Stochastic Arithmetic Implementations of Artificial Neural Networks

BY

Jeffrey A. Dickson

A thesis
presented to the
University of Manitoba
in partial fulfillment of the requirements
for the degree of

Master of Science

Department of
Electrical and Computer Engineering
University of Manitoba
Winnipeg, Canada 1992

ISBN 0-315-77880-6

Canada

STOCHASTIC ARITHMETIC IMPLEMENTATIONS OF ARTIFICIAL NEURAL NETWORKS

BY

JEFFREY A. DICKSON

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

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Abstract

Artificial Neural Networks require highly parallel computing implementations to be effective. This thesis examines the application of stochastic arithmetic to neural networks. Stochastic arithmetic uses values encoded as probabilistic pulse streams. It is shown that this arithmetic requires only simple digital logic gates. Stochastic arithmetic neural networks are demonstrated. In addition, a novel design for in-situ learning aritifical neural networks employing stochastic arithmetic is presented. The circuits require simple hardware to implement. The hardware is simulated and shown to successfully learn sample problems.

Acknowledgements

I would like to thank my advisors, Professor H. C. Card and Professor R. D. McLeod for their guidance in this work. I would like to particularly single out Professor McLeod for his advision during my Master's thesis and two NSERC Undergraduate Summer Research Scholarships. This thesis represents a very small portion of what I have learned and done during my association with him. To his chagrin it is pretty much all that I have written up. I feel fortunate to have been able to call him my advisor, and, most of all, friend.

I thank Brion Dolenko for sharing his neural network expertise, and David Blight, Gord McGonigal and Bob Pelletier for discussions.

Finanical support from the Natural Sciences and Engineering Research Council of Canada and equipment loans from the Canadian Microelectronics Corporation are greatfully acknowledged.

CONTENTS

List of Figures

Chapter 1

Introduction

The area of Artificial Neural Networks experienced a great resurgence of interest in the 1980's. In the past it has experienced two boom periods, only to have them go bust.

The first boom started in 1943, when McCulloch and Pitts proposed a simple model of neuron operation [2]. This model attracted much interest because of its simplicity.

The second boom was due to the networks of neurons that could learn to solve problems, given the inputs and the desired output. While effective for some problems, Minsky and Papert showed in the book *Perceptrons* [3] that these networks were unable to solve a wide range of problems. Interestingly some researchers believed they knew the solution to the problem, but they could not find the procedure to train the networks. Researchers left the field in the face of this roadblock, and the field became dormant.

In the 1980's new results, such as a learning algorithm that addressed the problems of the networks discussed in *Perceptrons*, started a resurgence in the

field. The availability of computers assisted simulation and exploration. There has been a tremendous amount of simulation of neural networks in recent years.

There has been comparatively little study of hardware implementation. Artificial Neural Networks are fine-grained, massively parallel systems; however, most of the work has been by simulation on sequential computers. General purpose computers are not optimized for the calculations of neural networks; they require specialized hardware. The hardware ranges from a few special purpose processors to large arrays of processors implemented on an integrated circuit. Most promising is the latter approach: highly parallel computing structures implemented on a single chip or multi-chip systems.

Networks can consist of thousands of computational elements. It is necessary to implement these processors efficiently. There are two approaches to implementing artificial neural networks in hardware: analog and digital. Each method has its disadvantages and advantages.

Analog circuitry permits high density implementation as the operations required in neural networks can be implemented using few transistors. This allows for many computational units to be included on one chip. Communication between units is typically via a single wire carrying a voltage or current. However, there are many drawbacks to analog implementation. Analog hardware does not produce high accuracy arithmetic. Analog values are susceptible to noise, and mismatch between transistors can have a serious effect on arithmetic. Device matching and connection impedance make it difficult to communicate analog val-

ues between chips, so multi-chip systems are not feasible. The amount of circuitry that one can implement on a single chip limits the size of the networks. Storage elements are very difficult to implement in analog hardware. In addition, analog circuit design and implementation is not well supported by CAD tools and fabrication houses as is digital. While analog circuitry is an interesting medium for neural network implementation, these drawbacks restrict its effectiveness at this time.

Digital hardware can perform arithmetic operations with a high degree of accuracy. Unfortunately, this accuracy comes at the expense of hardware size and computation speed. A single multiplier to perform an 8 bit multiplication can consume a large percentage of the available chip area. Another drawback of digital hardware is that communication of binary values requires many wires. There are many benefits of digital hardware. It is very easy to implement memory to store values. Digital signals are not as susceptible to noise as are analog signals. Finally, digital circuit design and fabrication is well-established.

With the tradeoffs between analog and digital circuits, some implementations are hybrids. Analog hardware performs the arithmetic and digital hardware is used for off-chip communication and data storage. The drawback is that the interface between analog and digital carries a large overhead.

This thesis examines a unique implementation architecture: stochastic arithmetic. Stochastic arithmetic encodes values as the probability of a pulse in a pulse stream. While the hardware considered in this thesis is digital, stochastic

arithmetic allows low area implementation of the hardware required to perform the arithmetic required for artificial neural networks.

The organization of this thesis is as follows:

Chapter 2 gives a brief introduction to artificial neural networks. A brief history and motivation is presented. An example of an artificial neural network training procedure is also presented.

Chapter 3 examines stochastic arithmetic. The hardware for the arithmetic required for neural networks is examined.

Chapter 4 discusses the implementation of stochastic arithmetic neural networks and presents simulation results. These networks are trained off-line. The data from a trained network is then loaded into a hardware implementation.

Chapter 5 is the main contribution of this thesis. The networks examined in Chapter 4 are extended to hardware implementations that can train themselves.

Chapter 6 presents the conclusions and discusses future work.

Chapter 2

Artificial Neural Networks

Modern computers perform complex calculations and can store and retrieve information with speed and accuracy. However, the most advanced computer is humbled by the problem solving power of even the most primitive brain. An Artificial Neural Network is a computation paradigm inspired by neural biology. The ultimate goal is to apply the methods and models of the brain to create computers capable of what the brain can perform that conventional computers cannot.

One area where the brain excels is pattern matching. People can recognize objects easily, even when presented with partial information. Moreover, people are able to grasp the concept of a book and can correctly classify a book that they have never seen. The exact process by which this done is not fully understood. Computers with such a capability are desired in many applications, such as weapon systems, handwriting recognition, and computer vision.

Artificial Neural Networks are attractive because it is not necessary to understand the problem well enough it solve it. It is only necessary to present it to the

neural network in the appropriate form. For example, in training a network to perform character recognition one presents the inputs and desired outputs and has the network learn to perform the classification. Conventional approaches require study of pattern recognition algorithms.

An interesting and dramatic example of the application of artificial neural networks was the work by Pomerleau at CMU[4]. He used a neural network controller for driving a vehicle along a winding road. The network trained with data from a video camera and a range finder. The resulting network could navigate the vehicle at a speed of 5 km/hr. This was twice as fast as previous attempts using conventional algorithms. Perhaps more significantly, the research required far less time to complete, since it was not necessary to spend time devising the necessary algorithms.

The basic computational element in the nervous system is the neuron. The neuron receives inputs from other neurons through synaptic interconnections. When the net input to the neuron is above a certain threshold, the neuron generates an output signal. The signal, encoded as a train of pulses, propagates through the filamentary wire of the neuron: the axon. At the end of the axon are more synaptic junctions that communicate the activation signal to other neurons. The firing threshold is not uniform, neither is the contribution of each synaptic junction to the net input. The strength of these contributions are modified by learning. The neuron compensates for the lack of complexity in computation with complexity of connection. The brain uses this simple element millions and millions

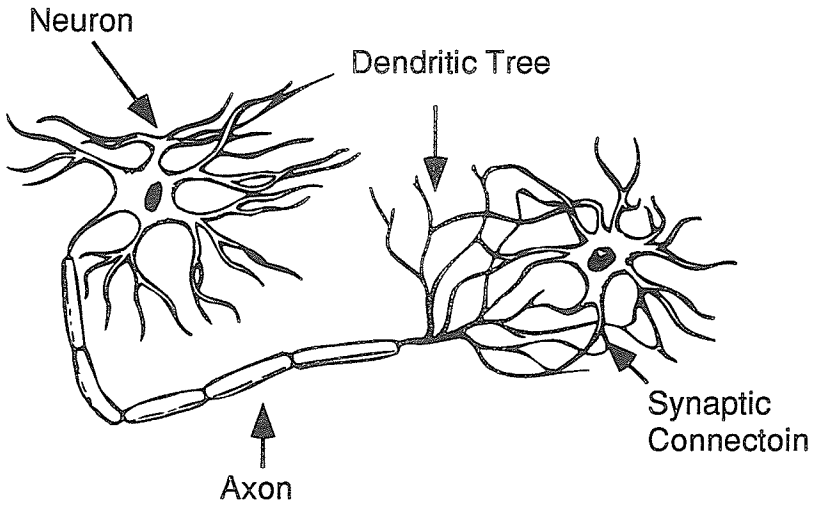


Figure 2.1: Simplified neural biology

of times with great effect. Figure 2.1 shows a simplified drawing of two neurons.

McCulloch and Pitts first formalized a simple model of neural operation in 1945 [2]. Figure 2.2 shows a block diagram of the McCulloch-Pitts neuron. The inputs to the neuron are weighted by the strength of the synaptic connection and summed. The resulting sum passes through a nonlinearity to produce the output.

The calculation of the net input is shown in Equation 2.1. The output of the previous neuron is multiplied by the synaptic "weight". The Θ term is a bias, which models the threshold of a biological neuron. The net input is passed through the nonlinearity function f to determine the output in Equation 2.2.

$$n_j = \sum_i w_{ij} o_i + \Theta_j \tag{2.1}$$

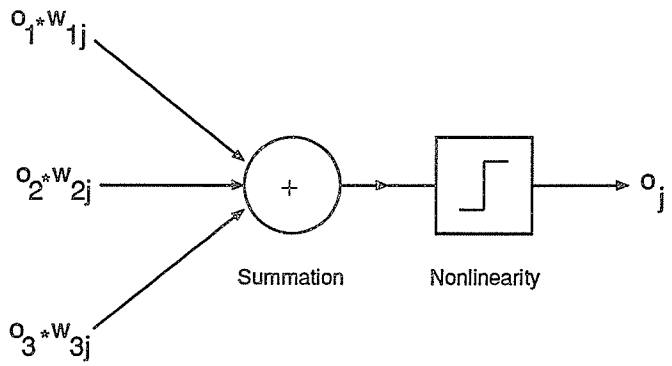


Figure 2.2: Model of the McCulloch-Pitts neuron. The outputs of the input neurons are multiplied by the interconnection weights and summed. The sum passes through a nonlinearity to determine the neuron output.

$$o_j = f(n_j) \tag{2.2}$$

2.1. Network Architecture

There are many different architectures of artificial neural networks. This thesis is limited to feed-forward networks. A feed-forward network has neurons arranged in layers, shown in Figure 2.3. Each layer has neurons that receive their inputs from the previous layer and propagate their outputs to the next layer. Neurons in the same layer do not communicate with each other. The first layer is called the input layer: it receives the inputs from the outside world. The final layer is the output layer. Neuron values in this layer are the considered result of the network. The layers between the input and output are called hidden layers. Networks without hidden layers can only solve a restricted problem set.

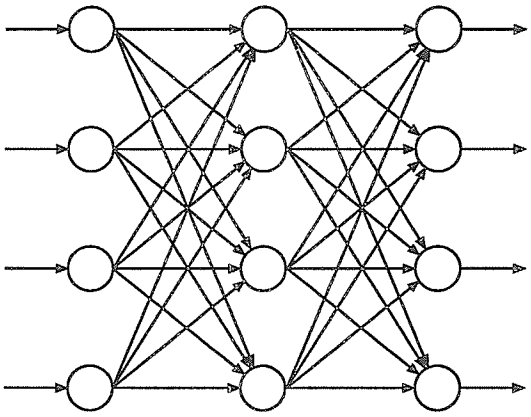


Figure 2.3: Feed-forward network architecture

There is no formula to calculate the number of hidden layers and units needed to solve a particular problem. Too few units and a network will be unable to solve the problem. Likewise too many units can cause problems[5]. Some learning algorithms, such as cascade-correlation [6], add units as needed during learning. Back-propagation, the learning algorithm considered in this thesis, requires the architecture to be specified in advance.

2.2. Training

The interconnection weights must be set to produce the correct outputs. One method is to set the weights explicitly, using apriori knowledge of the data. Typically this method is only useful for simple problems. A more sophisticated and useful method is to train the network by example. Each input and desired output is presented to the network and the weights modified to produce the correct response.

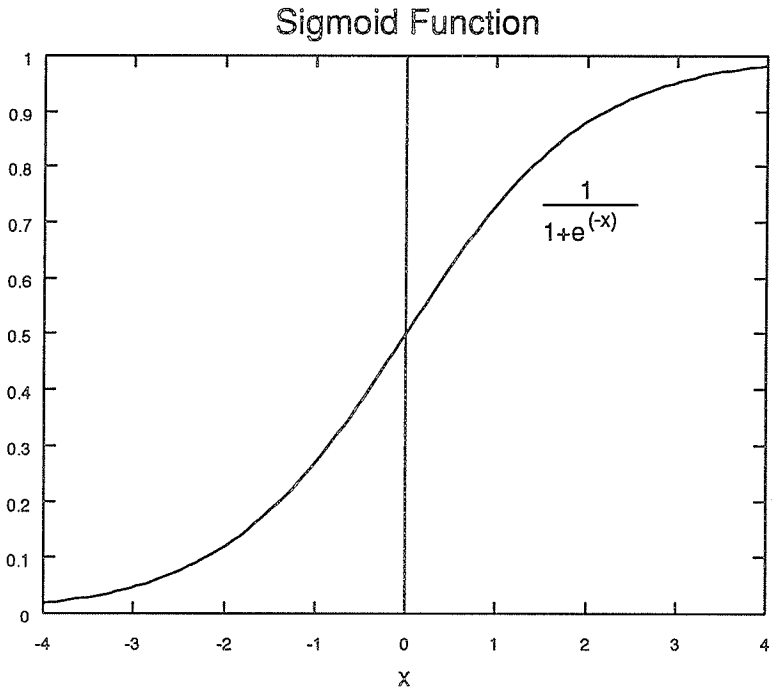


Figure 2.4: The Sigmoid nonlinearity

There are many different techniques to train a network. A popular is Back-propagation[5], the discovery of which started the current boom in neural networks.

The goal of this learning algorithm is to minimize the error at the outputs by adjusting the synaptic weights. The error is commonly defined as the total sum of squares error measure. The error for a single training pattern, summed over all the output neurons, is shown below. The factor of 1/2 is included to cancel the factor of 2 that comes out of the derivative.

$$E = \frac{1}{2}\sum_j (t_j - o_j)^2$$

where t_j is the desired output and o_j is the output of neuron j. Each weight is modified in proportion to the negative gradient of this error with respect to the weight:

$$\Delta w_{ij} \propto -\frac{\partial E}{\partial w_{ij}}$$

By applying the chain rule the derivative of the Error with respect to the weights can be determined.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial n_j}\frac{\partial n_j}{\partial w_{ij}}$$

For the output units the first factor is:

$$\frac{\partial E}{\partial o_j} = t_j - o_j$$

Calculating the other terms:

$$\frac{\partial n_j}{\partial w_{ij}} = o_i$$

$$\frac{\partial o_j}{\partial n_j} = f'(n_j)$$

where $f(x)$ is the nonlinearity of the neural activation. For a sigmoidal neuron it can be shown that: [1]

The updates for the weights from the hidden layer to the output layer is:

$$\Delta w_{ij} = \eta \delta_j o_i$$

where η is the constant of proportionality, called the *learning rate.*

For weights to output units $\delta_j = (t_j - o_j) f'(n_j)$.

Updates to the connection weights of the hidden units require further application of the chain rule. The δ's for the hidden units turn out to be the weighted summation of the $\delta's$ from the units each hidden unit is connected to in the next layer.

$$\delta_i = f'(o_j) \sum_j \delta_j w_{ij} \tag{2.3}$$

[1]The simplicity and continuity of the sigmoid's derivative is one of the main reasons that it is commonly selected for neural network activations.

Training using back-propagation has two phases. First the inputs are presented to the network and the output is calculated. Next the output is compared with the training data to compute the error. The error is propagated backward through the network to compute the weight derivatives for all the units.

The weight derivatives can be accumulated and applied after presentation of the set of training data. Each pass through the training set is called an *epoch.* Alternatively the weights can be updated after each individual training vector.

The training procedure is repeated until the total error is reduced to an acceptable level. It is possible that the error cannot be sufficiently minimized. This can be due to being trapped in local minima or the network having too few units or layers. Possible solutions are to try again with a different set of initial weights, different learning rates, or different network sizes.

2.3. Example of Network training

This section demonstrates the training of a simple feed forward network using back-propagation. The problem is to train a network to perform the XOR operation on two inputs.

The network architecture is shown in Figure 2.5. The network has one layer of two neurons, one for each input. These connect to a hidden layer of two inputs, which in turn connect to one output neuron. There are six weights and the bias values for the two hidden units and the output unit to determine in this network.

The weights are initialized to random values, and the network is trained using

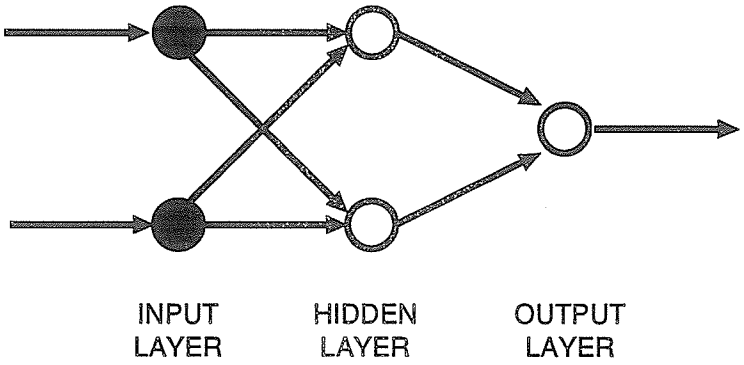


Figure 2.5: Network architecture for the XOR problem. The network consists of two hidden units and one output unit.

the back-propagation algorithm. The evolution of the training procedure is shown in Figure 2.6. The training error, weights, and bias values are plotted against the number of passes, or epochs, through the training data. After 900 passes through the training set the training error has reached zero, and the network has learned the XOR problem.

The output space of the trained network is shown in Figure 2.7. The training data is located at the corners of the graph. The analysis of the decision space is instructive because it shows how the network will respond to inputs not included in the training set (assuming analog inputs). The steep trough in the decision space indicates that the network has made a distinct classification, either one or zero. The trough is located in the region where the two inputs are very similar. This is reasonable since the XOR function is zero when the inputs are equal. The network classifies inputs that are very dissimilar with an output of 1, shown by the high regions of the graph around (0,1) and (1,0).

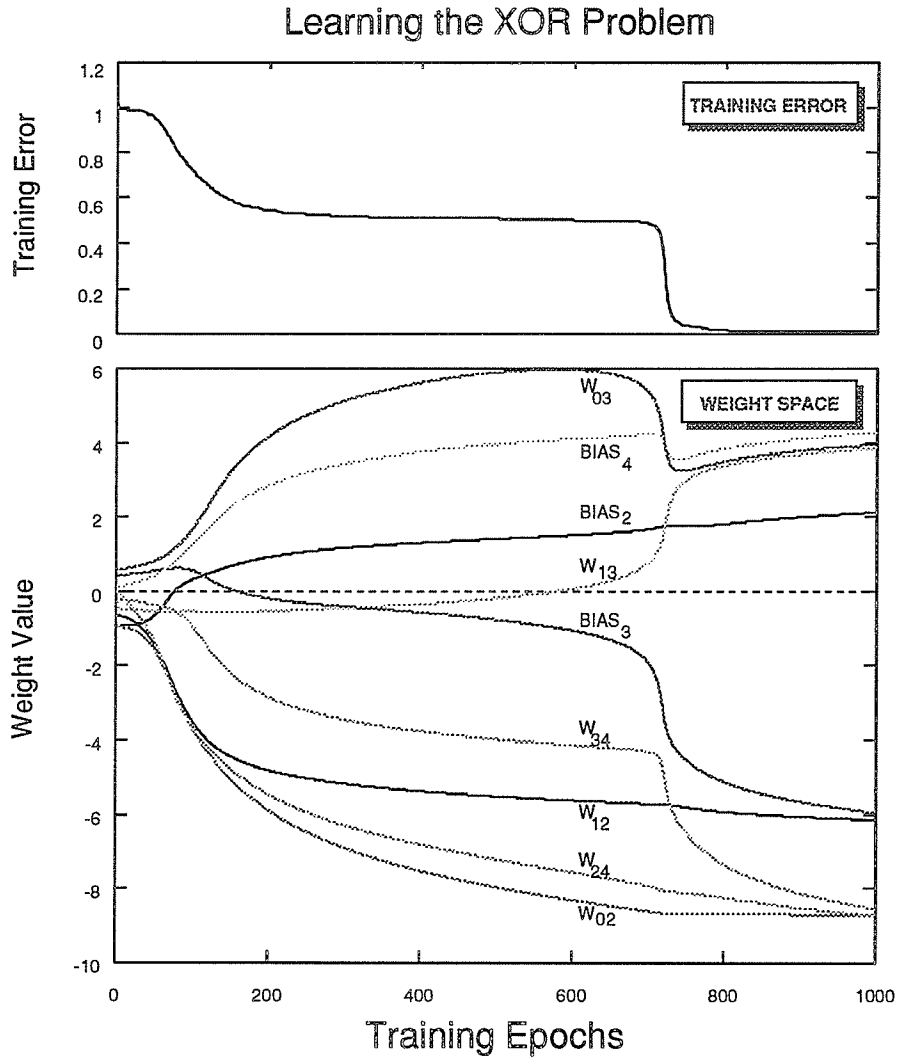


Figure 2.6: Training evolution of the XOR problem

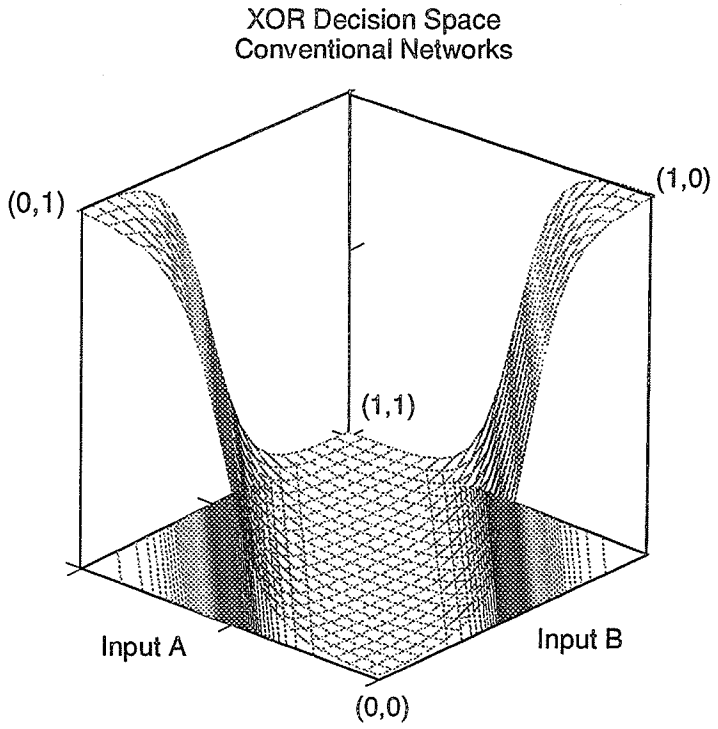


Figure 2.7: Output space for the XOR problem

2.4. Conclusion

This chapter has presented a brief introduction to artificial neural networks, specifically feed forward network trained using back-propagation of error. It was not intended as a comprehensive review of the field, but as a foundation for the rest of the thesis. More information on artificial neural networks is available from a variety of sources, such as reference [7].

