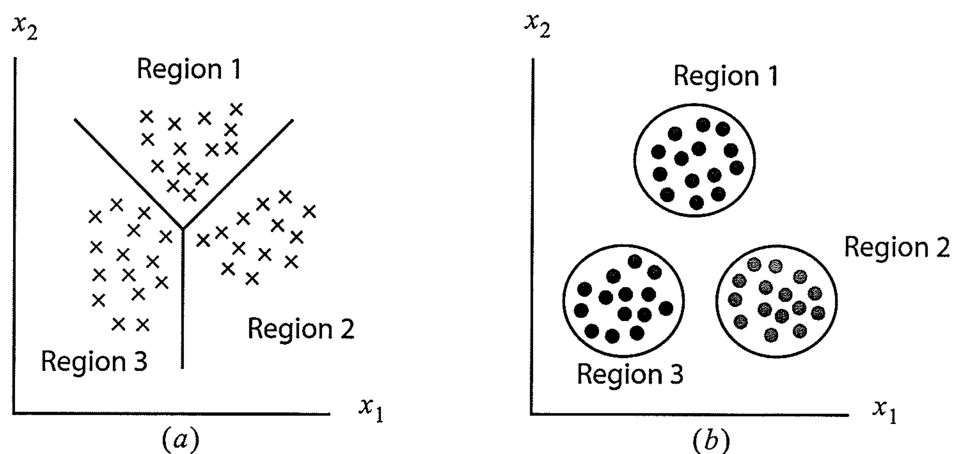


Figure 5.1: Decision regions formed by a) an MLP network and (b) an RBF network.

vectors, while RBF nets compute the distance from the input vector to the prototype vector.

In this thesis only RBF networks are considered, so only the underlying theories for this category of network will be discussed. MLP networks have also successfully implemented in a stochastic architecture ([9], [10], [11], [13]).

5.1 Radial Basis Function Network Theory

Radial basis function networks originated as a technique for performing exact interpolation on a set of data points [2]. This problem requires that every multi-dimensional input vector be mapped to a corresponding one dimensional target vector. The data set consists of N input vectors $\boldsymbol{x}^n$ (the notation in this discussion will be adapted from [2], where ***n* is an index** as opposed to an exponent), that are mapped to a corresponding group of target vectors t^n. The goal of the network is to find a function $h(\boldsymbol{x})$ such that:

$$h(\boldsymbol{x}^n) = t^n, n = 1, \ldots, N \tag{5.1}$$

The radial basis function approach introduces a set of N basis functions that take the form $\phi(\|\boldsymbol{x} - \boldsymbol{x}^n\|)$ where $\phi(x)$ is a non-linear function. The nth function depends on the distance (generally Euclidean) between $\boldsymbol{x}$ and $\boldsymbol{x}^n$. The result is a weighted sum of all of the basis functions:

$$h(\boldsymbol{x}) = \sum_{n=1}^{N} w_n \phi(\|\boldsymbol{x} - \boldsymbol{x}^n\|) \tag{5.2}$$

The most common form of the basis function is the Gaussian function:

$$\phi(x) = e^{\frac{-x^2}{2\sigma^2}} \tag{5.3}$$

These are termed *localized* basis functions, which means that $\phi \to 0$ as $|\boldsymbol{x}^n| \to \infty$. There are many other choices of basis function that have this property, but will not be discussed here.

It is straightforward to generalize these functions for M output variables. This requires a mapping from $\boldsymbol{x}^n \rightarrow \boldsymbol{t}^n$ where $\boldsymbol{t}^n$ has components t_k^n. There are now M network transfer functions:

$$h_k(\boldsymbol{x}^n) = t_k^n = \sum_{n=1}^{N} w_{kn}\phi(\|\boldsymbol{x}-\boldsymbol{x}^n\|) \tag{5.4}$$

The network topology for a radial basis function network is shown in Figure 5.2. This is classified as a two layer RBF network because there are two layers of adjustable parameters. The first layer of adjustable parameters is composed of the mean values for the basis functions. The second layer is composed of linear weights as well as bias units. The bias units ensure the that input vector values and output values are in the same range. More general forms of these networks (i.e. with more layers) are not normally considered.

By introducing some modifications to the exact interpolation problem, we arrive at the RBF neural network model. The goal of the RBF network is to provide a smooth mapping from input vectors to output vectors by choosing the number of basis units according to the complexity of the problem being considered (not the size of the input data set as before). The higher the number of basis functions, the more oscillatory the output will be. These modifications to the exact interpolation case are as follows:

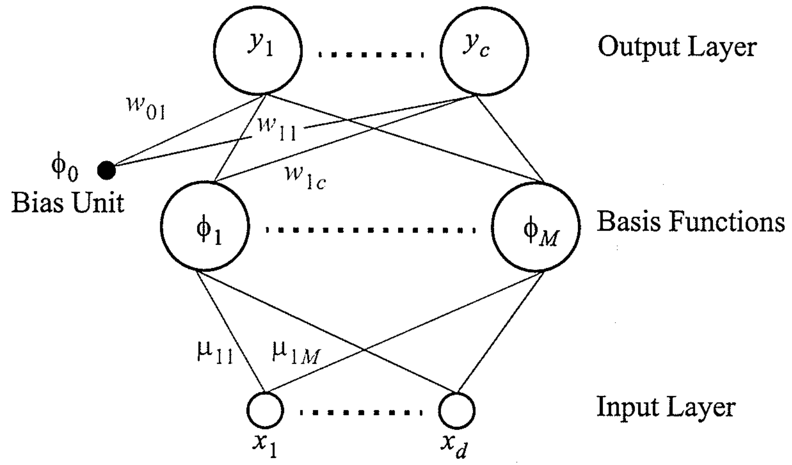


Figure 5.2: RBF Neural Network diagram.

1. The number, M, of basis functions does not need to equal the number of, N, data points. The value of M is typically much less than the value of N.
2. The centres of the basis functions and linear output layer weights are no longer based on specific input vectors, they are determined during the training process of the network.
3. As opposed to having a common variance σ^2, each unit will have a different value which is determined as part of the training process.
4. Bias parameters are included in the linear sum. These bias parameters account for a possible difference in mean value of the input vectors and the mean value of the target vectors.

When these changes are applied to the exact interpolation model, we arrive at the transfer function of the RBF network:

$$y_k(\boldsymbol{x}) = \sum_{j=1}^{M} w_{kj}\phi_j(\boldsymbol{x}) + w_{k0} \tag{5.5}$$

In the case of this thesis Gaussian basis functions of the form:

$$\phi_j(\boldsymbol{x}) = \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{\mu}_j\|^2}{2\sigma_j^2}\right) \tag{5.6}$$

are used, where $\boldsymbol{x}$ is the d-dimensional input vector (with components x_i), and $\boldsymbol{\mu}_j$ is the mean vector for the jth basis function (with components μ_{ji}).

The basis functions can be generalized to have full co-variance matrices Σ_j, in which case the RBF unit transfer function becomes:

$$\phi_j(\boldsymbol{x}) = \exp\left\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_j)^T\Sigma_j^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_j)\right\} \tag{5.7}$$

There is a trade off between using fewer basis functions with more adjustable parameters (full covariance Gaussians have $d(d+3)/2$ parameters) and more units with fewer adjustable parameters (e.g. radially symmetric Gaussians have $(d+1)$ adjustable parameters). Networks with more adjustable parameters take longer to train than networks with

fewer parameters. However, fewer units may be required to accurately represent a data set if they have a full covariance matrix. RBF networks in general also have a layer of weights between the basis function units and the output layer.

5.2 RBF First Layer Network Training

There are many methods by which a neural network can be trained from an initial state (with random weight vectors) into a state capable of correctly categorizing input vectors. RBF networks are trained in a two stage process, the first layer weights (μ_{ji}) and the second layer weights (w_{kj}) are trained separately.

There are two sets of input data presented to a neural network. The first is used during the training process, and is called the *training set.* The second is used to verify the neural network, and is called the *test set.* The training set is presented to the network in a random order to prevent biasing the network from learning all of one set of data, and then learning all of another. Each vector in the training set is also usually labelled with the desired response of the network. The training set is presented to the network multiple times, each presentation is referred to as one *epoch* of training. The training data is reordered after each epoch.

Several simplifications of the basis units can be made at this stage depending on the application. The RBF units can be reduced to hyperspherical basis functions (i.e. without a full covariance matrix) to reduce the number of trainable parameters. This means that multiple units of difference variances may be used to cover a single data cluster. To further reduce complexity, each unit can be fixed to have the same variance, while using potentially even more basis units.

The number of units required is determined by the complexity of the data set that is to be represented. There are several techniques that can be used to determine the number of basis functions. A common technique is called *growing.* Using growing the network is trained with a given number of units, then an error term is calculated. A new unit is then added and the network is retrained and the error recalculated. This process continues until there is a negligible difference in the error. Conversely, one could start with a large network and *prune* it until the error is affected significantly.

The first layer of weights are trained in an unsupervised learning process, where the mean of the RBFs gravitate to the centres of data clusters. After these parameters are determined, the second layer of weights is trained with a supervised process using the targets values of each input vector. First layer training determines all of the free parameters of the basis functions (μ_j and σ_j), while second layer training determines the value of the linear weights.

The goal of the training process is to minimize the total network error, which is defined as a function of error vs. weight parameters. The error function is a multi-dimensional surface dependent on the weight values of the network. Adjusting the weight parameters changes the position of the network on the error surface. There are several techniques for unsupervised training of neural networks, each with certain advantages and disadvantages.

5.2.1 Hard Competitive Learning

Hard competitive learning (HCL) is the simplest form of unsupervised training. It could also be described as the "dumbest" training technique. The approach is simple; first, compute the distance from the input vector to each mean vector

$$h_i = \|x^n - \mu_i\| \tag{5.8}$$

Then select the unit which has the smallest value of h_i, that has the best match to the input vector. This unit is termed the 'winner' of the competition. The smallest value of h_i is labelled h_i^*, and the index of the winning unit is i^*. Since we wish to reinforce the association of unit i^* and the current training vector x^n, only the values for μ_i^* are adjusted. The amount of adjustment is:

$$\Delta\mu_{i^*} = -\varepsilon(x^n - \mu_{i^*}) \tag{5.9}$$

where ε is the learning rate, and is generally chosen to be small. The purpose of the learning rate is to adjust this change so that the network does not get stuck in a local minima on the error surface, and also to allow the network to train at a sufficiently fast rate.

Since the HCL process is so simple, there are likely to be some drawbacks. One primary problem is that the network is very sensitive to the initial values of the μ_j vectors. For

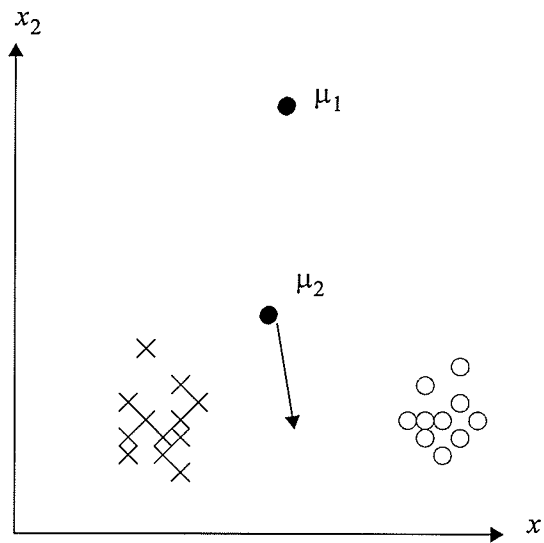


Figure 5.3: Isolation in an HCL training scheme.

example, consider two clusters of input data in a two dimensional input space as shown in Figure 5.3. Let us assume that we wish to represent these data clusters using two RBF units. Now suppose that one of the units is initialized to be near the centre of both clusters of data, while the other unit is initialized to the outer regions of the input space. The first unit will always be closer to both sets of input data, so it will always win every competition. This unit will eventually end up representing the mean of the data clusters, while the other unit will never move from its initial position. This is termed *isolation*.

To counteract isolation, a 'conscience' term can be added to the distance calculation which is based on the number of times a unit wins a competition.

5.2.2 Frequency Sensitive Competitive Learning

Frequency sensitive competitive learning (FSCL) attempts to add a level of fairness to the training process. This is accomplished by punishing units that win too many competitions and favoring units that win too few.

