Soft Competitive Learning Using Stochastic Arithmetic

by Bradley D. Brown

A thesis submitted to the Faculty of Graduate Studies
in partial fulfillment of the thesis requirements
for the degree of

Master of Science
in
Electrical Engineering

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

© Bradley D. Brown 1998

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-32908-9

Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

SOFT COMPETITIVE LEARNING USING STOCHASTIC ARITHMETIC

BY

BRADLEY D. BROWN

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

of

MASTER OF SCIENCE

Bradley D. Brown ©1998

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

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

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 also 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.

Bradley D. Brown 1998

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give your address and the date.

Contents

1. Introduction ____ *1*

1.1 Information Coding ____ **2**

2. Processing Element Design and Analysis ____ *3*

2.1 Multiplication ____ **4**

2.2 Squaring ____ **4**

2.3 Addition and Subtraction ____ **5**

2.4 State Machine based Computational Elements ____ **6**

2.4.1 Stanh Function ____ 8

2.4.2 Linear Gain Function ____ 10

2.4.3 Exponentiation Function ____ 12

2.5 Division ____ **14**

2.5.1 Unipolar Division ____ 14

2.5.2 Bipolar Division ____ 15

2.5.3 Increased Speed through Stepped Velocity ____ 15

2.5.4 Increased Speed through Scaled Processing Time ____ 16

2.6 Digital to Probability Converter (DPC) ____ **17**

2.7 Probability to Digital Converter (PDC) ____ **18**

2.8 Generation of Digital Noise ____ **18**

2.9 Effects of Processing Elements on Higher Order Statistics ____ **19**

2.9.1 Autocorrelation function of a Bernoulli Sequence ____ 20

2.9.2 Effects of Addition on State Machine Output Sequences ____ 21

3. Application of Stochastic Arithmetic to Soft Competitive Learning ____ *22*

3.1 Example Problem ____ **22**

3.1.1 Soft Competitive Learning Network Layer ____ 25

3.1.2 Linear Output Network Layer ____ 27

3.2 Baseline Solution Using Conventional Computation ____ **29**

3.3 Solution Using Stochastic Computation ____ **34**

3.3.1 Problems Encountered ____ 44

4. Conclusions and Future Work ____ *47*

5. Literature Cited ____ *49*

Abstract

Stochastic arithmetic principles have been known for many years (Gaines, 1969). The original motivation for considering computation using stochastic arithmetic was the simplicity of the computational elements involved. Pioneers in the areas of machine computation were faced with hardware implementations that were large, power hungry, and relatively unreliable. Stochastic arithmetic held out the possibility of carrying out complex computations with very simple hardware. Modern technology is now capable of fabricating systems with millions of computational elements at amazingly low costs. While this would appear to make stochastic processing irrelevant, there is a class of systems which require vast quantities of processing elements.

Artificial Neural Networks (ANNs) are massively parallel systems which can benefit from technology which allows implementation of an unusually large number of simple computational elements on a single integrated circuit. Unlike traditional computation devices such as microprocessors, neural networks are also characterized by a tolerance for much less accurate computation. Stochastic arithmetic may be performed by computational elements which are both very small and compatible with modern VLSI design and manufacturing technology.

Stochastic arithmetic provides a number of benefits over other computing techniques:

- very low computation hardware area
- fault tolerance
- simple communications over one wire per signal
- simple hardware implementations allowing very high clock rates
- capability to trade off computation time and accuracy without hardware changes

There are some disadvantages, such as the variance inherent in estimating the value of a stochastic signal and the increased number of clock cycles required to accomplish a given computation. The potential of massive parallelism driven by the small circuit areas involved may alleviate some of these disadvantages.

This thesis presents a number of stochastic computational elements, several of which are introduced for the first time in this thesis, and an analysis of their operation. The applicability of stochastic arithmetic to neural networks is demonstrated through the successful implementation of a sample problem, optical character recognition, using stochastic computation. While the accuracy, power and speed characteristics of stochastic computation may not compare favorably with more conventional binary radix based computation, the low circuit area requirements make them attractive for VLSI implementation of ANNs.

Results are presented for an example ANN application. Optical character recognition is performed on the characters in the E-13B MICR (Magnetic Ink Character Recogonition) font. The ANN is composed of two layers, the first layer being a set of Soft Competitive Learning (SCL) sub-networks and the second being a set of fully connected linear output

neurons. Each of the seven columns of each character is connected into a separate sub-network of SCL neurons, with each sub-network sharing a common set of weights in order to reduce overall network complexity. The fully connected linear output layer is trained using the delta rule and is pruned to less than 25% of its initial connections, also to reduce network complexity. After the output layer pruning, the network is implemented with 54 weights in the SCL layer and 130 weights in the output layer, for a network total of 184 weights.

The learning performance of the SCL layer is evaluated through the technique of transforming the SCL layer image interpretation back into the image space by multiplying SCL neuron activations by their receptive fields (weights). While a baseline double precision floating point implementation of the SCL layer trained to a squared error of 4.77, a stochastic implementation with 8 bit weights and 11 bit probability estimators trained to a squared error of 5.26 in approximately 35 million clock cycles.

The linear output layer was trained to a total network squared error of 0.545 in the case of the floating point implementation and 0.715 in the case of the stochastic implementation. The stochastic implementation required approximately 188 million clock cycles to prune and train the linear output layer. This number of clock cycles represents an order of magnitude improvement over the floating point implementation on the PC used for the simulations assuming clock frequency parity.

Network generalization capabilities were compared based on the network squared error as a function of the amount of noise added to the input patterns. The stochastic network maintains a squared error within 10% of that of the floating point implementation for a wide range of noise levels.

Acknowledgements

There are many people who have contributed to the process that I went through to develop this thesis. In a central position is my advisor Dr. Howard Card who is responsible not only for the original inspiration for the research area, but more importantly for pushing me to get it over the finish line. Dr. Card's efforts in respect of this thesis are greatly appreciated and will never be forgotten. My wife, Rosemin Brown, and our two children, Celina and Sarah, have been very patient and have sacrificed a lot of quality family time in order to allow me the time necessary to complete the thesis.

I would also like to thank the members of my thesis committee, Dr. R. D. McLeod and Dr. A. Alfa, for taking on the task of reviewing the thesis on short notice and for their comments and questions.

There are many other people who contributed to this thesis through technical discussions, assistance in development of software, and fighting with the tools required to bring it all together. Those students in the Ganglion group, the VLSI group, employees and others who made contributions to this thesis; Dean McNeill, Alex McIlraith, Derek Ross, Gord McGonigal, Jeff Dixon, Brendan Frey, and Zaifu Zhang all made special contributions.

A special thank you goes to my business partner, David Fletcher. He tolerated the time and effort that I put into this thesis without a single complaint, even while he shouldered extra duties in order to make it possible. His assistance in teaching me to use and work around problems in the graphics tools used is also gratefully acknowledged.

Support from the departments of Electrical and Computer Engineering and Graduate Studies is gratefully acknowledged.

This research was supported by NSERC.

List of Figures

Figure 1.1.1 Example Stochastic Signal 3
Figure 2.1.1 Bipolar and Unipolar Multipliers 4
Figure 2.2.1 Squaring Circuits 5
Figure 2.3.1 Adder 5
Figure 2.4.1 Generic Linear State Machine State Transition Diagram 7
Figure 2.4.2 Stanh State Transition Diagram and Symbol 9
Figure 2.4.3 Plot of Stanh(4, x) and tanh(2x) 9
Figure 2.4.4 Plot of Stanh(16, x) and tanh(8x) 10
Figure 2.4.5 Linear Gain State Transition Diagram and Symbol 11
Figure 2.4.6 Linear Function Gain v.s. Control Probability k 11
Figure 2.4.7 Examples of the Linear Gain Function 12
Figure 2.4.8 Exponentiation State Transition Diagram and Symbol 13
Figure 2.4.9 Examples of the Exponentiation Function 13
Figure 2.5.1 Unipolar Divider and its Symbol 14
Figure 2.5.2 Bipolar Divider and its Symbol 15
Figure 2.5.3 Comparison of Unipolar Divider with and without Stepped Velocity 16
Figure 2.6.1 Digital to Probability Converter Circuit and its Symbol 17
Figure 2.7.1 Probability Estimator (PDC) and its Symbol 18
Figure 2.9.1 Autocorrelation Function of a Bernoulli Sequence 21
Figure 2.9.2 Autocorrelation of Sexp Function and Sum 22
Figure 3.1.1 MICR E-13B Font with Noise 23
Figure 3.1.2 ANN Structure for the Sample Problem 25
Figure 3.1.3 SCL Layer Interpretation Transformed back into Input Space 27
Figure 3.2.1 NMSE, σ, and Noise during Learning of SCL Layer 30
Figure 3.2.2 Conventional SCL Network Layer Learning Results 31
Figure 3.2.3 Progression of NMSE and Pruning in the Output Layer 34
Figure 3.3.1 Stochastic SCL Neuron Circuit 35
Figure 3.3.2 Stochastic SCL Neuron Normalization Circuit 35
Figure 3.3.3 Stochastic Linear Output Layer 36
Figure 3.3.4 Learning in the SCL Neuron Synapse 37
Figure 3.3.5 Learning in the Linear Output Layer Synapse 38
Figure 3.3.6 NMSE, Sigma, and Noise during Learning in the SCL Layer 40
Figure 3.3.7 Stochastic SCL Network Layer Learning Results 41
Figure 3.3.8 Progression of NMSE and Pruning in the Output Layer (Stochastic) 42
Figure 3.3.9 Comparison of Conventional and Stochastic Network Results 43
Figure 3.3.10 Network MSE vs. Noise 44
Figure 3.3.11 Modified Stochastic SCL Neuron 45
Figure 3.3.12 Modified SCL Normalization Layer 46

1. Introduction

It is the intention of this thesis to present techniques for processing of information in a format which is well suited to implementation of neural networks using binary digital hardware. Stochastic pulse streams are a suitable alternative for neural network implementations for a number of reasons:

- Variable precision is available without hardware changes (only signal observation time needs to be modified).
- Only a single wire is needed per logical signal.
- The signal format is robust in the presence of noise/single bit faults.
- The hardware complexity of the computational elements is low, allowing the implementation of very large networks in VLSI.
- Re-timing (pipelining) may be applied easily in order to maximize system clock rate.

Artificial Neural Networks (ANNs) are parallel processing systems that are loosely modeled after the brain. ANNs have capability to learn useful functions as opposed to conventional computing systems which must be explicitly programmed to perform an algorithm which has been developed to perform a particular function. The processing elements in an ANN, called neurons, are interconnected with synapses which have associated weights. When an ANN learns, its weights, which store its "knowledge", are modified. The learning can be either supervised (stimulus patterns are provided along with desired output values), or unsupervised (stimulus patterns provided without any information on the desire output values).

Conventional computing systems use binary radix arithmetic. Circuits work with groups of binary digits (bits) which, together as a group, represent a single number. In the integer format, a set of N bits can represent either I $[0, 2^N-1]$ (unsigned integers) or I $[-2^{N-1}, 2^{N-1}-1]$ (signed integers). Each of the bits in the group is assigned a significance that is two raised to the power of its position in the group, beginning at 2^0 for the least significant bit (LSB). While the binary radix representation is very compact, circuits designed to compute using numbers in this format tend to be large. The variation in the significance associated with each of the bits means that a single bit error, in either communications or computation, could cause anything from a very small corruption to a very large corruption of the number represented.

Stochastic arithmetic uses a representation which involves a sequence of bits. All of the bits have the same significance (unary coding). The value of the represented number is contained in the average (primary statistic) of the bits in the sequence. Higher precision computation is accomplished by observing the sequence of bits for a longer time in order to allow a lower variance estimate of the average to be made. Due to the uniform significance of the bits in the sequence, the significance of a single bit error is small and decreases with higher precision computations. Stochastic arithmetic has been applied to

the implementation of ANNs by others (Dickson et. al., Tomberg et. al., Tomlinson et. al., Van den Bout et. al.)

1.1 Information Coding

The problem of coding information has been handled in digital hardware in many ways, driven by varying requirements. Factors used in the selection of a coding technique include:

- Properties of the information to be encoded, eg.:
  - Number class (Integer/Real/Complex)
  - Dynamic range requirements
  - Resolution requirements
- Signal bandwidth requirements (how much information, how fast)
- Error tolerance, eg.:
  - error detection/correction capability
  - error sensitivity

Traditional digital computers, having high bandwidth and efficiency requirements, use almost exclusively bit parallel binary representations of information, with the number of bits depending on the specific number class and application involved. The dynamic range and resolution are rather rigidly fixed. In order to add error tolerance to the highly error sensitive binary coding, redundancy is systematically added to the information.

Stochastic pulse stream information coding addresses the specific requirements of digital neural network implementations, specifically simplification of interconnectivity and computation hardware, while providing the added benefit of variable precision representation capability.

A binary stochastic pulse stream is a sequence of binary digits, or bits, in which the information is contained in the primary statistic of the bit stream <X>, or the probability of any given bit in the stream being a logic 1. Observation of the information carried on a given signal line is therefore a probabilistic process itself. Observation of a continuous stream of ten 1 bits on a signal X may appear to convey the information that the expected value of X (*X*) is one, but all that can really be said about the observation is, for instance, that $0.9 < X <= 1.0$ with a certain probability. The precision within which a signal value may be estimated, and the certainty of estimate are quantities which vary both with the signal observation time (number of bits observed) and with the higher order statistics of the signal X.

An example signal might look like this:

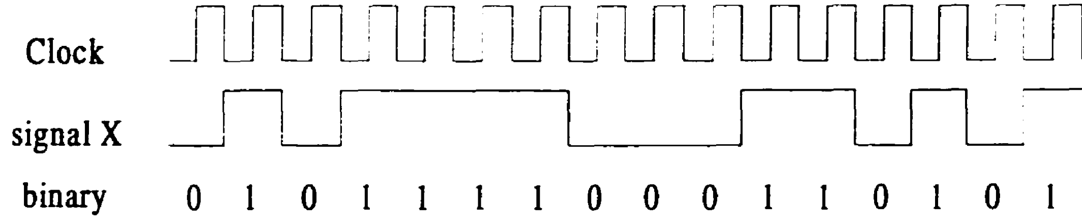


Figure 1.1.1 Example Stochastic Signal

Observation of just the first two bits of this stream implies that $P_{(X=1)}$ is 0.5, however observations of other possible sets of two bits yield all three other possibilities. Take note that this example signal is stationary, ie. it represents a constant value. Obviously the observation of the first two bits of the signal provides some information, but certainly not that $P_{(X=1)}$ is 0.5.

In this thesis stochastic pulse streams are used to carry signals coded in two formats, bipolar and unipolar. These formats are exactly the same as those described by Gaines (1969) as representation I (unipolar) and representation III (single line bipolar). In the unipolar format the information carried in a stochastic stream of bits X is:

$$X = P_{(X=1)} \qquad [1]$$

In the bipolar format:

$$X = 2 \cdot P_{(X=1)} - 1 \qquad [2]$$

2. Processing Element Design and Analysis

This section deals with a number of signal processing elements useful for application to neural network circuits. The analysis of all of the processing elements are predicated on the assumption that the input signal/signals are Bernoulli sequences. This means that the probability of a given bit being a 1 is independent of the values of any previous bits. While all of the processing elements are analyzed with the assumption that their inputs are Bernoulli sequences, note that the elements' processing functions are evaluated only with respect to their outputs primary statistic. The outputs are not, in general, Bernoulli sequences. It is also assumed, in the case of a processing element with multiple inputs, that the inputs are uncorrelated with each other.

2.1 Multiplication

Multiplication of two Bernoulli sequences is actually the one of the simplest operations. In the bipolar stochastic pulse stream coding format an XNOR gate performs a full 4 quadrant multiplication. In the unipolar stochastic pulse stream coding format an AND gate performs a one quadrant multiplication.

Figure 2.1.1 Bipolar and Unipolar Multipliers

For the bipolar configuration:

$$P_{(Y=1)} = P_{(A=1\&B=1)} + P_{(A=0\&B=0)} = P_{(A=1)} \cdot P_{(B=1)} + P_{(A=0)} \cdot P_{(B=0)}$$
$$= P_{(A=1)} \cdot P_{(B=1)} + \overline{P_{(A=1)}} \cdot \overline{P_{(B=1)}} \qquad [3]$$

$$\text{where } \overline{P} \equiv 1 - P$$

$$P_{(A=1)} = \frac{[A+1]}{2} \qquad P_{(B=1)} = \frac{[B+1]}{2}$$

$$\therefore \quad P_{(Y=1)} = \frac{[(A+1)\cdot(B+1)+(1-A)\cdot(1-B)]}{4} = \frac{(A\cdot B+1)}{2}$$

$$\therefore \quad Y = 2 \cdot P_{(Y=1)} - 1 = A \cdot B \qquad [4]$$

In this case the output bit stream Y will be a Bernoulli sequence. In the event that only one of the inputs is not a Bernoulli sequence, [4] still holds. The output will not, however, be a Bernoulli sequence.

For the unipolar configuration:

$$Y = P_{(Y=1)} = P_{(A=1\&B=1)} = P_{(A=1)} \cdot P_{(B=1)} = A \cdot B \qquad [5]$$

2.2 Squaring

Squaring a signal is very closely related to the process of multiplying two signals together. The catch is in the requirement that the two input sequences for a multiplier are

uncorrelated. Clearly attempting to square a signal A by connecting it to both inputs of an XNOR gate will result in an output signal equal to 1 *always*, while connecting it to both inputs of an AND gate will result in an output signal equal to A. This is because the two inputs to the multiplier are correlated with each other. In the case of an input stream which is a Bernoulli sequence making a "copy" of the input sequence which is uncorrelated with it is very simple - delay it by one clock cycle:

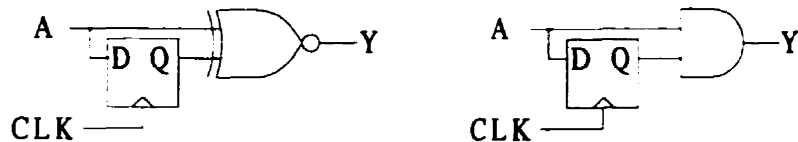


Figure 2.2.1 Squaring Circuits

$$Y = A \cdot A = A^2$$

Note that in cases where the input sequence is not a Bernoulli sequence, but has an autocorrelation function with a limited width spike, creation of an uncorrelated "copy" may be accomplished by delaying the sequence with a shift register containing one stage for every two bits of width in the autocorrelation spike.