Chapter 3

Stochastic Arithmetic

Stochastic arithmetic is not a new idea. It was developed in the 1960's by several groups seeking a new method of computing[8]. The research was motivated by the difficulty in assembling large computers, either analog or digital, in that era. Without a convenient means of programming digital computers, analog computers were considered easier to use. To multiply two numbers with digital hardware required programming using punch cards or setting switches – in analog you simply patched two wires into the multiplier. Stochastic arithmetic was proposed to allow digital hardware to be used as conveniently as analog hardware. The arrival of programming languages and powerful digital hardware changed the future of computing. Analog and stochastic computers were run over by the great digital steamroller.

This chapter shows how stochastic arithmetic is used to perform computations. The motivation for using stochastic arithmetic in artificial neural networks is discussed in the next chapter.

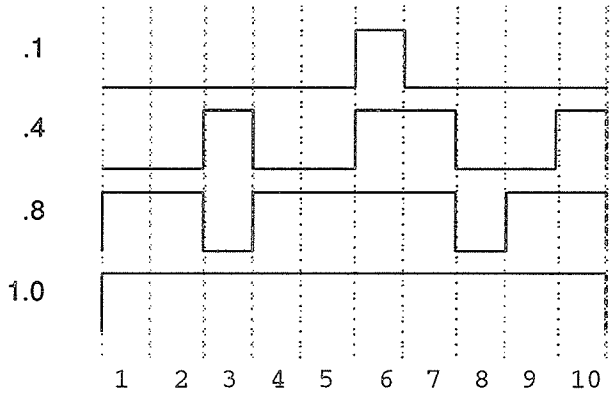


Figure 3.1: Examples of pulse stream representations. Values are represented by the probability of a high pulse in a pulse stream.

3.1. Stochastic Encoding

Stochastic arithmetic represents numbers as a probability of a pulse in a stream of pulses. For example, the value of .5 could be represented by a pulse stream with an equal chance of an individual pulse being a 1 or 0. Some examples of pulse representations are shown in Figure 3.1.

There are different ways of encoding a number for stochastic arithmetic. This paper deals primarily with the representation shown in Figure 3.1, where numbers are encoded as the probability of a one in a binary pulse stream. This representation is unipolar: only values between 0 and 1 can be encoded. Other representations can encode bipolar values or use more than one line for signalling.

Stochastic arithmetic offers the advantages of analog and digital computation. Like analog, pulse representation requires only one line to carry the values, and the size of the hardware to perform arithmetic computations is comparable to

analog hardware. Contrast this to digital hardware, where large buses are generally required to communicate data, and the hardware requirements of arithmetic are excessive. Analog circuitry is plagued by problems such as device matching, which makes intra-chip, and in some cases inter-chip, communication impossible. Stochastic computing, which requires only simple digital hardware, does not suffer from these drawbacks.

Stochastic computing has some unique benefits. Consider a pulse stream that to represent 11 values: (0,.1,.2,.3,.4,.5,.6,.7,.8,.9,1). A pulse length of 10 is required. If it is required to extend the system to handle 20 values: (0,.05,.1,...,.90,.95,1) all that is required is to double the length of the pulse: no hardware modification is necessary. In addition, the inherent noise can be beneficial to neural networks.

3.1.1. Addition

It is not possible to perform exact addition with pulse streams because of limitations of the representation. The maximum value that can be expressed is 1 (i.e. all pulses high), so OR gate addition cannot handle a sum greater than 1. One solution is to perform weighted summation. For example, the weighted summation of A and B would be $\frac{A}{2} + \frac{B}{2} = \frac{A+B}{2}$. The factor of two scaling on the inputs prevents the final summation from exceeding 1. The summation hardware can be a simple counter.

Another approach is to use approximate addition, sacrificing accuracy for hardware simplicity. Figure 3.2 shows that the OR gate truth table provides an ap-

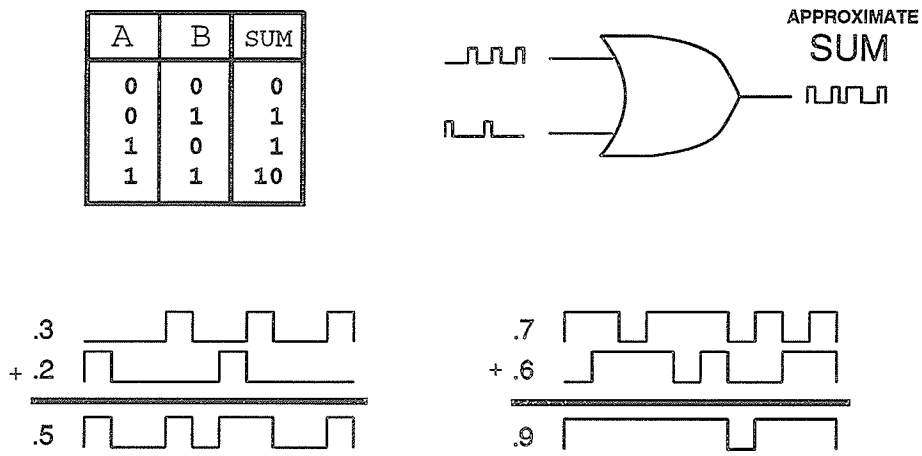


Figure 3.2: Pulse Stream Addition. Addition can be approximated for small inputs using an OR gate. For inputs approaching one the OR gate adder saturates to 1.

proximation of addition. The output of an OR gate is given by:

$$\begin{aligned} A \text{ or } B &= A\overline{B} + \overline{A}B + AB && (3.1) \\ &= A(1-B) + (1-A)B + AB \\ &= A - AB + B - AB + AB \\ &= A + B - AB && (3.2) \end{aligned}$$

Thus for $A << 1$ and $B << 1$ the AB term is small and the output of the OR gate is approximately $A + B$. For large A and B the output of the OR gate saturates to 1. This results in a saturating nonlinearity that we will see is useful for artificial neural networks. Figure 3.2 shows two examples of pulse streams added using the OR gate.

More than two numbers can be added using an OR gate with more inputs.

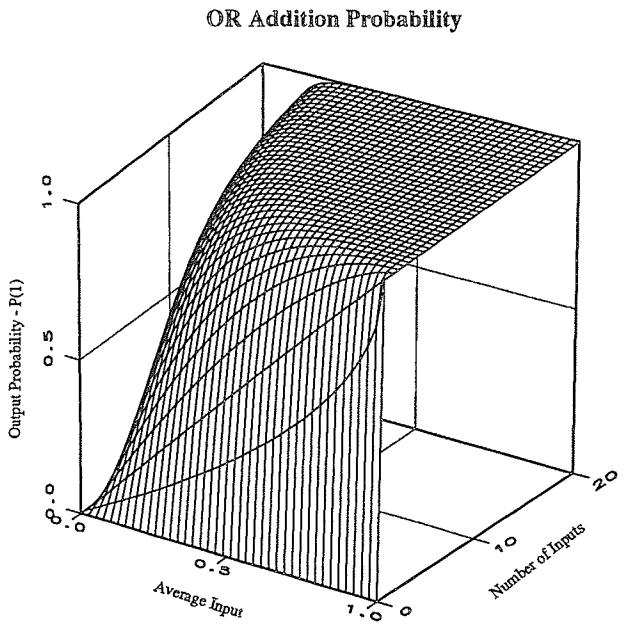


Figure 3.3: Output probability of OR gate addition. Note that the probability of a one output quickly approaches one as the number of inputs increases. For OR gate addition to be effective for large fan-in, the inputs must be small.

Because the maximum sum of any number of inputs is 1 (limited by the representation), the OR gate addition will be accurate for a smaller range as the number of inputs increases. The output for an OR gate with n inputs is given by:

$$Output = 1 - \prod_{n}(1 - i_n)$$

Figure 3.3 shows the output probability of the OR gate for increasing numbers of inputs. As the number of inputs increases the output of the OR gate addition saturates for a greater range of input. For large fan-in the inputs must be small.

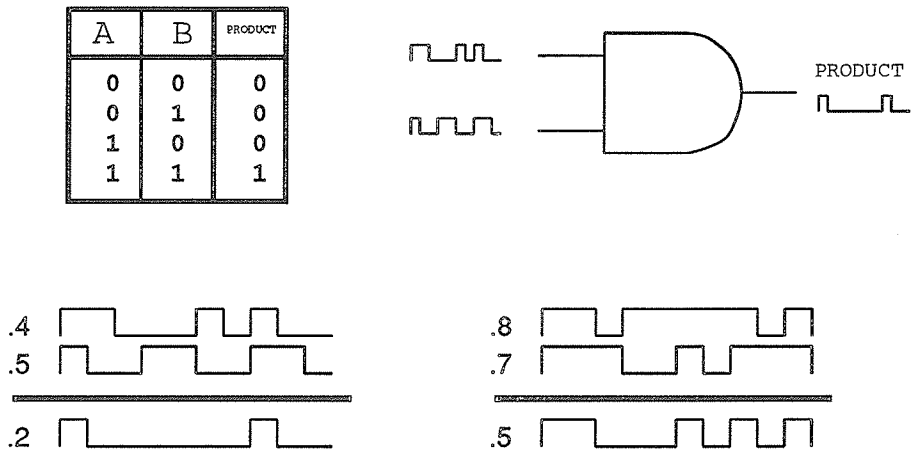


Figure 3.4: Pulse Stream Multiplication. The AND gate can be used for multiplying two pulse streams.

3.1.2. Multiplication

Since product of two numbers less than or equal to one is guaranteed to be less than one, multiplication using stochastic arithmetic does not suffer from the limitations of the representation as did addition. Calculation of the product of two (or more) pulse streams requires only simple hardware. Figure 3.4 shows that the AND function performs the multiplication of two pulse streams. The output of an AND gate is given by:

$$A \text{ and } B = AB \tag{3.3}$$

The product of n inputs can be computed by a n-input AND gate.

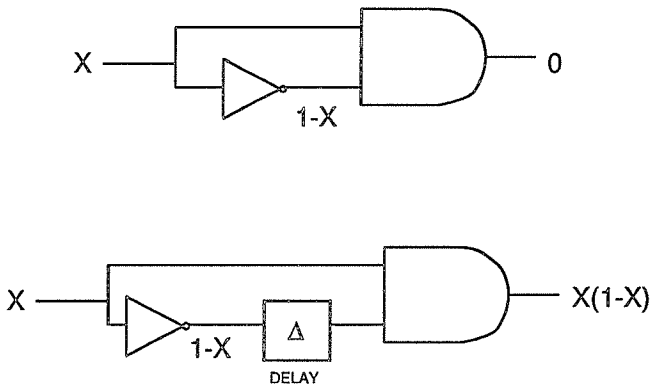


Figure 3.5: Removing temporal dependency with delay elements

3.1.3. Pulse Independence

It is important that pulse streams used in stochastic arithmetic be free from temporal dependency. Two bit streams are dependent when the probability of a one or zero *at a given instant in time* is a function of another bit stream. This can has a significant impact on calculations.

The impact of temporal dependency is best illustrated with an example. Consider calculating the product X(1-X) using stochastic arithmetic. A stream X with the desired probability is generated. The term (1-X) is computed by passing X through an inverter. The product of X and (1-X) is then calculated using a two input AND gate, as shown in the upper schematic of Figure 3.5. However, the output of the AND gate will always be zero, independent of the value of X. When there is a high pulse (logic 1) on X, the inverter will always produce an opposite (logic 0) for the (1-X) term. The inputs to the AND gate will either be 01 or 10, which will always result in a 0 output.

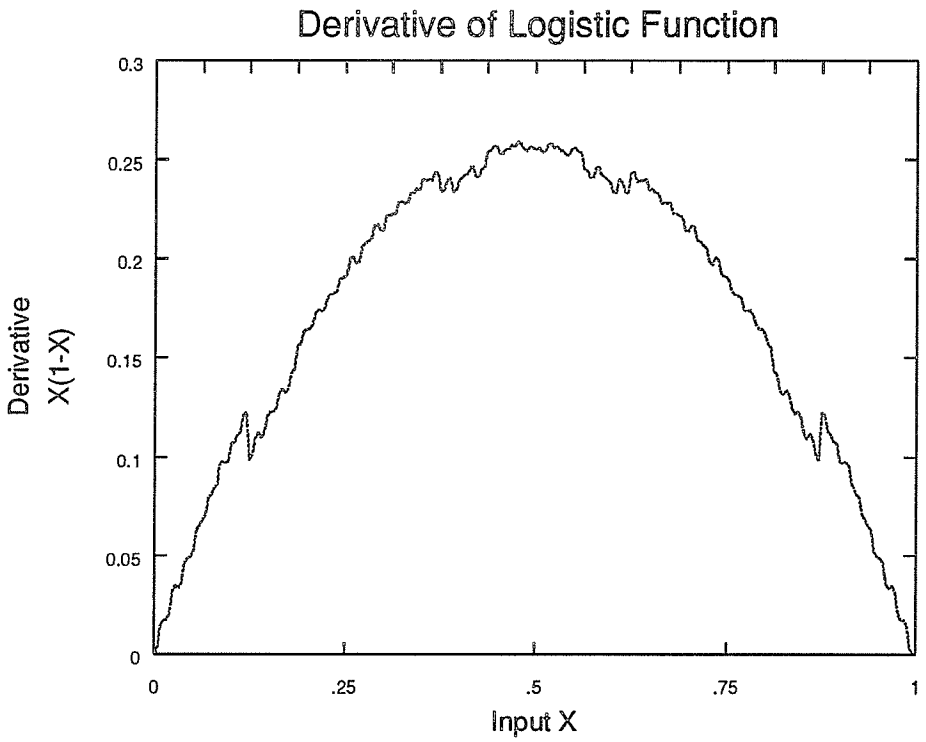


Figure 3.6: Computing the derivative of the logistic function. Glitches in the output are due to the pulse generation circuitry. Averaging the output will eliminate the glitches.

The dependency can be removed by adding a delay element to one stream, as shown in Figure 3.5. Because the pulse streams have no temporal correlation (if properly generated) and the delay removes the temporal dependency, the AND gate will produce X(1-X). Figure 3.6 shows a simulation of this circuit. The inputs have eight bit resolution, and the output had been averaged over four $2^8 - 1$ clock cycles for smoothing. Averaging over greater time will increase the accuracy of the arithmetic.

3.1.4. Division

This section shows that stochastic arithmetic can perform mathematical calculations more complex than addition or multiplication. Division, for example, is a sophisticated but useful operation and is not trivial to implement in binary arithmetic with digital hardware. An approximation to division using a J-K flip flop is possible [8]. This section demonstrates a powerful error minimization method to produce the quotient of two numbers [8].

Let P_o be an approximation of the quotient, and e is the error in this approximation.

$$P_o \approx \frac{P_1}{P_2}$$

$$ERROR = e = P_2 P_o - P_1$$

Note that e^2 is positive and bounded below by zero. If P_o is changed so that the derivative of (e^2) is negative, then the error will be minimized.

$$\begin{aligned} e^2 &= (P_o P_2 - P_1)^2 \\ \frac{d(e^2)}{dt} &= 2(P_2 \dot{P}_o)(P_o P_2 - P_1) = 2P_2 \dot{P}_o e \end{aligned}$$

For the derivative of e^2 to be negative we need:

$$2P_2 \dot{P}_o e < 0$$

$$\textit{since} \qquad P_2 > 0,$$

$$\textit{then} \qquad \dot{P}_o = -\alpha e = -\alpha(P_o P_2 - P_1)$$

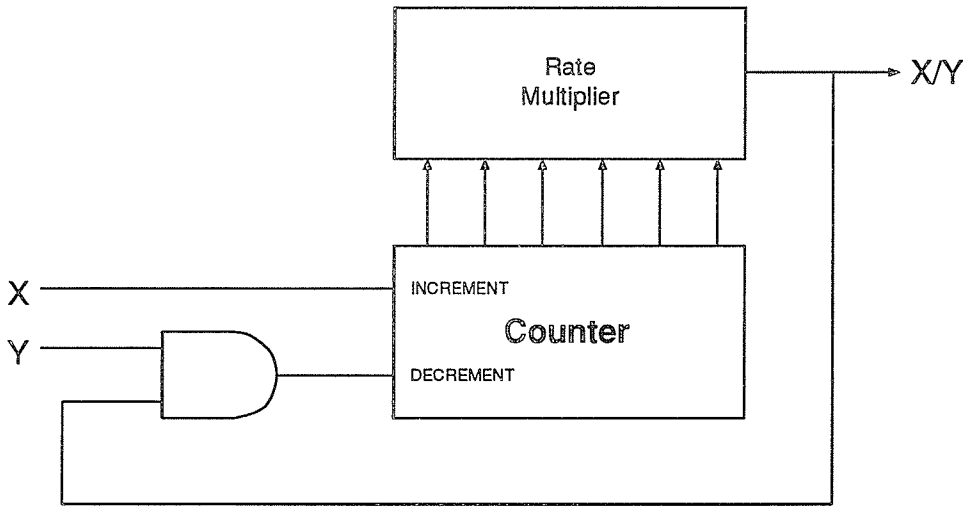


Figure 3.7: Circuit for stochastic division.

Therefore to minimize the error the output P_o must be changed by $P_1 - P_oP_2$. An AND gate computes the product P_oP_2 and P_o is updated using the up/down counter arrangement shown in Figure 3.7. Figure 3.8 shows a simulation of the circuit, using an eight bit counter. Initially the approximation is 0. The error correction improves the approximation until the error in minimized. After 1500 clock cycles the quotient (the value in the counter) settles to the final answer. The oscillations in the output are due to the variance present in the random streams, which is discussed in the next section. The variance can be reduced by averaging.

3.2. Pulse Stream Generation

An essential component of stochastic arithmetic is the generation of the pulse streams for use in the arithmetic operations. The generation must be efficient in both time and area so as not to offset the benefits of the stochastic arithmetic.

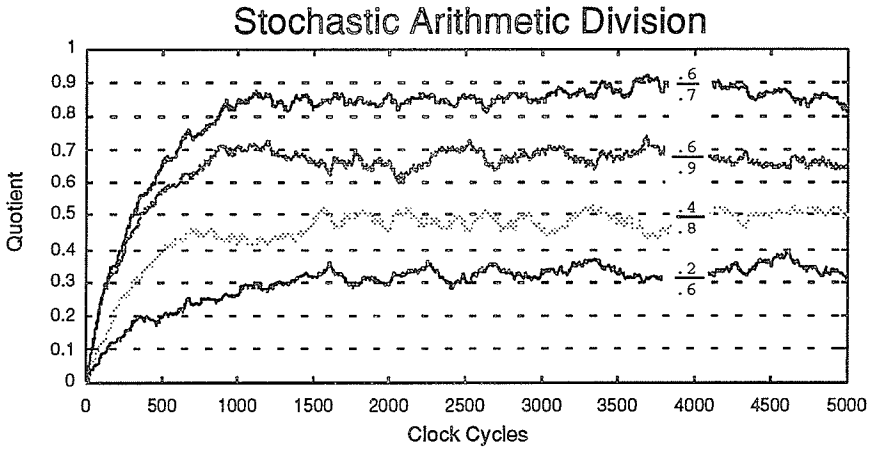


Figure 3.8: Simulation of the Stochastic Arithmetic Division circuit.