FSCL also starts by computing the distance from the mean vectors to the current input vector:

$$h_i = \|x^n - \mu_i\| + b_i \tag{5.10}$$

but adds a bias to the distance term (this is not the same as the bias units in the network). The bias term is computed from three parameters

$$b_i = -c\left(\frac{1}{M} - p_i\right) \tag{5.11}$$

where M is the number of units, p_i is the proportion of competitions that a unit has won, and c is a constant called the *bias factor*. The mean vectors of the winning unit are adjusted in the same manner as before (5.9), but the p_i term has to be adjusted after each competition for each unit:

$$\Delta p_i = B(1 - p_i),\ 0 < B \ll 1 \tag{5.12}$$

where B is a constant which controls the rate at which the penalty term changes. This type of penalty works, but only if the penalty constant C is chosen appropriately.

A superior method of fairness is achieved with a multiplicative penalty term as opposed to an additive one:

$$h_i = F(u_i)\|x^n - \mu_i\| \tag{5.13}$$

where $F(u_i)$ is any monotonically increasing function of u_i (representing the number of competitions a unit has won) and is referred to as the fairness function. $F(u_i)$ is initialized to 1, and is commonly chosen to be

$$F(u_i) = u_i \tag{5.14}$$

where the value of the function is simply the number of competitions the unit has won [6]. Again, the amount of weight adjustment for the winning unit is the same as in the HCL case.

There are drawbacks to FSCL however. If one group of data is more densely populated, multiple units could end up being assigned to it. This is because each unit is forced to win an approximately equal number of competitions. While this is a better result than the isolation problem, it is still not an optimal solution [6].

5.2.3 Soft Competitive Learning

Soft competitive learning (SCL) adjusts the mean vectors of each basis unit after every competition. The adjustment is proportional to each unit's output response for that

particular data point, as well as the distance of the data vector to the unit's mean vector. The basis functions are normalized so the total probability that a data point is represented by a particular unit is always 1. The functions ϕ_j are now modified as follows:

$$\phi_j(x^n) = \frac{e^{-\frac{\|x^n - \mu_j\|}{2\sigma^2}}}{\sum_{i=1}^{M} e^{-\frac{\|x^n - \mu_i\|}{2\sigma^2}}} \tag{5.15}$$

and the weight adjustment factor becomes:

$$\Delta\mu_j = -\varepsilon(x^n - \mu_j)\phi_j(x^n) \tag{5.16}$$

where ε is the learning parameter. In this scheme it is much less likely for units to become isolated.

Consider the case of the two input data cluster shown in Figure 5.4. Suppose that we have the same initialization problem as before: two data clusters close together, one unit centred in between them, and the other unit in the outlying region. Initially the close unit (μ_2) moves the largest amount, but μ_1 will also begin to move slowly towards the data clusters. As the first unit makes its way towards the data sets, its response will start to grow,

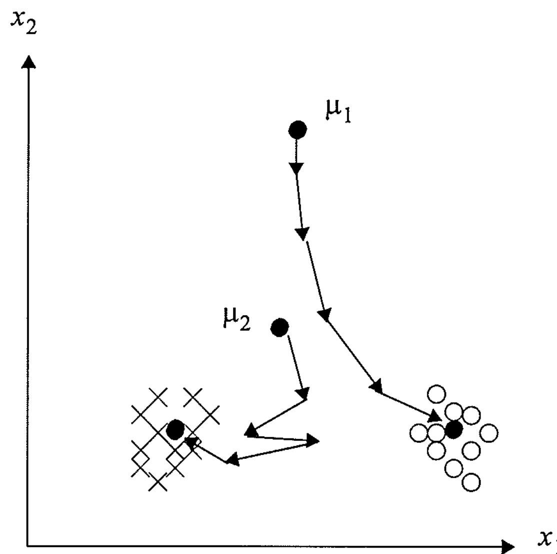


Figure 5.4: Simplified diagram of mean vector convergence in SCL example.

increasing its approach speed. As this is happening, the second unit will tend towards one of the data groups. Eventually, both units will centre on one of the clusters.

However, there are cases where one unit will remain isolated. If the learning parameter or the variance parameter is chosen to be too small, a unit that is far from the data clusters will take an extremely long time to converge to one of the data clusters.

5.3 Second Layer Training

The second layer of linear weights are trained in a supervised fashion. The objective of this stage of learning is to minimize the difference between the output values, y_k, and the desired target vector, $\boldsymbol{t}^n$, for a each input vector.

Before second layer training can begin, an error function must be defined for the network. This is the function that is to be minimized for best possible performance of the network. A commonly used error function is the sum-of-squares error function defined as

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{c}\{y_k(\boldsymbol{w}, \boldsymbol{x}^n) - t_k^n\}^2 \tag{5.17}$$

where N is the number of data vectors in the training set, t_k^n is the target value of output k for input value $\boldsymbol{x}^n$ and c is the number of output units. The factor of $^1/_2$ is included to simplify the calculation of the error gradient.

One method is to calculate the entire gradient and analytically find a solution for the optimal weight vector. A more feasible method is to start with random weight vectors, and move along the error surface until a minimum is found. It is desirable to be the absolute minimum, but this is not the case in general. This approach is termed the *error-correction learning rule* or the *delta rule* [26]. The amount of weight adjustment is:

$$\Delta w_k = -\varepsilon(y_k - \boldsymbol{t}^n)\boldsymbol{x}^n. \tag{5.18}$$

The weights are adjusted in proportion to the difference between the output of the unit, the desired value, the strength of the input, as well as a learning rate parameter.

Neural Networks Based on Stochastic Hardware

This chapter describes the hardware implementation of the stochastic circuits used to implement an RBF ANN using soft competitive learning. The application of these circuits will be discussed in the following chapter.

There are two major hardware components on which the system is implemented: a host PC, and a field programmable gate array (FPGA). The host PC is responsible for creating input data, transferring this data and commands to the neural network system, and collecting output data from the network. The FPGA module will receive the data from the PC, perform the neural computations and send the results back to the PC.

6.1 Development Platform

The FPGA used in this thesis research is the Altera Stratix EP1S40F780CS, which contains approximately 46000 registers, 41000 logic elements, 3.4M memory bits, and 180 I/O pins (90 input and 90 output).The chip also includes 12 PLL clock synthesizers that are user configurable.

Within the FPGA module there are two sub-components: the stochastic circuits, and a micro controller. The Nios micro controller is a soft-core micro controller that controls and sequences the stochastic hardware. This unit also receives data from and transmits

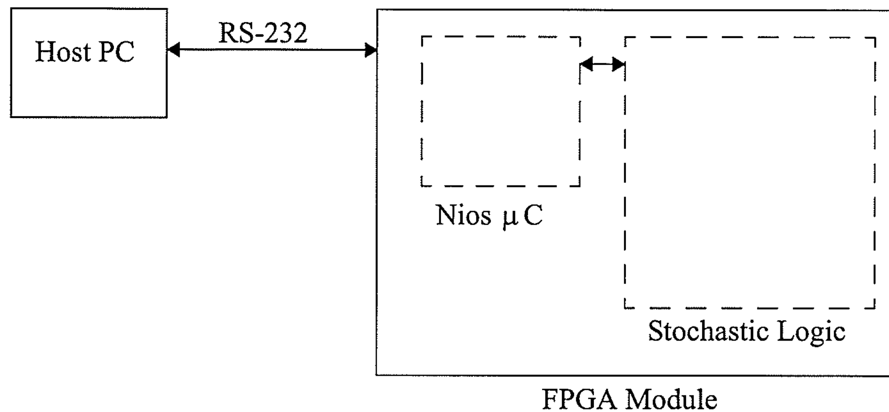


Figure 6.1: Hardware component block diagram.

data to the host PC. Figure 6.1 shows a block diagram of the major hardware components. A soft-core micro controller is a micro controller that is specified in software and then programmed onto the FPGA.

The biggest feature of the Nios development kit, in general, is the extremely flexible nature of the Nios core, which contains essentially drag and drop modules such as I/O ports, UART modules, and memory controllers. This system is well suited to developing specialized logic hardware (either graphically or using hardware description languages). The evaluation board for the Nios micro controller also has 2 serial ports, an LCD display, and several LEDs for displaying information (as well as many other peripheral device interconnects that are unnecessary for this research).

The input and output vectors are transmitted using one of the serial port modules. The other serial port is used to send information to the terminal window on the PC, as well as for programming the Nios processor. The FPGA is programmed using the JTAG port.

A picture of the Nios evaluation board is shown in Figure 6.2.

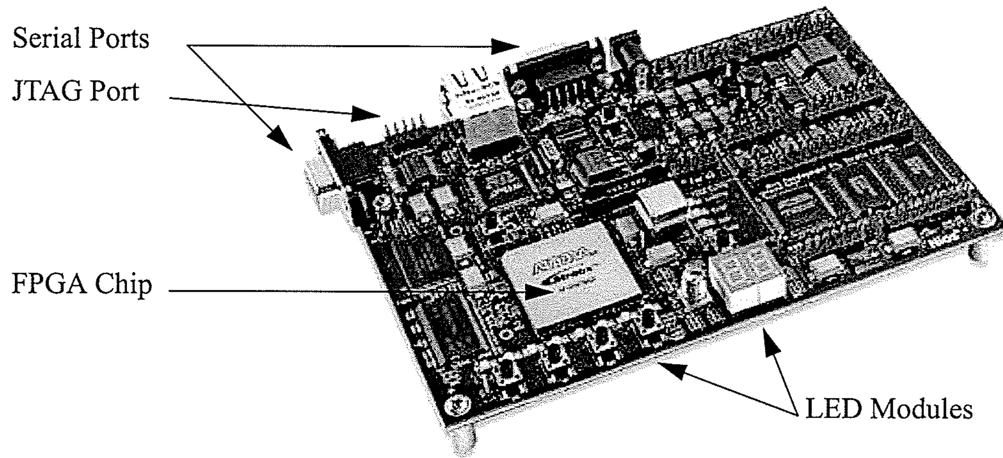


Figure 6.2: Nios Development Board with Stratix FPGA. Photo ©Altera Corp.

6.2 RBF Network Model Modifications

The neural network model used in this thesis differs from the conventional model in a few ways. The changes were possible due to the limited range of the data values present in the system, and the absence of labels for the training data (all of the data considered in the results section is unlabelled).

The transfer function for an RBF neural network is:

$$y_k(\boldsymbol{x}) = \sum_{j=1}^{M} w_{kj}\phi_j(\boldsymbol{x}) + w_{k0} \tag{6.1}$$

where $\phi_j(x)$ is:

$$\phi_j(\boldsymbol{x}) = \exp\left(-\frac{\|\boldsymbol{x}-\mu_j\|^2}{2\sigma_j^2}\right) \tag{6.2}$$

and M is the number of neurons.

The bias terms w_{k0} can be discarded in this work because the input vectors are in the same fixed range as the output vectors.

The value of σ^2 is variable, but it will be the same value for all units. Furthermore, this value will not be determined during training but set and varied according to how much noise is desired in the system.

There is also no linear output layer implemented in this system. The reason for this is that there is absolutely no prior knowledge of the input data. It is therefore not possible to train the output layer weights in a supervised fashion without a set of target vectors.

Taking into account these modifications, the revised network transfer function becomes:

$$y_k(x) = \phi_j(x) \quad (6.3)$$

making the output of each basis function the total response of the neuron. $\phi_j(x)$ is also substituted with the stochastic exponential function:

$$\phi_j(x) = \exp(-2G\|x - \mu_j\|) \quad (6.4)$$

where G is the implementation of the σ^2 parameter. A simplified block diagram of the RBF circuit is shown in Figure 6.3. This architecture can be scaled indefinitely, with the

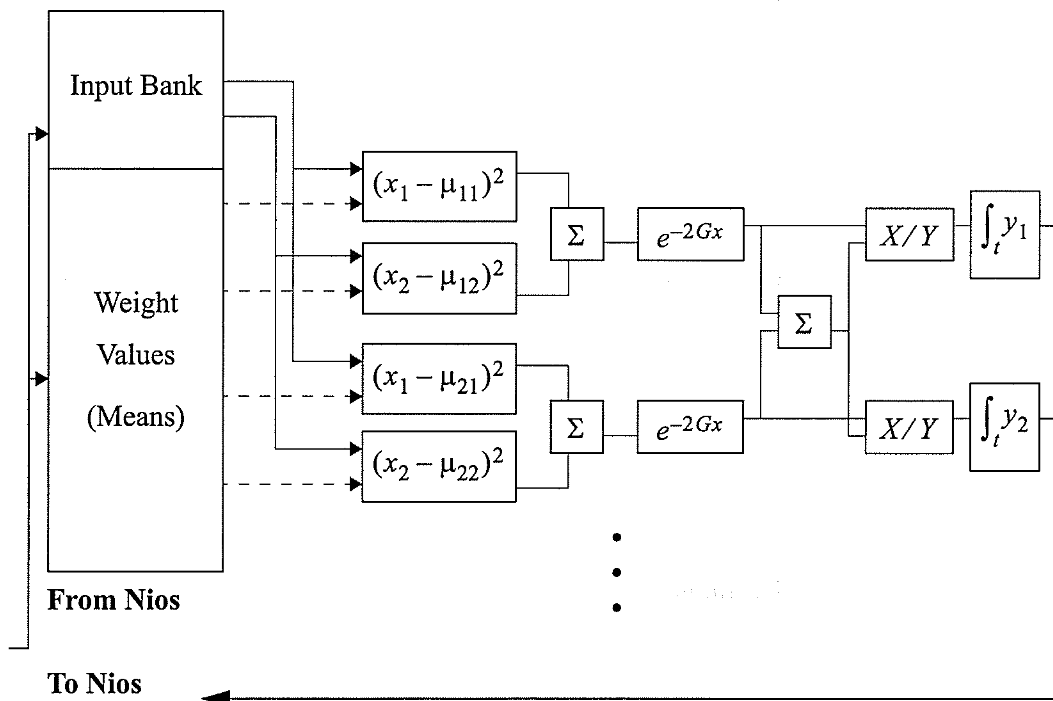


Figure 6.3: Hardware component block diagram of SANN.

network limited only by the size of the FPGA module. All communication lines indicated in red are a single wire. Black communication lines are the width of the system's resolution (eight bits in this case).