2.3 Addition and Subtraction

Addition and subtraction are very different operations to multiplication and squaring in that they are not closed operations on the interval [-1, 1] for bipolar signals or [0, 1] for unipolar signals. As such, it is not possible to perform them exactly, independent of a scaling operation. Addition with scaling is very simple to perform. A multiplexer which randomly selects a given input I_i with some probability S_i such that $\sum_{\forall i} S_i = 1$ will generate an output with a probability which is a scaled sum of the input probabilities:

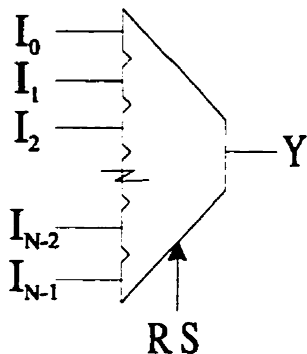


Figure 2.3.1 Adder

yields $$P_{(Y=1)} = \sum_{\forall i} S_i \cdot P_{(I_i=1)} \qquad [6]$$

and for bipolar signals,

$$\because Y = 2 \cdot P_{(Y=1)} - 1 = 2 \cdot \left[\sum_{\forall i} S_i \cdot (I_i + 1) \frac{1}{2} \right] - 1 = \sum_{\forall i} S_i \cdot I_i \qquad [7]$$

If a bipolar input is required to be subtracted, as opposed to added, then it can be multiplied by -1 before it enters the multiplexer (simple logic inversion). Generation of an appropriate random selection is relatively trivial if the number of inputs to the adder is a power of two. If the number of inputs is not a power of two there are a few design possibilities. One is to increase (pad) the number of inputs by adding in extra input connections to a single 0 signal (noise bit for bipolar 0, logic 0 for unipolar). This option will reduce the output value (decreased scaling constant), leading to sup-optimal performance. The option employed is to generate a random binary variable with the appropriate statistics (P(RS = i) = 1/i) by using a counter with a modulus equal to the number of inputs, driven to either increment or maintain its state each cycle based on a single random noise bit.

2.4 State Machine based Computational Elements

A major goal of this thesis was to develop stochastic computational elements based on relatively simple state machines. Gaines (1969) described the use of an ADDIE (ADaptive Digital Element) for both estimation of the statistics of a stochastic signal and for generation of arbitrary functions. There are similarities between the construction of the function generators presented in this thesis and the ADDIE technique reported by Gaines. Both may be viewed as being based on a saturating counter, that is a counter which will not increment beyond its maximum state value or decrement below its minimum state value. In the ADDIE, however, the state of the counter is controlled in a closed loop fashion. The transition probabilities between the states of the counter are not important. Due to the conversion of the state of the ADDIE into a Bernoulli sequence and its comparison to the input sequence, equilibrium is forced at a counter hyperstate dependent only on the input signals primary statistic. The use of a closed loop system certainly has advantages, but it does require that the output of the counter is converted into a stochastic pulse stream in order to implement the closed loop feedback, a hardware intensive undertaking. The state machines presented in this section do not use closed loop feedback. While specific output functions may be generated through either deterministic or stochastic functions of the counter state (as in Gaines, 1969), no closed loop is present. Certain functions are also generated by changing the counter from a deterministic state machine (albeit with a stochastic control input) to a stochastic state machine. By making the transition probabilities in the state machine functions of both the control inputs and also a set of stochastic variables, many degrees of freedom are added to the functions generated. The generalized form for the state machine to be analyzed in this thesis is illustrated in figure 2.4.1.

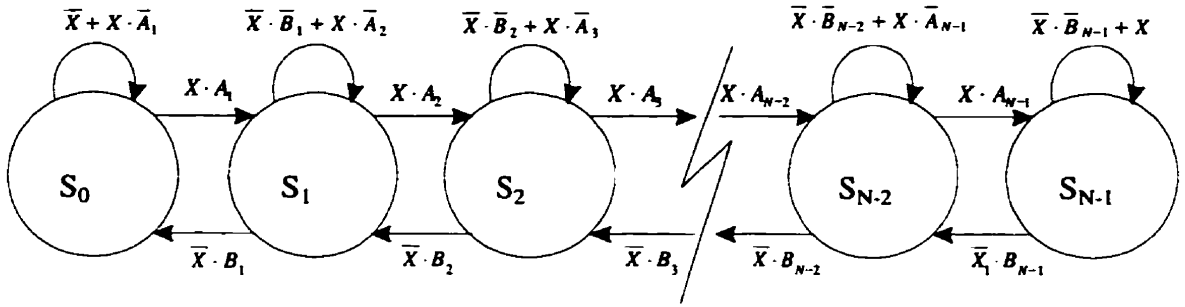


Figure 2.4.1 Generic Linear State Machine State Transition Diagram

The basic form of the state machine is a set of states arranged in a linear form (saturating counter). Transitioning from the first state to the last state must occur through a set of transitions through all of the intermediate states. The state transitions are driven by an input stochastic bit stream X (assumed to be a Bernoulli sequence) and 2N-2 statistically independent control variables A_n, B_n (also assumed to be Bernoulli sequences).

Given that $A_n, B_n \in \Re(0,1]$, we know that the system is ergodic and will have one single stable hyperstate. Solving for this stable hyperstate is relatively simple given a few conclusions which may be drawn from the existence of one single stable hyperstate. These conclusions are that the individual state probabilities in the hyperstate must sum to unity, and that the probability of transitioning from state *i*-1 to state *i* must equal the probability of transitioning from state *i* to state i-1.

Using the notation

$$P_i \equiv P < state = S_i >$$

and assuming for convenience

$$A_0 \equiv B_0 \equiv A_N \equiv B_N \equiv 1$$

$$P_{i-1}\cdot X\cdot A_i = P_i\cdot \overline{X}\cdot B_i \qquad \therefore P_i = \frac{X\cdot A_i}{\overline{X}\cdot B_i}\cdot P_{i-1} \qquad \therefore P_i = \frac{X^i\cdot\prod_{j=0}^{i}A_j}{\overline{X}^i\cdot\prod_{j=0}^{i}B_j}\cdot P_0$$

$$\sum_{i=0}^{N-1}P_i = 1 \qquad \therefore \sum_{i=0}^{N-1}\left[\frac{X^i\cdot\prod_{j=0}^{i}A_j}{\overline{X}^i\cdot\prod_{j=0}^{i}B_j}\cdot P_0\right] = 1 \qquad \therefore P_0 = \left[\sum_{i=0}^{N-1}\left[\frac{X^i\cdot\prod_{j=0}^{i}A_j}{\overline{X}^i\cdot\prod_{j=0}^{i}B_j}\right]\right]^{-1}$$

$$\therefore P_0 = \left[\frac{\overline{X}^{N-1} \cdot \prod_{j=0}^{N} B_j}{\overline{X}^{N-1} \cdot \prod_{j=0}^{N} B_j} \sum_{i=0}^{N-1} \left[\frac{X^i \cdot \prod_{j=0}^{i} A_j}{\overline{X}^i \cdot \prod_{j=0}^{i} B_j} \right] \right]^{-1} \qquad \therefore P_0 = \left[\frac{\overline{X}^{N-1} \cdot \prod_{j=0}^{N} B_j}{\sum_{i=0}^{N-1} \left[X^i \cdot \prod_{j=0}^{i} A_j \cdot \overline{X}^{N-1-i} \cdot \prod_{j=i+1}^{Ni} B_j \right]} \right]$$

$$\therefore P_i = \left[\frac{X^i \cdot \prod_{j=0}^{i} A_j \cdot \overline{X}^{N-1-i} \cdot \prod_{j=i+1}^{N} B_j}{\sum_{i=0}^{N-1} \left[X^i \cdot \prod_{j=0}^{i} A_j \cdot \overline{X}^{N-1-i} \cdot \prod_{j=i+1}^{N} B_j \right]} \right] \qquad [8]$$

Using the probabilities of the individual states, P_i, it is possible to synthesize output functions by forming logic functions of the states, and possibly additional stochastic variables. Due to the mutually exclusive nature of the states, exact sums of their probabilities may be formed through simple use of the logic OR function.

It is important to note the one of the strengths of the proposed state machine based computational elements can also be a significant weakness. Depending on how the output function is formed, there can be significant correlations between the output of the state machine in a given clock cycle and the output in the next clock cycle (ie. Output sequence is not a Bernoulli sequence). While the these correlations do not affect the primary statistic of the output, they do affect the variance of estimates made of the primary statistic. There is also the issue of compatibility between the output of the state machine and other computational elements in the event that further processing is required.

2.4.1 Stanh Function

An approximation to the tanh function, Stanh, with both input and output signals coded as bipolar stochastic signals may be implemented using a state machine of the form introduced in section 2.4. The associated A_n and B_n are all unity and the output Y is a digital 1 whenever the state machine is in the right half of its possible states (N is even).

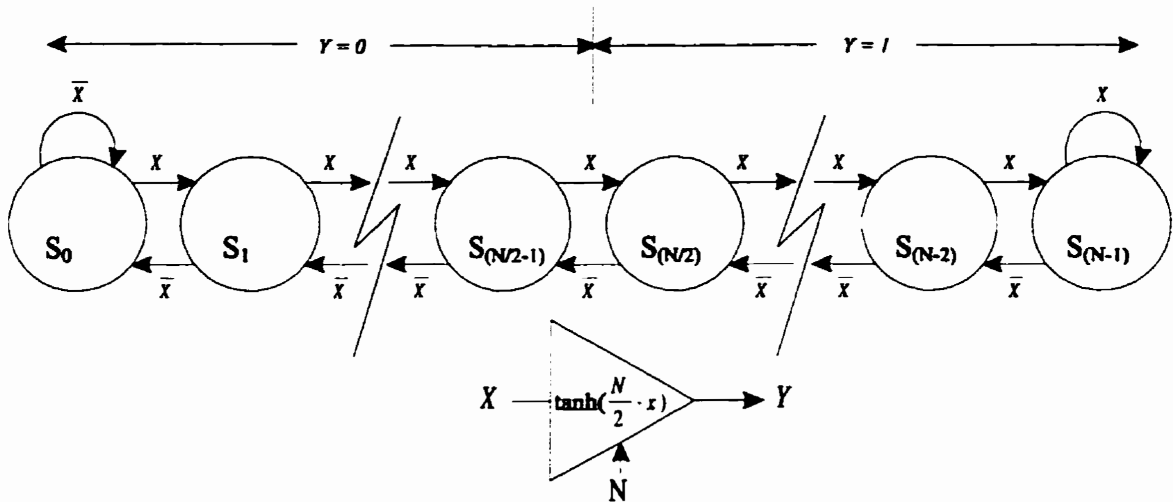


Figure 2.4.2 Stanh State Transition Diagram and Symbol

The total number of states is N
$A_n = 1$ for $n \in [0, N]$
$B_n = 1$ for $n \in [0, N]$
Output Y = 1 for all S_i where $I \in [N/2, N-1]$

In this configuration the approximate transfer function is:

$$\mathrm{Stanh}(N, x) \cong \tanh(xN/2)$$

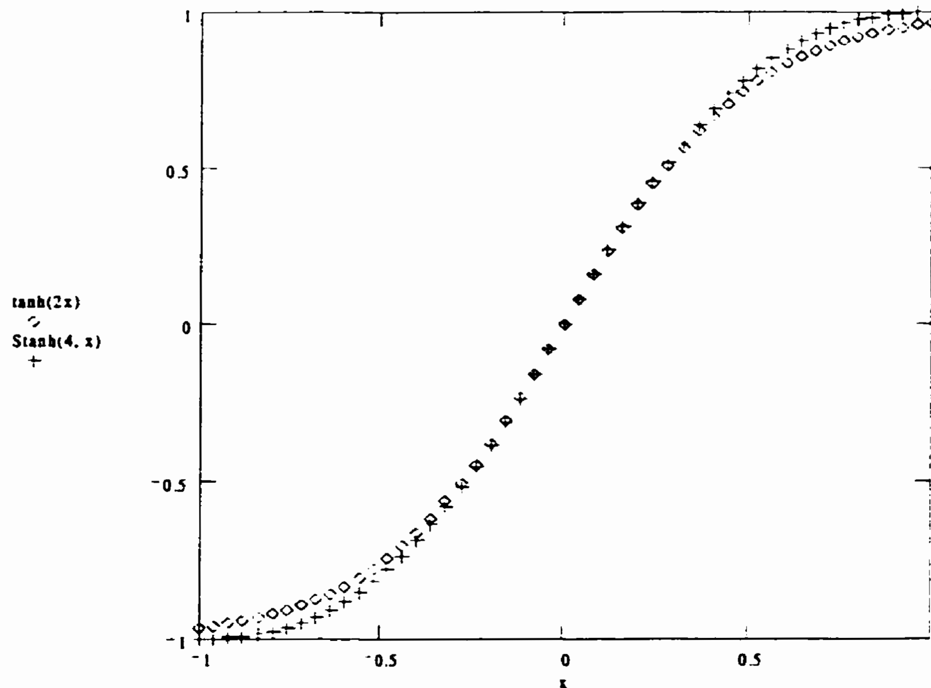


Figure 2.4.3 Plot of Stanh(4, x) and tanh(2x)

For small values of N the approximation to a tanh function is rather poor, but for large values of N the approximation is much better. Figures 2.4.3 and 2.4.4 illustrate a few examples of the transfer function of the Stanh function and their fidelity with the corresponding tanh function.

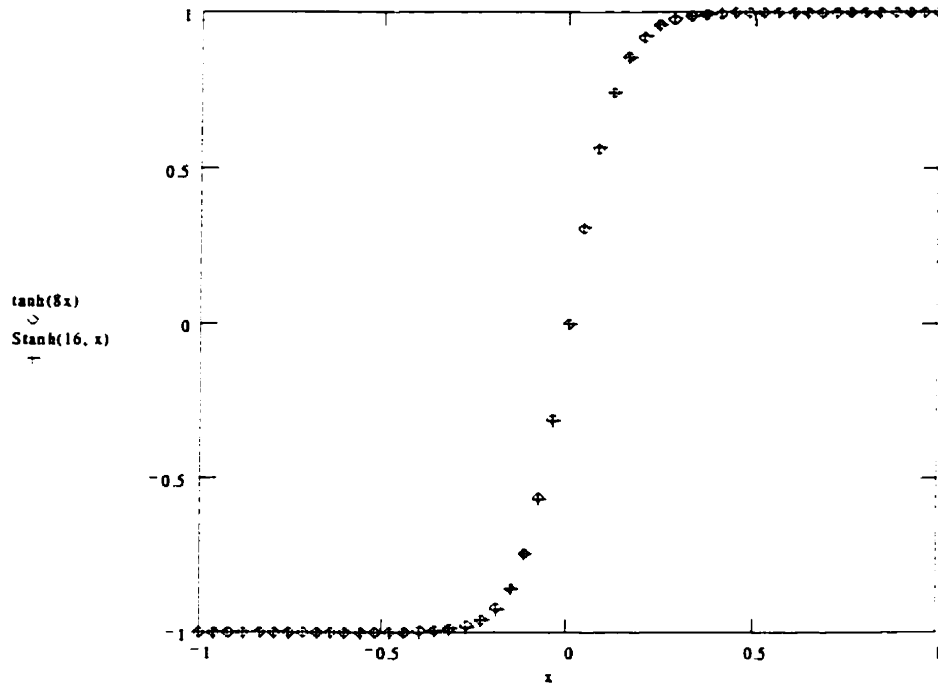


Figure 2.4.4 Plot of Stanh(16, x) and tanh(8x)

2.4.2 Linear Gain Function

An approximation to a linear gain function (with saturation) with both input and output signals coded as bipolar stochastic signals may be implemented using a state machine of the form introduced in section 2.4. While multiplication by a factor whose absolute value is less than unity is quite easy, multiplication by a factor whose absolute value is greater than unity is non-trivial. The specific conditions for this state machine are:

The total number of states is N
$A_n = 1$ for $n \in [0, N/2 + 1]$, $A_n = K$ for $n \in [N/2 + 1, N-1]$
$B_n = 1$ for $n \in [N/2, N]$, $B_n = K$ for $n \in [0, N/2 - 1]$
Where K is a stochastic control variable, $P_K \in (0, 1]$
Output Y = 1 for all S_i where $i \in [N/2, N-1]$

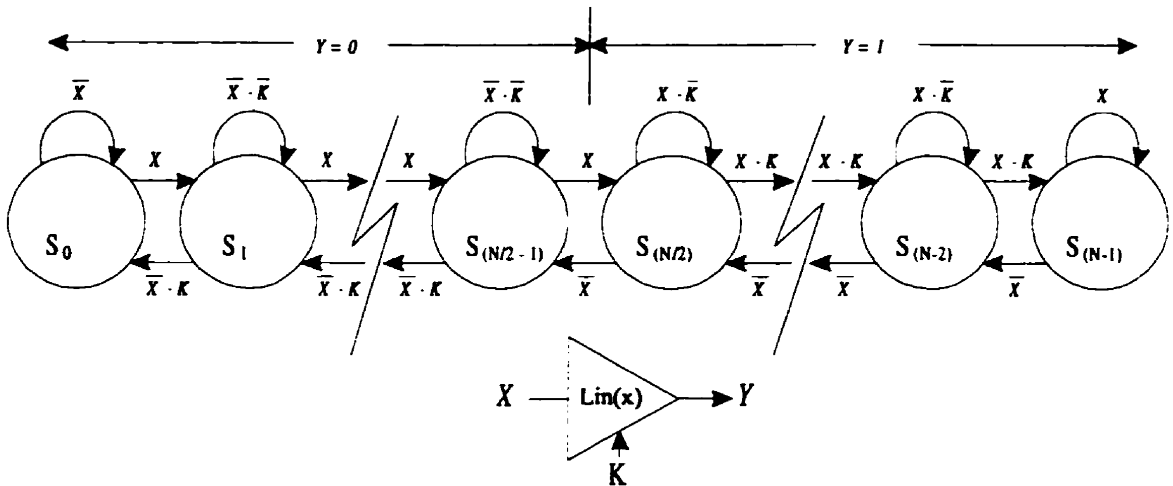


Figure 2.4.5 Linear Gain State Transition Diagram and Symbol

In this configuration the approximate transfer function is:

$$\mathrm{Lin}(N, K, x) \cong x \cdot G(N, K) \text{ for } x \in [-G(N, K)^{-1}, +G(N, K)^{-1}]$$
$$\mathrm{Lin}(N, K, x) \cong -1 \text{ for } x \in [-1, -G(N, K)^{-1}]$$
$$\mathrm{Lin}(N, K, x) \cong 1 \text{ for } x \in [+G(N, K)^{-1}, +1]$$

Where G(N, K) is the gain, a non-linear function of the primary statistic of the stochastic control variable K. The logarithm of G(N, K) is near to a linear function of K. It is also relatively independent of N, the number of states in the state machine, except when K is large. Figure 2.4.6 illustrates a few examples of the variation of G(N, k).