In general, to generate a pulse with a probability P(1)=p the following procedure will produce a random stream:

- Generate a random number, R, such that $0 \leq R \leq 1$
- If $p < R$, output a 1 else output a 0.

Figure 3.9 shows a block diagram of a rate multiplier to generate weighted pulse streams.

In digital hardware R and p are usually represented as a binary integer. If the maximum value in the weight register is M, and the value stored in the weight register is p, then the probability of a pulse should be P(1) = p/M.

Generation of a true random number is extremely difficult. In digital hardware the problem is amplified by the necessity to use a minimum of the silicon resources. A common technique to generate a random number is using a linear feedback

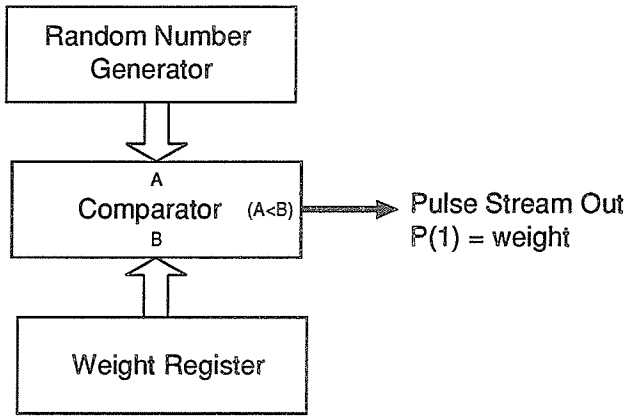


Figure 3.9: A pulse stream of desired valued is generated by comparing the desired weight with a random number.

shift register (LFSR). These structures have been studied extensively for VLSI implementation and found to produce correlated streams. High-quality random numbers are essential.

A solution to this problem is to employ a particular configuration of a Cellular Automata (CA) register. A CA is a set of registers whose next state is governed by nearest neighbor connections. Hortensius [9] has shown that certain arrangements of CAs possess maximal length sequences with superior random number properties compared to the LFSR. The maximal length CAs are composed of cells that are governed by the following state equations:

$$\begin{aligned} \textit{Rule } 90: \; O(t+1,x) &= O_{t,x-1} \oplus O_{t,x+1} \\ \textit{Rule } 150: \; O(t+1,x) &= O_{t,x-1} \oplus O_{t,x} \oplus O_{t,x+1} \end{aligned}$$

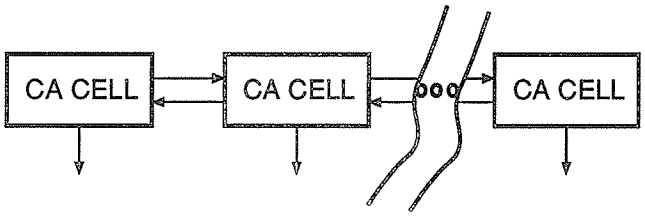


Figure 3.10: Block diagram of cellular automata

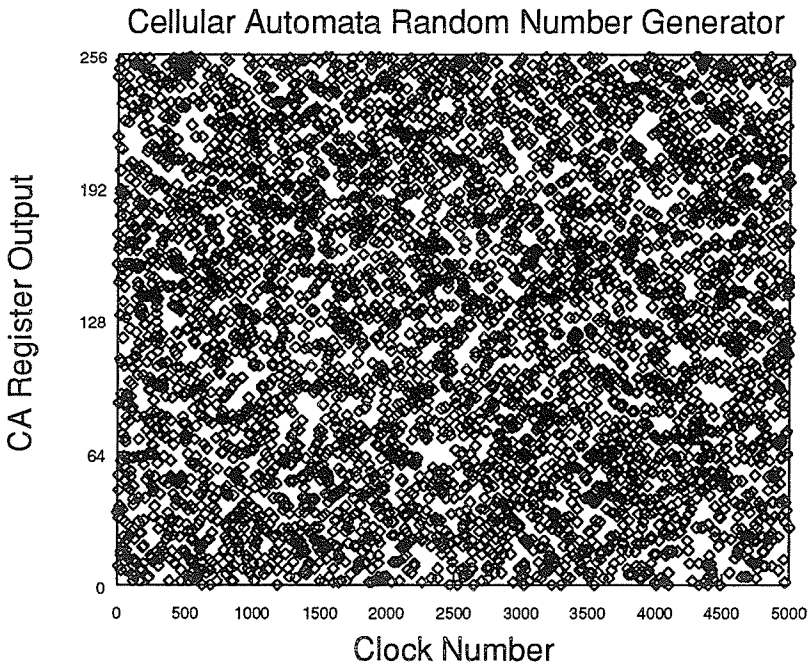


Figure 3.11: Distribution of random numbers from a cellular automata register

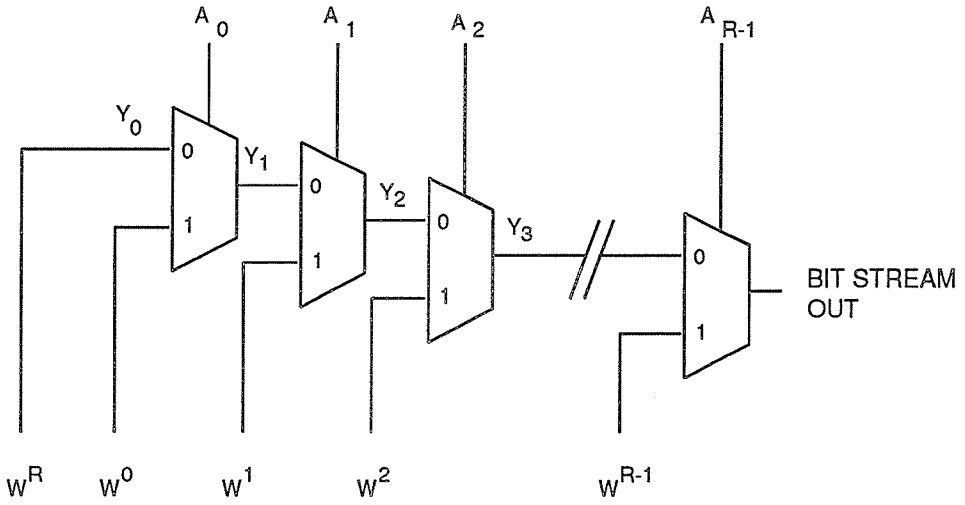


Figure 3.12: Rate Multiplier Schematic

The rate multiplier used in this thesis originates out of research in VLSI pseudo-random test pattern generation [10]. The schematic is shown in Figure 3.12. The circuit takes a binary set of weights and a set of random bit streams with P(1)=.5, and produces a weighted bit stream with P(1) between 0 and 1 in increments of $1/(2^{R-1}+1)$.

For $R = 3$ the circuit analysis is as follows:

$$
\begin{aligned}
Y_0 &= W^R \\
Y_1 &= \overline{A_0} Y_0 + A_0 W^0 \\
&= \overline{A_0} W^R + A_0 W^0 \\
Y_2 &= \overline{A_1} Y_1 + A_1 W^1 \\
&= \overline{A_1 A_0} W^R + \overline{A_1 A_0} W^0 + A_1 W^1 \\
Y_3 &= \overline{A_2} Y_2 + A_2 W^2
\end{aligned}
$$

$$= \overline{A_2 A_1 A_0} W^R + \overline{A_2 A_1} A_0 W^0 + \overline{A_2} A_1 W^1 + A_2 W^2$$

Since $P(A_x) = .5$ and $P(\overline{A_x}) = 1 - .5 = .5$,

$$P(Y_{R=3}) = \frac{1}{2}^R W^R + \frac{1}{2}^R W^0 + \frac{1}{2}^{R-1} W^{R-2} + \frac{1}{2}^{R-2} W^{R-1}$$

For arbitrary R, the general equation is:

$$P(Y_R) = 2^{-R} W^R + \sum_{i=1}^{R} 2^{-i} W^{R-i}$$

The output of the rate multiplier for seven different desired output densities is shown in Figure 3.13. The controlling weight register was eight bits wide, and thus the densities could range from 0 to 255. Inspection of Figure 3.13 shows that while there is a variance in the output, the mean value is equal to the desired density. Figure 3.14 shows the output of the rate multiplier averaged over 16 periods. The variance in the pulse stream adds noise to the computations.

3.3. Conclusion

This chapter presented a method of arithmetic using pulse trains called stochastic arithmetic. Stochastic arithmetic allows calculations using only simple gates. The next chapter will examine the use of stochastic arithmetic in neural networks, where the ability to implement arithmetic with small circuitry is desired.

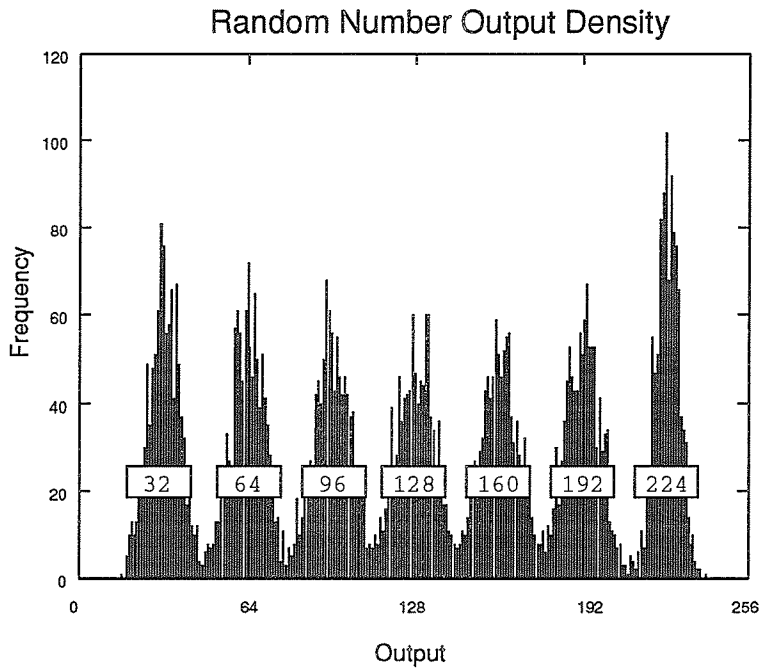


Figure 3.13: Probability density for 8172 cycles of an 8 bit random number generator for 7 desired output densities.

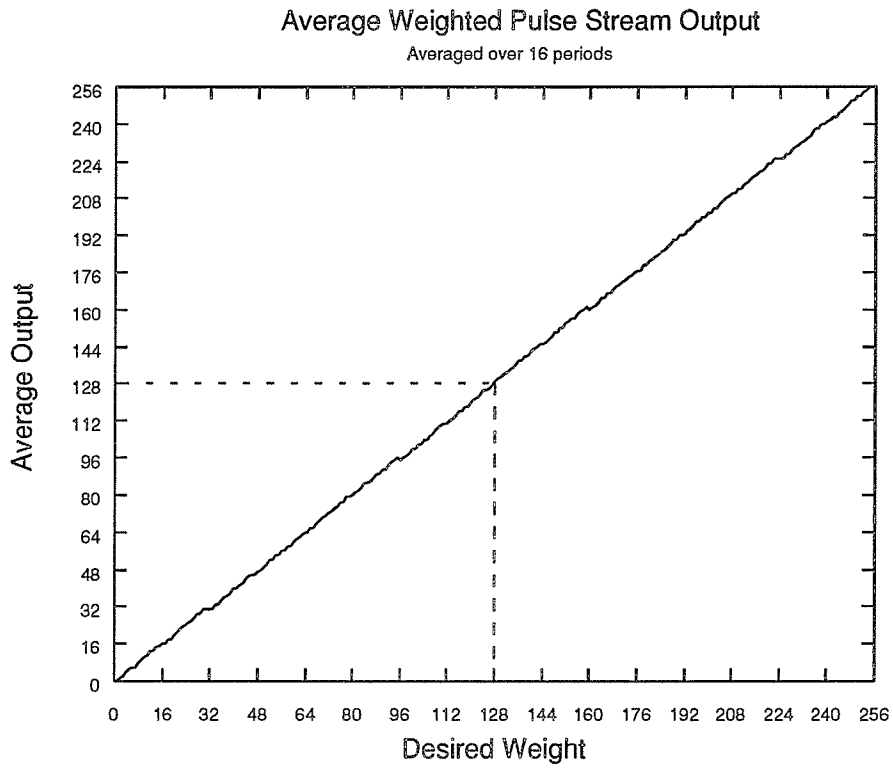


Figure 3.14: Output of Rate Multiplier averaged over 16 periods

Chapter 4

Stochastic Arithmetic Neural Networks

Stochastic Arithmetic Neural Networks use Stochastic Arithmetic to perform the network calculations. This chapter explores the theory, operation, and implementation of artificial neural networks employing stochastic arithmetic.

4.1. Biological Motivation

The brain uses trains of pulse to communicate information. Examples of neural pulse trains for different levels of activation are shown in Figure 4.1. This comparison is for interest sake only - the signalling of the brain is probably much more complex than the simple pulse representation considered in this thesis.

4.2. Stochastic Arithmetic meets Neural Networks

The main motivation of applying stochastic arithmetic to neural networks is the ability to implement arithmetic operations with high density using digital circuitry. The simplicity of the hardware can allow addition and multiplication, the

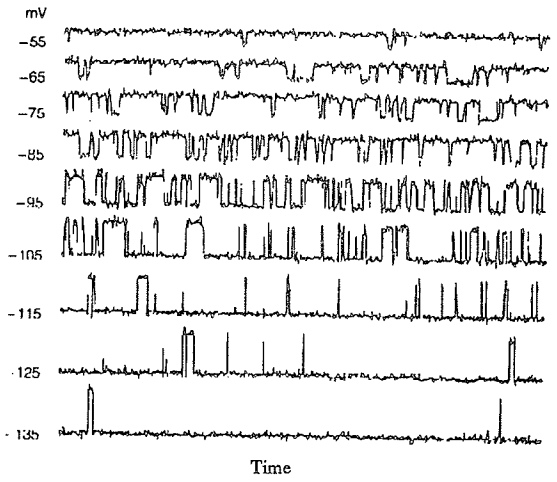


Figure 4.1: Biological motivation [1]

two most important operations in neural network hardware, to be implemented in small areas. Related work to that of this chapter can be found in [11, 12]. The work in this chapter is based on research by Tomlinson et. al. [13]

The implementation of the feed forward networks in this thesis requires the computation of a dot product consisting of an addition and a multiplication. As discussed in the previous chapter these operations can be computed by the OR gate addition and AND gate multiplication, shown in Figure 4.2.

4.2.1. Activation Function

The circuit shown in Figure 4.2 does not support negative synaptic weights, due to the unipolar nature of the pulse stream representation that is employed. To

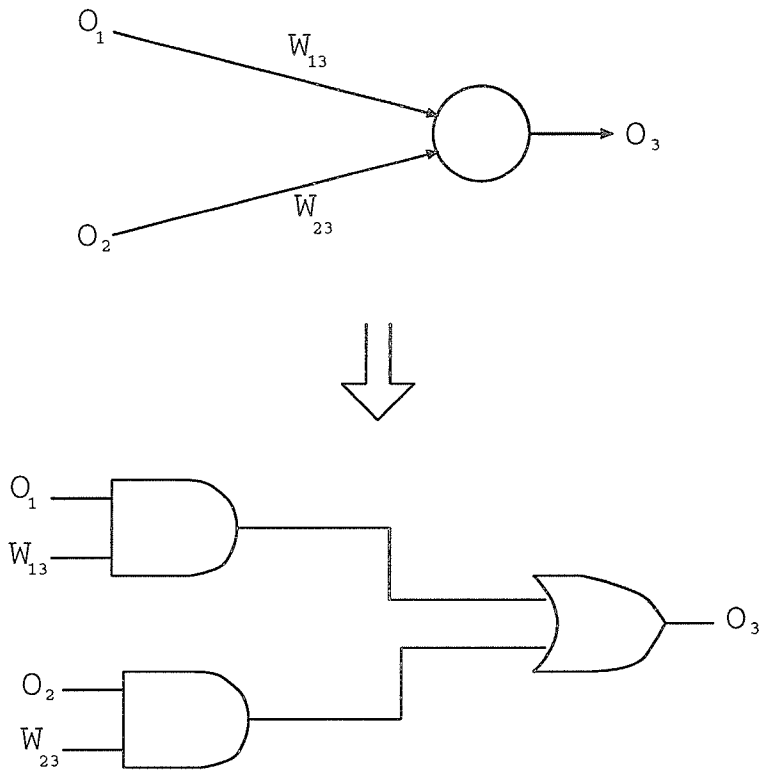


Figure 4.2: Pulse Stream Implementation of the basic neural network operation. The output of the input neurons are multiplied by the synaptic weights using the AND gate multiplier. The summation and non-linearity are performed by the OR gate addition.

accommodate negative weights, each synapse output is separated into two distinct nets: the excitatory and the inhibitory nets. Therefore, each neuron will have two net input terms. Each neuron j will combine the excitatory net input n_j^+ and the inhibitory net input n_j^- to determine the neuron output o_j. The net inputs are calculated using the OR gate addition; these are then combined to result in the final output stream using a simple logic function. These variables can be written in terms of the network values as:

$$n_j^+ = 1 - \prod_{w_{ij}>0} (1 - w_{ij}o_j) \tag{4.1}$$

$$n_j^- = 1 - \prod_{w_{ij}<0} (1 + w_{ij}o_j) \tag{4.2}$$

The two nets are not referred to as negative and positive because the net output of the neuron is not $o_j = n_j^+ - n_j^-$. Unfortunately there is no simple means of performing subtraction in stochastic arithmetic. Moreover negative and positive nets would require accommodation of negative neuron outputs. The output of the neuron is determined by Equation 4.3. The hardware required for this computation is shown in Figure 4.3.

$$o_j = n_j^+(1 - n_j^-) \tag{4.3}$$

The saturating effect of the OR gate addition provides the saturating nonlinearity of the neurons. Figure 4.4 shows the *average* output of a two-input

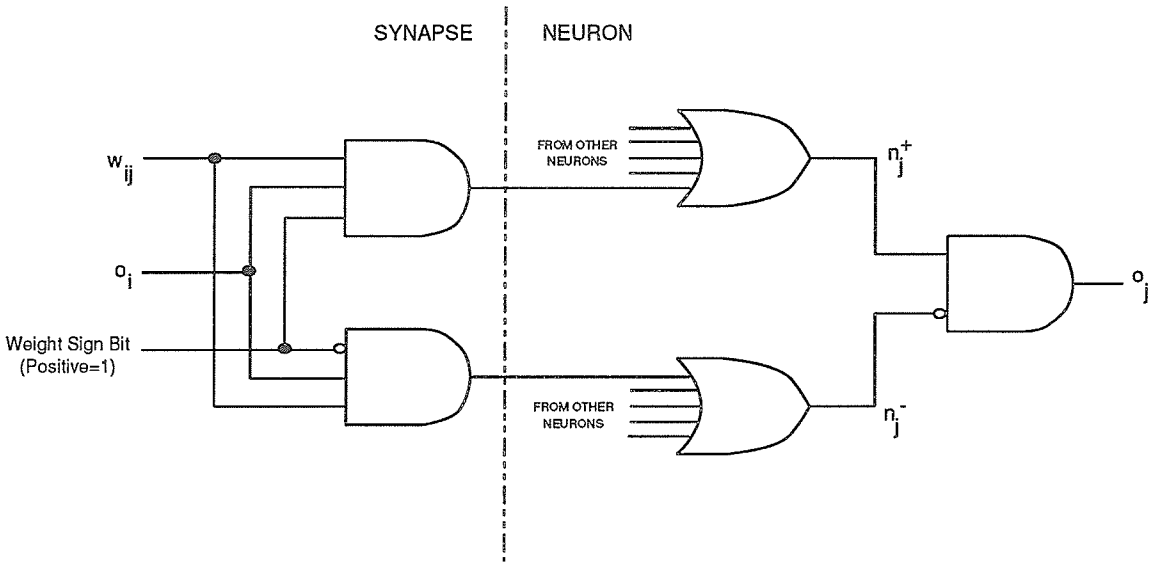


Figure 4.3: Negative and Positive Weights

neuron for the possible inputs. The average output is presented because the nonlinearity is a function of the inputs – the OR gate addition will give a different output for inputs of .4 and .2 than it will for .55 and .05, although the net input is the same.

In conventional digital approaches, the sigmoid nonlinearity is performed using a table look-up or power series expansion. Both approaches take time and area to perform. The computation of the nonlinearity function using OR gate addition is free.