There are also several control registers that are not shown in the simplified diagram. These registers contain the gain parameter, the number of clock cycles of processing time, as well as the number of learning cycles (analogous to the learning rate).

There are other control parameters that are stored in the memory of the μC including the number of bytes in the set of network parameters, the number of data bytes, as well as an operation mode parameter. To facilitate circuit debugging and operation analysis there are three modes of operation available to the system user. These are:

1. Training mode '0': Weights are not updated, or retrieved after each training vector.
2. Training mode '1': Weights are updated, and retrieved after each training vector.
3. Training mode '2': Weights are not updated, but retrieved after each training vector.

6.3 Neural Network Implementation

The hardware circuits for the stochastic system were designed using the Quartus II development environment, which is a graphical hardware design suite packaged with the Nios Development Kit. Each stage of the stochastic network is described below.

6.3.1 Input Stage

The input stage of the stochastic system is separated into two buses, the XBUS and the WBUS. The XBUS connects the Nios micro controller to the input data registers that are arranged in a FIFO configuration. The WBUS connects the controller to the weight modules that store the mean values and system operating parameters. These are also arranged in a FIFO configuration.

After the training and test vectors are generated by the host PC, they are downloaded to the FPGA. They are received by the Nios micro controller and then transferred to the network one vector at a time where they are stored in D-type flip flops (DFFs). These are then connected to 8-bit variable probability generator (VPG8) units (from Section 4.3). At this point, the input vectors are available as a properly formatted stochastic bit stream.

6.3.2 Cellular Automata

The formula for determining the number of CA outputs required is given by the following expression (specific to this method of implementation):

$$numbits = n(d \cdot res + n + divres + 1) + res(d + 1) + 1 \tag{6.5}$$

where n is the number of neurons, d is the dimensionality of the input, *res* is the resolution of the VPG units (8 in this case), and *divres* is the resolution of the divider units (11 in this case). The CA units developed are built in stages of 64 bits. While not optimal, this gives a reasonable size building block for constructing various sizes of network.

A 2 neuron network with 2-D inputs and 8 bits of precision (assuming 11-bit dividers) requires 85 CA taps. A 5 neuron network with 5-D inputs and 8-bits of precision requires 334 CA taps.

Each 64-bit CA is made out of 4 16-bit modules which have interconnected feedback signals. One or more of the four modules for each CA has one bit set during initialization. This CA initialization process is necessary because the rule of the CA is such that it returns a '0' for an input of all '0's. If the CA is not initialized then it will remain in a state of all '0's.

The truth table from the feedback module is written in AHDL (Altera Hardware Description Language) which is a proprietary Altera hardware design language. The other hardware components are constructed with the graphic design component of the Quartus II development environment. A simplified high level block diagram of a CA can be seen in Figure 6.4.

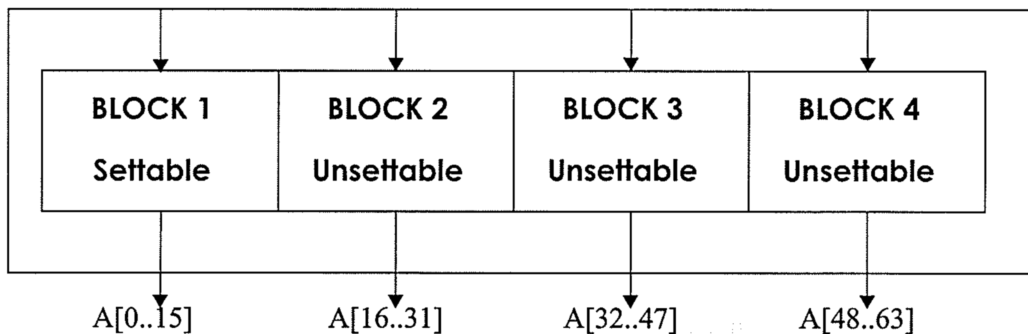


Figure 6.4: Block Diagram of 64-bit CA.

6.3.3 'Weight' Module

The weight values are loaded into 'weight' modules, which are implemented with an 8-bit register and a stochastic counter. The register stores the initial value of the weight, and the counter modifies this value during the training process. The counter has feedback that controls a count enable input (GN in Figure 6.5), limiting the count at either end of the counter's range. A block diagram of the weight module can be seen in Figure 6.5. Each weight module is connected to a VPG8 unit to generate a stochastic bit stream.

There have been many approaches to training stochastic and pulsed neural networks in the past ([5], [10], [11]). Studies have also been done where the weights are generated with conventional computational techniques built into the network [9]. This was shown to give a very high density, though inflexible, network.

From (5.16) the equation for updating the weights is:

$$\Delta\mu_{ik} = \varepsilon y_k(x_i - \mu_{ik}) \qquad (6.6)$$

where ε is the learning parameter, y_k is the output of the neuron, and x_i and μ_{ik} are the input and mean values, respectively.

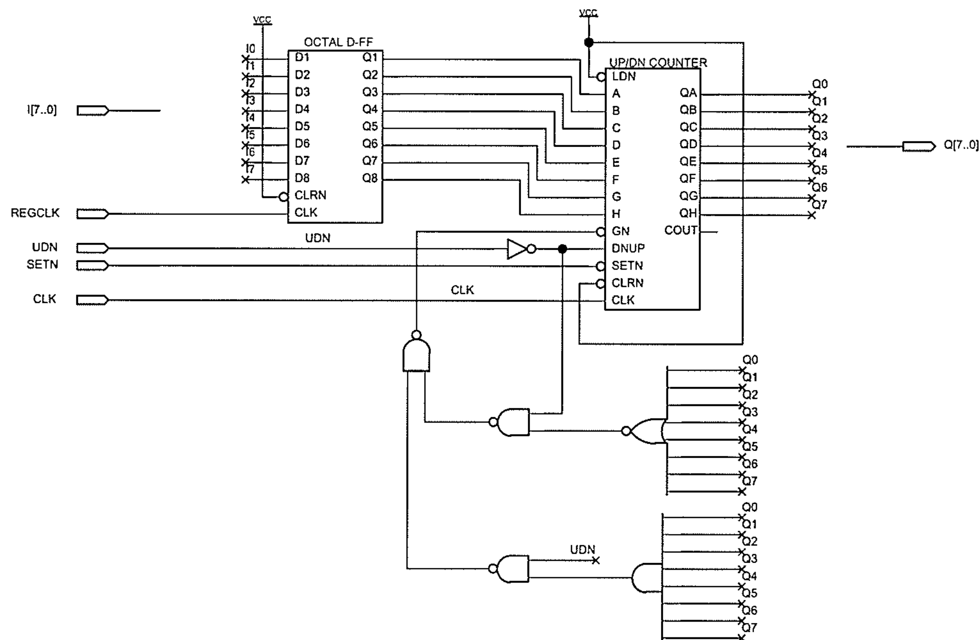


Figure 6.5: Weight module block diagram.

Since the Nios processor is available for sequencing, the learning rate is implemented as a set number of clock cycles as opposed to a stochastic signal. The reasoning for this is as follows. The learning rate may be interpreted as the number of allowable clock cycles that the weight parameter can change during a single learning stage. Since the total number of states of the weight modules of 255, the total number of state changes during a learning stage would be $255 \cdot lrate$. $lrate$ is the hardware implementation of the learning rate ε. As a stochastic parameter, the $lrate$ stream would be subject to the same variation as any other stochastic stream, but does not benefit from a large integration period to reduce the coefficient of variation. The generating probability of the $lrate$ stream is also generally small, on the order of 16/255 or less, which results in $CV > 24\%$ for 255 clock cycles.

To start the training process, the Nios processor asserts a learning enable signal (called 'LEARN') that gates the weight update clock. On each training clock cycle, if the y_k value is '1', the direction of the weight change depends on the value of the $(x_i - \mu_{ik})$ parameter. This signal is generated on the fly by the 'sumsquare' module (Section 6.3.4). If the value of the signal is '1' then the weight will increase, if the value is '0' then the weight will decrease. So if,

$$P(N = 1) = N \tag{6.7}$$
$$P(X = 1) = X \tag{6.8}$$
$$P(W = 1) = W \tag{6.9}$$

then,

$$P_{up} = \frac{1}{2}Y(X + (1 - W)) \tag{6.10}$$

and

$$P_{down} = \frac{1}{2}Y((1 - X) + W) \tag{6.11}$$

so therefore,

$$\Delta W = \varepsilon(P_{up} - P_{down}) \tag{6.12}$$

giving

$$\Delta W = \varepsilon Y(X - W) \tag{6.13}$$

Another common practice during the training process is *weight pruning.* If a neural network is large then there is a chance that not all of the weights will make a significant contribution to the output. The process of weight pruning involves finding the weights that are valued below some threshold and removing them from the network model. The goal of this process is to reduce the number of free parameters that must be determined. This is a very difficult operation to implement in hardware, and no gains in circuit area are achieved once the network is already programmed into the FPGA.

6.3.4 Sumsquare Module

The 'sumsquare' modules are the units which calculate the $\|x_i - \mu_{ij}\|$ value. Input to each module is the ith stochastic input stream, the ij^{th} weight stream, and a stochastic select signal S. The weight stream is inverted and summed with the input vector (using a 2-input MUX). The stream is then squared and transferred to a summing circuit. A diagram of the sumsquare module is shown in Figure 6.6.

Two DFF modules are required to properly isolate the bit streams because of the dual-clocking system implemented in this architecture. If only a single DFF is used, the result is a stream of '1's.

After the vector norms are calculated they are summed with a RISM summer and used as input to an exponential function unit.

6.3.5 Exponential Function Module

The exponential function module is implemented using the VHDL hardware description language. The inputs to this module include a CLOCK signal, an 11-bit G parameter

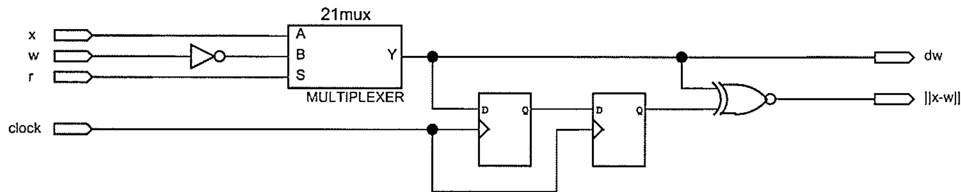


Figure 6.6: Sumsquare module block diagram.

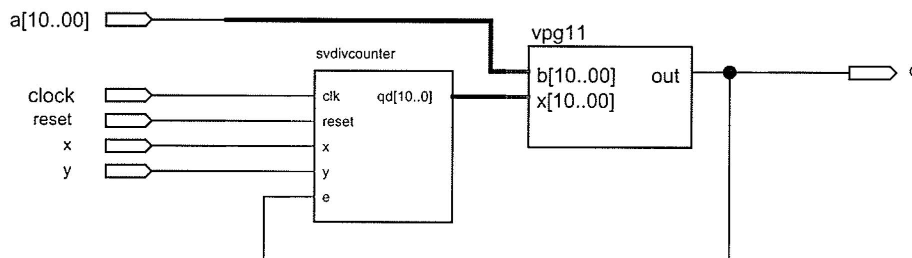


Figure 6.7: 'Svdiv' module block diagram.