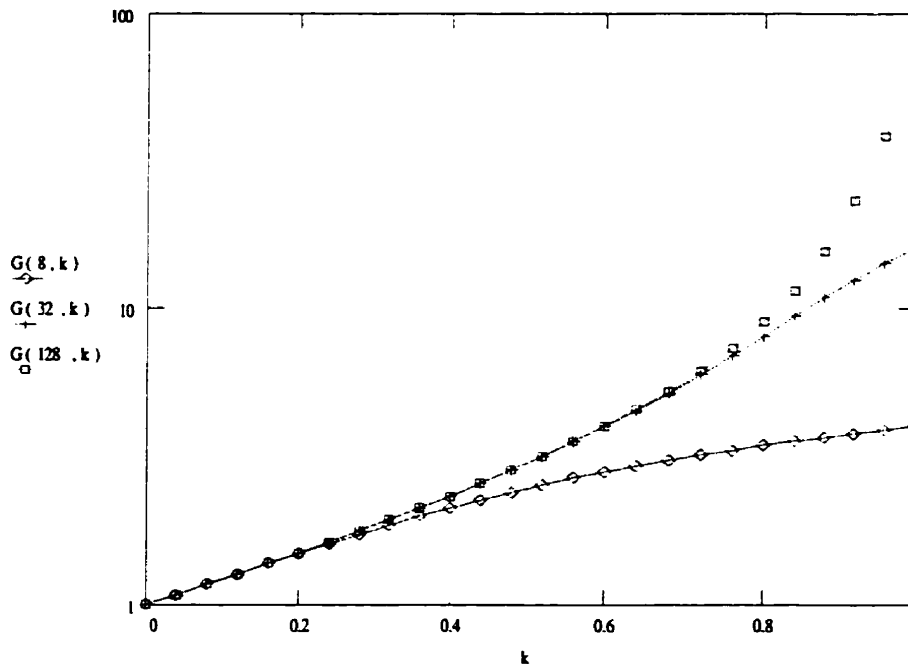


Figure 2.4.6 Linear Function Gain v.s. Control Probability k

Some examples of the linear gain function are plotted in figure 2.4.7. Note that as K increases (higher gain), the number of states N must be increased in order to maintain a reasonably linear transition from Y = -1 to Y = +1. In the limit, when K is one, the linear gain function becomes the same as the tanh function with the gain factor being dependent on the number of states (N).

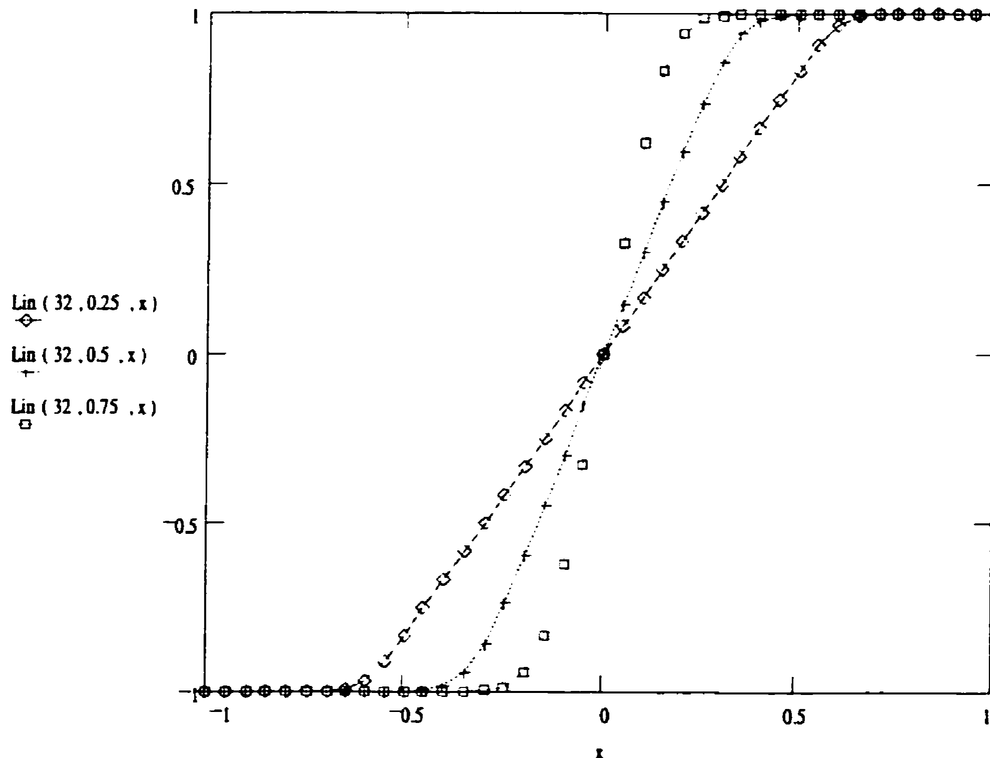


Figure 2.4.7 Examples of the Linear Gain Function

2.4.3 Exponentiation Function

An approximation to an exponentiation function, with the input coded as a bipolar signal and output signal coded as a unipolar signal, may be implemented using a state machine of the form introduced in section 2.4. The range of the function is [0, 1], and thus the approximation is valid only for inputs greater than zero. The specific conditions for this state machine are:

The total number of states is N
$A_n = 1$ for all n, $B_n = 1$ for all n
Output Y = 0 for all S_i $i \in [N\text{-}G, N\text{-}1]$
where G is a gain parameter (integer)
For all other S_i, Y = 1

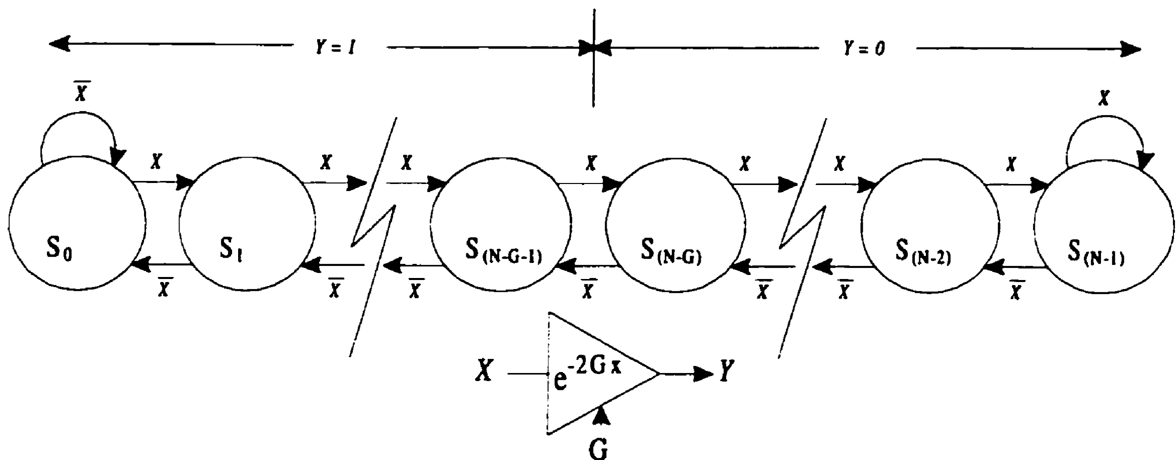


Figure 2.4.8 Exponentiation State Transition Diagram and Symbol

In this configuration the approximate transfer function is:

$$\text{Sexp}(N, G, x) \cong \exp(-2 \cdot G \cdot x) \quad \text{for } x \geq 0$$
$$\text{Sexp}(N, G, x) \cong 1 \quad \text{for } x < 0$$

Figure 2.4.9 illustrates a few examples of the Sexp function. Note that, not surprisingly, the approximation is poorest around x = 0. As the second and third curve illustrate, it is important that the number of states N be significantly greater than the gain parameter G. In fact, Sexp(N, N/4, 0) = 0.75 (ideal would be 1). As N is made larger the approximation improves. Dropping N to 2·G would degrade the function to being a specific case of the Stanh function, although one must account for the difference in output signal interpretation (unipolar vs. bipolar).

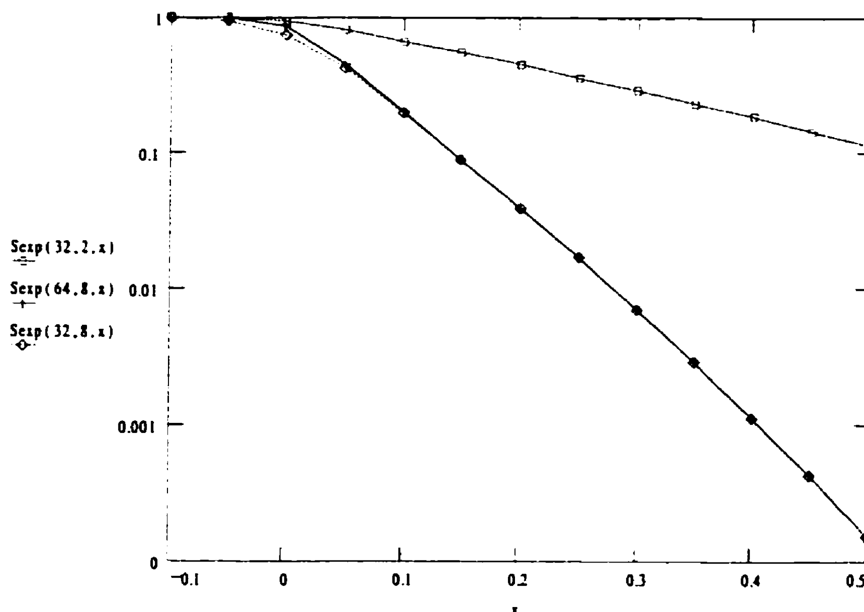


Figure 2.4.9 Examples of the Exponentiation Function

2.5 Division

Division may be accomplished in only an approximate form in the stochastic number representation schemes presented. The only technique available is gradient descent using a saturating counter as an integrator. The reason for the technique only being approximate is that, when the estimated quotient is at the limits for the counter state, the probability of a counter increment cannot be balanced with the probability of a counter decrement. A full analysis of division may be found in Gaines (1969).

2.5.1 Unipolar Division

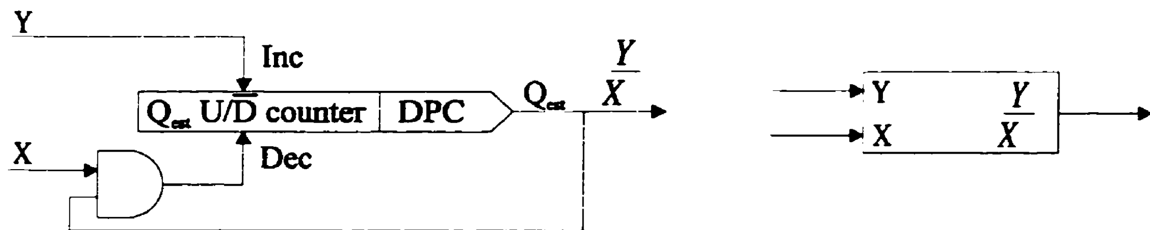


Figure 2.5.1 Unipolar Divider and its Symbol

Unipolar division is performed by performing gradient descent on a error function E:

Defining $x = P_{(X=1)}$, $y = P_{(Y=1)}$ and $q = P_{(Q=1)}$,

$$E = x \cdot q - y \quad [9]$$

$$\frac{\partial E^2}{\partial t} = 2 \cdot E \cdot x \cdot \dot{q} \quad [10]$$

setting

$$\dot{q} = -\alpha \cdot E = -\alpha\left(x \cdot q - y\right) \quad [11]$$

forces

$$\frac{\partial E^2}{\partial t} = -2 \cdot \alpha \cdot E^2 \cdot x < 0 \quad [12]$$

where α is a positive parameter related to the rate at which the counter state (estimated quotient) changes. Given that E^2 is a always positive, and that it is bounded below by 0, [12] indicates that E [9] must be driven to zero.

Implementation of equation [11] may be accomplished by incrementing the counter when Y is a one, and decrementing the counter when both X and Q are ones.

2.5.2 Bipolar Division

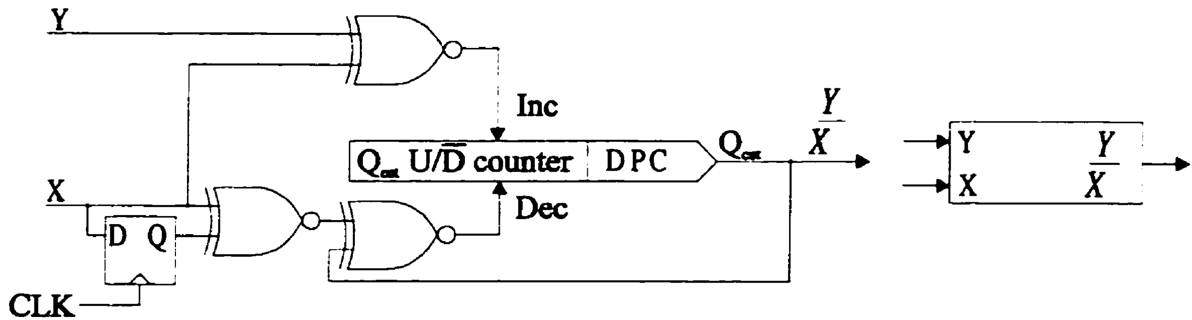


Figure 2.5.2 Bipolar Divider and its Symbol

Bipolar division is performed by performing gradient descent on a error function E:

$$E = x \cdot q - y \qquad [13]$$

$$\frac{\partial E^2}{\partial t} = 2 \cdot E \cdot x \cdot \dot{q} \qquad [14]$$

setting

$$\dot{q} = -\alpha \cdot x \cdot E = -\alpha\left(x^2 \cdot q - x \cdot y\right) \qquad [15]$$

forces

$$\frac{\partial E^2}{\partial t} = -2 \cdot \alpha \cdot E^2 \cdot x^2 < 0 \qquad [16]$$

Implementation of [15] may be observed in the circuit diagram (Figure 2.5.2).

2.5.3 Increased Speed through Stepped Velocity

Computation of a quotient can be time consuming, particularly if the numerator and denominator involved are very small. In this case many clock cycles can occur in which the quotient estimate is neither incremented or decremented because there is effectively no information available to indicate a discrepancy between the required quotient and the estimated quotient. In these cases the "velocity" of convergence may be sequenced through a decreasing sequence of values, starting with a counter step size of N/2 (MSB of

the counter) and then decreasing this by a factor of two at each step. The number of clock cycles at each step is successively increased by a factor of two at each step, allowing the counter to traverse the same "amount of ground" at each step of the process. Figure 2.5.3 illustrates the difference in the results when a signal with a probability of 0.2 is divided by the sum of a set of six signals whose probability adds up to 0.8.

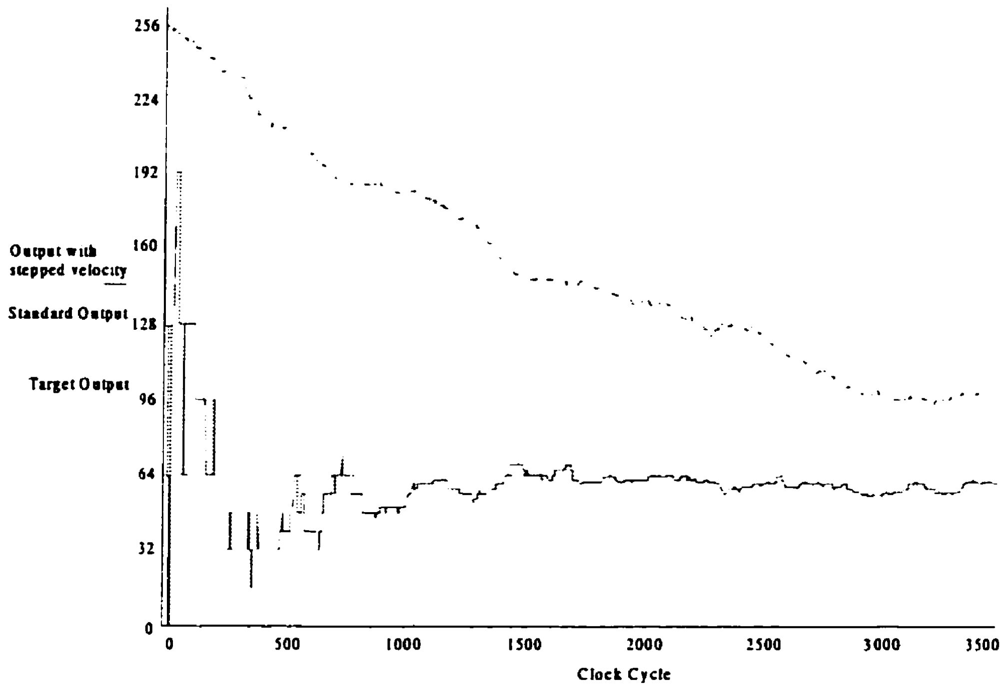


Figure 2.5.3 Comparison of Unipolar Divider with and without Stepped Velocity

2.5.4 Increased Speed through Scaled Processing Time

As previously mentioned, computation of a quotient where the numerator and denominator are very small values can be very slow. This creates a problem where the dynamic range of the numerator and denominator is large. Rather than always allowing the quotient to be computed for a fixed time which is sufficient to achieve acceptable precision with the smallest numerator/denominator which will be encountered, it is much more efficient to scale the processing time as required. This allows the system to spend, on average, much less time calculating quotients. This optimization may or may not be practical in the context of the system in which the computation is taking place. If a number of dividers are operating in parallel with different operands, the potential time savings may be very low. If, however, there is only a single divider in the system, or if all of the dividers in a given layer operate simultaneously with similar operands, the technique can greatly speed computation.

2.6 *Digital to Probability Converter (DPC)*

Figure 2.6.1 illustrates a simple circuit for implementing a binary digital to probability conversion. The technique essentially uses a chain of weighted adders implemented with multiplexers. Each multiplexer select line must be driven by a single noise bit N with a probability of ½. Each adder in the chain generates an output probability that is one half of the sum of the probability from the next earlier stage in the chain and a bit from the digital word to be converted. The result is similar to that of an R-2R ladder digital to analog converter. The transfer function, with a zero terminated chain, is:

$$P_{(y=1)} = \tfrac{1}{2} \cdot (B_{N-1} + \tfrac{1}{2} \cdot (B_{N-2} + \ \dots\ \tfrac{1}{2} \cdot (B_0 + 0)$$
$$= B/(2^N)$$

where N is the number of bits in B (and stages in the chain).