One potential problem of using the OR gate neurons mentionned earlier is as the number of inputs to a neuron (and thus the OR gate) increases, the probability of a 1 output for a given input will rise. This was shown in Figure 3.3. This result suggests that the weights in a neural network must be very small to prevent the

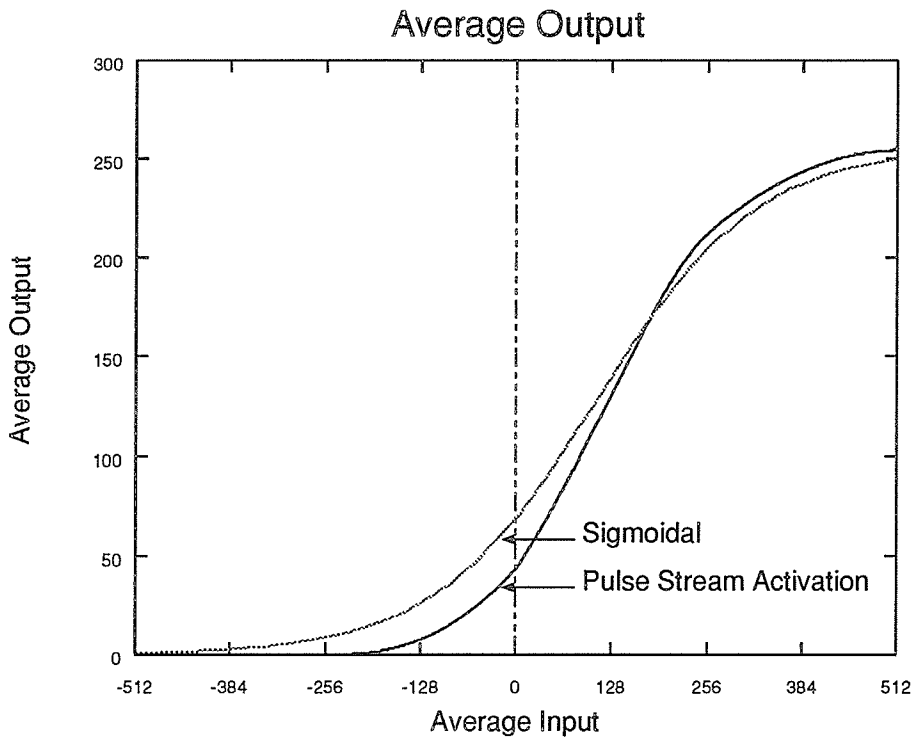


Figure 4.4: Activation function of a two-input pulse stream neuron. The average output is shown for all possible combinations of two inputs resulting in a certain net input. A translated and scaled sigmoidal activation function is shown for comparison.

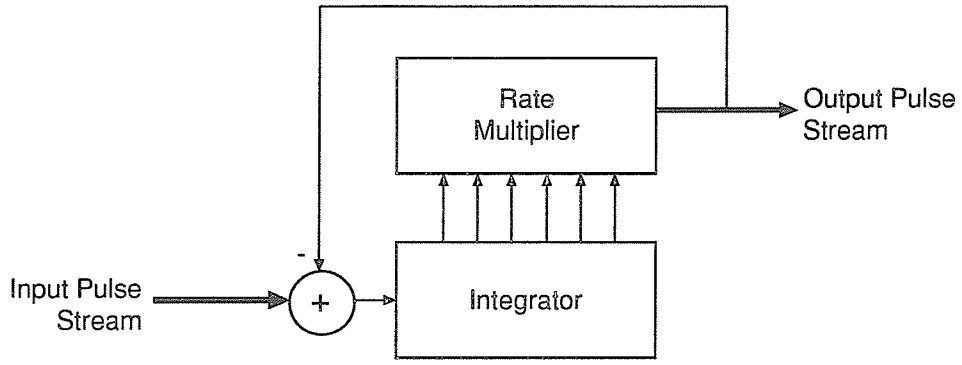


Figure 4.5: Block diagram of the re-randomizer circuit. The counter controls the density of the output stream. If the output is high when the input is low, the counter is decremented. If the output is low and the input is high, then the counter is incremented. If the input and output are equivalent, then there is no change.

neurons from constantly saturating.

4.3. RERANDOMIZER

To prevent correlation between pulse streams from earlier layers the output stream of each neuron must be "re-randomized." This is accomplished by an adder controlling an output stream that is configured to follow the input stream, as shown in Figure 4.5. If the output is high when the input is low, then the value of the counter is decremented. If the output is low when the input is high, then the counter is incremented. If the output and the input are equivalent then there is no change.

The output of the re-randomizer circuit for different input densities is shown in Figure 4.6. Note that the output is not constant but varies due to the variance

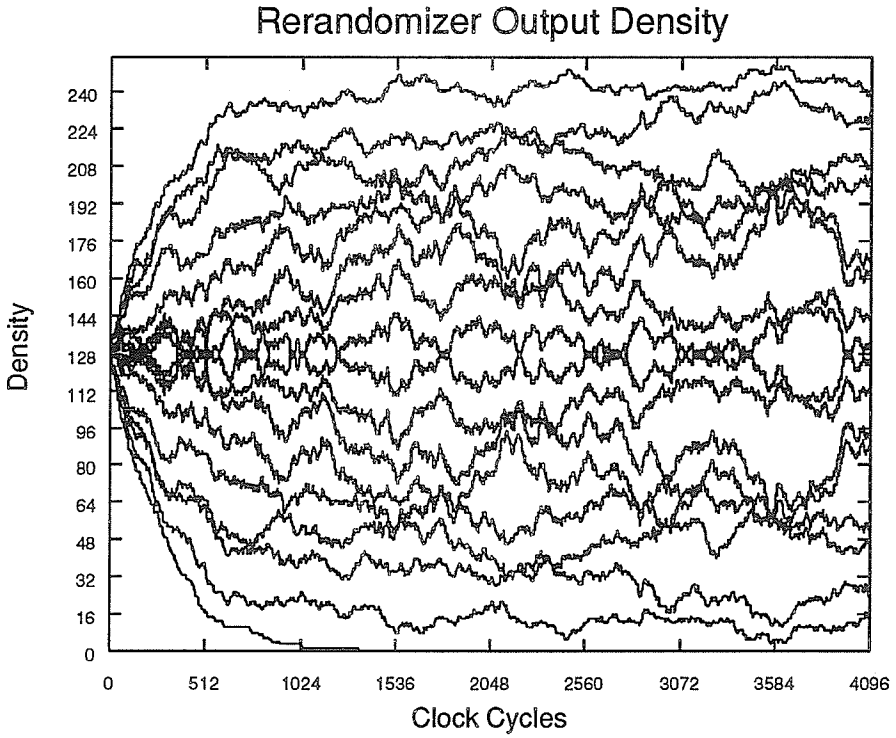


Figure 4.6: Test of the re-rerandomizer circuit. The re-randomizer was reset to the mid-point before each test. The target densities are 0 to 240 in steps of 15.

inherent in the random number generation. In applications where this variance is not desirable it is possible to increase the time constant of the integrater.

Figure 4.7 shows the re-randomizer circuit modified to produce a smoother output. The lower bits on the counter/integrater are ignored. This divides the output value by a factor of two for each bit shift. Changes in the lower bits due to the variance have no direct effect on the density output by the rate multiplier. Figure 4.8 shows the output of this arrangement for one bit and for two bit shifts. Not only does the output become smoother as more bits are ignored, but it also

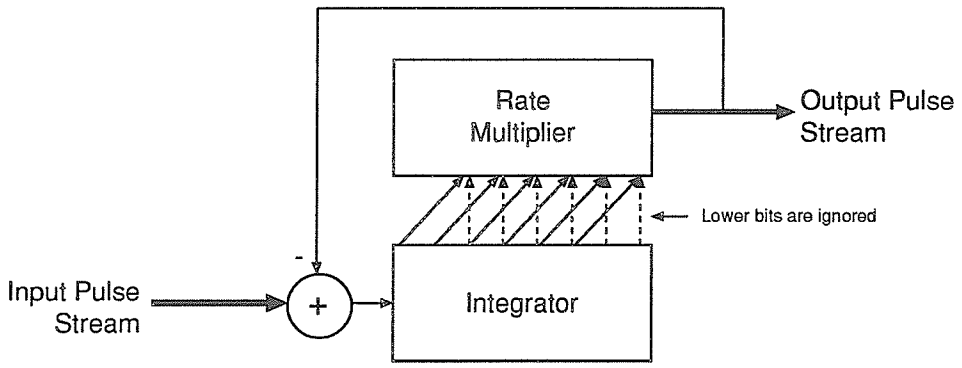


Figure 4.7: Block diagram of the re-randomizer circuit modified for output smoothing. After integrating normally until the input and output streams have equal densities, the value in the integrater is shifted to the left and the lower bits are ignored.

takes longer for the re-randomizer to settle the correct output value.

The output of the re-randomizer can be both rapid and smooth using a hybrid arrangement. Initially the re-randomizer is operated without smoothing, allowing it to quickly count to the target output. After allowing time for the integrater to reach the correct output, the value in the integrater shifts by n bits and the first n bits are not passed to the rate multiplier. This approach has the benefits of the quick response of the non-filtered re-randomizer and the smooth output of the filtered re-randomizer. The output of a re-randomizer employing this approach is shown in Figure 4.9.

4.4. Training Stochastic Arithmetic Neural Networks

The following weight update derivation follows standard back-propagation [5] and is taken from Tomlinson et al [13]. The procedure performs gradient descent over

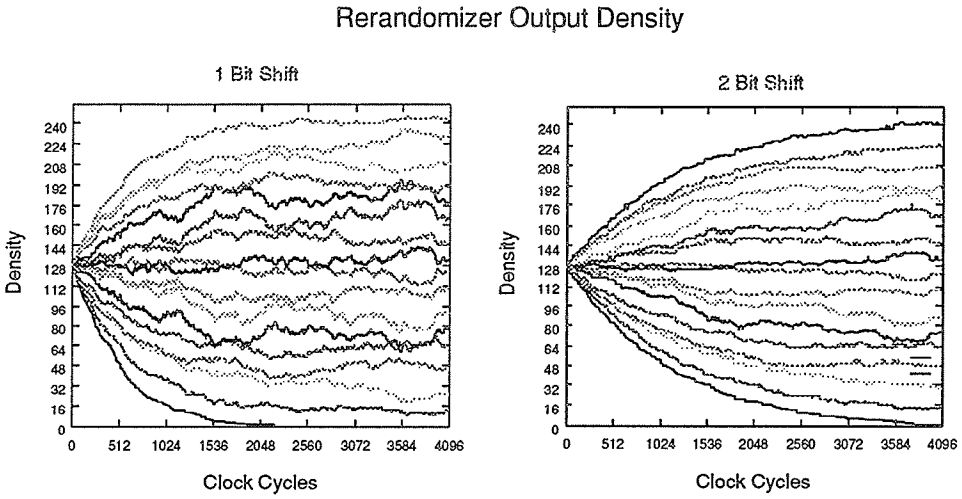


Figure 4.8: Output of the filtered re-randomizer

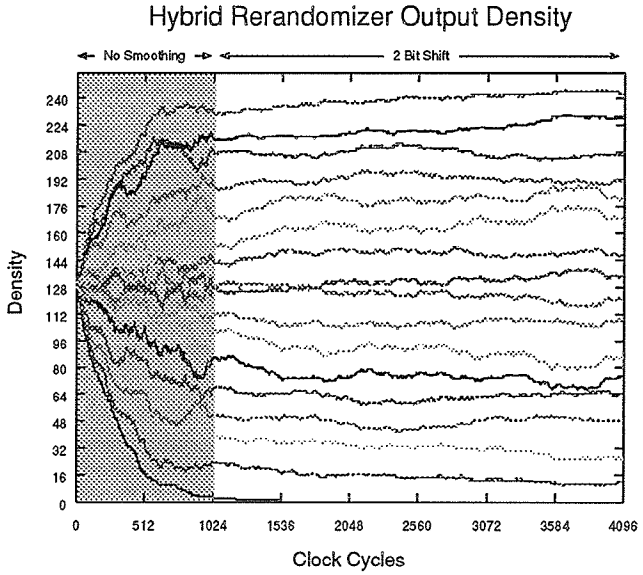


Figure 4.9: Output of the hybrid filtered re-randomizer. Integration operates normally for 1023 clock pulses, when two extra low order bits are added to the integrater

the sum-squared error measure given by Equation 4.4.

$$E = \frac{1}{2}\sum_j (t_j - o_j)^2 \tag{4.4}$$

where t_j is the target value from the training data and actual o_j is the output of neuron j. To minimize the error, each weight is modified in proportion to the negative gradient of the error with respect to each weight, given by Equation 4.5.

$$\Delta w_{ij} \propto -\frac{\partial E}{\partial w_{ij}} \tag{4.5}$$

Since the output of a neuron is not a simple function of its input, but a function of the contribution from the positive and negative streams, the derivative must be considered separately for positive and negative weights.

For positive weights, w_{ij}^+, the chain rule results in Equation 4.6, while negative weights w_{ij}^- require Equation 4.7.

$$\frac{\partial E}{\partial w_{ij}^+} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial n_j^+}\frac{\partial n_j^+}{\partial w_{ij}^+} \tag{4.6}$$

$$\frac{\partial E}{\partial w_{ij}^-} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial n_j^-}\frac{\partial n_j^-}{\partial w_{ij}^-} \tag{4.7}$$

Let:

$$\epsilon_j = -\frac{\partial E}{\partial o_j} \tag{4.8}$$

$$\delta_j^+ = -\frac{\partial E}{\partial n_j^+} = \epsilon_j \frac{\partial o_j}{\partial n_j^+} \tag{4.9}$$

$$\delta_j^- = -\frac{\partial E}{\partial n_j^-} = \epsilon_j \frac{\partial o_j}{\partial n_j^-} \tag{4.10}$$

For output neurons and the sum-squared error measure (4.4) the error is simply the difference between the training data and the network output:

$$-\frac{\partial E}{\partial o_j} = \epsilon_j = t_j - o_j \tag{4.11}$$

For the hidden layers the error is propagated back through the network. Each of the k output neurons connected to hidden unit j will contribute to this error. Using the chain rule:

$$\frac{\partial E}{\partial o_j} = \sum_k \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial n_k^+}\frac{\partial n_k^+}{\partial o_j} + \sum_k \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial n_k^-}\frac{\partial n_k^-}{\partial o_j} \tag{4.12}$$

Using Equation 4.1:

$$\frac{\partial n_k^+}{\partial o_j} = \frac{\prod_{w_{jk}>0}(1 - w_{jk}o_j)(-1)(w_{jk})}{1 - w_{jk}o_j} \tag{4.13}$$

$$= \frac{n_j^+(-w_{jk})}{1 - w_{jk}o_j} - \frac{(-w_{jk})}{1 - w_{jk}o_j} \tag{4.14}$$

$$= \frac{w_{jk}(1 - n_k^+)}{1 - w_{jk}o_j} \tag{4.15}$$

$$\frac{\partial n_j^+}{\partial w_{ji}} = \frac{\prod_{w_{ji}>0}(1 - w_{ji}o_i)(-1)(o_i)}{1 - w_{ji}o_i} \tag{4.16}$$

$$= \frac{n_j^+(-o_i)}{1 - w_{ji}o_i} - \frac{(-o_i)}{1 - w_{ji}o_i} \tag{4.17}$$

$$= \frac{o_i(1 - n_j^+)}{1 - w_{ji}o_i} \tag{4.18}$$

Similarly from Equation 4.2:

$$\frac{\partial n_k^-}{\partial o_j} = \frac{(-1)w_{jk}(1 - n_k^-)}{1 - w_{jk}} \tag{4.19}$$

$$\frac{\partial n_j^-}{\partial w_{ji}} = \frac{o_i(1 - n_j^-)}{1 + w_{ji}o_i} \tag{4.20}$$

From Equation 4.3:

$$\frac{\partial o_j}{\partial n_j^+} = 1 - n_j^- \tag{4.21}$$

$$\frac{\partial o_j}{\partial n_j^-} = -n_j^+ \tag{4.22}$$

The resulting weight update equations are:

$$-\frac{\partial E}{\partial w_{ji}^+} = \epsilon_j(1 - n_j^-)(1 - n_j^+)(\frac{o_i}{1 - w_{ji}^+o_i}) \tag{4.23}$$

$$-\frac{\partial E}{\partial w_{ji}^-} = \epsilon_j n_j^+(1 - n_j^-)(\frac{o_i}{1 + w_{ji}^-o_i}) \tag{4.24}$$

Note that it is possible for the denominator in the above equations to equal zero, which is unacceptable. A scaling factor of .95 is added to the $w_{ji}o_i$ term to prevent the denominator from approaching zero. These weight update equations

are further modified with the inclusion of a learning rate factor. In addition, the weights must be restricted to the range [-1,+1].

The error back-propagation summation is given by:

$$\frac{\partial E}{\partial o_j} = \sum_{k,w_{ij}>0} \left(\frac{\partial E}{\partial o_k}(1-n_k^-)(1-n_k^+)\frac{w_{kj}^+}{1-w_{kj}^+ o_j} \right) + \sum_{k,w_{ij}<0} \left(\frac{\partial E}{\partial o_k}(n_k^+)(1-n_k^-)\frac{w_{kj}^+}{1-w_{kj}^+ o_j} \right) \tag{4.25}$$

Where $\frac{\partial E}{\partial o_k}$ is simply the error from the layer immediately above.

4.5. Pulse Stream Neural Networks

This section presents the results of stochastic arithmetic neural networks applied to three problems: the XOR problem, the four bit parity problem, and a hexadecimal character recognition problem.

The networks were simulated using C++ code interfaced with the XERION[14] neural network simulator library. The networks can be simulated on two levels: probability and pulse stream. The probability level models the network activations as probabilities; the pulse level models all activations as pulses and directly emulates a hardware implementation of these networks.

4.5.1. The XOR Problem

The XOR problem is a frequently applied test of neural networks. A network consisting of two input neurons, two hidden neurons, and one output neuron was

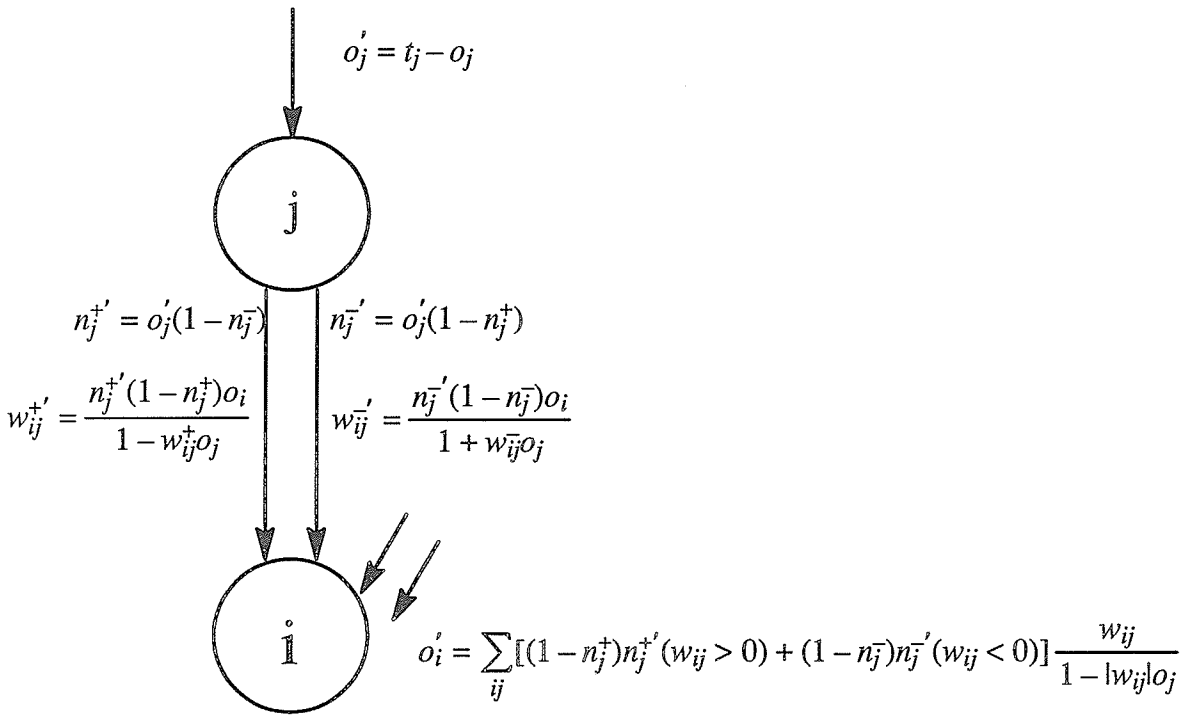


Figure 4.10: Equations for training.