(exponential function gain), an UP/$\overline{\text{DOWN}}$ signal, and a RESET signal. All exponential functions are reset to zero after processing each input vector. Code for the implementation of the exponential function is detailed in Appendix A.1. The output from the exponential function is multiplied by the stochastic sum of all exponential units and then used as input to the stepped velocity divider unit.

6.3.6 RISM Summer Module

The RISM summer implements the random incremental selection method that is described in Section 3.2.1. The RISM summer is implemented as detailed in Figure 3.4. The RISM summer was also modelled in VHDL code but used substantially more logic resources than a module implemented using graphical hardware components.

6.3.7 Stepped Velocity Divider Module

The stepped velocity divider units estimate the quotient of the ith neuron output divided by the sum of all n neurons. The inputs to this module include a RESET signal, a CLOCK signal, the dividend (X) and divisor (Y) bit streams, an 11-bit vector of random signals A[10..00], and the feedback error bit E. The 11 random signals drive a VPG11 unit that generates the E signal. A diagram of the stepped velocity divider is shown in Figure 6.7. The 'svdivcounter' is the heart of the divider module and is specified in VHDL. Code for the 'svdivcounter' module is shown in Appendix A.2.

6.3.8 Output Integrators

The output integrators are responsible for accumulating the bit pulses of the output streams, and sending this result to the Nios processor. The output counters have 3 inputs: a RESET signal, an increment signal, and a 16-bit vector representing the number of clock cycles for the integration period. The reason this is not implemented as an 'off-the-shelf' counter is because the output needs to be selected according to the integration period. The output selector routes the 8 most significant bits to the output bus of the counter. The counter is specified in VHDL and the code is detailed in Appendix A.3.

The output of the stochastic network is handled using various multiplexing units that are controlled by the Nios processor. In a 5 neuron network, there are 25 weight values and 5 output values that are read by a single 8-bit port on the controller. The details of the implementation of these units will not be discussed further because they are beyond the scope of the functionality of the stochastic ANN.

6.4 System Sequencing

The clock is controlled by the 'clk_gate' module, which controls a tri-state buffer gating the clock. The 'clk_gate' module counts clock pulses present at the output stage of the stochastic network, and when the correct number of pulses has been counted turns off the clock signal. A signal is also asserted to the Nios micro controller that the system is finished processing the current input vector.

There is also a provision for the stochastic system clock to be clocked in single cycles by the micro controller. This is used during the learning stage and the output gathering stage where only a few clock cycles are required.

There are five sequential stages of the stochastic system that must to be clocked in the correct order for proper operation. These are: the CA, the sumsquare modules, the exponential modules, the normalization dividers, and the output counters. The system is clocked by two clocks that are 180° out of phase. The clock frequency of the system is 50MHz.

6.4.1 Controller Software and Signals

The controller software is programmed into the Nios processor memory. After power-up, the controller waits for a command character input from the host PC. There are six command characters that the controller interprets and these are summarized in Table 6.1.

Table 6.1: Command Character Summary for Stochastic Control Software

Character	Function	Description
R	Reset	Resets the system by zeroing all counters.
I	Initialize CA	Clears the CA and asserts the INIT_CA line, which initializes the CA. The CA is then clocked two times the NUM_CYC parameter which completes the initialization process.
S	Start Processing	Sends each data vector to the network, evaluates result, and sends set of outputs to host PC.
G	Get Weights	Retrieves the values in all weight module registers.
L	Load Data	Causes the controller to enter 'data receive' mode. The first two bytes received indicate number of data vectors to be transmitted.
W	Load Weights / System Parameters	Causes the controller to enter 'system parameter receive mode'. The first two bytes indicate the number of bytes to be transmitted. These values are then clocked into the stochastic system registers.

The system requires several control lines to sequence the clocks, counters, CA, and output circuitry. There are two control lines for the CA that are used to reset it and initialize it, these are RST_CA and INIT_CA, respectively. There are also two control lines for the weight modules: W_SET, and W_REG_CLK. The W_REG_CLK line loads the front-end register of the weight module, and the W_SET line loads the counter with the value in the register. There is also a RESET signal, which controls the state of all of the counters in the system (excluding the weight value counters).

The clock is controlled with three signals: CLOCK_START, SINGLE_CLK_EN, and SINGLE_CLOCK. The CLOCK_START signal clears the counter that gates the system clock, this allows the clock signal to propagate throughout the system until the counter is

full again. The SINGLE_CLK_EN signal selects the system clock or the SINGLE_CLOCK control line. The SINGLE_CLOCK line is a clocking signal that is controlled by the Nios processor. The Nios processor and the stochastic circuits have separate dedicated clock signals from the on-board clock generating circuitry. There are also bus control lines that handle tri-state buffers and multiplexors.

6.5 Hardware Allocation and Circuit Area

The way that logic circuits are physically implemented on the FPGA is the metric applied in determining the efficiency of the layout of a circuit. To understand this, some FPGA terminology must be explained.

The basic element of the FPGA is called the logic cell (LC). This building block is composed of a 4-input look-up-table (LUT) which can be programmed with an arbitrary logic function, a register, and a carry-chain element. The register provides one bit of storage, and the carry-chain provides carry signals for cascaded operations (such as addition). The flip-flop is configurable as a D, T, JK, or SR type flip-flop. An LC can be used as just a register, or just a combinational function, or as a combination of the two. The LC is the unit that will be used to assessing total circuit area.

6.5.1 Circuit Area of Stochastic Elements

Every element in the stochastic system has a circuit size associated with it. These values are found using the Quartus II Project Navigator utility, which lists logic resources required by each component of the design. Not all of the same components use exactly the same logic resources due to the fact that the compiler optimizes for speed and circuit area. Ostensibly, the speed gains from using more LCs may offset the cost of using that LC, provided there are sufficient logic resources available. Table 6.2 summarizes the logic cell allocation of each stochastic processing element. The total LCs column summarizes the total number of LCs for a particular module. The LUT-Only and FF-Only columns give the number of LCs that are only required for these respective elements.

The CA is the largest component of the stochastic system after the Nios controller. The overhead of circuit routing from the perspective of hardware layout becomes an issue as CAs become larger. A CA30 with nearest neighbour connectivity uses only 1 LC per bit

Table 6.2: Summary of Hardware Requirements for Stochastic Logic Elements

Logic Element	Total LCs	LUT-Only LCs	FF-Only LCs
128-bit CA30	128	0	0
128-bit CA38490	137	9	9
384-bit CA38490	425	41	41
Stepped Velocity Divider Counter	120	23	20
8-bit VPG	9	9	0
11-bit VPG	10	10	0
Summing/Squaring Module	9	8	1
Exponential Unit	37	26	10
Weight Module	23	7	8
RISM Adder	9	4	5
Intra-Count Adder (2-1 MUX)	8	8	0
Input Register	8	0	8
Output Counter	96	81	0
Output Multiplexor	16	16	0
16-bit BR Multiplier	417	0	0
16-bit BR Divider	433	433	0
16-bit Adder/Subtractor	17	17	0
Nios Processor	7920	5080	1599

cell. A 128-bit CA38490 uses a total of 137 LCs. The majority of these use both a register and LUT, while nine use only the register and nine use only the LUT portion of the LC. A 384-bit CA uses a total of 425 LCs, 343 of which use both a register and LUT, while 41 use only the register, and 41 use only the LUT.

For a five dimensional, five neuron network, the total area required per neuron is 492 LCs. There are also several circuit elements whose cost can be amortized across all neurons,

these are the input units, the CA, the normalization divider circuits, and the output multiplexor. The area per neuron of these units is 136.6 LCs. The output multiplexor for the weights is not included because it would not be strictly necessary in the final implementation of a system such as this.

For a five dimensional five neuron network, the total amount of the FPGA used is 27%, which equates to 11140 LCs. For a two dimensional two neuron network, the total amount of the FPGA used is 22%, which is 9157 LCs.

The Nios processor is not a stochastic element, but is included for reference. The Nios processor in this project is adapted from a basis design provided by Altera. Due to this, there are many features included in the processor that are not required for this design (i.e. Ethernet controller, IDE interface, etc). These features increase the size of the controller, leading to an overall system that is larger than strictly necessary.

In comparison, a conventional 16-bit BR multiplier circuit uses 417 LCs, and 16-bit divider uses 433 LCs, while a 16-bit adder/subtractor uses 17 LCs. The actual size of these modules depends on the speed at which the resulting calculation is desired versus the allowable circuit size. Building sequential circuits will reduce the total area (less combinational logic) but increase the processing time. The sizes specified are for speed optimized units meaning that there is no sequential logic used.

6.6 Processing Time

The total number of clock cycles required to process each input vector is set by the user. This parameter varies according to the accuracy desired in the computations. The system, as implemented, supports 8 processing times ranging from 2^8 cycles to 2^{16} cycles. Since the system runs completely in parallel, the same number of clock cycles are required to run a 2 neuron network as to run a 10 neuron network. A conventional ANN process requires processing time proportional to $d \cdot n$ where n is the number of neurons and d is the dimensionality of the input vector. This is an advantage for stochastic architectures from an expandability point of view.

Assuming that multiplication, division and addition are all a single operation for a processor, the number of operations required for an ANN can be determined by the following formula:

$$N_{operations} = n(2d + 29) - 1 \tag{6.14}$$

where n is the number of neurons and d is the dimensionality of the input vectors. This formula is specific to this project as it includes the simplifications introduced to the general case (e.g. removing the linear output layer). This is also assuming that the exponential operation is implemented as a 5th order Taylor series approximation.

Assuming 16-bit accuracy and single-cycle mathematical operations, a BRI network is significantly faster when implemented on specialized hardware available in this FPGA. In the absence of the Nios processor the final implementation of this system could be fine tuned for significantly less circuit area. The Nios controller provides a convenient platform for control in order to determine the overall performance of a stochastic ANN for later optimization.

Testing and Results

This chapter describes the problems that have been used to test the operation of the stochastic neural network and the results of these tests.

7.1 Test Problem

The problem created to verify the functionality of the stochastic system is a simple one. Consider a 2-dimensional input space containing several data clusters. The goal is to initialize the network to a random state, train the network with the data vectors, and determine if the network converges to the mean values of the input data distributions. The training data is generated using a normally distributed RNG with a variance of 0.05, the test data has a variance of 0.10. A sample 2-D input space with the mean vectors of the Gaussian units super-imposed on it can be seen in Figure 7.1.

7.1.1 Network Error

To gauge the system's effectiveness an error measure needs to be defined. The error cannot be defined in absolute terms, because there is no knowledge of the actual form of the input data. The error of the network will be defined to be the Euclidean distance from the current position of the weight vector of the 'winning' neuron to the current input vector.

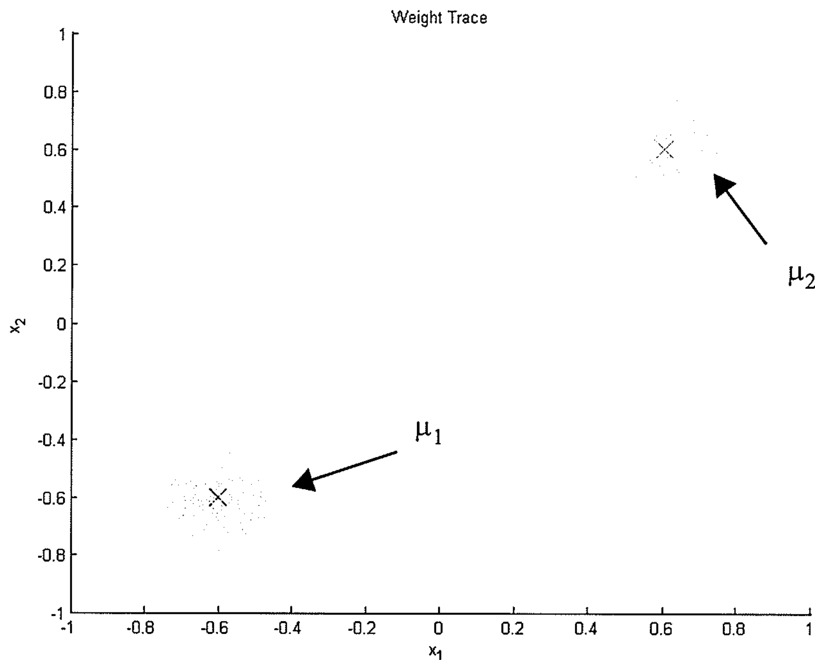


Figure 7.1: Sample 2-D Input Space.

The goal is that as each neuron takes more responsibility for a group of data points, the total error will be reduced. The expression for the error is:

$$d = \sqrt{\sum_{i=0}^{N-1} (x_i - w_{ki})^2} \tag{7.1}$$

The d value is calculated for all N training vectors, where the index k corresponds to the weight vector of the 'responsible' neuron for the current input. The total network error for one training epoch is then defined as

$$Error = \sum_{N} d_n \tag{7.2}$$

where there are N data vectors in the training set. After each training epoch the order of the input data is randomized before being presented again to the network.