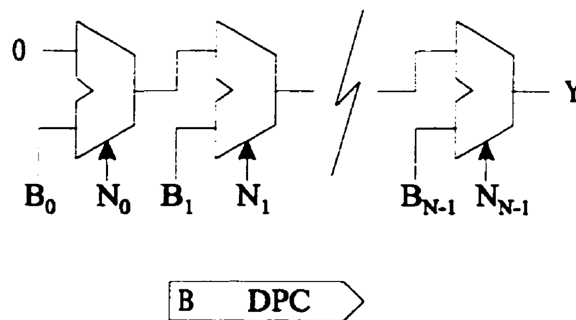


Figure 2.6.1 Digital to Probability Converter Circuit and its Symbol

This circuit benefits from simplicity of structure and, due to the nature of the signals involved, may be pipelined fairly easily in order to maintain very high processing speeds. In the event that the digital value B is a constant, the circuit may be pipelined by adding a flip flop at the output of each multiplexer (or every second one, etc.). In the event that B changes, proper statistics will require extra clock cycles to propagate to the output, based on the number of flip flops in the circuit. Given the number of clock cycles that computations are based on, this delay may not be a significant factor. If the temporary shift in the output statistics cannot be tolerated, an accurate pipelining technique may be used where the bits of B are pipelined as well. Doing this requires flip flops to be added in series with each bit of B in order to maintain the same number of flip flops in the path from the bit to the output as for all other bits. Note that the noise bits need not be pipelined.

This technique is similar to the variable probability generator presented by Gaines (1969). The only real difference in this presentation is the perspective, and its impact on the pipelining technique.

2.7 Probability to Digital Converter (PDC)

We really cannot directly convert an probability to a binary digital number, but we can generate an estimate for the probability. In Gaines (1969) the ADDIE was presented as an optimal estimator for the primary statistic of a signal when the input distribution is unconstrained. Figure 2.7.1 illustrates an ADDIE configured as a probability estimator. Use of this circuit in the context of unipolar or bipolar signals involves both differences in the format of the binary coded state of the ADDIE and in the connections to the saturating counter (integrator) inside the ADDIE which generates the estimate. When estimating unipolar coded signal values, the state of the integrator is interpreted as an unsigned number. When estimating bipolar coded signal values, the state of the integrator is interpreted as an offset binary number. Note that the CE signal is a count enable and the U/D signal is an up/down control signal for the saturating counter.

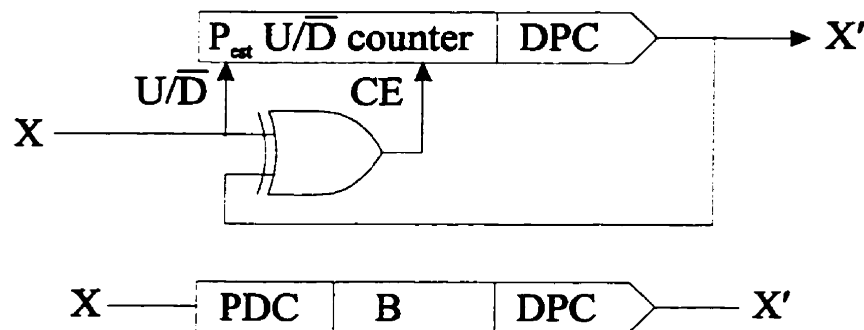


Figure 2.7.1 Probability Estimator (PDC) and its Symbol

The X′ output from the circuit is the stochastic output from the current integrator state. It provides a "re-randomized" output bit stream with the same primary statistic ("signal value") as the input bit stream X. Its use is desirable when the X input is not a Bernoulli sequence and further processing with the signal is desired.

2.8 Generation of Digital Noise

A central element of a digital stochastic processing system is a source of digital noise. Many of the processing elements presented require digital noise bits where the probability of each bit being a one is fixed at ½. Gaines (1969) describes a number of potential sources of digital noise. While noise sources based on quantum mechanical effects may provide noise which meets all of the statistical requirements, generation of noise using these techniques is not very amenable to standard VLSI processes. A source of pseudorandom digital noise, the linear feedback shift register (LFSR), described by Gaines (1969) and by many others, is built upon standard digital components.

The LFSR generates noise which has an autocorrelation function which is a set of repeating delta functions with each delta function separated by the number of bits in the LFSRs sequence length. While this is near ideal, there are a few properties of the LFSR which are sub optimal for use in a stochastic processing system. The bits in the state of the LFSR are all correlated with each other with small shifts (one for nearest neighbor bits in the shift register). While the use of XOR combinations of various bits in the LFSR state may be used to generate bit sequences with any desired shift, use of this technique would require either the centralization of a noise generator or the distribution of multiple LFSR state bits to each stochastic processing element that requires the noise. The routing overhead of such techniques could be significant. Distributing the flip flops of the LFSR amongst the stochastic processing elements would only partially solve the problem. The LFSR requires feedback from the last element in the shift register back to the first element in order to generate a maximal length sequence. Routing this feedback path, along with multiple register neighbor bits to the physical location where a noise bit is required would still take a toll on routing resources.

A more appropriate pseudorandom digital noise generator, based on a cellular automata (CA), is given in Hortensius et. al (1989). One of the desirable properties of this noise generator is that the bit sequences from adjacent bits in the state of the CA are not, in general, shifted from each other by only a single clock. This property likely eliminates the requirement to use sets of the register bits to form a single noise bit. The CA described in Hortesius et. al (1989) is also characterized by the presence of only localized (nearest neighbor) feedback. These characteristics, taken together, make it likely that the individual flip flops that make up the CA register could be placed in a circuit wherever a noise bit is needed. The various flip flops could then be all tied together after placement in a nearest neighbor fashion in order to minimize the burden of routing the nearest neighbor feedback connections. This process is exactly what is commonly done in current full scan test design practices (Fetherston et. al, 1998). After the CA register bits are tied together and their total number is known an appropriate CA rule would then have to be configured into the logic surrounding each CA flip flop in order to generate a maximal length pseudorandom sequence.

In this thesis a single 32 bit CA is used to generate all of the digital noise bits used by the stochastic processing elements.

2.9 Effects of Processing Elements on Higher Order Statistics

The state machine based processing elements presented earlier have outputs which, as mentioned at the beginning of section 2.4, are not in general Bernoulli sequences. While the outputs may not be Bernoulli sequences, they may still be processed by some functions. The multiplier and adder, in particular, do not require all of their inputs to be Bernoulli sequences in order for them to function properly. Ultimately, however, the correlations present in the state machine output sequences will lead to increased variance in any estimates made of the outputs' primary statistic. Note that while estimates of the

primary statistic will have higher variance, the variance of the outputs' higher order statistics will actually be reduced. One approach to reduce the correlations would be to resort to the technique given by Gaines (1969) where the output of the state machine is turned into a Bernoulli sequence. As mentioned earlier, this is a hardware intensive approach. The other approach is use characteristics of the processing typically performed after a state machine based computational element to reduce the effects of the correlations. A typical ANN is formed of layers of neurons. Each neuron performs a function on its total input. Typical functions for a neuron activation function would be a tanh or linear gain function, functions which have been implemented with state machine based computational elements. Between the layers of neurons are synapses which form the total input for the next layer of neurons. Each neuron generates a weighted sum of the outputs from its input neurons by using multipliers and a single adder. If this processing of the activations of the outputs of layer of neurons sufficiently reduces the effects of the neuron output correlations, then it may be justifiable to totally eliminate the conversion of the neuron output to a Bernoulli sequence.

2.9.1 Autocorrelation function of a Bernoulli Sequence

A Bernoulli sequence has, by definition, a sequence of bits whose probability of being a one is independent of any other bit in the sequence. This makes the definition of a Bernoulli sequence's autocorrelation function very simple.

Given that the probability of any given bit in the sequence is a one is p, and $\bar{p} = 1 - p$:

$$\Gamma(k) = \left(1 \text{ if } k = 0, 1 - 2 \cdot p \cdot \bar{p} \text{ if } k \neq 0\right) \qquad [17]$$

An experimental example, generated using a DPC to generate the bit sequence and plotting $\Gamma(k)$ over a sequence of 16K bits is in Figure 2.9.1. The generating probability for the sequence is ½, and therefore the expected value of the autocorrelation function is ½ for k≠0.

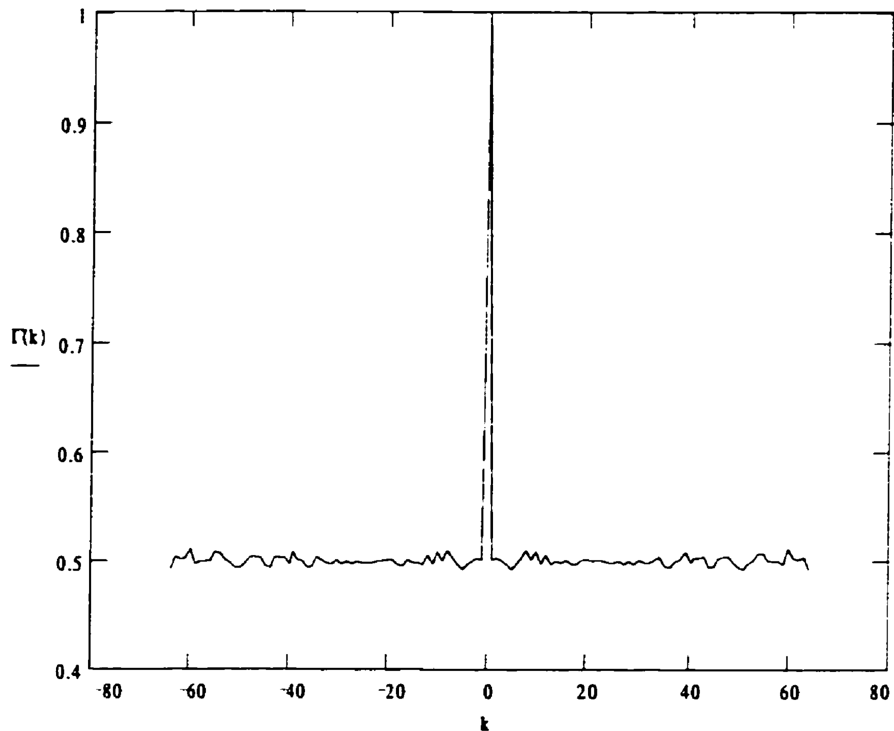


Figure 2.9.1 Autocorrelation Function of a Bernoulli Sequence

2.9.2 Effects of Addition on State Machine Output Sequences

Figure 2.9.2 illustrates both the autocorrelation function of an Sexp function (N=64, G=16, x = 0.0156) and of the output of a scaled stochastic summer which has eight inputs being fed by eight similar Sexp functions. Note that the spike in the autocorrelation of the summer output is one clock wide. The x input of all of the Sexp functions have been chosen to drive the probability of its output to be ½ (maximal variance). As the plot indicates the autocorrelation function of the Sexp output has a very broad spike centered at k=0. Lower gain leads to a much narrower spike. The plot also indicates that, to a very large extent, the generation of the sum of a similar set of Sexp function outputs does mask much of the correlation at the inputs. In this case it appears that the fact that the multiplexer which "throws away" seven out of every eight bit in each input stream is discarding many bits that are correlated with each bit taken. While some research has been done to devise ways to avoid this potential loss of information (Janer et. al. 1996), in this situation the loss of correlated bits is actually, in a sense, desirable. Results for the other state machine based computational elements were qualitatively similar.

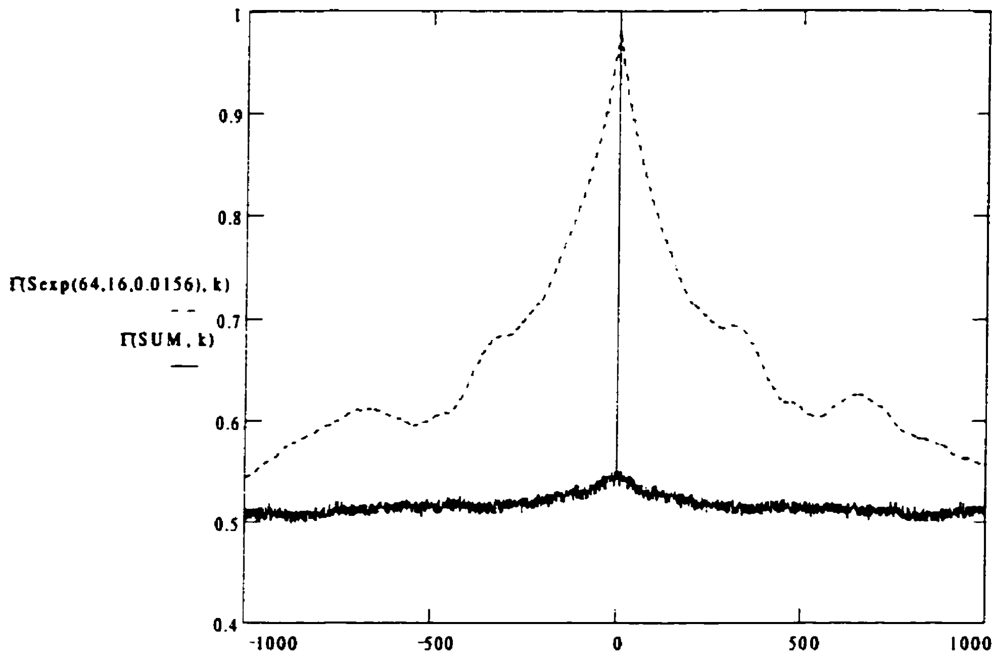


Figure 2.9.2 Autocorrelation of Sexp Function and Sum

3. Application of Stochastic Arithmetic to Soft Competitive Learning

In order to demonstrate the applicability of the computational elements presented, a sample ANN problem was implemented in both conventional (deterministic, floating point) and stochastic arithmetic. The intention is to show that stochastic arithmetic is a viable technique for ANN implementation, with results comparable to that for a conventionally implemented ANN.

3.1 Example Problem

The example problem considered is the optical recognition of MICR (Magnetic Ink Character Recognition) characters. These characters are commonly used on cheques for encoding information such as the identity of the financial institution and the account number. The font, ANSI standard E-13B, has been designed for both magnetic character and human optical recognition. While magnetic recognition is most commonly used by machines, optical recognition is sometimes used in conjunction with magnetic recognition. There are a total of fourteen characters in the E-13B font, ten digits plus four special characters. In the example problem we consider only those characters that are present in applications at the point of sale. In this case only three special characters are present since the fourth special character, "amount", is encoded during processing by the first depositing bank. The characters are in a 7x9 matrix, as illustrated in figure 3.1.1. Each row of figure 3.1.1 represents a specific example of the character set corrupted with a different amount of noise.

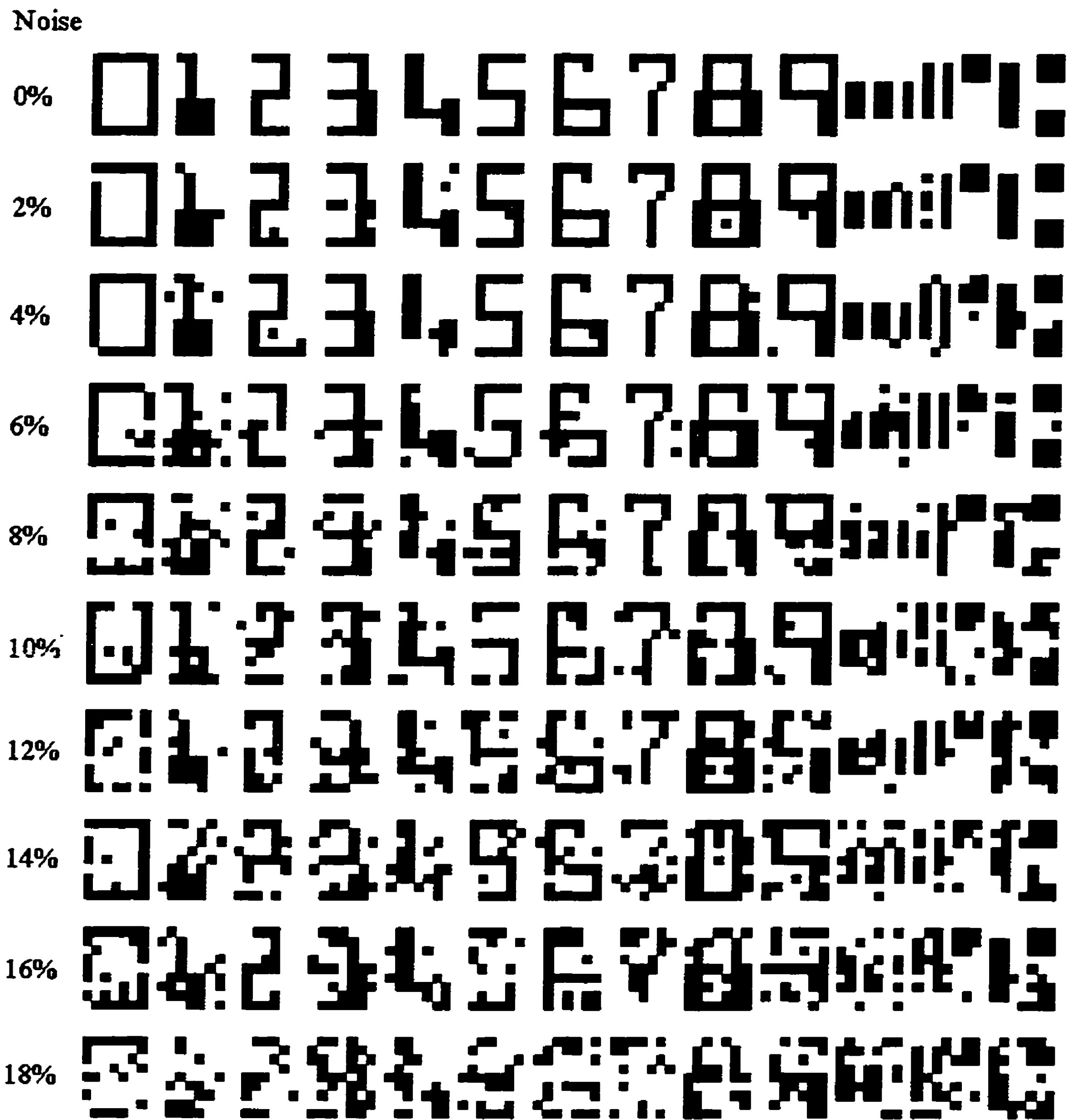


Figure 3.1.1 MICR E-13B Font with Noise

The goal of the ANN is to drive the activation of the output neuron corresponding to the presented character to a maximum level, while holding the other output neuron activations at a minimum level. The problem is know to be easy when the input characters are not corrupted by noise. In this example the performance of the ANN, implemented in both conventional an stochastic arithmetic, is compared when the ANN is presented with noisy input patterns.