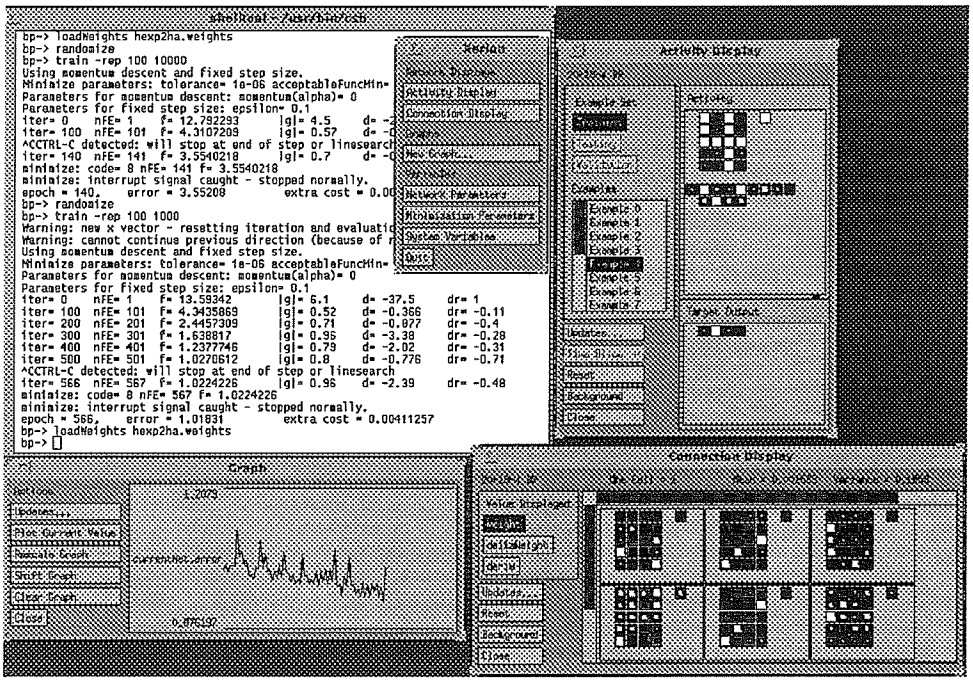


Figure 4.11: The user interface of the pulse stream neural network simulator

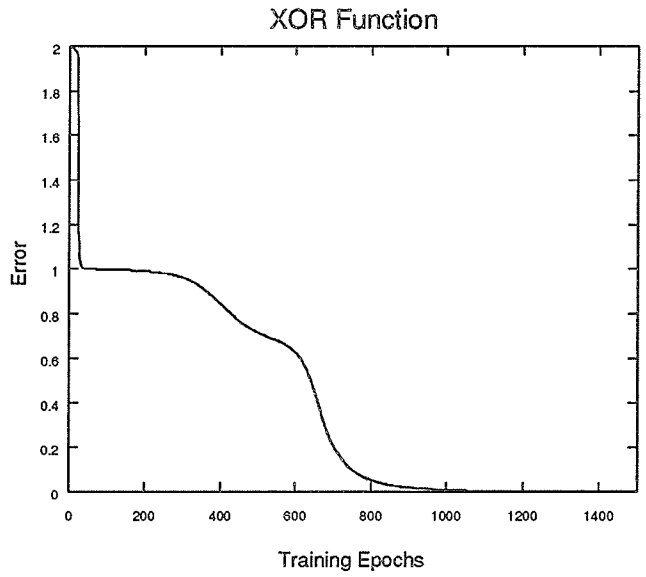


Figure 4.12: Training error for XOR problem.

trained using the weight update equations presented in the previous section. The total sum-squared (tss) error for the network is shown in Figure 4.12. After 1000 presentations of the training data the error approaches zero.

Figure 4.13 shows a Hinton diagram of the outputs of the trained network. The area of the white box is proportional to the value of the output. For example, a half filled box would equal represent an activation of 1/2. The figure shows the network produced the correct outputs.

Figure 4.14 shows the output of the neurons as the inputs are presented to the trained pulse stream network. The output is the value of the re-randomizer register. The output waveforms strongly resemble those of analog circuitry charging and discharging. Figure 4.15 shows the output of the neurons with "precharging"

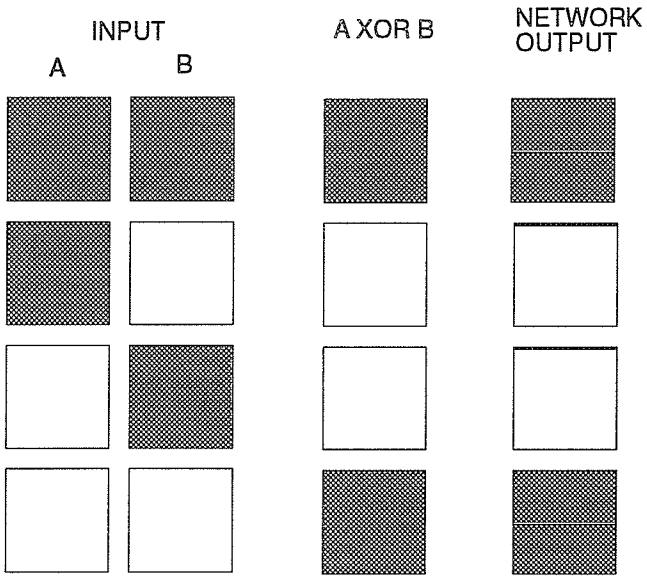


Figure 4.13: Hinton diagrams showing the training data and the output of the trained network.

the rerandomizers. With each presentation of an input vector the value of the rerandomizer is reset to 1/2.

4.5.2. Four bit parity

The four bit parity problem is a natural extension to the XOR problem. It increases the number of bits from two to four; therefore, the number of input vectors rises from 4 to 16. Experience from training with standard back propagation shows this to be an extremely difficult problem. The network used for the four bit parity problem consisted of two hidden layers of 12 and 8 units respectively. Figure 4.16 shows the training error, and the network outputs are shown in Figure 4.17.

The time evolution of the pulse stream network for the presentation of the

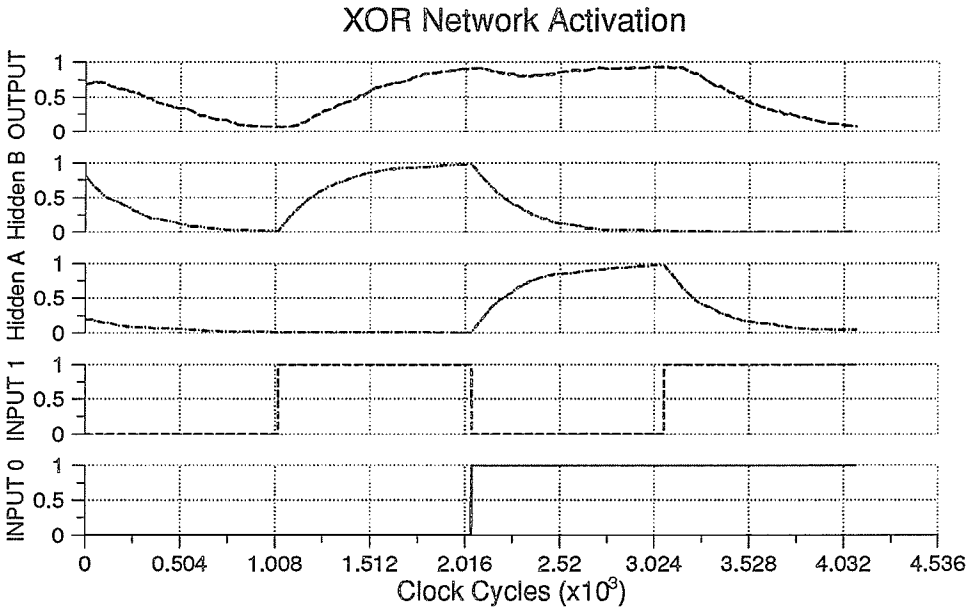


Figure 4.14: Network activation for the XOR problem. Shown are the two inputs, two hidden units, and the single output unit. The top three curves represent the value of the counter in the re-randomizer.

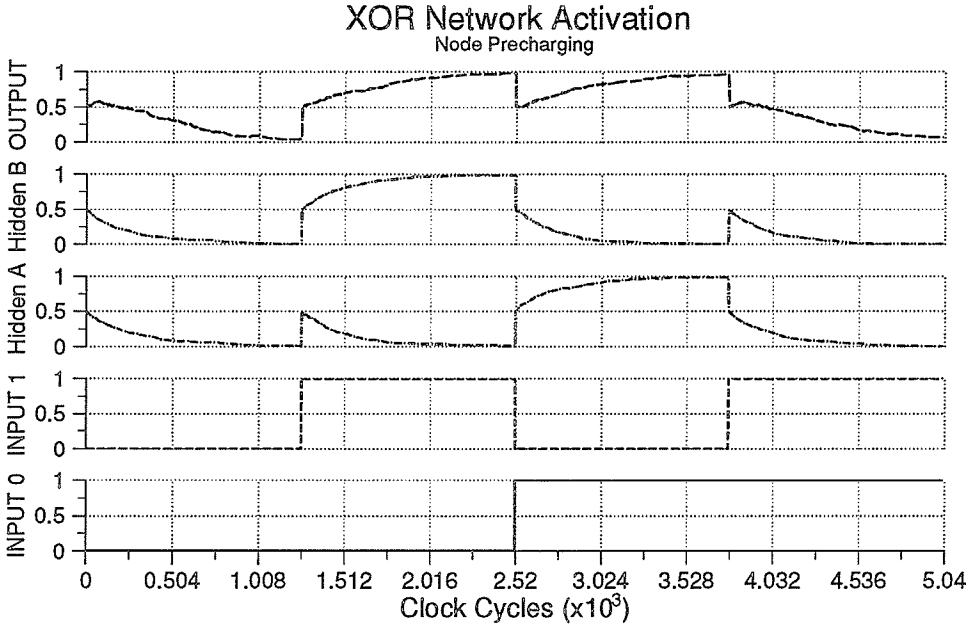


Figure 4.15: Network activation for the XOR problem. Shown are the two inputs, two hidden units, and the single output unit. The top three curves represent the value of the counter in the re-randomizer.

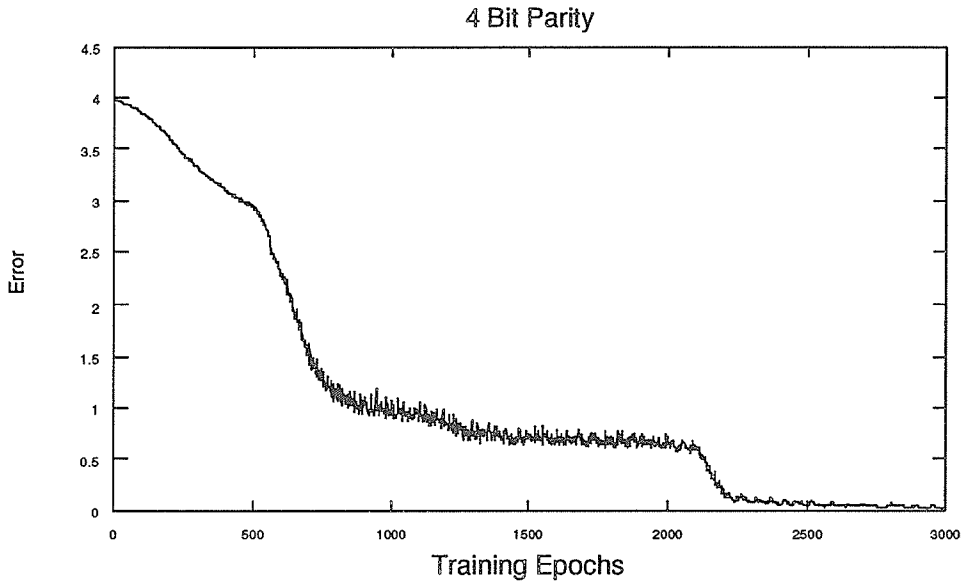


Figure 4.16: Training error for the four bit parity problem

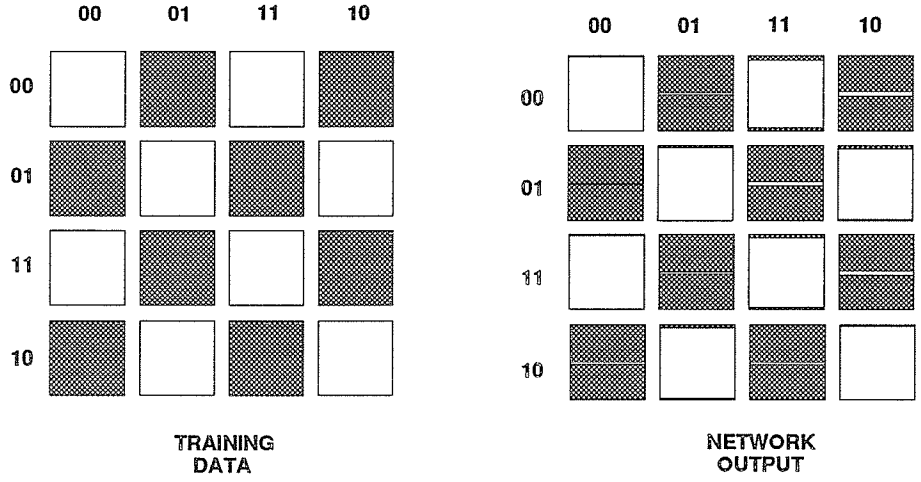


Figure 4.17: Training data (left) and network output (right) for the four bit parity problem

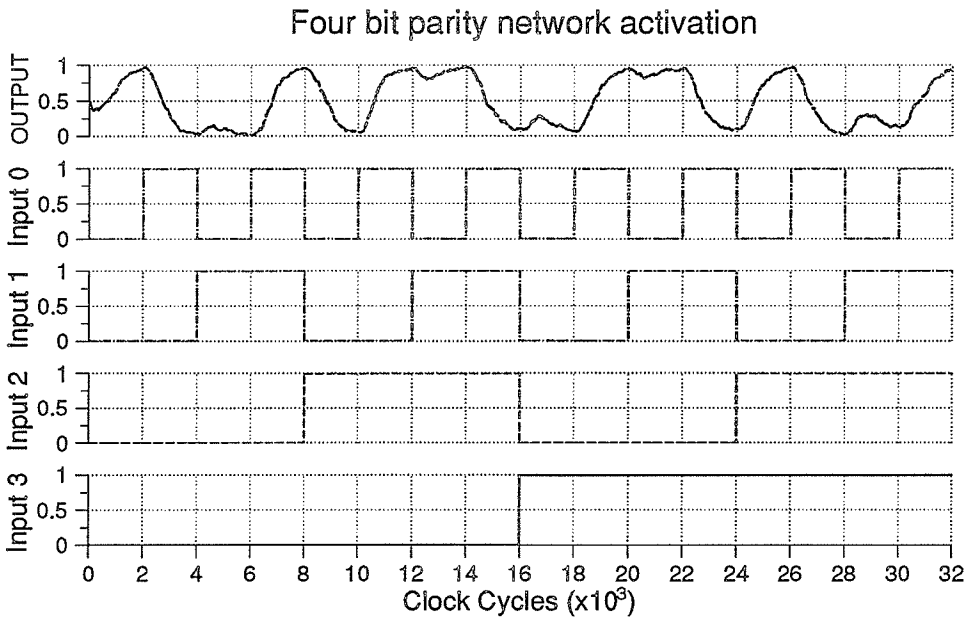


Figure 4.18: Network activation for the four bit parity problem

input vectors is shown in Figure 4.18.

4.5.3. Hexadecimal character recognition

The final example considered in this section is a hexadecimal character recognition problem. The input to the network is 20 bits, determined from a 4X5 matrix representing the hexadecimal character. The output is the 4 bit binary number corresponding to the input character. The network consisted of 1 hidden layer of 12 units and an output layer of 4 units. The training error is shown in Figure 4.19. The activation of the pulse stream network is shown in Figure 4.20 and a Hinton diagram of the outputs is shown in Figure 4.21.

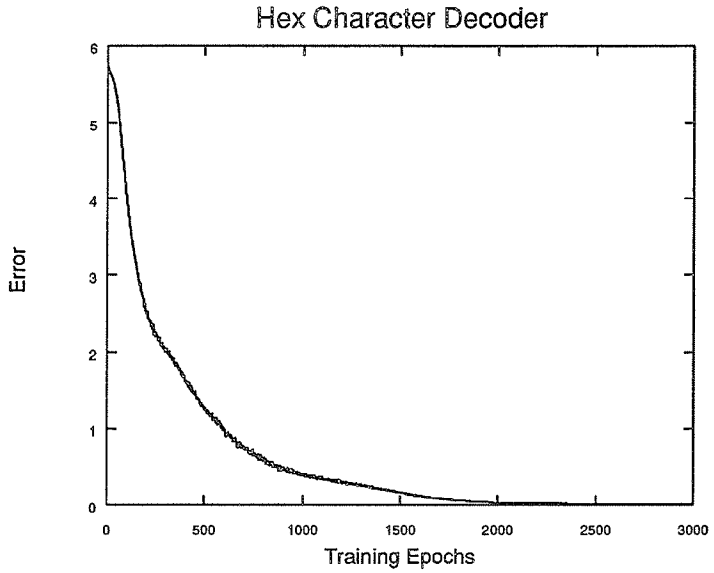


Figure 4.19: Training error for the hexadecimal character recognition problem

4.5.4. Impact of division

The equations for the learning algorithm contain division. Division is undesirable for a number of reasons. First, division by zero must be avoided. Second, division is a time consuming operation for most computers. Third, and most important, division is a difficult operation to implement in hardware. Implementation of an on-chip divider for each synapse and output would be prohibitive to the implementation of on-chip learning, discussed in the next chapter.

To determine whether division was necessary, the networks were trained with the division operation omitted. The results of the simulation are shown in Figure 4.22. The results show that eliminating the division does not impair the ability of the networks to minimize the error. The hexadecimal character recognition

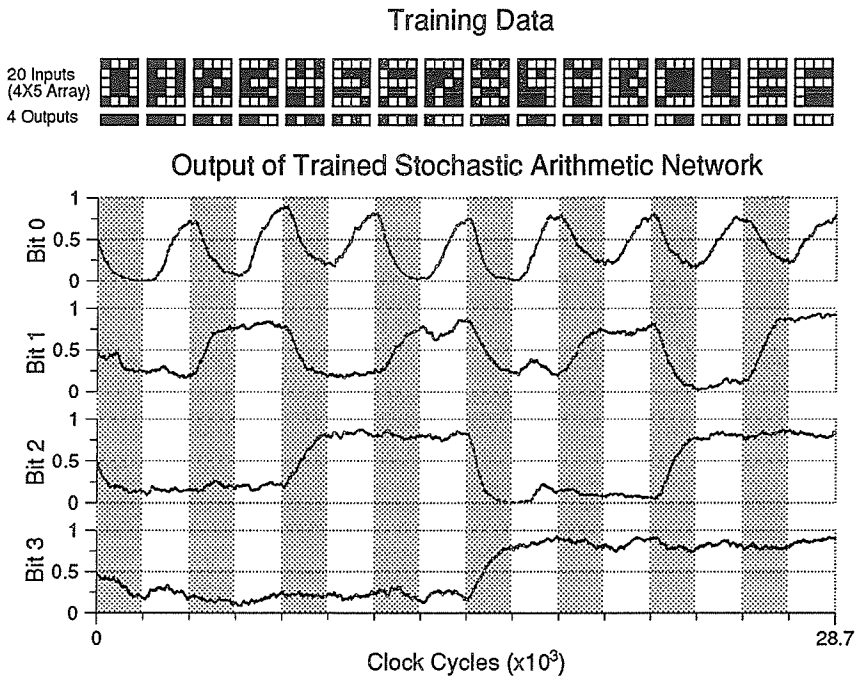


Figure 4.20: Network activation for the hexadecimal character recognition problem

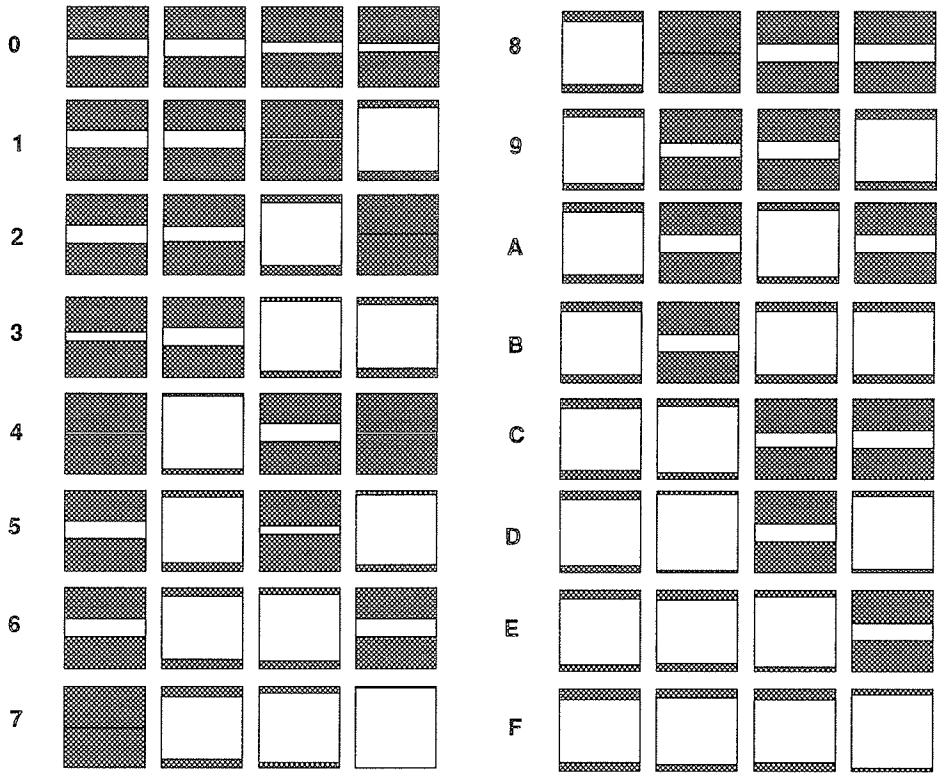


Figure 4.21: Network outputs for the hexadecimal character recognition problem

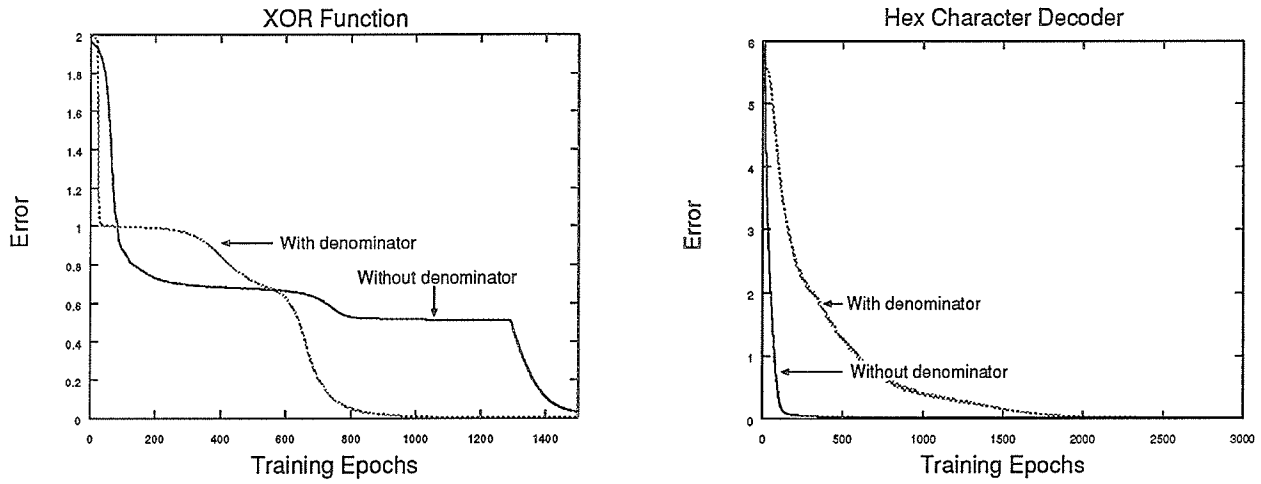


Figure 4.22: The impact on division on network training. The graphs show comparison of the training with and without the division term. As can be seen, the exclusion of the division does not impair the ability to minimize the error. This has significant benefits for the development of in situ learning

problem trained in fewer epochs without the division.