The network is initialized into a random state, and the input vectors are then applied to it. The error after each epoch of training is measured, and the training process is repeated until the error has stabilized to a small value.

7.1.2 Network Training Results

Before the hardware stochastic network is tested, a simulation is performed in software using conventional BR arithmetic. The goal of the simulation is to show the theoretical performance that the stochastic network is to emulate. The goal for the network training process is to alter the mean vectors of the Gaussian function to model the input data distribution, and by doing so reduce the error calculated after each training epoch. After the training process the output from the neurons should correctly identify which data cluster each input vector belongs to.

The learning rate for this simulation is $\varepsilon = 0.004$, there are 200 vectors in the training set, and 400 vectors in the test set. The actual means of the input data distributions are $(-0.6, -0.6)$ and $(0.6, 0.6)$ and the gain parameter of the simulated units is $G = 4$. During simulation, the network correctly converges to the proper mean values. An example is shown in Figure 7.2. The network error also decreases as expected for each training epoch, as seen in Figure 7.3.

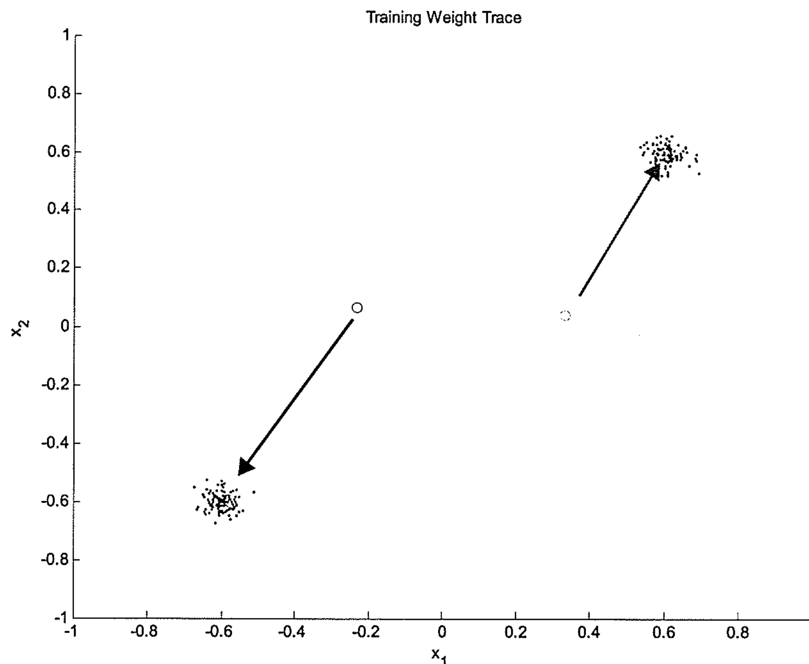


Figure 7.2: Software simulation training results. 2-D inputs, 2 data clusters.

The stochastic system is tested using an integration period of $t = 65535$ clock cycles. The number of training cycles during the test portion of the simulation is reduced to $t = 16383$ cycles. The number of vectors in both the training and test sets are 300, and the number of epochs is 25. The G parameter is initially $G = 500$, and is increased to $G = 1500$ during training. The training vectors are normally distributed with means of (-0.6, -0.6) and (0.6, 0.6), and a variance parameter of $\sigma^2 = 0.03$. The test vectors have the same mean values, but a variance parameter of 0.09. The learning rate parameter is initially set at 8 cycles and is reduced to 1 cycle during training. The number of stochastic system clock cycles to evaluate this data is 65535. System running time for these parameters is 2:36 minutes. A significant portion of this time is spent communicating with the host PC.

Figure 7.4 shows a plot of the input space with the weights in the initial ('o's) and final ('x's) positions for the stochastic hardware ANN. The network error in the system is shown in Figure 7.5. The network error is initially high, it does eventually decrease but there is still significant fluctuation. This is due to the stochastic nature of the system (pri-

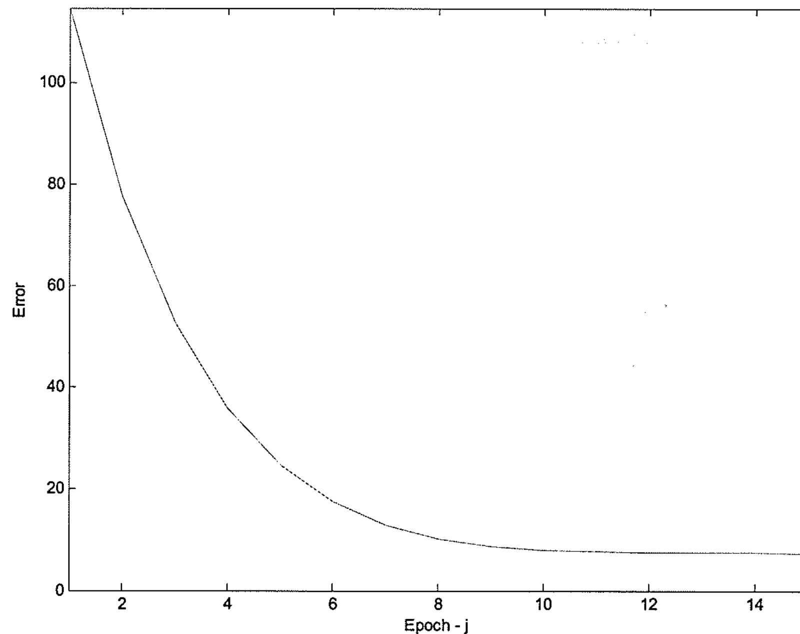


Figure 7.3: Software simulation. Network error vs. Epoch.

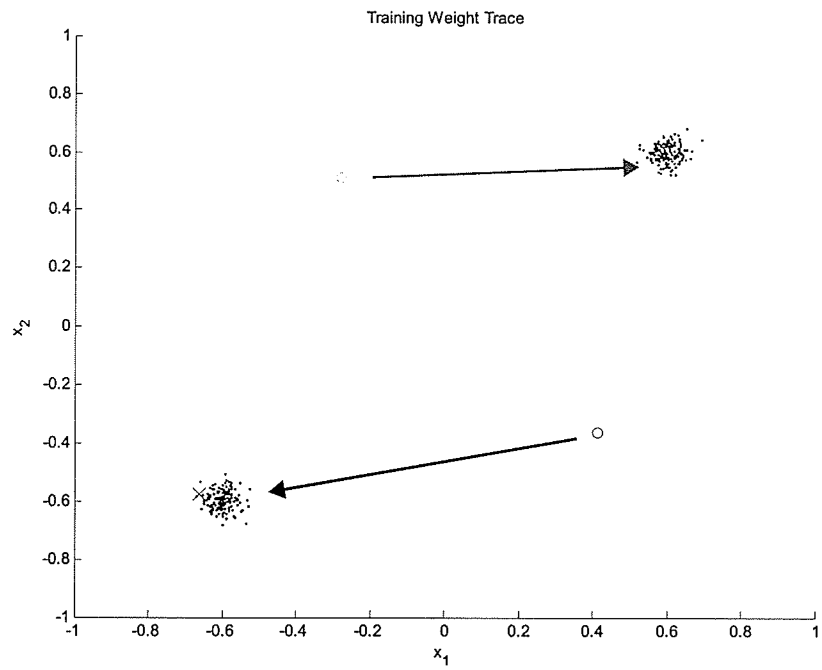


Figure 7.4: Hardware stochastic network training results. 2-D inputs, 2 data clusters.

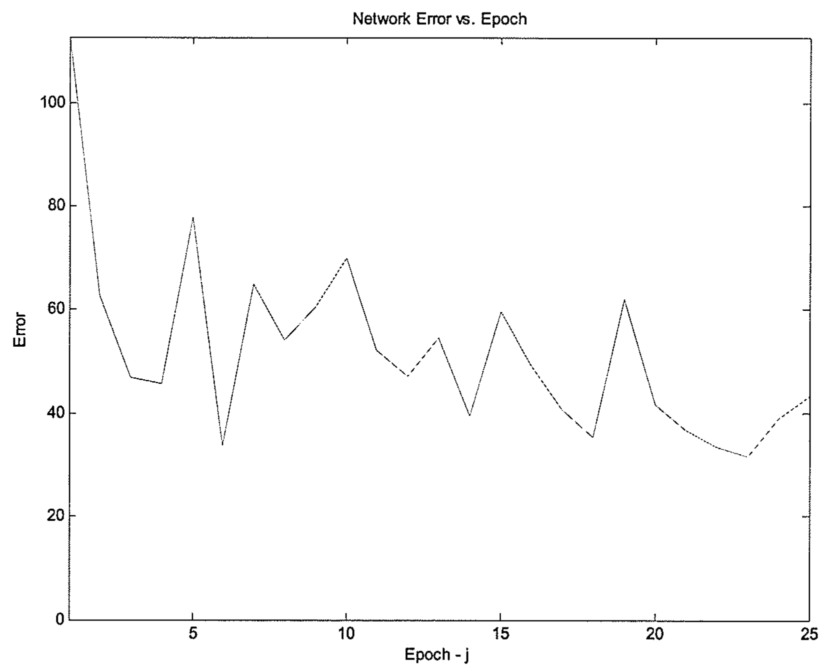


Figure 7.5: Hardware stochastic system. Network error vs. Epoch.

marily due to the outputs of the exponential functions). The weight values after each epoch are saved and because of this fluctuation, the values from the epoch with the least error are used in evaluating the test set.

The output of the network from evaluating the test set can be seen in Figure 7.6. The response of each neuron is shown for all data points. The red circles represent neuron '1' and the black circles represent neuron '2'. Many tests were performed, although only one set of results will be shown for this configuration of input data. The network will reliably represent the data with acceptable accuracy. In this simple case, the network will identify the test vectors correctly 100% of the time, if training is successful.

The network was also presented with 3 clusters of input data. Since the network in this case is limited to two neurons, it is desired that one neuron take responsibility for two of the clusters, while the other neuron takes responsibility for the 3rd cluster. A plot of the training process resulting for this case can be seen in Figure 7.7, the neuron outputs for this can be seen in Figure 7.8. The same network operating parameters as in the previous

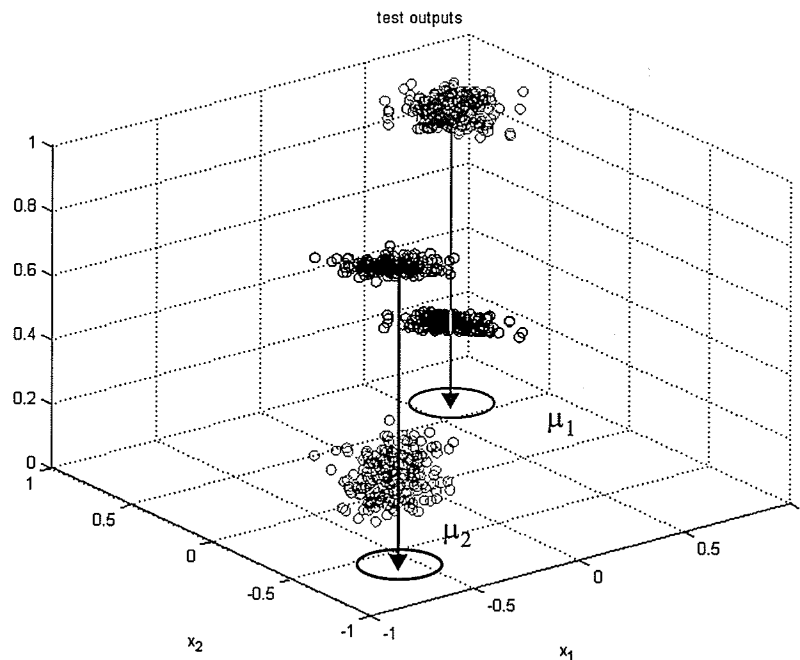


Figure 7.6: Hardware Stochastic System. Neuron response from test set.