The example ANN is implemented in two layers. The first layer is an unsupervised soft competitive learning network. The second layer is a simple set of linear neurons which classify the features detected by the first layer into the corresponding characters using supervised learning (delta rule). In order to simplify the resulting ANN and to accelerate training, the first layer is formed by segmenting the input space (7x9 pixels) into multiple sections of a common dimension. The goal was to train a soft competitive learning (SCL) network on all of the possible patterns in each of these sections and then, in the final network, to use the weights from this network to configure one SCL network per segment of the input space. Initially a partitioning of the input image into nine 3x3 pixels areas (two columns of overlap) was attempted. After some experimentation it was decided to use a partitioning of the 7x9 input character into seven columns of nine pixels each. A major reason for this decision was to allow for a more practical implementation. Commonly the incoming image of the MICR characters would be coming from a line scanner where the document would be passed by the line scanner. This would result in the image coming in one column at a time. Combined with the architectural decision that the SCL networks would share weights, the decision to partition the input character into columns with each column being processed by an identical SCL network would allow one to easily share the hardware implementing the SCL network. The same network would process each column in turn, generating a set of seven output vectors. Each of the output vectors would contain the activations of each of the SCL network neuron outputs. While it is know that sharing the weight vectors amongst the SCL networks will lead to sub-optimal recognition performance, it works well enough to allow comparison of the conventional and stochastic arithmetic implementations. Figure 3.1.2 illustrates the overall structure of the ANN, as implemented, with one SCL network for each column of the input space. The linear output layer is, initially, fully connected to the SCL networks. Also included in the linear output layer is a constant input (not shown in Figure 3.1.2) which allows the neurons to learn a bias term. When the linear output layer is trained its weights are pruned in order to simplify the network.

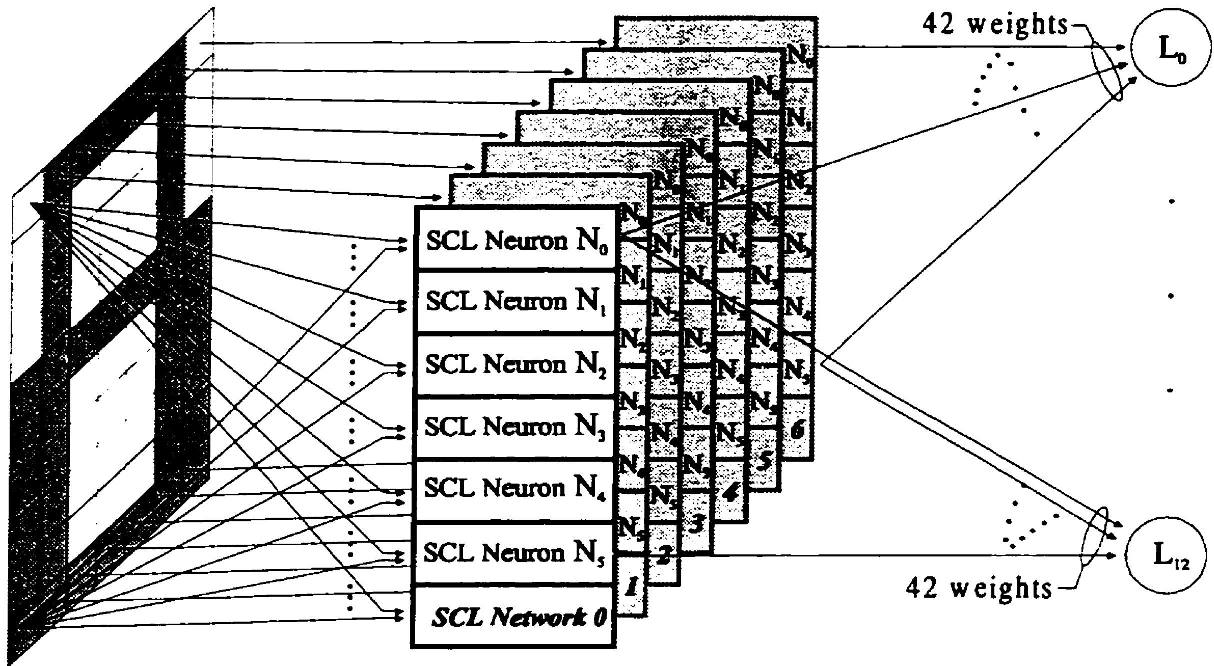


Figure 3.1.2 ANN Structure for the Sample Problem

3.1.1 Soft Competitive Learning Network Layer

The function of the SCL layer of the network is to find a set of prototype vectors to best describe the input space. Each neuron in the SCL layer generates a raw output, Y_i^j for the output of the *i*th neuron in response to the *j*th input pattern. Y_i^j is an estimate of the relative probability that a given input pattern X^j would be generated by a gaussian distribution centered at W_i with a sigma σ. Sigma is the same for all neurons in the SCL layer of the network, although it is not generally constant throughout time as learning progresses. k indexes each of the dimensions in the input pattern and the weight vector for the neuron.

$$Y_i^j = e^{-\frac{(dx_i^j)^2}{2\sigma^2}} \qquad [18]$$

$$(dx_i^j)^2 = \sum_{\forall k} (W_{ik} - X_k^j)^2 \qquad [19]$$

Based on the premise that the SCL neuron weight vectors are an accurate model for the input space, the input vector X^j must have been generated by (or be the responsibility of) one of the SCL neurons in the network. This leads to the generation of a set of normalized outputs for the neurons, N_i^j, which must sum to unity:

$$N_i^j = \frac{Y_i^j}{\sum_{\forall i} Y_i^j} \qquad [20]$$

The designation *Soft* Competitive Learning comes from the use of a learning rule which updates the weight vectors of all neurons (no selection of a "winner") based on the normalized probability that they are "responsible" for the presented input vector j. This learning rule is given by Hertz et. al.(1991):

$$\Delta W_{ik} = \eta N_i^j \left(X_k^j - W_{ik} \right) \qquad [21]$$

Where η is an adjustable learning rate parameter. These weight updates may be applied on an input patteren by pattern basis as indicated above, or in a batch mode:

$$\Delta W_{ik} = \eta \sum_{\forall j} N_i^j \left(X_k^j - W_{ik} \right) \qquad [22]$$

In the work of this thesis, the pattern by pattern update rule is used because it is more practical to implement in a hardware realization.

In order to facilitate observation of the performance of the SCL layer of the network it was convenient to create a special output function. This special output function, not part of the actual network implementation, allowed a natural observation of the SCL network operation. Due to the soft nature of SCL, it generates a distributed representation of the input space. By taking this distributed representation and transforming it back into the input space, we can get an idea of how well the network is performing. The technique is similar to the autoencoder technique for training a back-propagation network described in Haykin (1994). In the autoencoder technique a three layer network (input, hidden, and output) is made to learn the identity function, allowing the input pattern to also serve as the target output pattern. The hidden layer is typically made of fewer neurons than the dimensionality of the input pattern, forcing the hidden layer to generate a compressed representation of the input patterns. In our case we use the concept of transforming the SCL layers representation back into the input space as a means to visually evaluate the performance of the SCL layer. The SCL layer learning, however, is not driven in any way by any discrepancy that may exist between the input pattern and the result of encoding the SCL layer activations back into the input space. The transformed output, O, is a vector with the same dimensionality as the input vector X.

$$O_k^j = \sum_{\forall i} N_i^j W_{ik} \qquad [23]$$

In this thesis the above function is used both to visually observe the qualitative behavior of the SCL layer of the network, and to quantitatively calculate a performance measure, mean square error (MSE):

$$MSE = \sum_{\forall j} \sum_{\forall k} \left(O_k^j - X_k^j \right)^2 \qquad [24]$$

Figure 3.1.3 illustrates an example of the SCL layer interpreting an input pattern and the result of tranforming the interpretation back into the input space. While this is shown based on a single column from the input space, latter in the thesis complete characters will be shown which have gone through the same process, one column at a time.

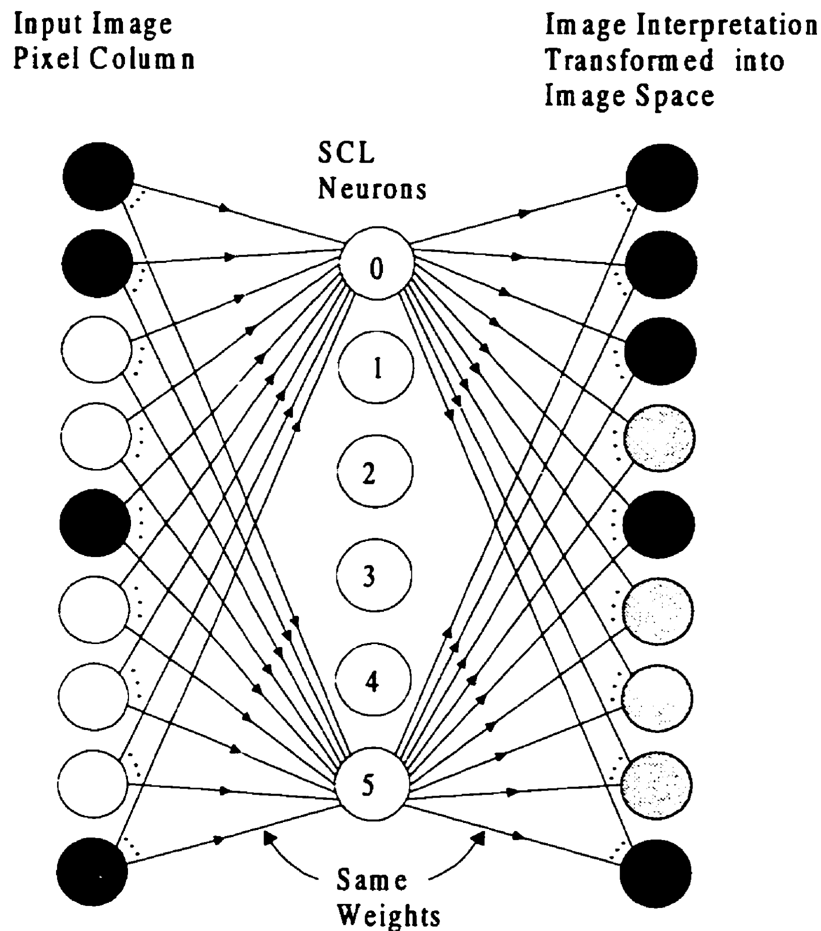


Figure 3.1.3 SCL Layer Interpretation Transformed back into Input Space

3.1.2 Linear Output Network Layer

The final layer of the ANN is a simple linear layer which is connected to the outputs of the SCL sub-networks (all seven of them), as well as to a fixed bias input. The linear

layer is trained by means of the delta rule. There is a single output neuron for each character that we want the network to recognize. A full description of this learning algorithm maybe found in Hertz et al. (1991).

In the linear neuron the output L is a simple sum of the products of each of the components of its weight vector W with the corresponding component of the input vector N.

$$L_i = \sum_l W_{il} \cdot N_l \qquad [25]$$

The input vector N is formed, as mentioned earlier, from the normalized activation vectors from the seven identical SCL sub-networks which analyze the seven columns of the input image. An additional dimension added to this is a single constant term to allow the linear units to learn a bias. For each input image X^j there is a corresponding N^j generated by the SCL layer of the network. For each of the possible input images we want the corresponding linear output neuron to be activated, while all other neurons should not be activated. This corresponds to the definition of a set of target patterns for the network, T, defined as:

$$T_i^j = \left(1 \; if \; i = j, 0 \, otherwise\right) \qquad [26]$$

Where j ranges through the thirteen possible input patterns. Defining an error function which indicates the degree to which the network is accomplishing its mission:

$$E(W) = \frac{1}{2} \cdot \sum_{\forall j} \sum_{\forall i} \left(T_i^j - L_i^j\right)^2 = \frac{1}{2} \cdot \sum_{\forall j} \sum_{\forall i} \left(T_i^j - \sum_{\forall l} \left(W_{il} \cdot N_l^j\right)\right)^2 \qquad [27]$$

Applying gradient descent, we can improve our current weights W by moving each weight component in direction which will decrease E(W):

$$\Delta W_{il} = -\eta \frac{\partial E}{\partial W_{il}} = -\eta \sum_{\forall j} \left(T_i^j - L_i^j\right) N_l^j \qquad [28]$$

η is an adjustable learning rate parameter, as before. The above equation is for a batch update of each weight. Using it involves evaluating the network response to all patterns (j's) while accumulating the appropriate change to the weight. It is much more convenient, though less mathematically rigorous, to apply the updates on a pattern by pattern basis:

$$\Delta W_{il} = -\eta \left(T_i^j - L_i^j\right) N_l^j \qquad [29]$$

The pattern by pattern update technique is the one used in this thesis. During training of the linear output layer, the measure of performance used is MSE (mean squared error). MSE is evaluated as:

$$MSE = \sum_{\forall j} \sum_{\forall i} \left(T_i^j - L_i^j\right)^2 \qquad [30]$$

3.2 Baseline Solution Using Conventional Computation

A C++ program was written to implement the SCL network and allow exploration of the design possibilities. Implementation of the computations in section 3.1 is through the use of standard double precision floating point arithmetic. The program takes in a script file that specifies the number of neurons in the SCL network and its input dimensionality. The script also specifies the offsets into the input character image (7x9) which the SCL network inputs should taken from. The script then controls a simulation of the network allowing the specification of a learning schedule, which includes specification of such things as:

- Learning rate (η) as a function of learning epoch number
- Sigma (σ) as a function of learning epoch number
- Pattern noise to be added as a function of learning epoch number
- Frequency of MSE calculation (number of epochs per MSE calculation)
- Frequency of weight vector observations (number of epochs between captures of weight vectors into an image file)

The script also controls other useful functions:

- Translation of the SCL layer interpretation of a given input character back into the input space (O function as shown in Figure 3.1.3).
- Calculation of MSE while sweeping the value of sigma (σ).
- When to build the second layer (linear output) of the network.

Noise was added to the input patterns as learning progressed in order to expand the training set beyond the 13 input patterns. The noise added was binary in nature, so that each pixel would be left undisturbed or it would be toggled (from black to white or vice-versa) with a certain probability. Note that while the patterns are formed of eight bit pixels, all of the training set patterns are formed of pixels with either the maximum or minimum values.

The conventional arithmetic implementation was very useful for exploring the design space and solutions without having to wait for the much longer execution times of the stochastic arithmetic implementation. Investigations lead to a choice of six SCL neurons per 1x9 column of input pixels. With this number of neurons the network was forced to find suitable compromises for the system weights. While larger numbers of neurons were

able to attain much lower MSE (virtually zero when the number approached the number of unique pixel columns in the input space), the increase in network complexity was deemed excessive. Also, a network with too many neurons would be expected to do a very poor job of generalizing. Without good generalization capabilities a network would perform poorly on novel data.

Figure 3.2.1 illustrates the evolution of sigma, noise, and NMSE as the SCL layer of the network learns. NMSE is simply a normalized version of MSE (normalized to the MSE before learning), used for convenience of scale. The MSE at the start of learning (NMSE = 1) was 15.05. The learning rate was held constant at 0.025. At the termination of learning a sweep of sigma was performed to determine the optimum value of sigma (based on MSE). The optimum value of sigma was found to be 0.747 with an MSE of 4.77.

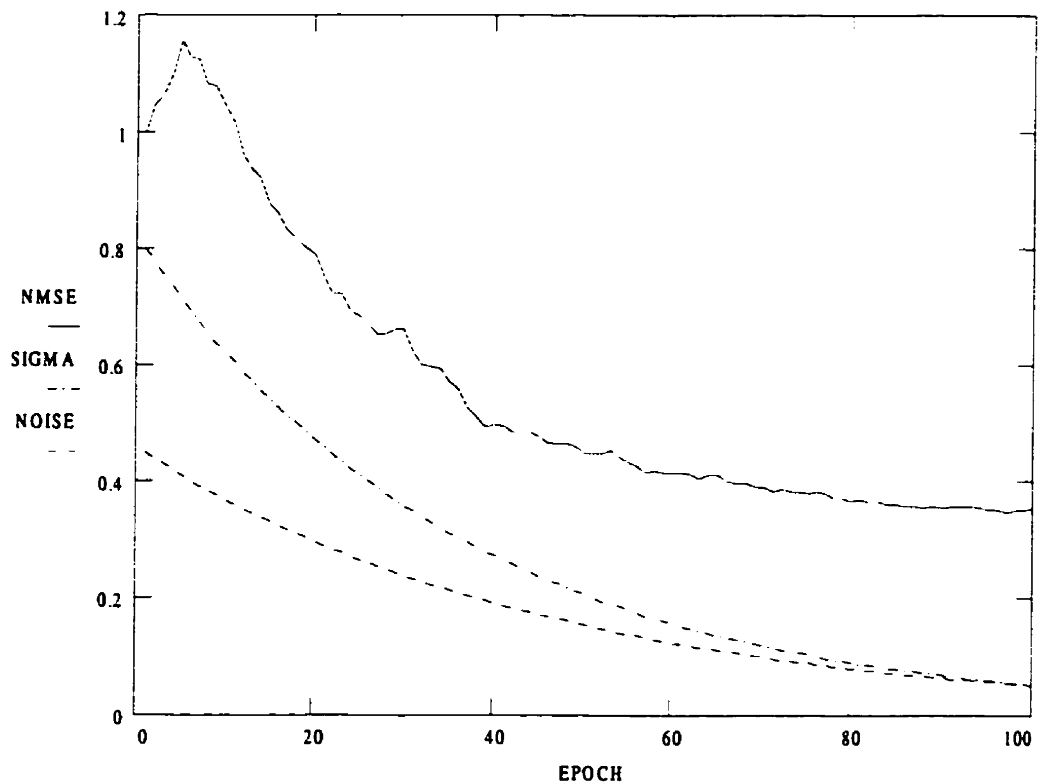


Figure 3.2.1 NMSE, σ, and Noise during Learning of SCL Layer

The trajectories of the weights, along with the results of encoding and decoding each character, as mentioned in section 3.1.1, are illustrated in Figure 3.2.2. The six strip images represent the weight vectors of each SCL neuron. Each column of pixels represents the state of the weights at a particular learning epoch, starting with the initial random weights on the left and ending with the final weights on the right (epoch 100).