4.5.5. Comparison with conventional hardware

The limitations of the stochastic representation, specifically the limited range of the synaptic weights, is not without its cost. The four bit parity example required two hidden layers of 12 and 8 stochastic units. Conventional arithmetic using back propagation, sigmoidal neurons, and synapses of unlimited range required only 1 hidden layer of 10 units to solve the parity problem. The addition of the extra layer, which provides scaling to accommodate the limited synaptic connections, is crucial for the success of these stochastic arithmetic networks. While the network complexity of the networks is greater for stochastic arithmetic, the

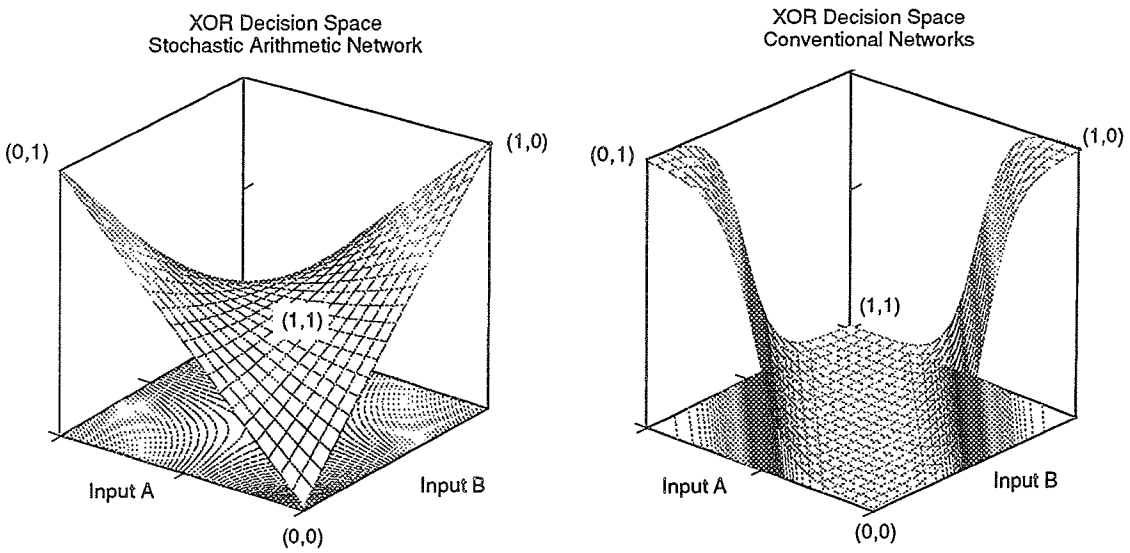


Figure 4.23: Comparison of the decision spaces of Stochastic Neural Networks and conventional sigmoidal Backpropagation Networks.

hardware complexity is still significantly less. Since the hardware requirements of the stochastic arithmetic approach are significantly less than conventional digital approaches, the cost of adding more computing elements is not severe.

It is also instructive to compare the decision spaces of Stochastic Arithmetic and conventional sigmoidal networks. Figure 4.23 compares the XOR problem decision space of the stochastic arithmetic net (section 4.5 and the sigmoidal network (section 2.3).

While the stochastic network may not have been a good classifier for this particular problem it is interesting to compare the solution with the analog XOR function. Figure 4.24 shows that the stochastic arithmetic, trained on only the four binary inputs, has correctly predicted the function for the entire input space.

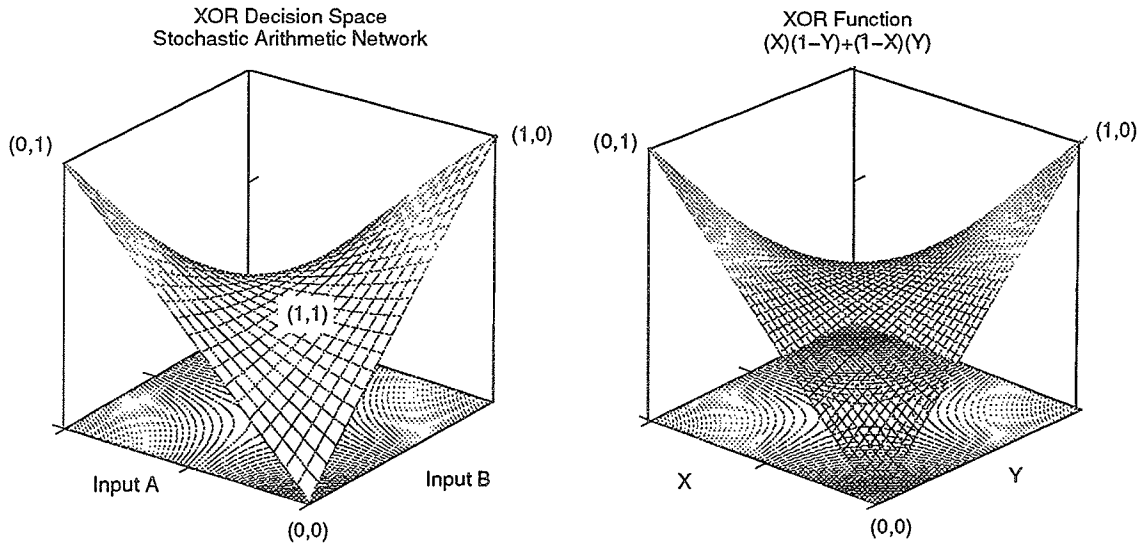


Figure 4.24: Comparison of the stochastic arithmetic network and the XOR function

The logic gates used in the stochastic arithmetic network coincidentally produced the complete solution.

4.5.6. Weight Resolution

During training and simulation of network outputs the resolution of the weights was 8 bits plus 1 sign bit. Thus there were were $2^8 - 2$ possible positive values, $2^8 - 2$ negative values, and the zero. Increased resolution requires more hardware due to larger weight registers and expanded rate multipliers. Reduced resolution requires less hardware.

To investigate the weight resolution necessary to solve an arbitary problem a network was trained to perform the hexadecimal character recognition problem.

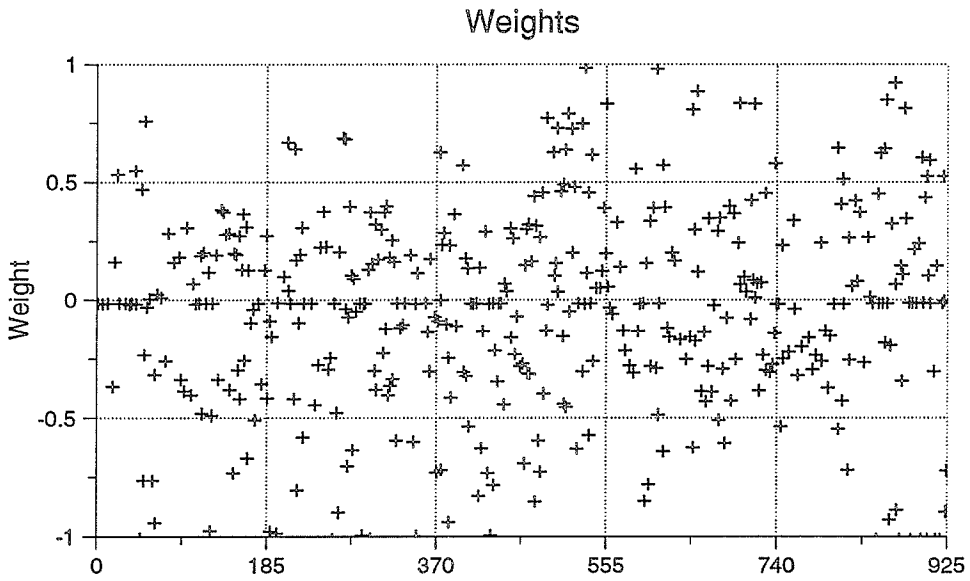


Figure 4.25: The weight space of a two layer network trained to solve the Hexadecimal OCR problem. The values of the weights in the network are plotted.

The resulting weights are shown in Figure 4.25. Note that the distribution of the weights favours numbers of small magnitude. This is expected due to the large fan-in to the OR gate adders.

The weights were successively mapped to weights of smaller resolution until the network produced incorrect outputs. Figure 4.26 shows that only 5 distinct weights(+1, +.5, 0, -.5, -1) were needed in the network to solve the problem. This weight set requires little hardware to implement, since the (+1,-1) weights are direct connections and the 0 weights imply no connection. Further experimentation is required to determine the effect of this quantization on generalization performance.

The weight distribution in Figure 4.25 suggests that it may be beneficial to

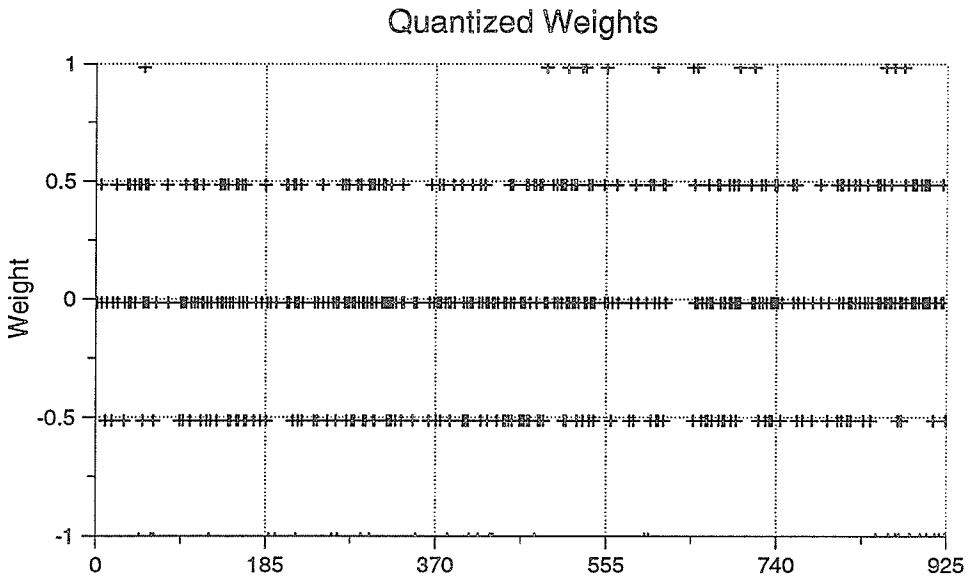


Figure 4.26: The weight space after quantization to (+1, +.5, 0, -.5, -1). The network continues to produce the correct outputs.

employ a nonlinear set of weights for these networks. The upper magnitudes could be represented by only a few weight values, while more values would be located closer to zero. The OR gate saturation characteristic means that weights should be kept small, and there is no point in accommodating large weights that will encourage saturation. A nonlinear rate multiplier could be constructed using AND gate multiplication of predetermined pulse streams. For example, the set of pulse streams densities 0, 1/4, 1/3, 1/2, 1 could generate weights of 0, 1/24, 1/12, 1/8, 1/6, 1/4, 1/3, 1/2, 1.

4.6. VLSI Implementation

This section discusses the implementation of stochastic arithmetic neural networks in VLSI hardware.

4.6.1. OR gate Addition

For fan-in to the neurons greater than four inputs, OR gate addition has severe drawbacks. The large OR gates consume area and time. More important, it is undesirable to have many lines coming into a neuron for summation. One solution is to multiplex the data on one line. Another method to address this problem is to distribute the OR gates among the synapses. Both these methods add complexity and size to the system, and slow down the operation of the network.

The solution is to use a wired OR gate. A wired OR gate requires no active circuitry to compute the OR function. The inputs are simply wired together, as shown in Figure 4.27. To avoid contention between HIGH and LOW signals, it is necessary to convert the inputs to the OR gate from [0,1] to [Z,1], where Z is the high impedance state. The three transistors required for this are also shown in Figure 4.27. This circuit would be placed on the n_j^+ and n_j^- outputs of each synapse. The final pmos transistor should be sized to provide adequate drive, and it may be wise to include pull-down transistors at each synapse to accelerate circuit operation.

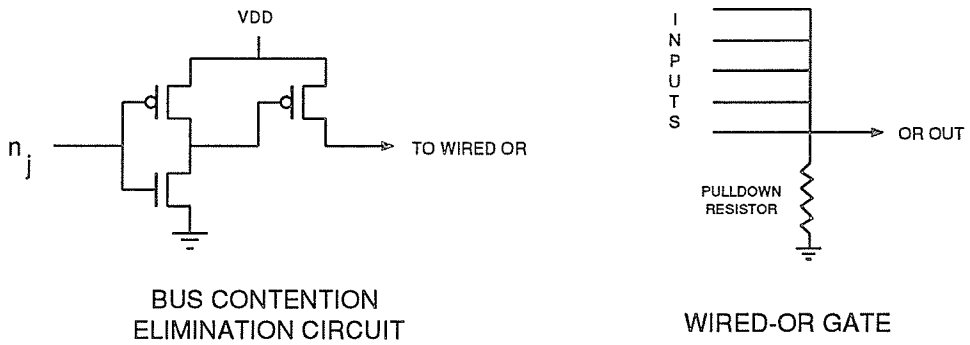


Figure 4.27: The wired OR gate and bus contention circuit.

4.6.2. Random Number Generation

Random numbers will be required at every synapse and neuron to provide a set of bits to use in the generation of a weighted pulse stream. It would be impractical to include a CA based random number generator at each rate multiplier. One solution is to use a shift register and shift a random number to each rate multiplier. This approach has the drawbacks of requiring area for the registers and time for the shifting. The method proposed here is based on the same principles of temporal independence discussed in Chapter 3. All synapses extending from a neuron will use the same random number, since each communicates its result to a unique neuron. Delay registers between neurons provides temporal independence. Figure 4.28 shows this method. Only one random number generator is required.

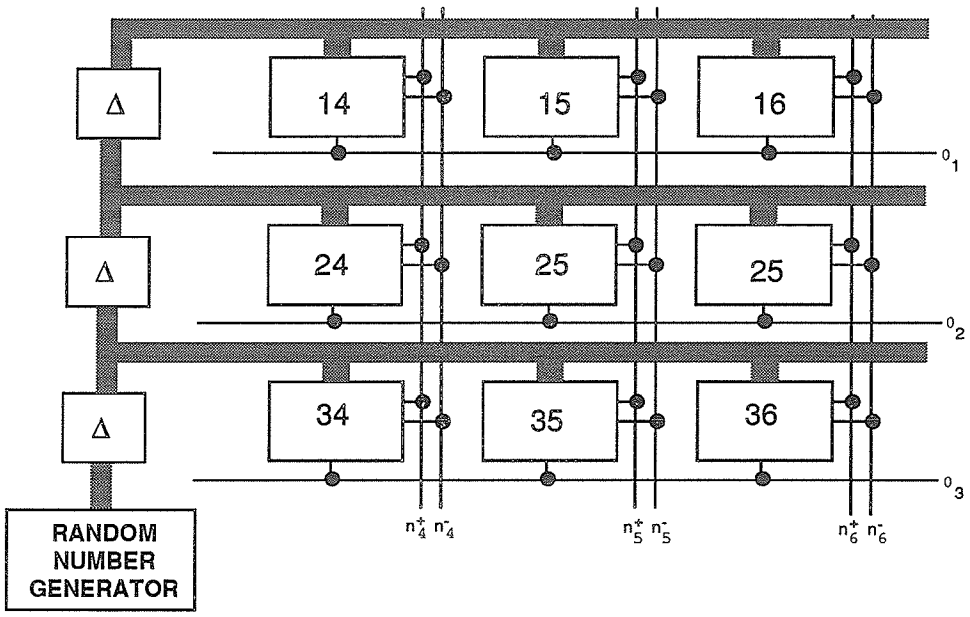


Figure 4.28: Block Diagram of integrated circuit

4.6.3. Chip Implementation

To demonstrate hardware pulse stream networks, a custom VLSI implementation of a simple network was designed and fabricated. The chip implements a network of two hidden layers, each with four hidden units, as shown in Figure 4.29. The weights are represented with an eight bit storage register and can be loaded serially from off-chip. The random number generator has not been placed on the chip to enable experimentation with alternate generation methods. The random number used in the synapses and rerandomizers is stored in a register local to each unit. These registers are connected in parallel and are separated by a delay register. By using a delay between registers, each random number can be used by each pulse generation unit (synapse or rerandomizers). The fabricated chip is shown

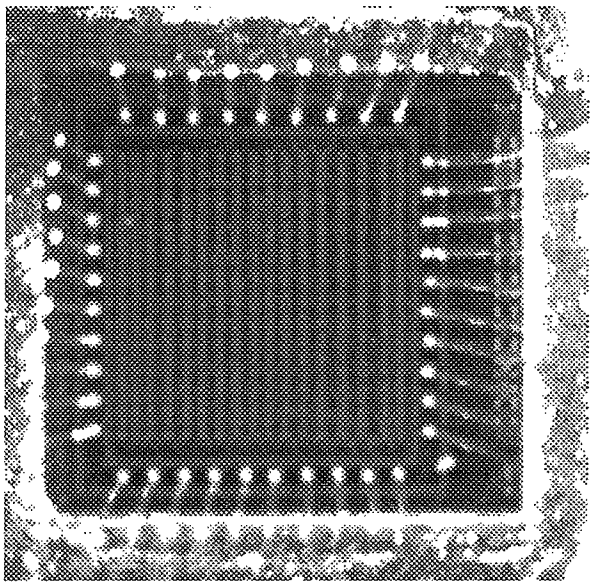


Figure 4.29: VLSI Implementation of Stochastic Arithmetic Neural Networks. The chip implements a two layer network of 4 units per layer

in Figure 4.29.

4.7. Conclusion

This chapter has discussed stochastic arithmetic implementations of neural networks. The training rule was discussed and applied to three problems. The hardware necessary to implement these networks was also presented.

While the results of this chapter have shown stochastic arithmetic to be a viable implementation option for neural networks, further experimentation should be performed. Larger and more complex problems should be trained on the network. More detailed comparisons with other implementations should also be performed.

Chapter 5

In situ Learning

The previous chapter examined the operation of Stochastic Neural Networks whose weights were determined by off-line training. This chapter presents an extension to that work: Stochastic Arithmetic Neural Networks that perform the learning in situ using on-chip circuitry.

It is desirable in many applications to have the learning performed by the hardware. First of all, the training procedure typically requires many passes through the input data and is generally very time consuming especially since it is usually performed on sequential computers. In addition, if certain neurons are not functioning in the hardware, the learning algorithm can compensate by adjusting the weights to the other neurons. Neural networks can also be found in control applications, where a network must adapt according to inputs which may not be available for off-line training. In situ learning frees neural networks of complex external hardware required for training thereby increasing their flexibility and areas of application.

However, learning is rarely found in dedicated neural network hardware. Net-

works with in situ learning must calculate the error at the output, propagate it backwards towards the inputs, and modify the synaptic connections accordingly. In addition to extra hardware required to perform the weight updates, the communication complexity alone is enough to make in situ learning intractable, especially for digital implementations with wide data buses.

This chapter derives the equations for in situ learning, describes the hardware necessary to implement the equations, and presents simulations of the hardware as it learns two of the problems from the previous chapter.

5.1. Previous Work

The only previous stochastic arithmetic implementation with in situ learning is the work by Eguchi et al. at Ricoh[15]. The networks were like the networks described in the previous chapter. Their learning seems to be based on a simple delta learning rule.

Their implementation differs in many aspects from the approach described in this chapter. There was only a single neuron on each chip, with the weights stored in on-chip static RAM, indicating that the implementation was not fully parallel. Random number generation used linear feedback shift registers.

The efficacy of the learning rule described in [15] could not be duplicated for the training problems examined in this thesis.

5.2. Derivation of the in situ Learning procedure

5.2.1. Calculation of the Error

It Chapter 4 the error at the output neurons was

$$Error = t_j - o_j \tag{5.1}$$

This presents two difficulties for stochastic arithmetic. First, the pulse representation has no simple implementation of subtraction. Unlike the other neural network operations of addition and multiplication, subtraction can not be performed by a simple gate. Second, only unipolar quantities are available. A single stream can not represent the error when $t_j > o_j$ and $t_j < o_j$.