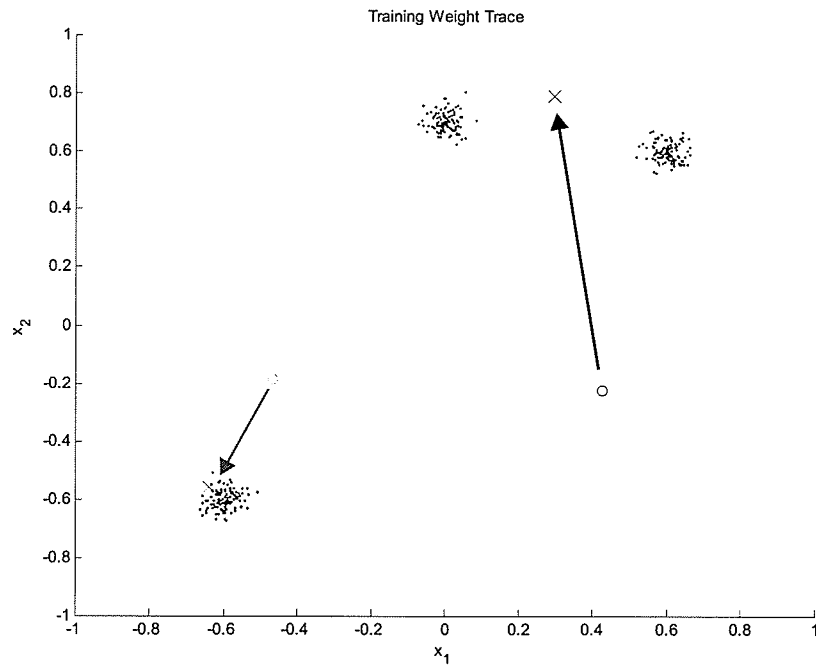


Figure 7.7: Training results for 3 input clusters.

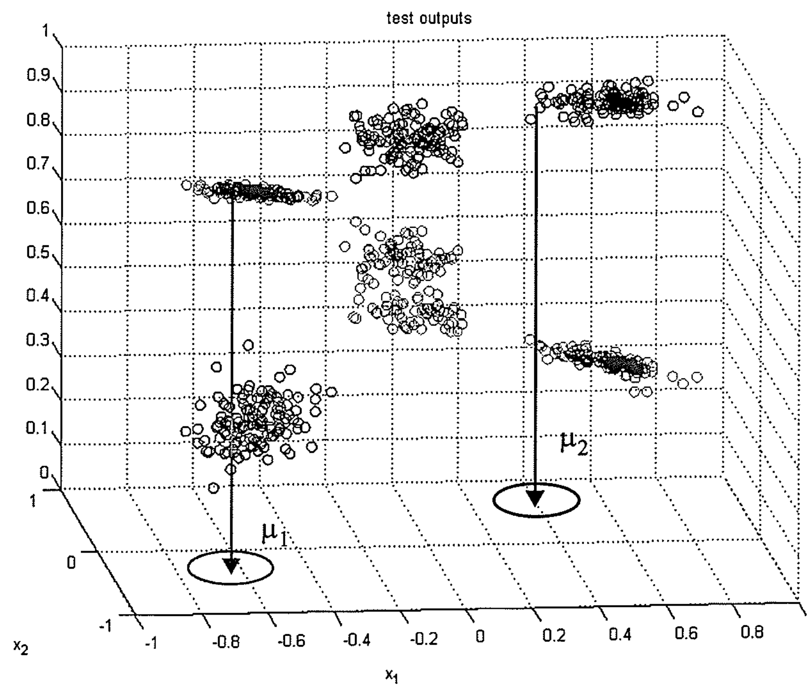


Figure 7.8: Stochastic system response from test vectors. 3 Data clusters.

case were used. The classification rate for this problem is 100% for the neuron covering a single data cluster, and 85% for the neuron covering two data clusters, on average. Other configurations of input data clusters are also tested but the results are not shown here. The results indicate that the SANN and the BRI network have similar classification capabilities.

7.2 A Simulated Visual Environment

To further demonstrate the usefulness of the system, a more complex problem was also considered. The problem is based on the studies performed by McNeill in [6]. Consider an artificial environment that might be encountered by a mobile robot that consists of several point sources of light (e.g. LEDs). The robot has 5 sensors that observe the light inputs. Initially, the system is trained with input patterns corresponding to the five lights, where each neuron should give a high output when it's corresponding light is on. For the test case, four new light sources will be added at the corners of the input grid. The goal is for the system to generalize information learned in the training process to give spatial information about the four new visual inputs. The system should also be able to generalize information when Gaussian noise is added to the sensor readings. The physical representation of light sources and sensors is shown in Figure 7.9.

7.2.1 Input Generation

The input patterns are synthesized values based on the theoretical intensity of a light source at a distance. The theoretical magnitude of the light intensity at the sensor plane is

$$\frac{1}{d^2} = \frac{1}{\sqrt{(l^2 + r^2)}} \tag{7.3}$$

The r value varies depending on the sensor under consideration. The intensity of a light source on different points of a grid is calculated as the Euclidean distance of the source to that point. The intensity of light on a 5x5 grid is defined below:

$$I = \frac{1}{\begin{bmatrix} (l^2+8r^2) & (l^2+5r^2) & (l^2+4r^2) & (l^2+5r^2) & (l^2+8r^2) \\ (l^2+5r^2) & (l^2+2r^2) & (l^2+r^2) & (l^2+2r^2) & (l^2+5r^2) \\ (l^2+4r^2) & (l^2+r^2) & (l^2) & (l^2+r^2) & (l^2+4r^2) \\ (l^2+5r^2) & (l^2+2r^2) & (l^2+r^2) & (l^2+2r^2) & (l^2+5r^2) \\ (l^2+8r^2) & (l^2+5r^2) & (l^2+4r^2) & (l^2+5r^2) & (l^2+8r^2) \end{bmatrix}} \quad (7.4)$$

The reading of each sensor is based on this matrix. A 'sensor window' of the form

$$SW = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad (7.5)$$

is applied to the matrix corresponding to the current active light source. For example, when calculating the sensor readings when the centre light source is active, the window is centred on element I_{33} of the matrix. If the bottom right light source is activated then the mask is centred on element I_{22}. For simplicity, the l parameter is set to 1 for all experiments.

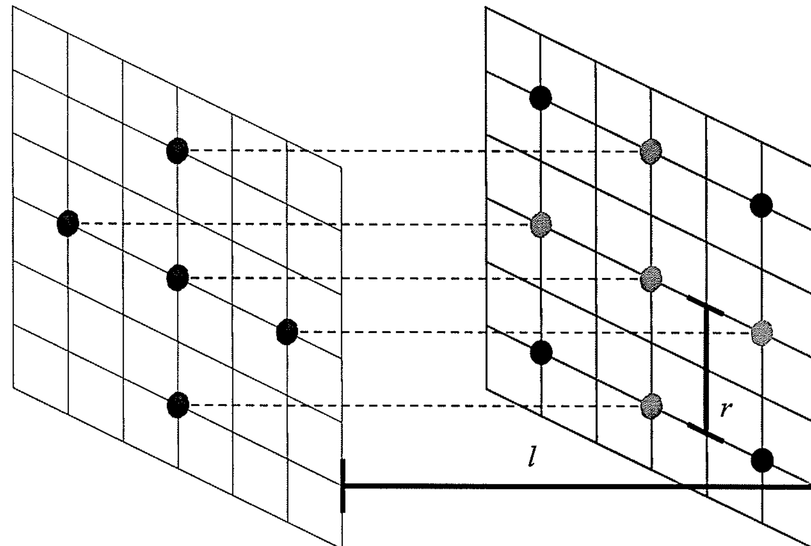


Figure 7.9: Sensor and light input grid used for network training. Sensors are denoted by the red dots, training light sources are green, test light sources are blue.

7.2.2 Results

This problem was also simulated in software using conventional arithmetic. The BRI ANN gives very promising results that are hoped to be repeated with the stochastic system.

Gaussian additive noise was also applied to the simulated sensor readings to emulate the noise present in a real world environment. The output of the simulations can be seen in Figure 7.10 with no noise added and in.Figure 7.11 with noise added. The noise had a mean of zero with a variation of $\sigma^2 = 0.1$

The plots show the test vector response of each neuron, arranged corresponding to the sensors that they represent. While the inputs for the training set are applied in a random order, the inputs for the test case are applied in a set order. This does not affect correct operation of the network because the random ordering is required only for training purposes. There are 9 sets of input vectors, 50 vectors per light module, giving a total of 450 vectors in the test set. The vectors are presented to the network starting with the sensor readings corresponding to the top-left light being on. The active light then switches to the centre of the

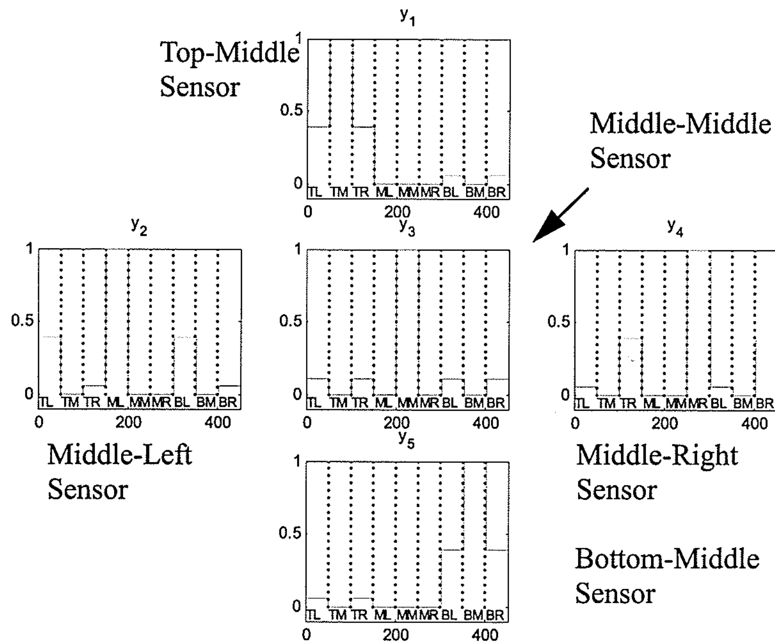


Figure 7.10: Output from each neuron after training.

top row, and continues in a left-to-right, up-to-down pattern (the active light is also labelled in the bottom of each column). The neuron representing the top sensor is located in the top of the cross (labelled 'y_1').

The mean of the output for each neuron is also computed for each input pattern. These values for the simulated BRI network are shown in Table 7.1.

Each column in Table 7.1 corresponds to a single neuron (T = Top, M = Middle, B = Bottom, L = Left, R = Right). The output with the additive noise causes some variation in the output of the neurons, though the individual responses do not vary considerably.

This error is calculated in a similar way as in the simple problem discussed previously. The Euclidean distance from the current data point and the 'winning' neuron's weight vector is evaluated and summed. The error for the training process with the conventional arithmetic is shown in Figure 7.12. The error settles around 52 for this amount of additive noise. When there is no noise added to the input data, the error is minimized to zero.

The stochastic system is set up with similar parameters to the BRI system. There are 50 vectors per light source, for a total of 250 training vectors and 450 test vectors. The G

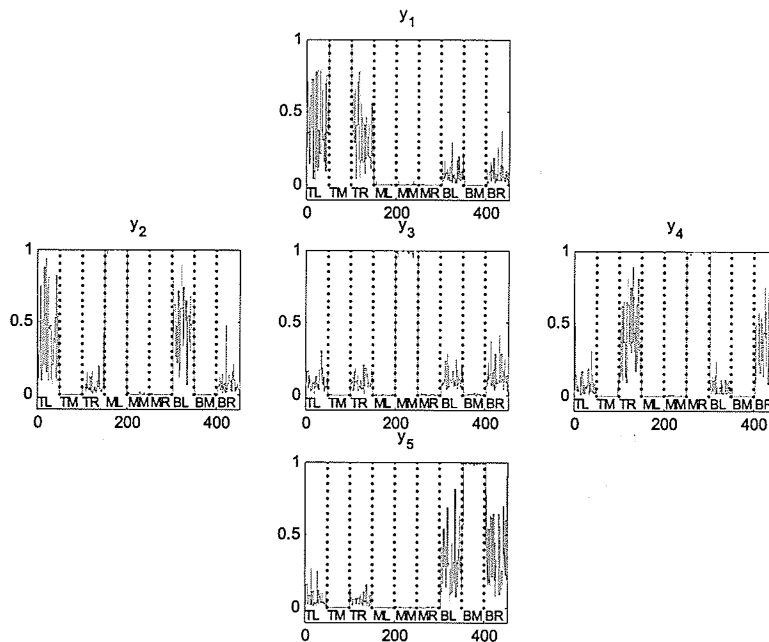


Figure 7.11: Output from conventional arithmetic ANN with additive noise.