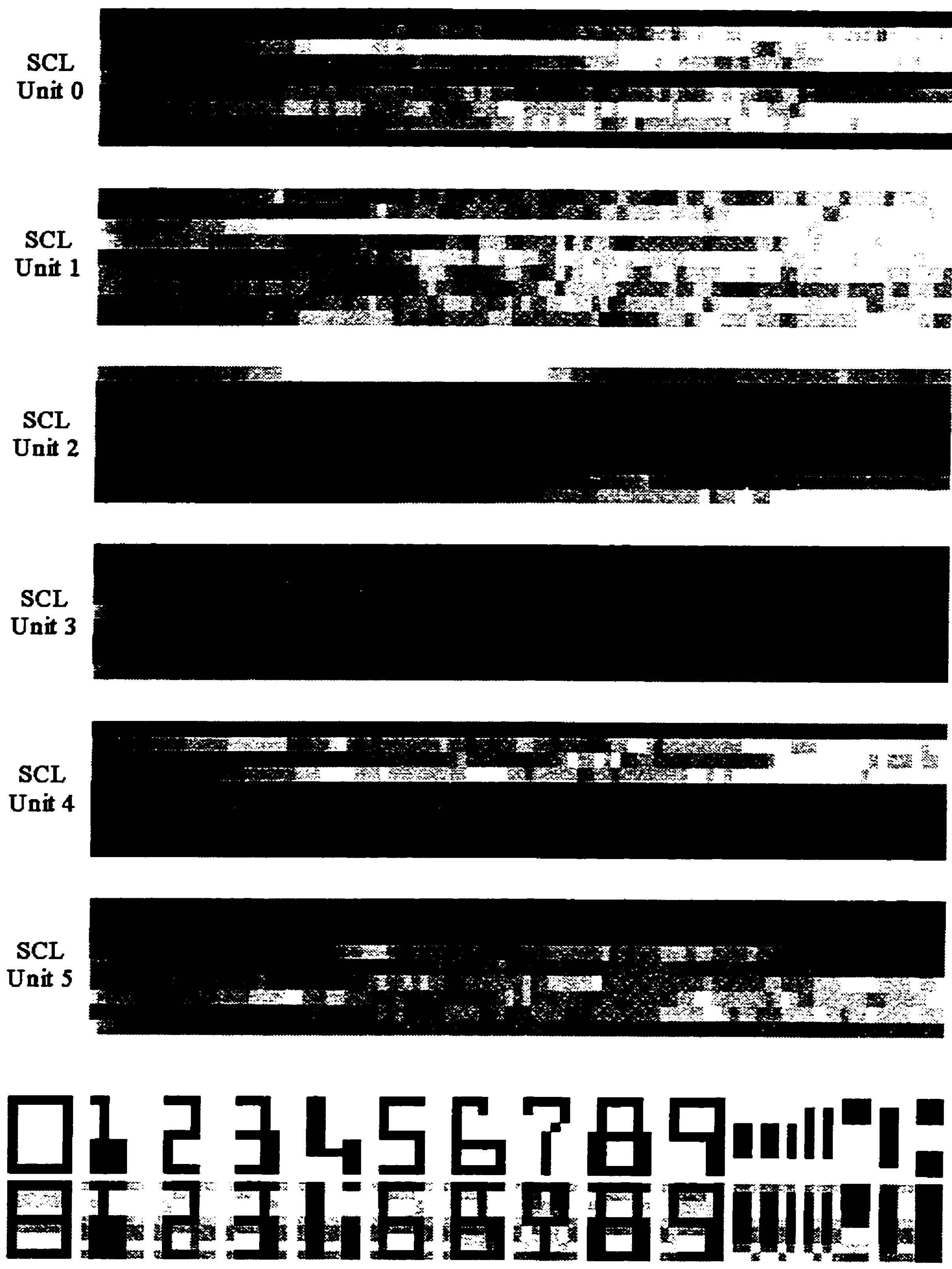


Figure 3.2.2 Conventional SCL Network Layer Learning Results

The original images of the E-13B font characters are provided for reference. The characters below them have been first evaluated by the seven identical SCL networks (one per column), and then reproduced by making a normalized activation weighted sum of the SCL neuron weights.

During learning the effects of large sigma and large amounts of training pattern noise are similar. Large sigma causes all neurons to learn in the direction of all input patterns at a similar rate. If sigma is very large one would expect all of the neurons to effectively learn towards the global average of all of the input patterns. This is one technique to prevent any neurons which start in an unused area of the input space from becoming stranded in a situation where they never learn due to their chronically low activations. Large amounts of noise added to the input patterns causes the input patterns to fill the entire potential input space. If the neurons have reasonable sigma values the expected result is that the neurons will distribute themselves evenly throughout the entire input space, regardless of their initial random starting states. As noise is decreased, the neurons would be expected to home in on areas of the input space surrounding the training patterns. This means that both large sigma values and large amounts of input pattern noise will help to prevent stranding any neurons in an unused region of the input space. Another potential problem is that two or more neurons may jointly claim a common area of the input space. In the absence of any noise in the activation evaluation or learning process, once two neurons have jointly claimed the same area of the input space nothing will ever cause them to be separated. Starting the neurons with random weights and never letting them home in too close to the input pattern average (through high sigma for too long) will lower the probability that more than one neuron will claim the same region of the input space. Note that pushing sigma to very small values causes the soft competitive learning to degrade towards being hard competitive learning (only one "winner" which has non-zero activation). The sigma which optimizes MSE in the SCL layer was found to be reasonably large, allowing the network to generate a distributed representation of the input space. Making sigma small causes the activations of the SCL neurons to become a one of N code, containing less information than the distributed representation of soft competitive learning.

Construction of the second layer of the network was performed using a pruning algorithm which started with a fully connected output layer. This meant that each output neuron had 43 inputs (6x7 + 1 for bias). In the final network this leads to a very large number of weights in the output layer. The SCL layer has only 54 weights (6 neurons x 9 dimensions), while a fully connected output layer has 559 weights (43 inputs x 13 neurons). The pruning algorithm involved the following steps:

1. Train the network until MSE is near minimum attainable
2. Set a threshold parameter THR to a small positive number (e.g. 0.01)
3. Set a minimum number of connections parameter C_MIN to a reasonable lower bound for the number of connections to a given output neuron (e.g. 10).

4. On a neuron by neuron basis, order all of the weights in the output layer from lowest to highest absolute value.
5. Remove all weights in a given neuron, starting with the smallest, which are less than THR. Regardless of weight magnitude, never remove weights when the number of remaining weights is at C_MIN.
6. Train the network again.
7. Repeat steps 4 to 6 until either a target number of weights have been pruned OR step 5 results in no further pruning.
8. If the target number of weights have not yet been pruned, increase THR and then repeat steps 4 to 7.

The pruning algorithm gave good results with little increase in MSE when the output layer connections were pruned to a uniform ten connections per neuron, resulting in 130 weights in the output layer.

It is realized that the pruning algorithm, while very useful when applied to a simulation of an ANN on a general purpose computer, is not very applicable to an actual hardware implementation. Unfortunately time did not allow for further investigations into a more appropriate fixed structure for the output layer.

A simple linear transform was applied to the normalized SCL neuron activations before applying them to the linear layer. This was done to allow the stochastic implementation to closely follow the conventional implementation. In the stochastic implementation we have the unipolar output from the normalization dividers driving into the linear layer which is implemented using bipolar signaling. The linear transform, from unipolar N to bipolar N (BN) is:

$$BN = 2 \cdot N - 1$$

In all of the descriptions of the linear output layer this transform is ignored.
Figure 3.2.3 illustrates the evolution of NMSE and the normalized number of weights pruned as the second layer is pruned. NMSE is, as before, MSE normalized to its own value at the start of learning. MSE is 11.85 before learning starts and 0.545 at the end of pruning and learning. NPRUNE is the number of weights pruned, normalized to the target number to be pruned (429). The value of the weight pruning threshold (THR) is also plotted.

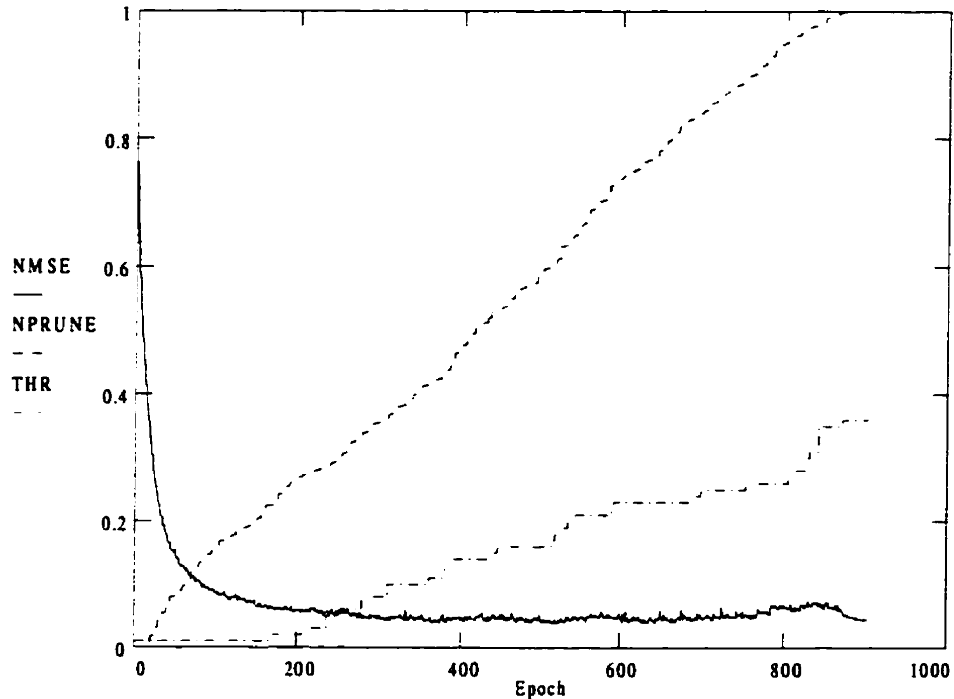


Figure 3.2.3 Progression of NMSE and Pruning in the Output Layer

Figure 3.3.8 (in the next section) illustrates the response of the thirteen output neurons to the sequential presentation of the thirteen characters of the E-13B font.

3.3 Solution Using Stochastic Computation

The C++ program described in section 3.2 was extended to implement the ANN using a clock cycle by cycle behavioral simulation of a stochastic processing circuit designed to implement the same functionality as that described in section 3.1 Implementation of the computations in section 3.1 using stochastic computational elements is illustrated in Figures 3.3.1, 3.3.2, and 3.3.3.

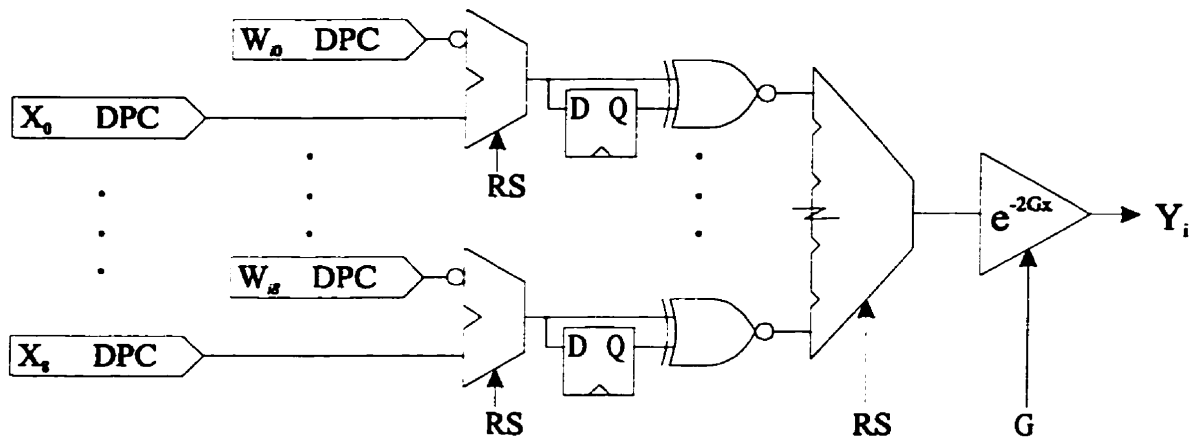


Figure 3.3.1 Stochastic SCL Neuron Circuit

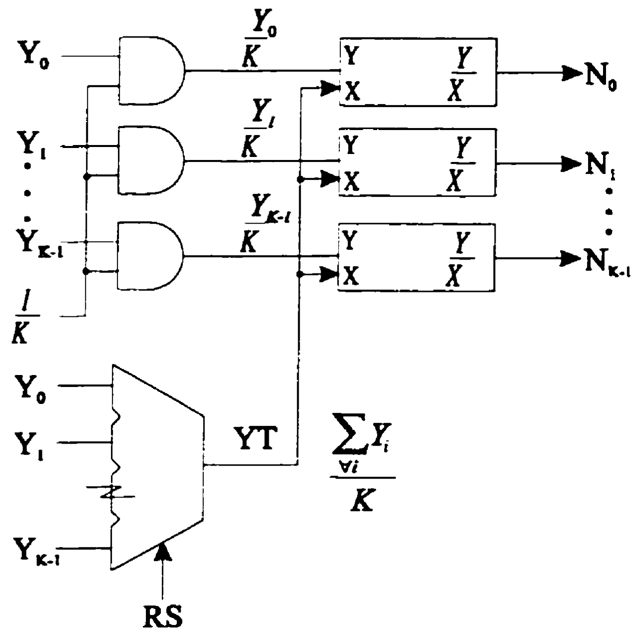


Figure 3.3.2 Stochastic SCL Neuron Normalization Circuit

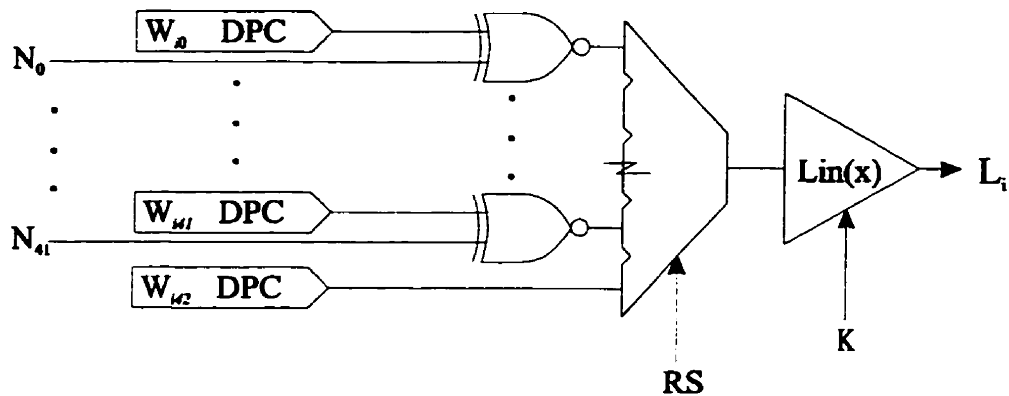


Figure 3.3.3 Stochastic Linear Output Layer

Figure 3.3.1 illustrates the SCL neuron implementation. Note that the bits from the DPCs on each of the input image pixels are used in parallel by all SCL neurons in a given SCL sub-network. The bits from the DPCs on each weight are shared by the corresponding neurons in all of the other SCL sub-networks. As mentioned previously, all image data is in an eight bit input format. Note that while the input training patterns consist of white and black pixels only, the circuitry provides for eight bit gray scale inputs because in a real application the input image is likely to be degraded and not have full contrast. The weights of the network are implemented as eight bit values. The divider quotient registers are configurable as to the number of bits used, the maximum number of states being called DIV_RES. In the learning performed in this example DIV_RES = 2048 (eleven bits). The binary variable G which controls the Sexp function in the SCL neurons is driven globally (to all SCL neurons) with an integer value based on the current desired value of σ in the network.

During learning in the SCL layer a schedule of operations was followed, starting with evaluation of the pattern:

1. The new input training pattern (perturbed with noise as required) is loaded into the input DPCs.
2. The divider states in the normalization layer were set to zero.
3. A digital word G is calculated based on σ and distributed to the Sexp function generators.
4. A control register called SCHEDULE is set to DIV_RES/8.
5. A control register called C_INT is set to 0.
6. A single clock cycle of the entire circuit is simulated.
7. If YT = 1, SCHEDULE is added into C_INT.
8. If C_INT overflows (capacity = DIV_RES - 1):
   - C_INT = 0 (natural in hardware implementation)
   - SCHEDULE is shifted right one bit
9. While SCHEDULE $\neq$ 0, repeat steps 6 to 8.

The increased "velocity" technique was implemented in the normalization dividers through the distribution of the SCHEDULE register value to each of the normalizers. The integrators in each normalizer would increment or decrement by a delta equal to SCHEDULE. When the evaluation is complete SCHEDULE = 0 and the normalized SCL neuron outputs are locked into the divider integrator registers. The technique of scaling the evaluation time to compensate for small divisors is implemented through step 7 above. YT is the divisor and step 7 causes the number of clock cycles in the evaluation to be proportional to 1/YT.