The solution is to split the error into two pulse streams, errorP and errorM. The stream errorP has a pulse when the training pulse stream is high and the output is low, while the errorM stream has a pulse when the output is high and the training stream is low. If the training and output pulses are the same, there is no error and both errorM and errorP are low. For binary training data this calculation of the errors will be exact.

5.2.2. Back-propagation of the error

The implementation of two error streams complicates learning, since there are two error signals in addition to the two net inputs (n_j^+, n_j^-). The equation for error propagation was given in Equation 4.25 (page 48) as:

$$\frac{\partial E}{\partial o_j} = \sum_{k,w_{ij}>0} \left(\frac{\partial E}{\partial o_k}(1-n_k^-)(1-n_k^+)\frac{w_{kj}^+}{1-w_{kj}^+ o_j} \right) + \sum_{k,w_{ij}<0} \left(\frac{\partial E}{\partial o_k}(n_k^+)(1-n_k^-)\frac{w_{kj}^+}{1-w_{kj}^+ o_j} \right)$$

It was shown in Section 4.5.4 that the denominator terms are unnecessary for effective learning. While division is possible with stochastic arithmetic (Section 3.1.4), it requires time and area to compute. Ignoring the division reduces the complexity of the circuits.

Since the error is split into two separate nets, errorP and errorM, the above equation must be modified. The errorP net will be modified by terms that produce positive results. Examining the equation shows that positive error will be contributed when the incoming error and the weight of the unit have the same sign (i.e. the product will be positive). Thus $errorP_{jk}$, the positive error at neuron j due to neuron k of the layer above is:

$$errorP_{ij} = \left[(1-n_j^+)(errorP_j)(1-n_j^-)_{(w_{ij}>0)} + (1-n_j^-)(errorM_j)(n_j^+))_{(w_{ij}<0)} \right] |w_{ij}| \tag{5.2}$$

Similarly, the errorM net will arise from terms that have a negative result when the error and weight have opposite signs:

$$errorM_{ij} = \left[(1-n_j^+)(errorM_j)(1-n_j^-)_{(w_{ij}>0)} + (1-n_j^-)(errorP_j)(n_j^+)_{(w_{ij}<0)} \right] |w_{ij}| \tag{5.3}$$

Each connection from a neuron will have a back propagated error signal associated with it. These errors must be added to produce the errorP and errorM values. This summation is performed like the summation of the net inputs to the neuron using OR gate addition. For large errors the addition will saturate, but as the network learns, the errors will become small and the addition will be accurate.

Let:

$$\delta^{++} = (1 - n_j^+)ErrorP(1 - n_j^-) \tag{5.4}$$

$$\delta^{--} = (1 - n_j^-)ErrorM(n_j^+) \tag{5.5}$$

$$\delta^{-+} = (1 - n_j^+)ErrorM(1 - n_j^-) \tag{5.6}$$

$$\delta^{+-} = (1 - n_j^-)ErrorP(n_j^+) \tag{5.7}$$

These variables are computed for each neuron and transmitted to the synapses that feed into it.

5.2.3. Weight Updates

The weight update equations, ignoring the division, are shown below:

$$-\frac{\partial E}{\partial w_{ji}^+} = \epsilon_j(1 - n_j^-)(1 - n_j^+)o_i \tag{5.8}$$

$$-\frac{\partial E}{\partial w_{ji}^-} = \epsilon_j n_j^+(1 - n_j^-)o_i \tag{5.9}$$

Recall that the weights are modified in proportion to the negative gradient of the error with respect to the weight. Therefore:

Weight	Condition	Weight Change
$w_{ij} > 0$	$\delta^{++}\ \overline{\delta^{-+}}\ \eta\ o_i$	$w \uparrow$
	$\delta^{-+}\ \overline{\delta^{++}}\ \eta\ o_i$	$w \downarrow$
$w_{ij} < 0$	$\delta^{+-}\ \overline{\delta^{--}}\ \eta\ o_i$	$w \uparrow$
	$\delta^{--}\ \overline{\delta^{+-}}\ \eta\ o_i$	$w \downarrow$

Table 5.1: Weight updates

$$\Delta w_{ij}^{+} = \eta \frac{\partial E}{\partial o_j}(1 - n_j^{-})(1 - n_j^{+})o_i \tag{5.10}$$

$$\Delta w_{ij}^{-} = \eta \frac{\partial E}{\partial o_j} n_j^{+}(1 - n_j^{-})o_i \tag{5.11}$$

The learning rate is determined by a pulse stream generated with $P(1) = \eta$. These weight equations must be modified for the two error streams.

$$\Delta w_{ij}^{+} = \eta(errorP - errorM)(1 - n_j^{-})(1 - n_j^{+})o_i$$

If η and o_i are both high then a positive weight will be increased when $\delta^{++} > \delta^{-+}$. This will occur when $\delta^{++} = 1$ and $\delta^{-+} = 0$. The weight will be decreased when $\delta^{++} = 0$ and $\delta^{-+} = 1$ and η and o_i are high. A similar analysis holds for negative weights. The weight update equations are summarized in Table 5.1.

5.2.4. Analysis of the Learning Procedure

The learning procedure described in Chapters 2 and 4 performed the weight updates after each epoch or after each individual training pattern. The learning

rules presented above are different in that the weight updates are made *during* pattern presentation. In effect, we are calculating:

$$\frac{dw_{ij}}{dt} = -\eta \frac{\partial E}{\partial w_{ij}}$$

Previous work has shown this form of weight update to be effective in speeding up learning[16].

The learning will have to deal with the variance inherent in the pulse streams. Learning is notorious for its requirement of high accuracy, and it is not clear that the noise will allow effective learning to occur. However, it has been observed that noise can be beneficial to learning [17].

Simulation of this learning algorithm is presented later in the chapter.

5.3. Hardware Implementation

This section describes the hardware required to implement in situ learning. In situ learning adds complexity in two ways: it requires hardware to compute and propagate the error signals, and it requires hardware to adjust the weights according to those signals. Figure 5.1 shows a block diagram of an in situ learning synapse. The synapse weight is held in an up/down counter, which is controlled by the error calculation hardware.

As shown in Figure 5.2 only simple gates are required to generate the error streams at the output units.

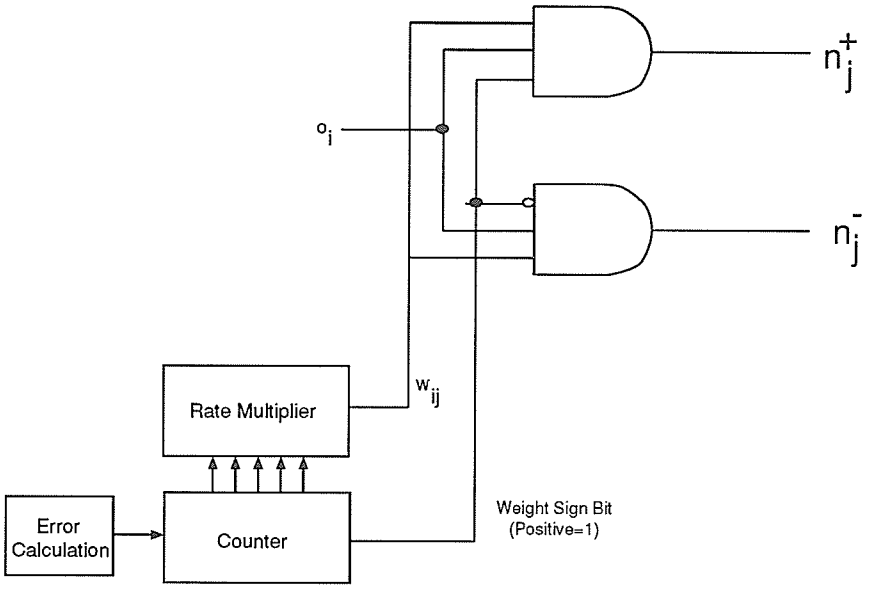


Figure 5.1: Block diagram of in situ learning synapse.

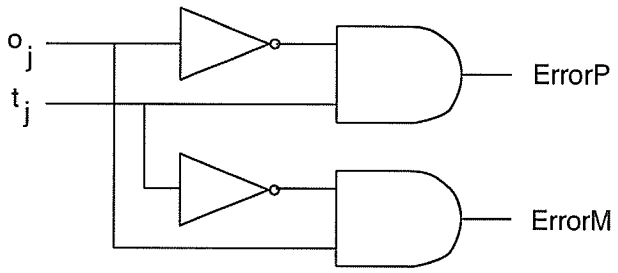


Figure 5.2: Hardware required for generating the error streams at the output neurons.

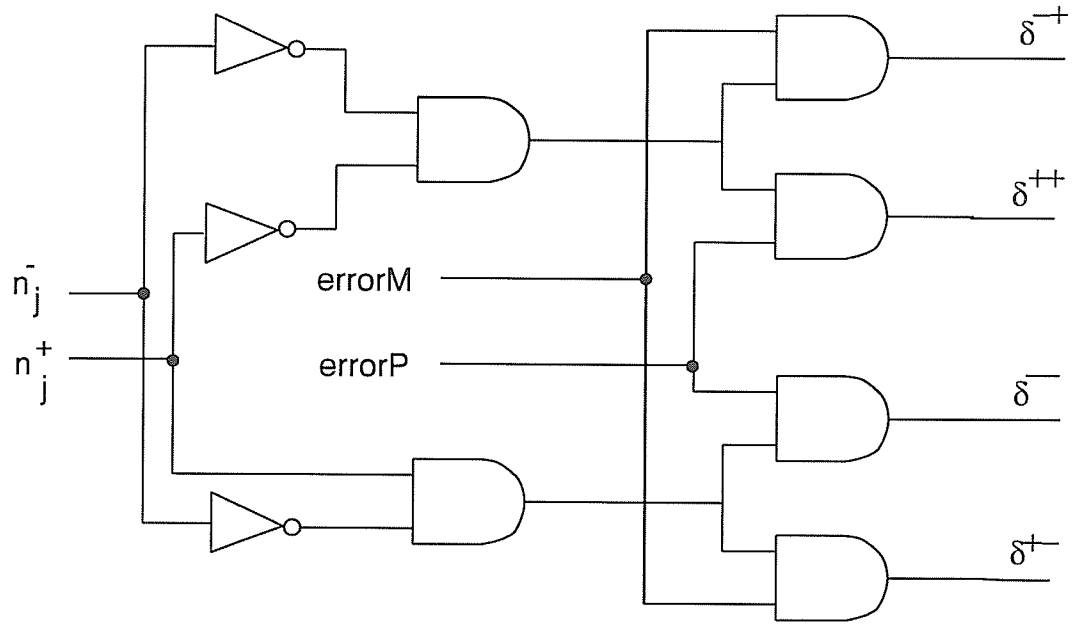


Figure 5.3: Hardware implementation of δ variables.

Each neuron will compute the δ nets and propagate through the synapses to the previous layer. Figure 5.3 shows the hardware required to compute these variables, as given by Equations 5.4 through 5.7.

The synapses are responsible for computing the error for back propagation and the weight changes. Figure 5.4 shows the hardware to compute the errorP and errorM nets. The errorP and errorM nets incoming to a neuron are summed using wired OR summation.

The weight change computation circuitry is a direct implementation of Table 5.1 and is shown in Figure 5.5. The output of this circuit controls an up/down counter that handles the weight modification.

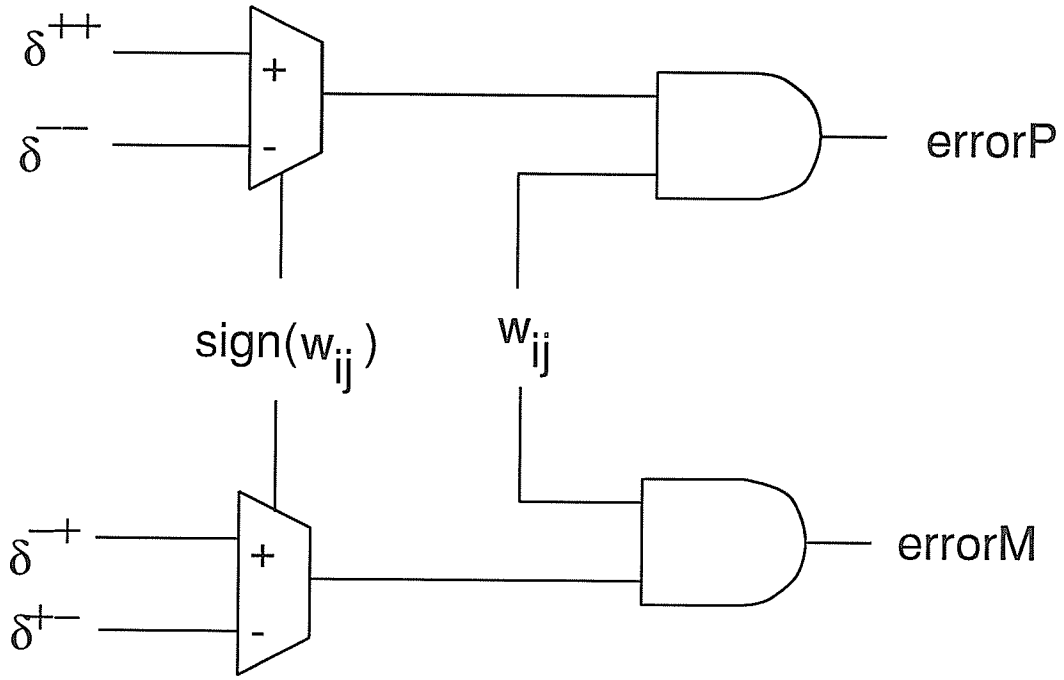


Figure 5.4: Hardware implementation of error back propagation.

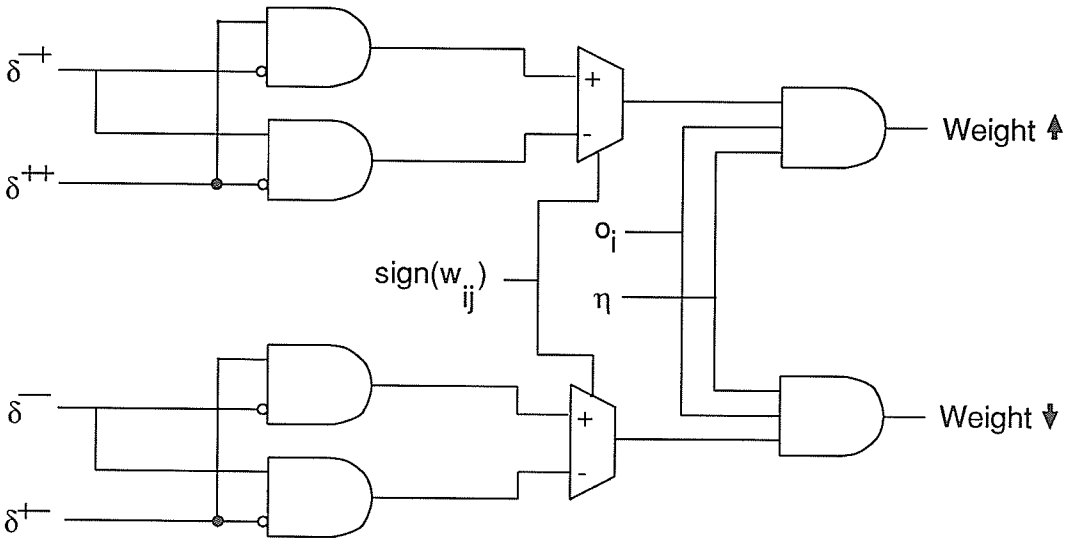


Figure 5.5: Hardware implementation of weight change computation.

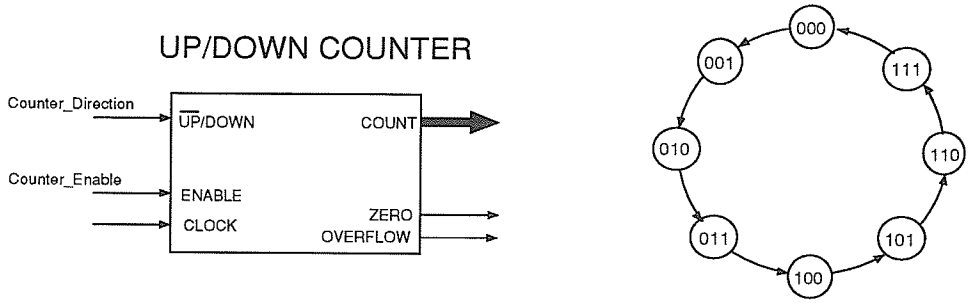


Figure 5.6: Up/Down counter building block

The basic element performing the weight updates is an up/down counter. Based on the results of the weight learning circuit, the counter will increment, decrement, or perform no operation.

A block diagram of a basic up/down counter is shown in Figure 5.6. The counter inputs are: direction control (0=UP, 1=DOWN), enable (1=COUNT), and clock. The outputs are the count (n bits), a zero flag (count=0) and an overflow flag (count=full range). Also shown in Figure 5.6 is the state diagram of a 3 bit counter (only the up direction is shown).

This counter must be modified for use in these networks. It must support negative values and it must represent values in sign magnitude form. Since the counter is a crucial element of the learning hardware, the circuitry for these enhancements will be presented in detail. The state diagram for a 3 bit counter that meets these requirements is shown in Figure 5.7. The up/down counter circuit can be found in many common references, such as [18].

When the number is positive, the up/down counter operates normally. For sign magnitude the number in the counter gets *larger* when a negative number in

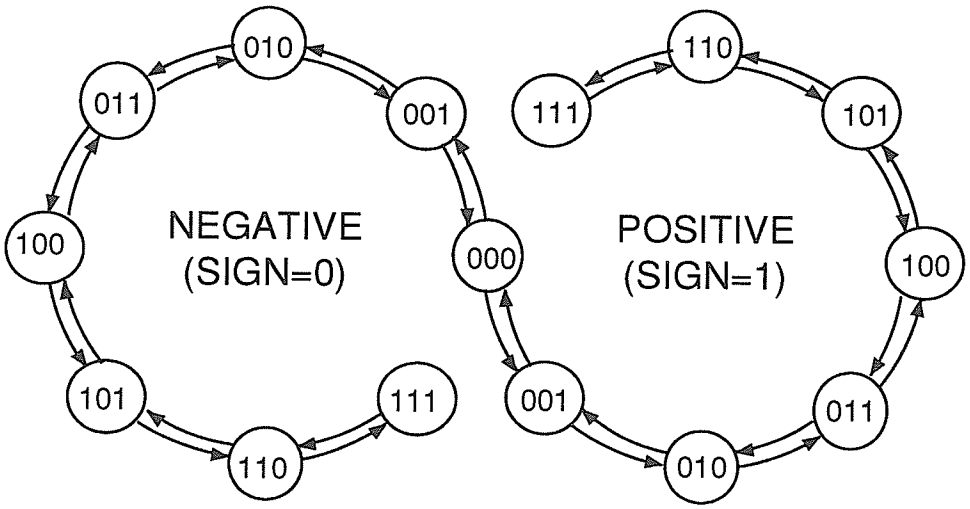


Figure 5.7: State transition diagram for a three bit sign magnitude counter. The full count state can only be left by decrementing the magnitude.

decremented. Likewise, incrementing a negative number will make the magnitude smaller. When the number is zero, the next value is always magnitude 1, so the counter must increment when the value is zero. The additional control logic to implement this behavior is shown in Figure 5.8.

Figure 5.8 also shows the logic necessary to make the counter stop at plus or minus full range. If the counter is at full range, the counter will be enabled only when the count is reduced in magnitude.

The sign of the weight can only change when the value passes through zero. If the DIRECTION line is low, then the counter is counting up and the next value is +1, which means the sign bit will be high. If the DIRECTION line is high, the next value will be -1 and the sign bit will be low. The sign is simply the inverse of the DIRECTION line if the count is currently 0; otherwise the sign will not

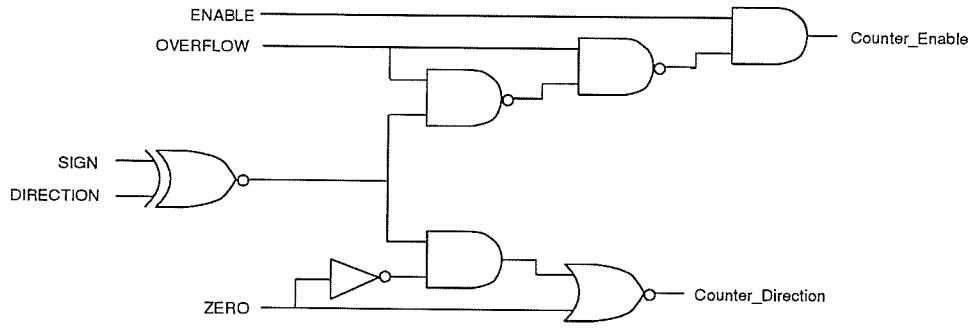


Figure 5.8: Counter control logic for sign magnitude up/down counter

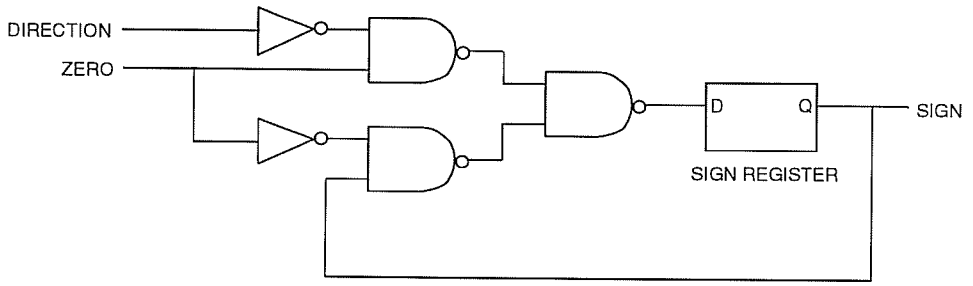


Figure 5.9: Sign logic for sign magnitude count

change. The logic for the sign is shown in Figure 5.9.

The only other additional hardware required for the counter is an XOR gate to perform $ENABLE = WEIGHT \uparrow \oplus WEIGHT \downarrow$. The direction input is connected to $WEIGHT \downarrow$.

5.4. Simulations of in situ Learning

The simulator used in Chapter 4 was extended to simulate learning at the pulse level using the equations derived in the previous section. The simulator is effectively a mixed mode gate/behavioral simulation of the hardware needed to

implement in situ learning. Except for counters and registers, the simulation was at the gate level.

5.4.1. The XOR and Hexadecimal OCR Problems

The performance of the in situ network on the XOR problem is shown in Figure 5.10. The training error is graphed for six different learning rates. The network consisted of 1 layer of three hidden units. Note that the training error was effectively zero after only 75 iterations through the training set. The network in chapter 4 required over 800 epochs to reach the same level of error reduction. This could have been due to a number of causes. The random noise present in the learning and activation pulse may have assisted learning, or the continuous weight updates may be responsible. It is likely that the learning improvement is due to a combination of these factors.

The training for the hexadecimal OCR problem is shown in Figure 5.11. The network consisted of one layer of 10 hidden units.

Figure 5.12 shows the effect of variance on the bit streams. If it were not for the variance of the pulse stream densities, the effective learning rate for each of the two graphs would have been the same. Observe, however, the learning curves are different.

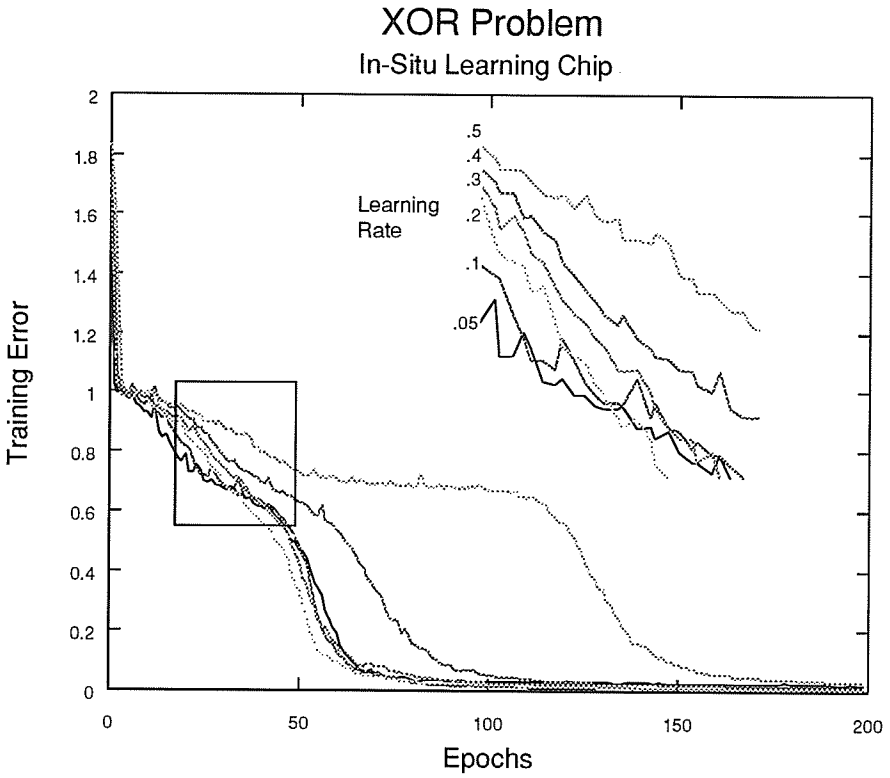


Figure 5.10: Simulation of in situ learning for the XOR problem.

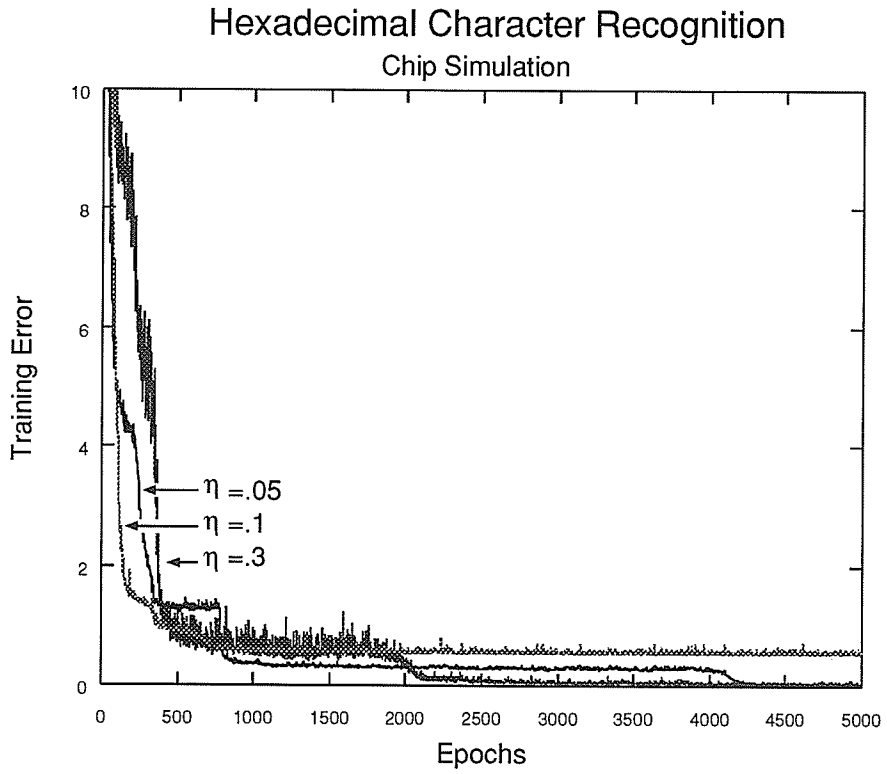


Figure 5.11: Simulation of in situ learning for the hexadecimal character recognition problem.

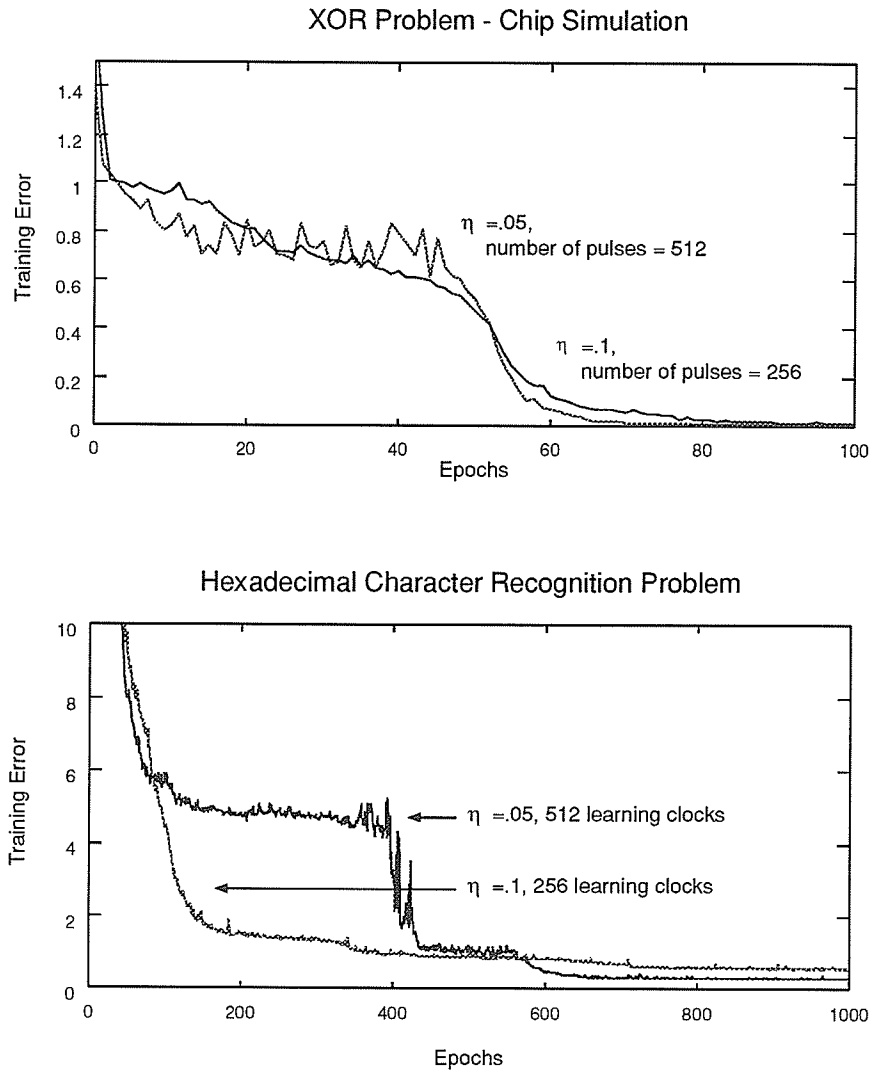


Figure 5.12: The impact of variance on in situ learning

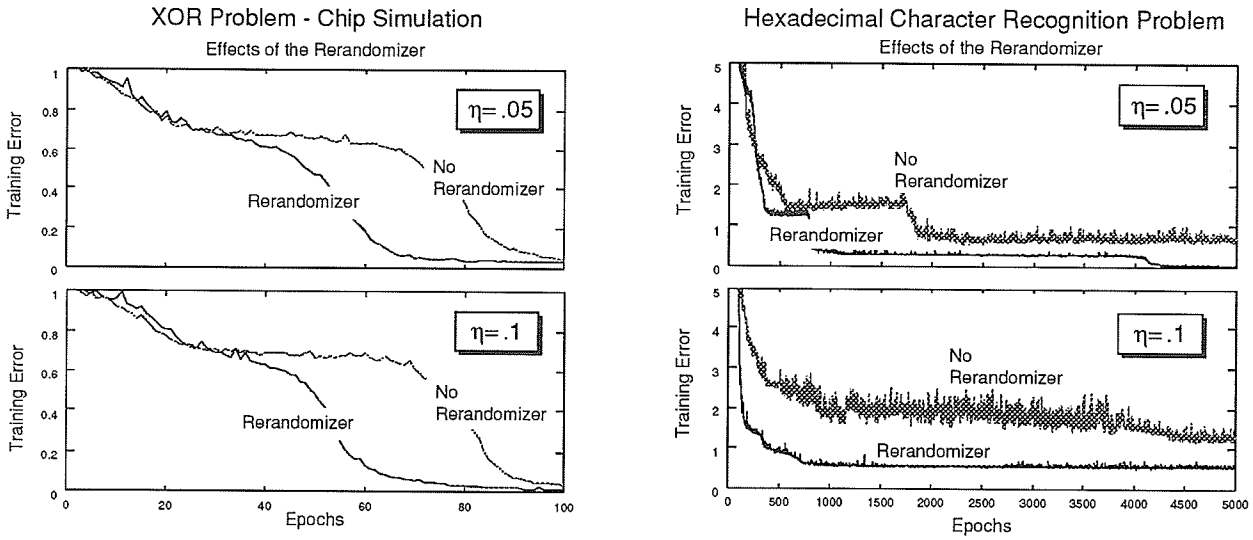


Figure 5.13: Effect of the rerandomizer on learning.

5.4.2. The Rerandomizers and in situ learning

Figure 5.13 shows the effect of the rerandomizers on the learning process. The left graph shows that the network successfully learned the XOR problem without the rerandomizers, although it took longer. The hexadecimal character recognition was unable to minimize the error without the rerandomizers. The noise on the training error curve for the situation without rerandomization indicates that the correlation noise had a detrimental effect on the learning.

The rerandomizer has another benefit for learning in addition to the removal of correlation. Note from the weight update rules of Table 5.1 that o_i is one of the factors. If the activation of the neuron becomes zero, then no weight change will take place. All weights from a neuron with activation of zero will remain constant. Even small, non-zero activations, will impair learning. The rerandomizers can be

designed so that the activation will not fall below a certain activation. As learning progresses, the full range of the rerandomizer can be employed to complete the training. Simulations have shown this to be a very powerful technique to improve learning.

5.4.3. Weight Resolution

For the previous simulations the weights had resolutions of eight bits. The variance of the pulse density, clearly evident in Figure 3.13 and Figure 4.5, suggests that far less resolution may be required. However, simulations of in situ learning for seven bit weights were largely unsuccessful. From this can be drawn two conclusions:

1. Despite the variance in the pulse density, the average pulse density was dominant.

2. Small weight changes are important.

The second point suggests that the nonlinear weight set discussed in the previous chapter may not be suitable for in situ learning.

5.5. VLSI Implementation

To investigate the area and time required of the in situ learning algorithm, the hardware presented in this chapter has been implemented as an integrated circuit. In 1.2 μm CMOS, approximately 200 in situ learning synapses could be

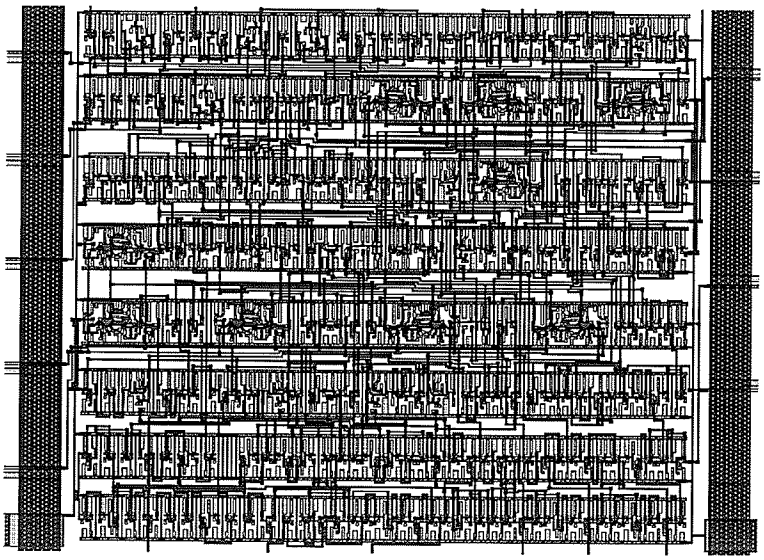


Figure 5.14: Layout of synapse with in situ learning.

implemented on a single chip. Because the design is digital, larger networks can be accomodated by simply cascading chips. At a conservative 25 MHz clock, the chip can process 100 000 patterns per second without the rerandomizers. With the rerandomizers enabled the design can process 25 000 patterns per second. The chip could learn the hexadecimal OCR problem in under 1 second. In comparison the algorithm from Chapter 4 takes 60 seconds to train on a Sun Sparc 2 workstation. The layout of the in situ learning synapse is shown in Figure 5.14.

5.6. Conclusion

This chapter has formalized in situ learning using stochastic arithmetic neural networks. The learning algorithm has been developed and the necessary circuitry for

a hardware implementation has been designed. Both have been verified through simulation. A VLSI implementation of the in situ learning synapses has been presented.

Chapter 6

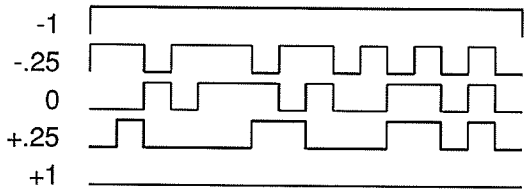
Conclusions and Future Work

The thesis has demonstrated the application of stochastic arithmetic to artificial neural networks. The hardware requirements of these networks was shown to be minimal; only simple gates were required to perform the arithmetic. This is very important for artificial neural network implementation, as area-efficiency results in larger networks and greater speed.

In addition, a novel in-situ learning neural network was presented. The learning hardware requires only simple digital gates to implement. This allows parallel implementation of the learning networks.

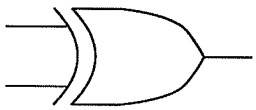
Future work should test the performance of these networks on larger problems. In addition, there are other pulse representations that could be used. A preliminary investigation into using a bipolar representation has been made. This representation uses an XOR gate for multiplication, shown in Figure 6.1.

The block diagram of the architecture envisioned for this representation is shown in Figure 6.2. This is not a fully parallel approach like the representation examined in this thesis. Each neuron stores the synaptic weights in local static



Multiplication

A	B	PRODUCT
0	0	0
0	1	1
1	0	1
1	1	0



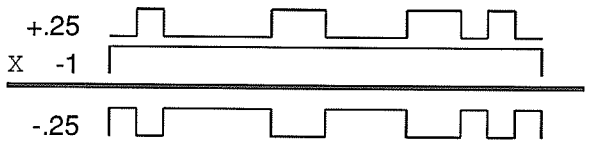


Figure 6.1: The XOR multiplication bipolar representation.

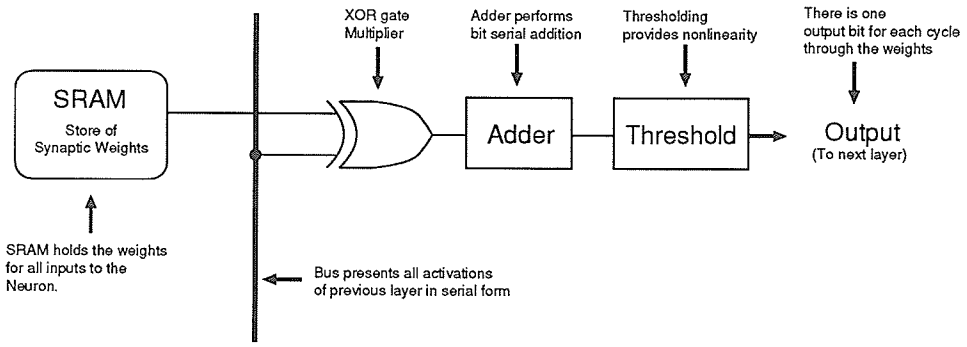


Figure 6.2: Block diagram of a neural network architecture using the bipolar representation.

RAM storage, and cycles through all the inputs accumulating the net input. A step nonlinearity with a gaussian random threshold is used to generate a sigmoid-like transfer function. While this architecture is not fully parallel, the synapses require no hardware since their function is incorporated into the neurons. In situ learning implementation with this representation is possible, although formal experiments have yet to be carried out.

Investigation should also be made into applying stochastic arithmetic to other neural network paradigms. Competitive learning algorithms would make an interesting subject for stochastic arithmetic.

BIBLIOGRAPHY

[1] B. U. Keller, R. P. Hartshorne, J. A. Talvenheimo, W. A. Catterall, and M. Montall. Sodium channels in planar lipid bilayers. *Journal of General Physiology*, 88(1), 1986.

[2] W. S. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[3] M. L. Minsky and S. A. Papert. *Perceptrons.* MIT Press, Cambridge, 1969.

[4] D. A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In D. S. Touretzky, editor, *Advances in Neural Information Processing I*, pages 305–313, San Mateo, 1989. Morgan Kaufmann.

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing.* MIT Press, 1986.

[6] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems II*, pages 524–532, San Mateo, 1990. Morgan Kaufmann.

[7] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation.* Santa Fe Institute Studies in the Sciences of Complexity. Addison Wesley, Redwood City, 1991.

[8] B. R. Gaines. Stochastic computing systems. In Julius T. Tou, editor, *Advances in Information Systems Science*, volume 2. Plenum Press, 1969.

[9] Peter Hortensius. *Parallel Computation of Non-deterministic alogorithms in VLSI.* PhD thesis, Department of Electrical Engineering, University of Manitoba, 1987.

[10] F. Brglez, C. Gloster, and G. Kedem. Hardware-based weighted random pattern generation for boundary scan. In *IEEE International Test Conference*, aug 1989.

[11] W. Wike and D. Van den Bout. Stonn: A stochastic neural network chip. In W. J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference.* MIT Press, 1990.

[12] Journi E. Tomberg and Kimmo K. K. Kaski. Pulse-density modulation technique in vlsi implementations of neural network algorithms. *IEEE Journal of Solid State Circuits*, 25(2):1277–1286, oct 1990.

[13] Max Stanford Tomlinson, Jr., D. J. Walker, and M. A. Silvilotti. A digital neural network architecture for VLSI. In *Proc. IJCNN-90*, pages 545–550, San Diego, CA, 1990.

[14] Drew van Camp, Evan E. Steeg, and Tony Plate. *XERION Neural Network Simulator.* Computer Science Department, University of Toronto, 1991.

[15] H. Eguchi, T. Furuta, H. Horiguchi, S. Oteki, and T. Kitaguchi. Neural network lsi chip with on-chip learning. In *Proceedings of IJCNN-91*, volume 1, pages 453–456, 1991.

[16] A. J. Owens and D. L. Filkin. Efficient training of the back propagation network by solving a system of stiff ordinary differential equations. In *International Joint Conference on Neural Networks*, pages (II) 381–386, jun 1989.

[17] A. von Lehman, E. G. Paek, P. F. Liao, A. Marrakchi, and J. S. Patel. Factors influencing learning by back-propagation. In *IEEE International Conference on Neural Networks*, pages 335–341, New York, 1988. IEEE.

[18] M. Morris Mano. *Digital Logic and Computer Design.* Prentice-Hall Inc., Englewood Cliffs, N.J., 1979.