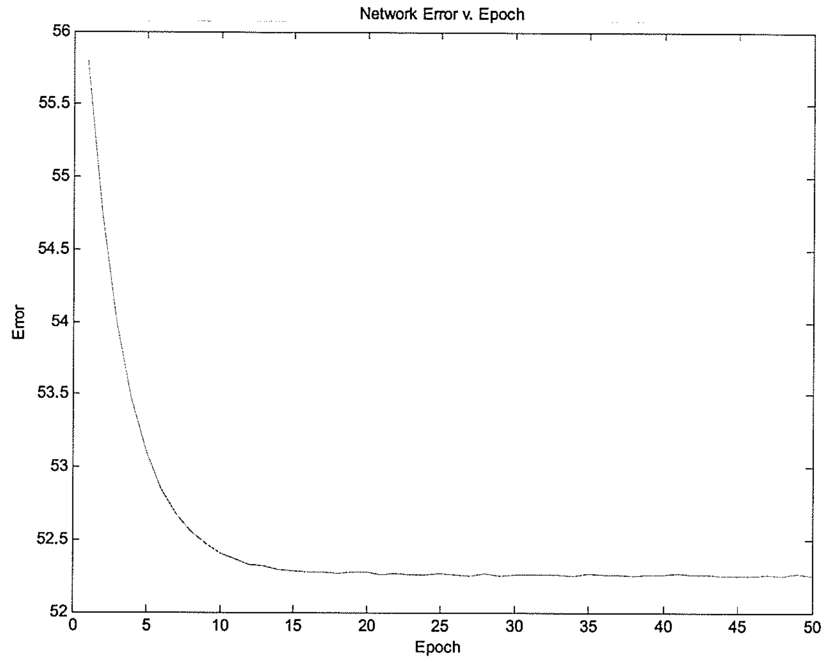


Figure 7.12: Error corresponding to training process of simulated BRI network.

Table 7.1: Averaged Response of Trained BRI Neurons. $\sigma^2 = 0.1$.

LIGHT	TM	ML	MM	MR	BM
TL	0.2119	0.1686	0.3530	0.1446	0.1220
TM	0.5923	0.0719	0.2012	0.0818	0.0529
TR	0.2351	0.1321	0.3274	0.1945	0.1110
ML	0.0769	0.5784	0.1984	0.0596	0.0868
MM	0.0651	0.0718	0.7383	0.0635	0.0613
MR	0.0706	0.0535	0.2114	0.5937	0.0708
BL	0.1136	0.2154	0.3359	0.1206	0.2145
BM	0.0650	0.0635	0.2469	0.0657	0.5589
BR	0.1168	0.1206	0.3961	0.1664	0.2001

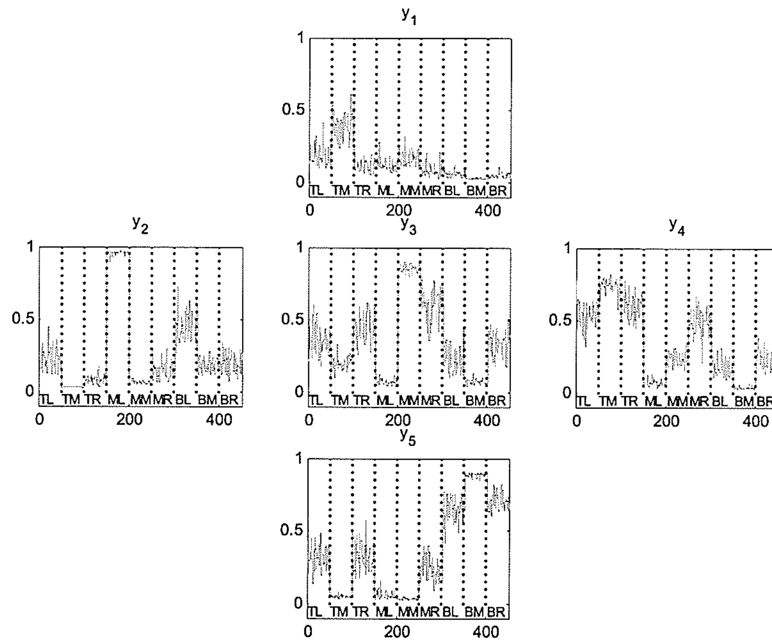


Figure 7.13: Output from trained stochastic network.

parameter of the exponential units varies from 1000 to 1850 (selected based on performance observations) throughout the training process. The learning rate is set to 2 cycles, giving a maximum possible weight probability adjustment of 0.008 per training vector. The output from a test of the stochastic network can be seen in Figure 7.13.

The training process of the stochastic network proved to be fairly unreliable due to the extreme fluctuation of the outputs. There are cases where when the same input is applied to the network, the output will change in value by up to 70%. This creates a huge problem during the learning process, where accurate results are necessary to properly adjust the weight values. On some occasions the network would correctly train itself to properly identify the test vectors, but the majority of times this was not achieved. As an example of a successful training operation, the output of the network when there is no additive noise is shown in Figure 7.13. The error associated with the training process for the stochastic network is shown in Figure 7.14. This error is initially very high, and stabilizes to a value that is much higher than in the BRI case.

To further test the operation of the stochastic network in spite of the poor training performance the weight parameters generated with the conventional arithmetic simulation were used. This procedure yielded satisfactory results for the application. Examples of network output can be seen in Figure 7.15 with additive Gaussian noise with $\sigma^2 = 0.05$ and Figure 7.16 with additive noise with $\sigma^2 = 0.10$.The average values of the output of the network for each test input (and added noise) is shown in Table 7.2.

When properly trained, the stochastic network reliably indicates which input source is active. The network is also capable of tracking changes in the input vectors that may occur over time. The error present in the network reaches its minimal value after approximately 40 epochs.

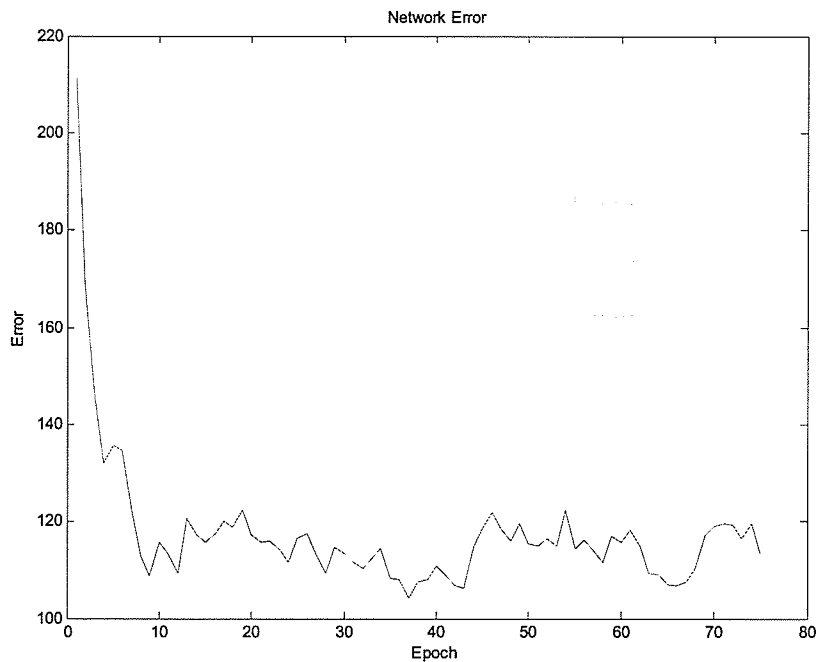


Figure 7.14: Error plot from training the stochastic neural network.

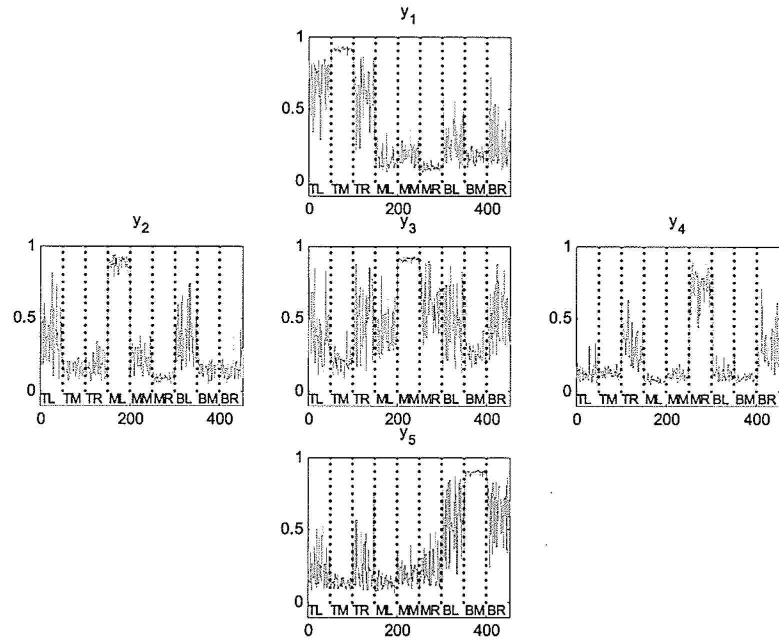


Figure 7.15: Output from pre-trained stochastic network, noise variance is $\sigma^2 = 0.05$.

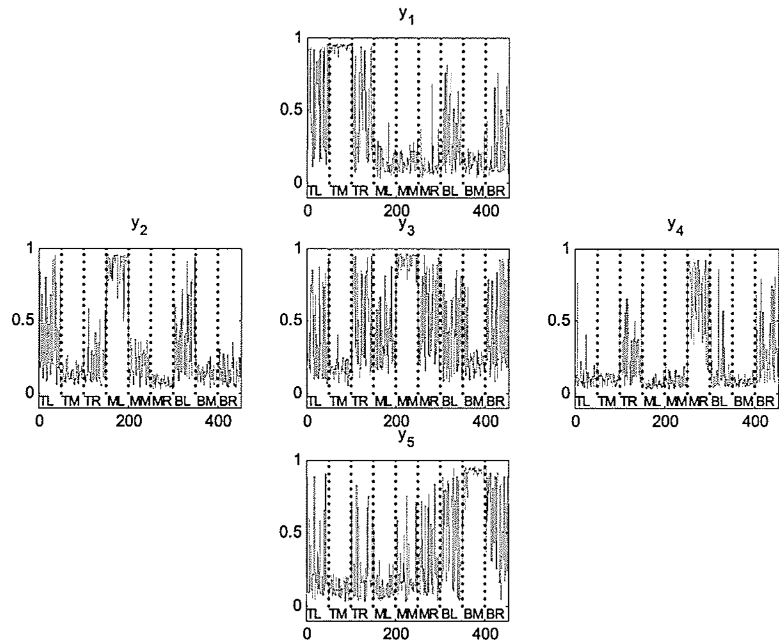


Figure 7.16: Output from pre-trained stochastic network, noise variance is $\sigma^2 = 0.10$.

Table 7.2: Averaged Response of Trained Stochastic Network. $\sigma^2 = 0.1$.

LIGHT	TM	ML	MM	MR	BM
TL	0.3655	0.1787	0.3857	0.0866	0.1427
TM	0.7107	0.0769	0.1206	0.0704	0.0657
TR	0.3214	0.1095	0.3798	0.1813	0.1826
ML	0.0748	0.6820	0.2089	0.0433	0.0819
MM	0.1027	0.1475	0.7283	0.0525	0.1076
MR	0.0544	0.0513	0.4006	0.5134	0.1354
BL	0.1705	0.2017	0.3456	0.0896	0.3674
BM	0.0919	0.0852	0.1275	0.0525	0.7562
BR	0.1645	0.1027	0.3931	0.1478	0.3522

8

Conclusions and Future Work

This thesis has studied stochastic architecture and its application to constructing dedicated circuitry for hardware neural networks. The test application for the circuits designed and constructed was a simulated vision problem for a mobile robot. This problem was attacked using both a traditional computational method and the proposed stochastic method. The results of these experiments are that the stochastic method, while comparable to the BRI method, yielded inferior results on the whole. In terms of processing time and result accuracy the BRI network outperformed the SANN.

This may be due to several factors including over simplification of the network model and/or too much variance in the computations. It is more likely that the variance in the calculations had the most detrimental effect on performance. The addition of a linear output layer would alleviate this somewhat.

The stochastic network was observed to perform adequately when supplied with weight vectors determined by a BRI network. Once the network is put in a learned state, it is possible to adjust to any changes in the input data that may occur over time. For example, if sensors were to deteriorate, the readings might change in value. This could be corrected by by allowing the network to continue to train a small amount.

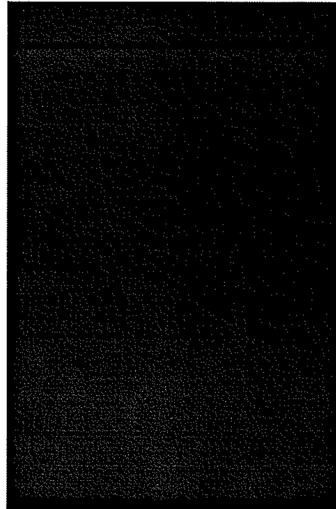
A study has been performed [5] using the full RBF neural network model that achieved satisfactory results identifying typed characters. The exclusion of the linear output layer in

the SANN examined in this thesis adds a significant amount of potential imprecision when the desire is for a hard decision. A linear output layer is only possible however, when apriori information about the input data is available. This could potentially alleviate the problem of high variance in the computations.

In conclusion, this type of simplified stochastic network model is not well suited to this particular problem. The conventional BRI implementation offers much better precision (in the absence of bit errors) and computation time when implemented on an FPGA device such as this one. Stochastic networks are better suited to recognition problems when labelled data is available ([5], [10], [11]). The extremely flexible nature of this FPGA system creates an environment where specialized hardware for a conventional computational architecture can be implemented just as easily as custom hardware for an atypical one. If one was restricted to a lower functionality FPGA then the stochastic architecture might be a more attractive alternative.

Future work in this area would include an automatic system generator implemented in a hardware description language. This would be quite feasible due to the extremely regular nature of the stochastic network architecture.

Further research could also be performed in the area of random number generation. A neighbourhood-of-3 CA would be more efficient to implement on an FPGA than the neighbourhood-of-4 CA considered.



References

[1] B.R. Gaines, "Stochastic Computing Systems," *Advances in Informations Systems Science,* J.F. Tou, ed., vol. 2, chapter 2, pp37-172, New York: Plenum Press, 1969.

[2] C.M. Bishop, *Neural Networks for Pattern Recognition*. Oxford: Clarendon Press, 1995.

[3] B.D. Brown and H.C. Card, "Stochastic Neural Computation I: Computational Elements," *IEEE Trans. Computers,* vol. 50, no. 9, pp. 891-905, Sept. 2001.

[4] B.D. Brown and H.C. Card, "Stochastic Neural Computation II: Soft Competitive Learning," *IEEE Trans. Computers,* vol. 50, no. 9, pp. 906-920, Sept. 2001.

[5] B.D. Brown, "Soft Competitive Learning Using Stochastic Arithmetic," M.Sc. thesis, Dept. of Electrical and Computer Eng., Univ. of Manitoba, 1998.

[6] D.K. McNeill, "Adaptive Visual Representations For the Autonomous Mobile Robots Using Competitive Learning Algorithms," Ph.D thesis, Dept. of Electrical and Computer Eng., Univ. of Manitoba, 1998.

[7] P.D. Hortensius, R.D. McLeod, and H.C. Card, "Parallel Random Number Generation for VLSI Systems Using Cellular Automata," *IEEE Trans. Computers*, vol. 38, no. 10, pp. 1466-1472, Oct. 1989.

[8] B. Shackleford, M. Tanaka, R. J. Carter and G. Snider, "FPGA Implementation of Neighbourhood-of-Four Cellular Automata Random Number Generators," *Proc. of the 2002 ACM/SIGDA 10th Int. Symposium on FPGAs*, pp. 106-112, 2002.

[9] Stephen L. Bade and Brad L. Hutchings, "FPGA-based stochastic neural networks - Implementation," *Proc. of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 189- 198, 1994.

[10] J.A. Dickson, R.D. McLeod, and H.C. Card, "Stochastic Arithmetic Implementations for Neural Networks with In Situ Learning," in *Proc. Int. Conf. Neural Networks*, pp. 711-716, 1993.

[11] J Zhao, J. Shawe-Taylor, and M. van Daalen, "Learning in Stochastic bit-stream neural networks," *Neural Networks,* vol. 9, pp. 991-998, 1996.

[12] H.C. Card and D.K. McNeill, "Gaussian Activation Functions Using Markov Chains," *IEEE Trans. Neural Networks*,vol. 13, no. 6, Nov. 2002.

[13] S.J. Min, E.W. Lee, S.I. Chae, "A Study on the Stochastic Computation Using the Ratio of One Pulses and Zero Pulses", *Proceedings of ISCAS,* London, pp.471-474, 1994.

[14] A. Dinu, M.N. Cirstea, M. McCormick, "Stochastic Implementation of Motor Controllers," *ISIE 2002. Proc. of the 2002 IEEE International Symposium on Industrial Electronics*, vol. 2, pp. 639 - 644, July 2002.

[15] H.C. Card, "Compound Binomial Processes in Neural Integration," *IEEE Trans. Neural Networks*, vol. 12, no. 6, Nov. 2001.

[16] H.C. Card, "Input Multiplexing in Artificial Neurons Employing Stochastic Arithmetic," *Neural Processing Letters,* vol. 15, pp. 1-8, Netherlands: Kluwer Academic Publishers, 2002.

[17] A. Astaras, R. Dalzell, A. Murray, M. Reekie, "Pulse-Based Methods for Probabilistic Neural Computation," *Proc. 7th Int. Conf. Microelectronics for Neural, Fuzzy, and Bio-Inspired Systems,* pp. 96-102, 1999.

[18] D.J. Mayes, A.F. Murray, H.M. Reekie, "Pulsed VLSI for RBF Neural Networks," *IEEE Int. Symposium on Circuits and Systems,* vol. 3, pp. 297-300, May 1996.

[19] H.C. Card, "Dynamics of stochastic artificial neurons," *Neurocomputing Letters*, vol. 41, pp. 173-182, 2001.

[20] H.C. Card, "Stochastic Radial Basis Functions," *Int. Journal Neural Systems,* vol. 11, no. 2, pp. 203-210, 2001.

[21] D.K. McNeill, H.C. Card, "Dynamic Range and Error Tolerance of Stochastic Neural Rate Codes," *Neurocomputing Letters*, vol 48, pp. 905-917, 2002.

[22] J.M. Quero, J.G. Ortega, C.L. Janer, L.G. Franquelo, "VLSI Implementation of a Fully Parallel Stochastic Neural Network," *IEEE World Congress on Comp. Intelligence,* vol. 4, pp. 2040-2045, 1994.

[23] J.G. Ortega, J.M. Quero,C.L. Janer,L.G. Franquelo, "Synaptic Weight Generation in VLSI Stochastic Neural Networks," *IEEE Conf. Neural Networks*,vol. 1, pp. 179 - 182, Dec. 1995.

[24] Y. Kondo, Y. Sawada, "Functional Abilities of a Stochastic Logic Neural Network," *IEEE Trans. Neural Networks,* vol. 3, no. 3, May 1992.

[25] A. Torralba, F. Colodro, E. Ibanez, and L.G. Franquelo "Two Digital Circuits for a Fully Parallel Stochastic Neural Network," *IEEE Trans. Neural Networks,* vol. 6, no. 5, Sept. 1995.

[26] S. Haykin, *Neural Networks, A Comprehensive Foundation*, New York: Macmillan College Publishing Company, 1994.

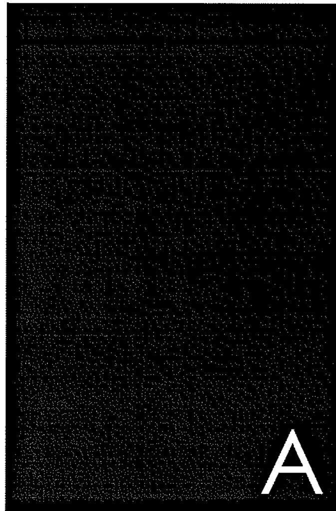
[27] S. Wolfram, "Statistical Mechanics of Cellular Automata," *Rev. Modern Physics*, vol. 55, pp. 601-644, 1983.

[28] S. Wolfram, “Random Sequence Generation by Cellular Automata,” *Advances in Applied Mathematics*, vol. 7, pp. 123-169, 1986.

[29] P. Horowitz, W. Hill, *The Art of Electronics*, *2nd Edition*, New York: Cambridge University Press, 1989.

[30] A. Papoulis, *Probability, Random Variables, and Stochastic Processes, 3rd Edition,* New York: McGraw Hill, 1991.

[31] D.W. Trim, *Calculus for Engineers*, Prentice Hall, 1998.



VHDL Source Code

This appendix contains VHDL code for selected components from the stochastic system.

A.1 sExp.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY sExp IS

  PORT
  (
    up   : IN std_logic;
    reset: IN BIT;
    clk  : IN BIT;
    G    : IN    std_logic_vector(10 downto 0);
    q    : OUT std_logic

  );
END sExp;

ARCHITECTURE arch OF sExp IS
  signal cnt : std_logic_vector(10 downto 0);

  BEGIN-- begin architecture
  PROCESS (clk,reset,up,G) -- reset and increment process
    BEGIN

      IF reset = '0' THEN-- reset counters
        cnt      <= "00000000000";
      ELSIF (clk'EVENT AND clk = '1') THEN
```

```
      IF up = '1' AND CNT < ("11111111111") THEN
        CNT <= CNT + "00000000001";
      ELSIF up = '0' AND CNT > "00000000000" THEN
        CNT <= CNT - "00000000001";
      END IF;
    END IF;
    IF CNT < G THEN
      q <= '1';
    ELSE
      q<= '0';
    END IF;
  END PROCESS;
END arch;
```

A.2 svdivcounter.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY svdivcounter IS
  PORT(
    clk    : IN  BIT;
    reset  : IN  BIT;
    x      : IN  std_logic;
    y      : IN  std_logic;
    e      : IN  std_logic;
    ocount : OUT std_logic_vector(11 downto 0);
    qd     : OUT std_logic_vector(10 downto 0)
  );
END svdivcounter;

ARCHITECTURE arch OF svdivcounter IS
    signal mult     : std_logic;
    signal inc      : std_logic;
    signal cnt      : std_logic_vector(11 downto 0);
    signal schedule : std_logic_vector(10 downto 0);
    signal c_int    : std_logic_vector(10 downto 0);
    signal old_cnt  : std_logic_vector(11 downto 0);

    BEGIN     -- begin architecture
    PROCESS (clk,reset,x,y,e,inc,mult) -- reset and increment process
      BEGIN
        mult <=   X XOR (y AND e); -- test if cnt state needs to be changed
        inc  <=  (X AND NOT(Y AND E)); -- direction of change

        IF reset = '0' THEN            -- reset counters
          cnt      <= "011111111111";-- and registers
          old_cnt  <= "011111111111";
          schedule <= "00100000000";
          c_int    <= "00000000000";
        ELSIF (clk'EVENT AND clk = '1') THEN
```

```
        IF inc = '1' AND mult = '1' AND CNT < ("11111111111" - SCHED-
ULE) THEN
          CNT <= CNT + SCHEDULE; -- increment counter, adjust C_INT
          C_INT <= C_INT + SCHEDULE;
        ELSIF inc = '0' AND mult = '1' AND CNT > SCHEDULE THEN
          CNT <= CNT - SCHEDULE; -- decrement counter
        ELSIF mult = '0' THEN
          CNT <= CNT;             -- do not adjust counter
        END IF;

        IF C_INT = "00000000000" AND inc = '1' AND mult = '1' THEN
          SCHEDULE(9 downto 0) <= SCHEDULE(10 downto 1);
          schedule(10) <= '0'; -- check C_INT, adjust SCHEDULE
        END IF;
      END IF;
    END PROCESS;
END arch;
```

A.3 outcount_wsel.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY outcount_wsel IS
  PORT(
    reset   : IN  std_logic;
    clk     : IN  std_logic;
    num_cyc : IN  std_logic_vector(15 downto 0);
    output  : OUT std_logic_vector(7 downto 0)
    );
END outcount_wsel;

ARCHITECTURE arch OF outcount_wsel IS
signal cnt: std_logic_vector(15 downto 0);
BEGIN
  PROCESS(clk,reset,num_cyc)
  BEGIN
    IF (reset = '0') THEN
      cnt <= "0000000000000000"; -- zero counter on reset
    ELSIF (clk'EVENT AND clk = '1') THEN
      IF cnt < num_cyc THEN
        cnt <= cnt+1;            -- increment counter on clock cycle
      END IF;
    END IF;
  END PROCESS;

WITH num_cyc SELECT
                                -- select
  output <= cnt(15 downto 8) when "1111111111111111",
            cnt(14 downto 7) when "0111111111111111",
            cnt(13 downto 6) when "0011111111111111",
```

```
            cnt(12 downto 5) when "0001111111111111",
            cnt(11 downto 4) when "0000111111111111",
            cnt(10 downto 3) when "0000011111111111",
            cnt(9 downto 2)  when "0000001111111111",
            cnt(8 downto 1)  when "0000000111111111",
            cnt(7 downto 0)  when "0000000011111111",
            cnt(7 downto 0)  when   OTHERS;
END arch;
```