During the evaluation, the counters which hold the weight values are prevented from changing states (LEARN = 0). After evaluation, the counters are enabled (LEARN = 1) and a number of clock cycles equal to 256 times the LRATE parameter is run. See Figure 3.3.4 for some details on control of the weight counters during learning. As elsewhere, the counters must be of the saturating type. The learning phase is:

1. A single clock cycle of the entire circuit is simulated.
2. For each neuron, if its normalized output N is a one:
   - For each weight, if the output of the subtractor (X-W) is a one, increment the weight
   - For each weight, if the output of the subtractor (X-W) is a zero, decrement the weight

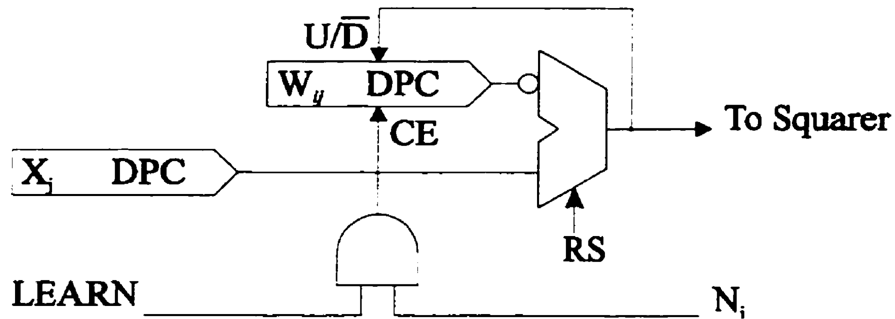


Figure 3.3.4 Learning in the SCL Neuron Synapse

It is fairly easy to see that the algorithm for updating the weight follows [21]:

repeating, for convenience

$$\Delta W_{ik} = \eta N_i^j \left(X_k^j - W_{ik} \right) \qquad [21]$$

$$P_{inc} = P_{(N=1)} \cdot \frac{1}{2}\left(P_{(X=1)} + P_{(W=0)}\right) = N \cdot \frac{1}{2} \cdot \left(1 + \frac{1}{2}(X - W)\right) \qquad [31]$$

and

$$P_{dec} = P_{(N=1)} \cdot \frac{1}{2}\left(P_{(X=0)} + P_{(W=1)}\right) = N \cdot \frac{1}{2} \cdot \left(1 + \frac{1}{2}(W - X)\right) \quad [32]$$

given that the number of clock cycles of learning is LRATE · 256, the expected change in the weight is:

$$< \Delta W >= 256 \cdot LRATE \cdot (P_{inc} - P_{dec}) = 128 \cdot LRATE \cdot N \cdot (X - W) \quad [33]$$

Due to the fact that the resolution of the weights are fixed at eight bits and the weights range logically over [-1, 1), the logical change in W is:

$$< \Delta W >= LRATE \cdot N \cdot (X - W) \quad [34]$$

where

$$LRATE \equiv \eta \quad [35]$$

Performing a similar analysis for the linear output layer synapse:

recall the learning rule:

$$\Delta W_{il} = -\eta\left(T_i^j - L_i^j\right)N_l^j \quad [29]$$

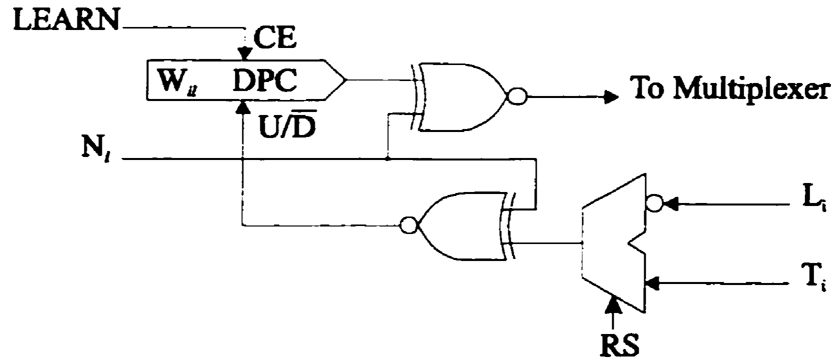


Figure 3.3.5 Learning in the Linear Output Layer Synapse

$$P_{inc} = P_{(N=1)} \cdot P_{(M=1)} + P_{(N=0)} \cdot P_{(M=0)} = \frac{1}{2} + \frac{N}{4}(T - L) \quad [36]$$

and

$$P_{dec} = 1 - P_{inc} = \frac{1}{2} - \frac{N}{4}(T - L) \quad [37]$$

given that the number of clock cycles of learning is LRATE · 256, the expected change in the weight is:

$$< \Delta W > = 256 \cdot LRATE \cdot (P_{inc} - P_{dec}) = 128 \cdot LRATE \cdot N \cdot (T - L) \qquad [38]$$

Due to the fact that the resolution of the weights are fixed at eight bits and the weights range logically over [-1, 1), the logical change in W is:

$$< \Delta W > = LRATE \cdot (T - L) \cdot N \qquad [39]$$

where

$$LRATE \equiv \eta \qquad [40]$$

The circuits implemented as detailed were observed to function correctly. There was, however, a significant issue with the large dynamic range of the signals in the SCL layer of the network. When σ was small (G large) the Y outputs of the SCL neurons were very small. The small total activations in each SCL sub-network caused the dividers in the normalization layer to converge very slowly. The technique of scaling the processing time with the inverse of the divisor in order to minimize overall processing time (see section 2.5.4) did cause the processing time to vary with the evaluation of different inputs with the network spending less time processing when convergence was fast. The technique was, however, insufficient to provide reasonable total processing times due to the dominance of the processing of worst case patterns (very small total activations). The worst case pattern was taking in excess of 1.5 million clock cycles to evaluate. The dynamic range problem was solved through a change to the SCL neurons and the normalization layer which brought evaluation times to a very consistent (and much smaller) number of clocks. Section 3.3.1 details the circuit adjustments made.

Figure 3.3.6 illustrates the evolution of sigma, noise, and NMSE as the SCL layer of the network learns. NMSE is simply a normalized version of MSE (normalized to the MSE before learning), used for convenience of scale. The MSE at the start of learning (NMSE = 1) was 21.25. The learning rate was held constant at 0.025. At the termination of learning a sweep of sigma was performed to determine the optimum value of sigma (based on MSE). The optimum value of sigma was found to be 0.46 with an MSE of 5.26.

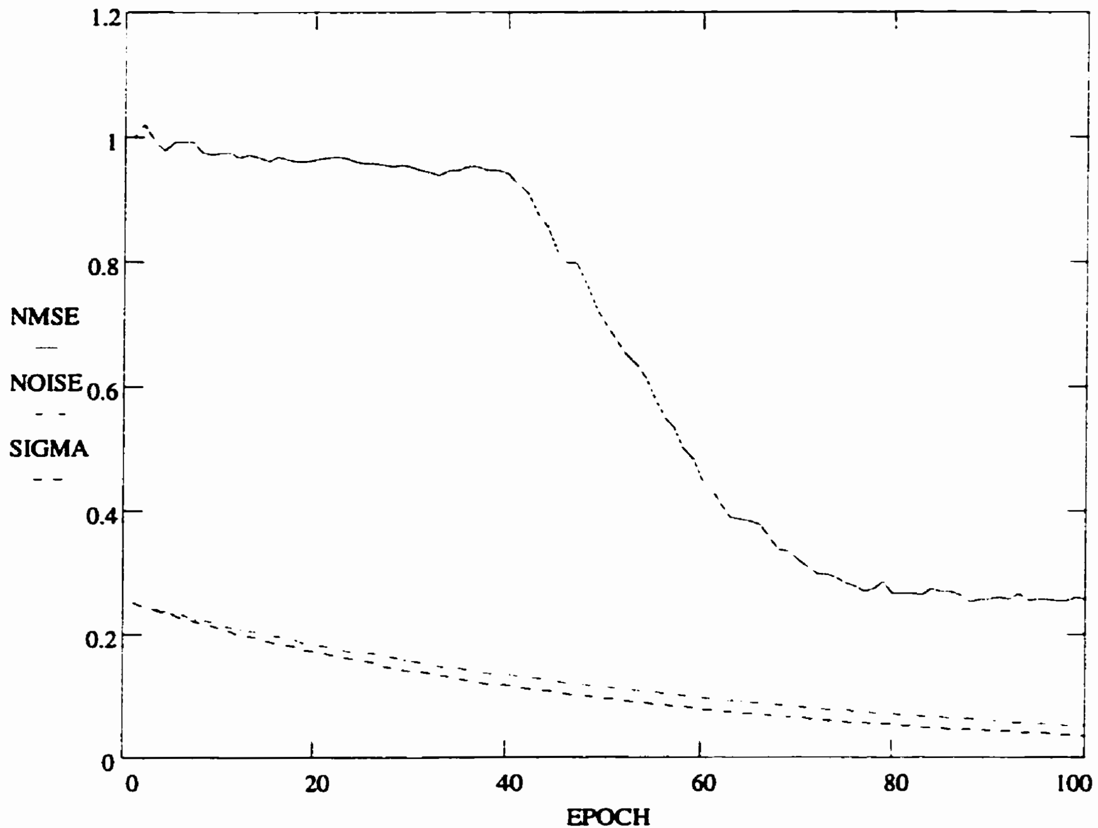


Figure 3.3.6 NMSE, Sigma, and Noise during Learning in the SCL Layer

The trajectories of the weights, along with the results of encoding and decoding each character, as mentioned in section 3.1.1, are illustrated in Figure 3.3.7. The six strip images represent the weight vectors of each SCL neuron. Each column of pixels represents the state of the weights at a particular learning epoch, starting with the initial random weights on the left and ending with the final weights on the right (epoch 100). The original images of the E-13B font characters are provided for reference. The characters below them have been first evaluated by the seven identical SCL networks (one per column), and then reproduced by making a normalized activation weighted sum of the SCL neuron weights.

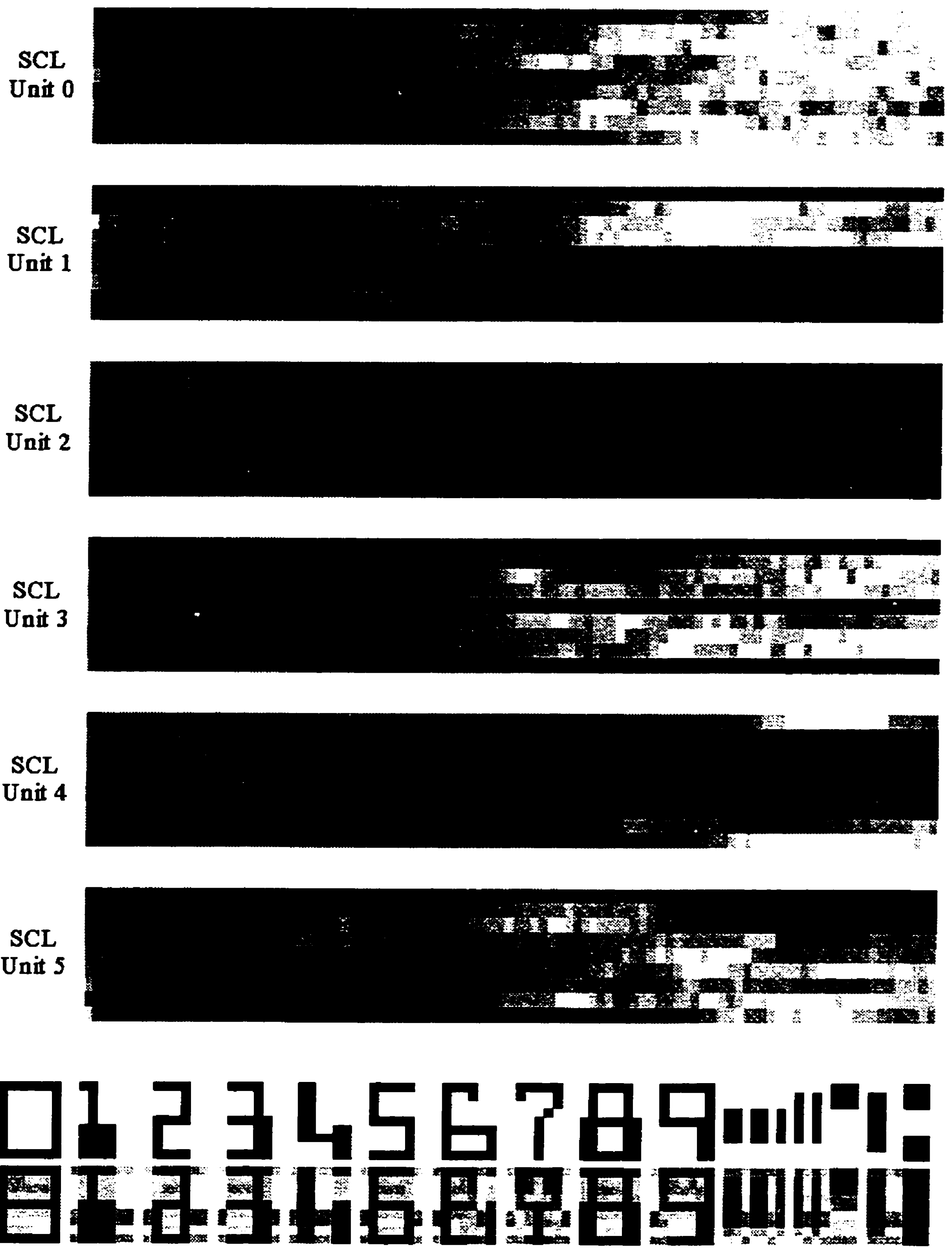


Figure 3.3.7 Stochastic SCL Network Layer Learning Results

Figure 3.3.8 illustrates the evolution of NMSE and the normalized number of weights pruned as the second layer is pruned. NMSE is, as before, MSE normalized to its own value at the start of learning. MSE is 17.46 before learning starts and approximately 0.715 at the end of pruning and learning. The value of the weight pruning threshold (THR) is also plotted. Pruning and learning in the linear output layer required 188 million clock cycles.

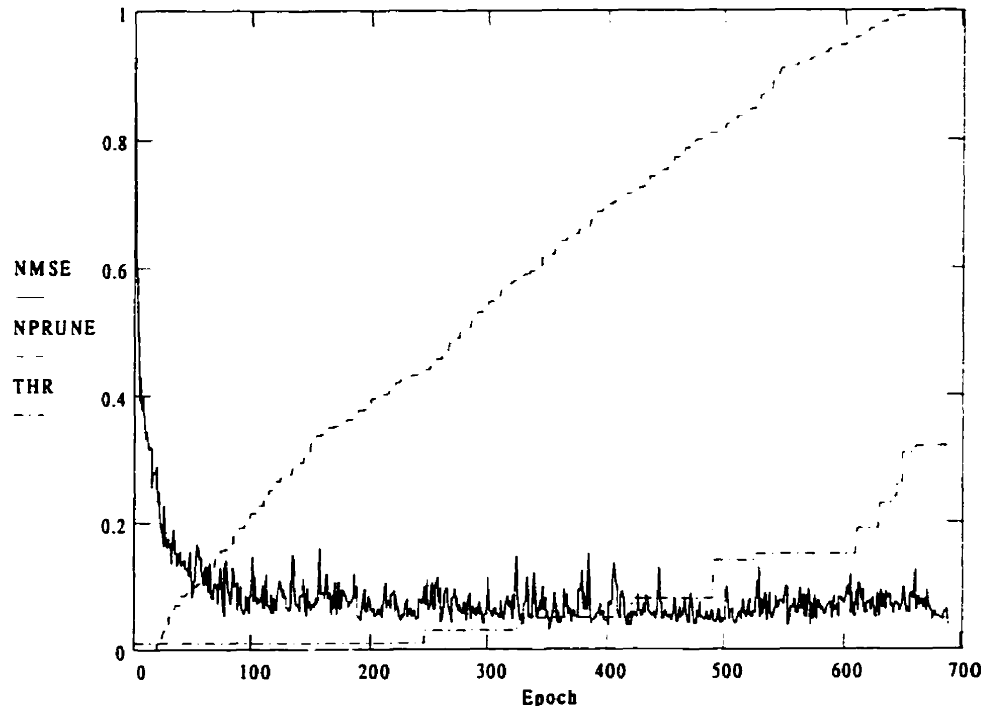


Figure 3.3.8 Progression of NMSE and Pruning in the Output Layer (Stochastic)

Figure 3.3.9 compares the results of having both the deterministic and stochastic networks evaluate the thirteen characters of the E-13B font (without noise) sequentially. Each trace represents the output of one of the linear output neurons. The straight line segments correspond to the outputs of the deterministic network which are constant during the entire presentation of each input pattern. The jagged lines are, of course, the outputs of the stochastic linear output neurons. The large spikes on the stochastic outputs typically occur at the beginning of each pattern evaluation when the gradient descent velocity is very high. Note that for the deterministic network all learning (SCL and linear layers) and pruning (linear layer only) were performed deterministically. For the stochastic network all learning and pruning was performed using stochastic arithmetic. No systematic comparison between pruning results for the deterministic and stochastic networks, in terms of the specific connections retained, was performed.

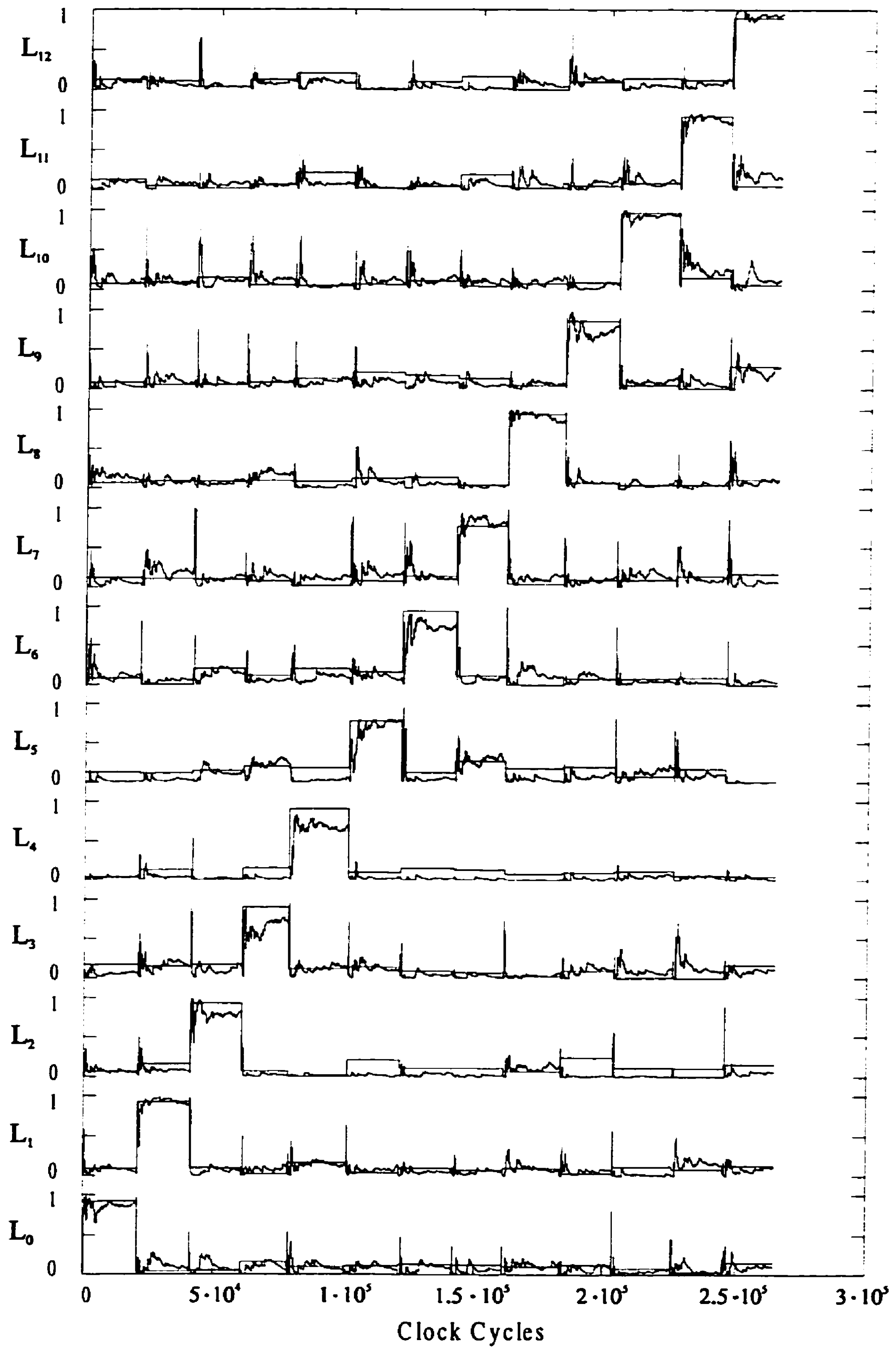


Figure 3.3.9 Comparison of Conventional and Stochastic Network Results

The generalization performance of the networks were evaluated by observing the effects of noisy input patterns on the network MSE. If a network is very good at recognizing corrupted input patterns its MSE will not be much higher than for a set of clean input patterns. MSE verses noise probability was measured by presenting one hundred sets of input characters for each noise probability. Figure 3.3.10 illustrates the performance of the two networks as a function of noise. The stochastic implementation tracks very closely to the deterministic implementation until the noise levels become very high.

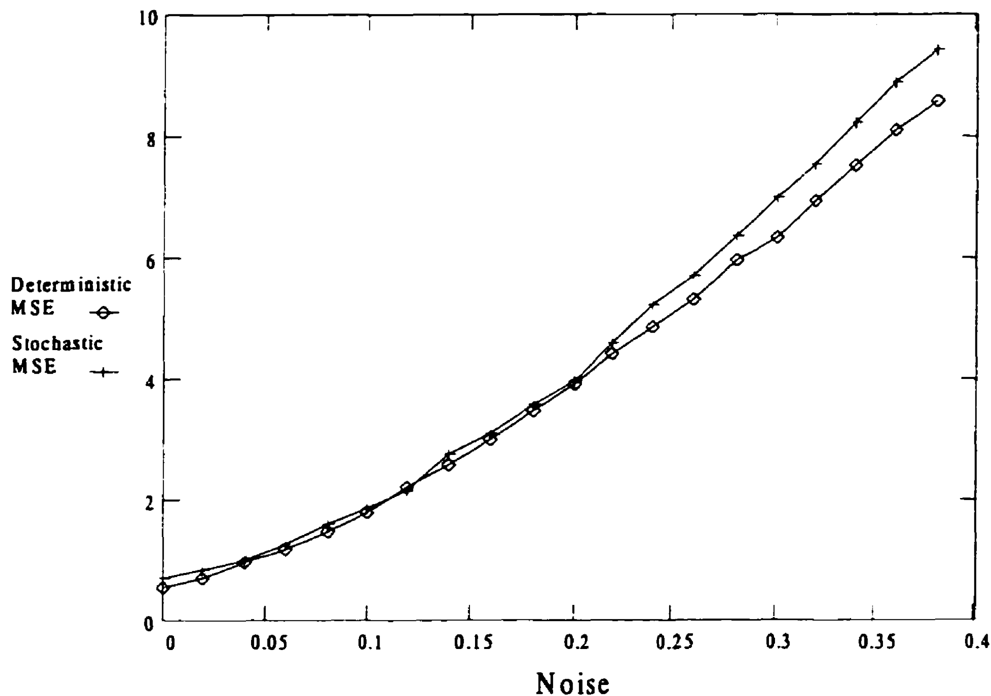


Figure 3.3.10 Network MSE vs. Noise

3.3.1 Problems Encountered

The observation that very low total activations were occurring in the SCL layer and leading to very long computation times led to investigations into a possible solution. It was observed that the total activations could be raised to more tractable magnitudes by performing the normalization division, effectively, before the exponentiation function. Recall:

$$Y_i^j = e^{\frac{(dx_i^j)^2}{2\sigma^2}} \qquad [18]$$

$$(dx_i^j)^2 = \sum_{\forall k}(W_{ik} - X_k^j)^2 \qquad [19]$$

$$N_i^j = \frac{Y_i^j}{\sum_{\forall i} Y_i^j} \quad [20]$$

Equation [20] may be rewritten as:

$$N_i^j = \frac{e^{\frac{(dy_{i})^2}{2\sigma^2}}}{\sum_{\forall i} e^{\frac{(dy_{i})^2}{2\sigma^2}}} = e^{-\left(\frac{(dy_{i})^2}{2\sigma^2} - Z\right)} \quad [41]$$

where

$$Z = \ln\left(\sum_{\forall i} e^{\frac{(dx_i^j)^2}{2\sigma^2}}\right) \quad [42]$$

Figure 3.3.11 illustrates the changes made to the SCL neuron circuitry to allow the subtraction of Z from the input to the Sexp function. Now computing Z, by itself, appears to be a difficult task. However, it is a simple task to compute Z based on gradient descent where we drive the sum of the SCL neuron activations to be equal to one. Figure 3.3.12 shows the corresponding changes made in the SCL normalization layer in order to generate Z.

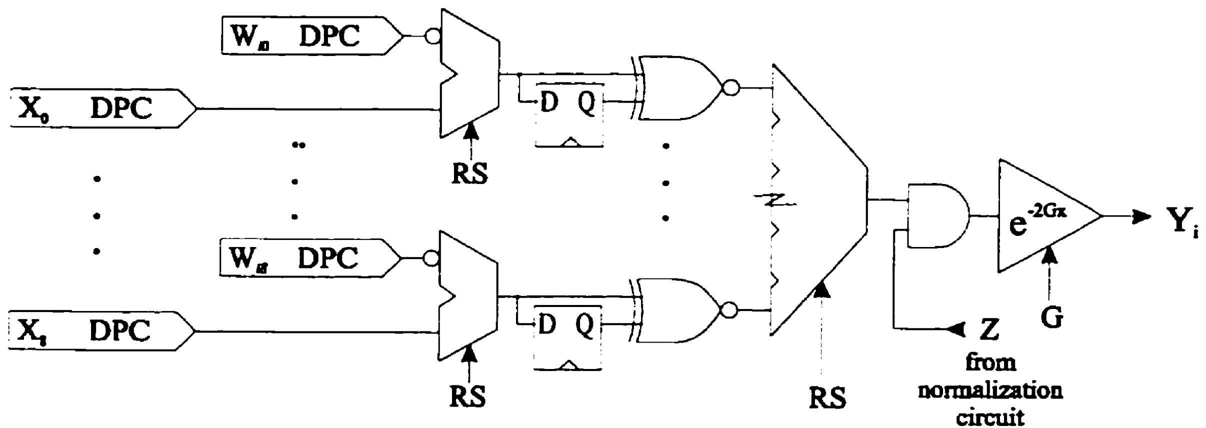


Figure 3.3.11 Modified Stochastic SCL Neuron

The use of the AND gate was to allow the feedback circuit to drive the neuron input towards zero while having as little effect as possible on the gain of the forward path from the synapses.

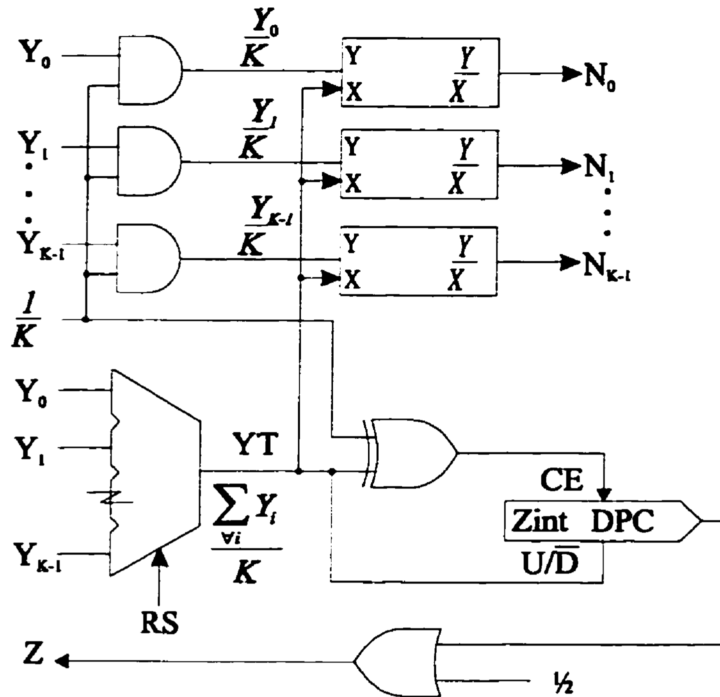


Figure 3.3.12 Modified SCL Normalization Layer

The Zint integrator which is used to generate the Z signal was set to have two more bits in it than the integrators in the divider circuits. While the dividers used the SCHEDULE register as their counter increment/decrement value, the Zint integrator used SCHEDULE/4. This was an important issue due to the high loop gain when sigma is very small. The smallest possible change in the value of Zint must be small enough that the SCL neuron activations do not change too much. The modified input to the exponentiation function is now:

$$\left[(dx)^2\right]' = Z \cdot (dx)^2 - (1 - Z) \quad [43]$$

where Z is a unipolar signal ranging over [1/2, 1].
Typically Z is close to 1 and therefore:

$$\left[(dx)^2\right]' \approx (dx)^2 - (1 - Z) \quad [44]$$

The result of these circuit modifications was that evaluations now take a fairly consistent number of clock cycles to compute. The approximate number of clock cycles is:

$$Nclocks \approx 2 \cdot K \cdot DIV_RES \quad [45]$$

where the factor of two comes from the schedule of increasing clock cycles at lower and lower gradient descent velocities (ending with DIV_RES clocks at the lowest possible velocity), and the factor of K comes from the scaling of processing time based on the magnitude of the dividers denominator. In the particular example ANN implemented [45] represents a reduction in evaluation clock cycles of approximately two orders of magnitude on what were some of the worst case input patterns.

4. Conclusions and Future Work

A number of stochastic computational elements were implemented and demonstrated through use in an example application. In the example application, an ANN for performing optical character recognition on the MICR E-13B font, both the normal computations of the network and the learning were performed using stochastic arithmetic. The results of the computations using stochastic computational elements compared well with conventional double precision floating point computations. The stochastic circuits used were simulated at the clock cycle level with the number of clocks for each computation being tracked.

The learning performance of the stochastic SCL layer, evaluated through the technique of transforming the SCL layer image interpretation back into the image space by multiplying SCL neuron activations by their receptive fields (weights), trained to within 10% of the final squared error of the double precision floating point implementation. Through the use of a feedback technique to reduce the dynamic range of the neuron activations, the clock cycle count for training the stochastic SCL layer was reduced. The learning performance of the stochastic network as a whole, evaluated through squared error on all of the characters in the font, was within a factor of 1.5 of the conventional implementation. Comparisons of the networks' generalization capabilities indicated very close agreement between the stochastic and the conventional implementations.

Computational clock cycle counts were quite reasonable, indicating performance more than an order of magnitude greater for the stochastic computation than the PC used with conventional arithmetic (this conclusion assumes clock frequency parity between the PC and the stochastic computations).

Overall the example ANN application demonstrated the capability to apply stochastic arithmetic and its advantages to the implementation of ANNs:

- very simple circuits and communications resulting in low circuit areas
- simple hardware implementations allowing very high clock rates
- fault tolerance
- capability to trade off computation time and accuracy without hardware changes

The penalties paid for using stochastic arithmetic, increased computation time and variance inherent in estimating the probabilities of stochastic pulse streams, were show to be compensated by other factors in the example ANN. The massive parallelism in the

ANN, supported by the low circuit areas of stochastic computational elements, more than compensated for the large number of clock cycles involved in any given single computation. The variance involved in estimating the primary statistic of a stochastic signal has been shown to not be a significant problem in the context of an ANN. Some ANN applications may even be able to use the variance as an advantage.

There are several opportunities for further work on this topic. While simulations of the stochastic circuits did, in most ways, provide accurate models of a realistic hardware implementation, there are a few deviations from a real hardware implementation. The most notable is in the generation of random digital numbers. The stochastic simulations all used a single 32 bit random number generator based on cellular automata (CA). While this is practical to implement in digital hardware, an actual hardware implementation of a stochastic ANN requires a much larger set of random bits in each clock cycle. Simulations reconciled this issue by cycling the CA multiple times per clock cycle of stochastic processing. A realistic hardware implementation would increase the size of the CA to the point that a sufficient number of random bits would be available for each clock cycle of stochastic processing. A fast gate level simulator has been written for this purpose and was used in early research on the state machine based computational elements. However its use was discontinued due to its slow execution time. Now that a reasonable circuit has been discovered for the example ANN, it would be useful to use the gate level simulator to validate the approach at the gate level.

The topic of generation of random selects over ranges that are not powers of two (for weighted stochastic summation) is an area that deserves further study. The demonstrated ability of the stochastic summer to reduce unwanted correlations generated by the state machine based computational elements was shown with a random select. Correlations in the random selects generated over ranges that are not powers of two could have a deleterious effect on this ability.

Another significant area for future work addresses the generation of new stochastic computational elements. While the current generic state machine was analyzed for the statistics of its hyperstate, a general procedure for choosing the appropriate stochastic parameters and output generation function has not been developed. A general procedure for designing the computational elements would be very useful for designing stochastic processing systems with even greater capabilities.

Finally, the new stochastic computational elements developed here need to be analyzed and compared with alternatives to evaluate performance in terms of circuit area, power consumption and accuracy.

5. Literature Cited

P. Denyer and D. Renshaw, VLSI Signal Processing: A Bit-Serial Approach. Addison-Wesley Publ. Co., Reading, MA. 312 p.

J. A. Dickson, R. D. McLeod, and H. C. Card, "Stochastic arithmetic implementations of neural networks with *in situ* learning," in Proc. Int. Conf. Neural Neworks, San Francisco, CA, 1993, pp. 711-716.

R. S. Fetherston, I. P. Shaik and S. C. Ma, "Testability features of the AMD-K6 Microprocessor," IEEE Design & Test of Computers, pp.64-69, July-September 1989.

B. R. Gaines, "Stochastic Computing Systems," Advances in Information Systems Science, vol. 2, J. F. Tou, Ed. New York: Plenum, 1967, pp. 37-172, ch. 2.

S. Haykin. 1994, Neural Networks A Comprehensive Foundation. Macmillan College Publ. Co., New York, NY. 696p.

J. Hertz, A. Krogh and R.G. Palmer. 1991, Introduction to the Theory of Neural Computation. Addison-Wesley Publ. Co., Redwood City, CA. 327 p.

P. D. Hortensius, R. D. McLeod and H.C. Card, "Parallel Random Number Generations for VLSI Systems Using Cellular Automata," IEEE Transactions on Computers, Vol. 38, pp. 1466-1472, October 1989.

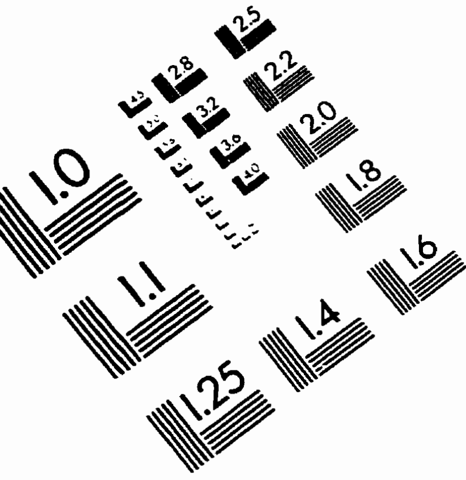
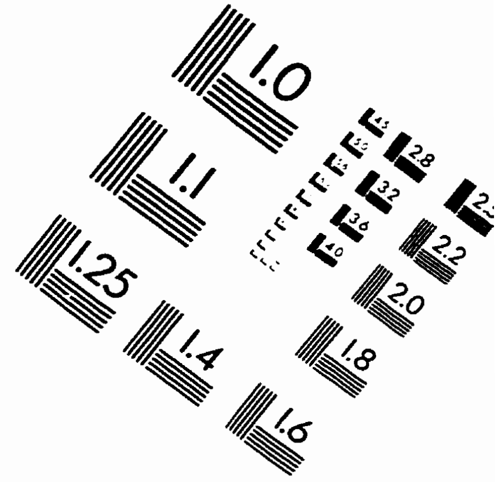
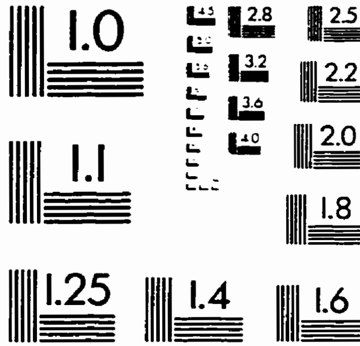
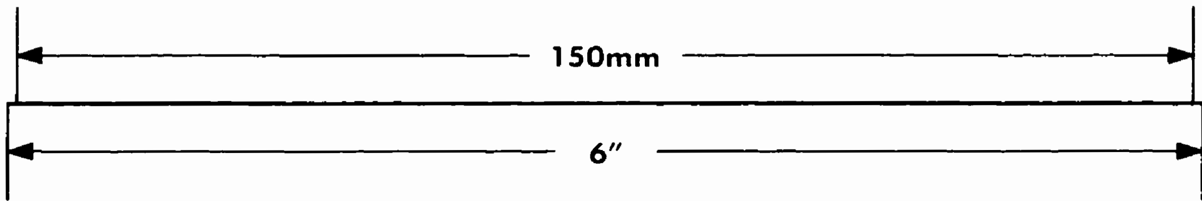
C. L. Janer, J. M. Quero, J. G. Ortega, and L. G. Franquelo. "Fully Parallel Stochastic Computation Architecture", IEEE Transactions on Signal Processing, Vol. 44, pp. 2110-2117, August 1996.

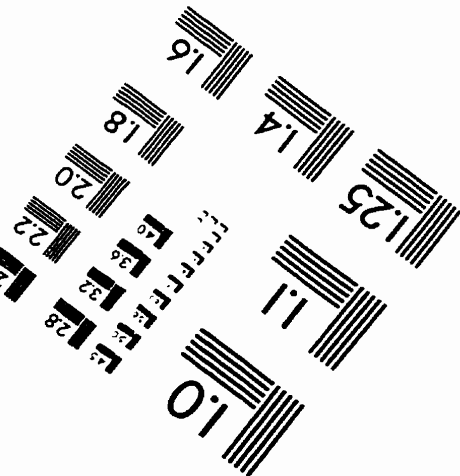
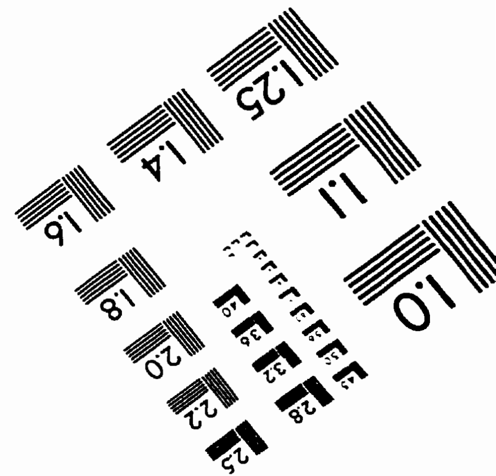
J. E. Tomberg and K. Kaski, "Pulse density modulation technique in VLSI implementation of neural network algorithms," IEEE J. Solid-State Circuits, Vol. 25, pp. 1277-1286, 1990.

M. S. Tomlinson, D. J. Walker, and M. A. Sivilotti, "A digital neural network architecture for VLSI", in Int. Joint Conf. Neural Networks, San Diego, CA, Vol. 2, 1990, pp. 545-550

D. E. Van den Bout and T. K. Miller, III, "A digital architecture employing stochasticism for the simulation of Hopfield neural nets," IEEE Trans. Circuits Syst., Vol. 25, pp. 1277-1286, 1990.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved