CIRCUIT LEVEL SIMULATION BASED TRAINING ALGORITHMS FOR ANALOG
NEURAL NETWORKS

Ismet Bayraktaroglu

Electrical and Electronic Engineering, M.S. Thesis, 1996

Thesis Supervisors: Assoc. Prof. Dr. Sina Balkir, Asst. Prof. Dr. Günhan Dündar

Keywords: Neural Networks, Circuit Simulation, Circuit Partitioning,

Computer Aided Design (CAD)

Neural networks have gained popularity in the last few years due to their success in diverse applications. However, many applications require real time, or very fast operation. This is possible only with dedicated neural network hardware. Due to the inherently parallel nature of neural networks, digital realizations have not been feasible. Analog realizations, on the other hand, have been plagued by non-idealities in circuit behavior. In order to consider such non-idealities circuit level simulations should be performed. The circuit simulator introduced in this thesis makes use of the fact that neural networks with multilayer perceptron architecture consist of many decoupled blocks. The whole circuit can be partitioned into blocks during simulation and each partition can be simulated separately. Although partitioned simulators have appeared in the literature previously, they are applied to neural networks for the first time. The simulator outperforms all conventional simulators in both speed and convergence criteria. This simulator allows us to do training on the circuit level. On-chip training is implemented on the software because the circuit simulator is incorporated into the Madaline Rule III. On-chip training on the simulator has never been done before and has yielded extremely successful results for the examples we have tried. The use of this software will solve most of the problems with analog neural network implementations.

ANALOG YAPAY SÝNÝR AÐLARI ÝÇÝN DEVRE SEVÝYESÝNDE BENZETÝM ÝLE EÐÝTME YÖNTEMLERÝ

Ýsmet Bayraktaroðlu

Elektrik-Elektronik Mühendisliði, Yüksek Lisans Tezi, 1996

Tez Danýþmanlarý: Doç. Dr. Sina Balkýr, Yard. Doç. Dr. Günhan Dündar

Anahtar Kelimeler: Yapay Sinir Aðlarý, Devre Benzetimi, Devre Bölüntüleme,

Bilgisayar Destekli Tasarým

Son yýllarda, bir çok uygulamadaki baþarýlarýndan dolayý, yapay sinir aðlarý sýkça kullanýlýr duruma gelmiþtir. Bir çok uygulama ise gerçek zamanda ya da çok hýzlý çalýþan yapay sinir aðlarý kullanmaktadýr. Bu ise sadece yapar sinir aðlarýna adanmýþ donýným ile gerçekleþtirilebilir. Yapay sinir aðlarýnýn doðasýnda paralel çalýþma olduðu için, sayýsal olarak gerçeklenmeliri iyi sonuçlar vermez. Analog olarak gerçeklenmeleri ise, gerçek devreler ile bunlarýn modelleri arasýndaki ayrýlýklardan dolayý bir çok sorunla karþýlaþmaktadýr. Bu sorunlarý deðerlendirebilmek için devre seviyesinde benzitim yapýlmasý gerekir. Bu tezde geliþtirilen devre benzeticisi, yapay sinir aðlarýnýn yapýsýnýn biri birine benzeyen ve biri birinden baðýmsýz bir çok parçaçýktan oluþmasýndan yararlanmaktadýr. Devre benzetimi sýrasýnda, yapay sinir að devresi, birbirinden baðýmsýz parçalara ayrýlarak, her parçanýn benzetimi ayrý ayrý yapýlýr. Literatürde parçalý devre benzeticileri bulunmasýda raðman, bunlarýn yapay sinir aðlarýna uygulanmasý ilk defa olarak gerçekleþtirilmiþtir. Gerliþtirilen benzetici, þu ana kadar gerçekleþtirilmiþ olan bütün benzeticileden hýz ve yakýnsama bakýmýndan daha baþarýlýdýr. Bu devre benzeticisi, yapay sinir aðlarýnýn devre seviyesinde eðitilmesini saðlamaktadýr. Madaline III Kuralý devre benzeticisi ile entegre edildiði için, devre seviyesinde eðitme iþlemini yapmak mümkün olmaktadýr. Devre seviyesinde, benzetim ile eðitim yapma iþlemi daha önceden hiç yapýlmadýðý gibi, denenen örnekler üzrinde çok baþarýlý sonuçlar elde edilmiþtir. Bu devre benzeticisinin kullanýmý, yapay sinir aðlarýnýn gerçekleþtirilmesi konusunda karþýlaþýlan bir çok sorunu ortadan kaldýracaktýr.

1. INTRODUCTION

Neural networks have found many applications in recent years. Research on neural networks started before the evolution of powerful computers. Neural networks are used when there is no algorithmic solution to a problem or a problem is too complicated to be solved by known algorithms [1]. Also, neural networks can be used when the definition of the problem does not exist, but the samples of inputs and corresponding outputs are available. Applications of neural networks in general can be divided in two classes: Pattern recognition and function approximation. Some examples of pattern recognition applications are ECG abnormality detection to detect dangerous arrhythmia, data reduction by deletion of irrelevant data for high-energy-physics, speech recognition, optical data interpretation, and sensor signal processing. Some examples of function approximation, on the other hand, are the inverted pendulum problem, plant control, robot (-arm) control, and sensor signal processing.

Most of these applications need real time processing. Neural networks are inherently parallel processors. When we implement a neural network with computers, they have to be implemented in a serial manner. Besides this, the complexity of operations required in neural networks makes it impossible to use neural network simulation models in many time critical applications. However, hardware implementations of neural networks can easily be made parallel. All parts of a neural network can be implemented in hardware. There are different approaches to hardware implementation. These are digital, analog and mixed signal implementations. Mixed signal implementations generally implement the neural network in analog hardware, whereas the inputs and outputs of such implementation are digital to enable easy interfacing with digital computers [2].

1.1. What Are Artificial Neural Networks

Understanding the ways in which the brain uses neuronal systems for pattern processing serves as key to the survey of both neural network models and neurocomputers. Artificial neural networks are systems that consist of a large number of simple processing units, called neurons like in the brain. Anyone can see that the human brain is superior to a digital computer in many tasks. A good example is the processing of visual information: a one-year old baby is much better than and faster at recognizing objects than even the most advanced AI system running on the fastest supercomputer. The brain has many other features that would be desirable in artificial systems [3].

- It is robust and fault tolerant, Nerve cells in the brain die every day without affecting its performance significantly
- It is flexible. It can easily adjust to a new environment by learning; it does not have to be programmed in Pascal, FORTRAN or C.
- It can deal with information that is fuzzy, probabilistic, noisy, or inconsistent.
- It is highly parallel.
- It is small, compact, and dissipates very little power.
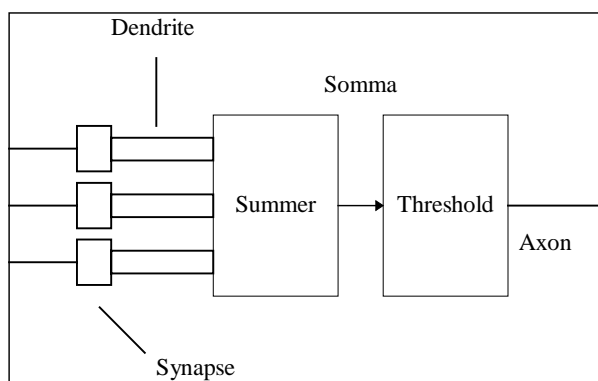
Figure 1.1. Simplified models of neurons: a biological neuron

The brain is massively parallel natural computer composed of approximately 10^{11} neurons, with each neuron connected to approximately 10^4 other neurons. The brain mainly consists of biological neurons.

The biological neuron is a complex analog computing device. Its shape and interconnection characteristics seem to determine its function in brain activity. Neuroscientists do not really understand these neuronal processes. Figure 1.1 shows the brain structure and the basic principle that nerve cells use for information processing. The neuron basically consists of a cell body, or soma, branching complex extensions that serve as inputs, or dendrites, and the output channel of the cells, or axon. The axon carries the electrical signal to other cells.

Table 1.1 shows the correspondence between the brain and artificial neural network. A neuron has generally a high-dimensional input vector and a simple output signal; this output signal is usually a non-linear function of the input vector and a weight vector. The function to be performed on the input vectors is hence defined by the non-linear function and the weight vector of neuron.

The weight vector is adjusted during the training phase by using a large set of examples and a learning rule. The learning rule adapts the weight of all neurons in a neural network in order to learn an underlying relation in the training examples. It may be clear that this type of finding a function to be performed by a system is completely different from programming a function [1].

Table 1.1. Correspondence between the brain and a neural network

Element	Brain	Artificial neural network
Organization	Network of neurons	Network of PEs
Components	Dendrites and axons Synapses Summer Threshold	Inputs and Outputs Weights Summation function Threshold function
Processing	Analog	Digital or Analog
Architecture	10-100 billion neurons	1-1,000,000 processors
Hardware	Neuron	Switching device
Switching speed	1 millisecond	1 nanosecond to millisecond
Technology	Biological	Silicon Optical Molecular

1.1.1. Types of Neural Networks

Many classes of artificial neural networks exist. Table 1.2 presents some of the best known neural network models, together with their properties. Neural networks can be characterized by a few properties: network topology, recall procedure, training/learning procedure, and input values. Designers have devised over 50 different types of neural network models. Of these, about 15 models are in common use [4]. In Hopfield model, a single layer of processing elements (PEs) forms the topology. The output from each PE feeds back to all its neighbors. In models like the Boltzmann Machine and the Back Propagation, the networks consist of one or more layers between the input and output PEs. In addition, in models like the Self-Organizing map the network connects a vector of input PEs to a two-dimensional grid of output PEs.

Table 1.2 Major neural networks and properties

Neural model	Primary applications	Strengths	Weakness
Hopfield Kohonen	Retrieval of data/images from fragments	Large-scale implementation	No learning, weights must be set
Perceptron	Typed-character recognition	Oldest neural network	No recognition of complex patterns; sensitivity to changes
Multilayer Perceptron/ Delta Rule	Pattern recognition	Simple network, more general than the Perceptron	No recognition of complex patterns
Back Propagation	Wide range: speech synthesis to analysis, and loan-application scoring	Most popular network, works well, and is simple to learn	Supervised training with abundant examples
Boltzman Machine	Pattern recognition (radar, sonar)	Simple network that uses noise function to reach global minimum energy state	Long training time
Counter-propagation	Image compression, statistical analysis, and loan scoring	Simple multiplier Perceptron, but less powerful than Back Propagation	Large number of PEs and connections
Self-Organizing Map	Mapping one geometrical region to another	Better performance than many algorithmic techniques	Extensive training
Neocognitron	Hand-printed character recognition	Sophisticated network that can identify complex patterns	Large number of PEs and connections

1.1.2.   Multilayer Perceptron

Most of the neural networks used for pattern recognition or function approximation applications use the multi-layer perceptron structure. The weights of this structure have to be determined before operation. Determination of the weights is done prior to operation, at the learning-phase, which in general requires the adaptation of weights iteratively to reduce the error (distance between the actual output and the desired output of the network) to zero. Different algorithms can be applied during the learning phase. The most commonly used and well-known algorithm is the backpropagation algorithm.

Figure 1.2 shows the structure of a multilayer neural network. This network is a feedforward network. The circles denote neurons whereas synapses are represented by the interconnections between the neurons. The circles in the input layer are just input nodes and do not perform a neuron function. This structure is a regular structure where every neuron is connected to every other neuron in the previous layer through synapses. Therefore, it yields itself easily to partitioning and automatic netlist generation.
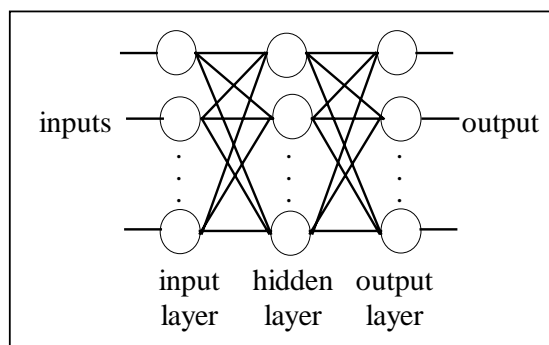
Figure 1.2 Two-layer feedforward neural network

A layer of perceptrons is created by connecting a number of perceptrons to the same input vector. Many layers can then be cascaded, with outputs of one layer connected to the inputs of the next layer, to form a network. The algorithm used to train neural networks is known as back-propagation [5].

1.1.3. Backpropagation

Backpropagation is the well known algorithm used to train the multilayer perceptron neural networks. The Backpropagation algorithm converges a set of weights that minimizes the mean-square error

$$J = E\left(\left\|d(X) - y(X)\right\|^2\right)$$
(1.1)

where $y(X)$ is the output vector of the last layer, $d(X)$ is the desired value vector, and X is the input vector. It is convenient to define "equivalent error" for each perceptron in the network. For perceptron m in the output layer, the equivalent error is given in equation 1.2, where y_m is the output of perceptron m, δ_j is the equivalent error of the jth perceptron, j indexes the set of all perceptrons that have inputs connected to perceptron m's output, and w_{jm} is the weight of the connection from perceptron m's output to perceptron j's input.

$$\delta_m = (d_m(X) - y_m(X))f'(s_m(X))$$
(1.2)

where $s_m(X)$ is the sum of the inputs to the perceptron m and $f(x)$ is the activation function of the perceptron.

For perceptron m in one of the other layers, the equivalent error is

$$\delta_m = f'(s_m(X))\sum_j \delta_j w_{jm}$$
(1.3)

Each weight is updated using the equation as for the single perceptron case, where i ranges over the inputs of perceptron m

$$w_{mi,new} = w_{mi,old} + 2\mu\delta_m x_i$$
(1.4)

where μ is the learning constant and it ranges between 0 and 1. Note that this is called back-propagation algorithm because the equivalent error is computed for the output using equation 1.2, and then propagated backward through the layers toward the input layer using

equation 1.3. As the equivalent error is computed during the backward propagation, the weights are updated using equation 1.4. There are many modifications to the backpropagation algorithm, most of which claims better convergence speeds. However, they are all based on the backpropagation algorithm. One of the modified versions of backpropagation algorithm is given in section 4.

1.2.   Neural network processors

Neurocomputers are essentially a parallel array of interconnected processors that operate concurrently. Each processing unit is primitive (that is, it consists of an analog neuron or a simple reduced instruction-set computer known as a RISC) and can contain a small amount of local memory. When we consider the basis of a neurocomputer the important design issues are parallelism, performance, flexibility and their relation to the silicon area.

There are different categories of neurocomputers [2].

- Neural network programs running on Von Neuman machines
- VLSI implementation
    - Digital
    - Analog
    - Hybrid

1.2.1.   Von Neuman machines

Von Nueman machines can be used to implement any kind of neural network structures. However, neural networks simulated on Von Neuman machines run in a series fashion, which does not let them to be used in real time applications. Very large networks (e.g. 1000's of neurons) may only be practical with specialized neural network hardware. While large general purpose parallel machines can certainly provide sufficient performance,

cheaper alternatives are also available with co-processor, or accelerator cards for computers.

1.2.2. Digital VLSI implementation

The digital neural network category encompasses many sub-categories including slice architectures, SIMD and systolic array devices, and RBF (radial basis function) architectures. For the designers, digital technology has advantages of mature fabrication techniques, weight storage in RAM, and arithmetic operations exact within the number of bits of the operands and accumulators. From the users point of view, digital chips are easily embedded into most applications. However, digital operations are usually slower than in analog systems, especially in the *weight x input* multiplication, and analog inputs must first be converted to digital. Table 1.3 shows some digital chips, their architectures, precision, capacity, and speed. The most common performance rating is the Connection-Per-Seconds (CPS), which is defined as the rate of multiplication and accumulate operations during recall process.

Table 1.3. Digital Neural Network Chips

Name	Architecture	Learn	Precision	Neurons	Synapses	Speed
NeuroLogix NLX-420	Feed forward Multilayer	no	1-16b	48x48	Off-chip	300CPS
HNC 100-NAP	General processor, SIMD, floating point	program	32b	100 PE	512K off-chip	250MCPS
Nestor/Intel NI1000	RBF	RCE, PNN	5b	1 PE	256x1024	40 k pat/s
Philips Lneuro-1	Feed forward Multilayer	no	1-16b	16 PE	64	26 MCPS
MCE MT19003	Feed forward Multilayer	no	13b	8	off-chip	32MCPS
Siemens MA-16	matrix ops	no	16b	16 PE	16x16	400MCPS

Slice architectures for neural networks provide building blocks to construct networks of arbitrary size and precision. For example, the NeuroLogix NLX-420 Neural Processor Slice has 16 processing elements. A common 16-bit input is multiplied by a weight in each PE in parallel. New weights are read from off-chip. The 16-bit weights and inputs can be user

selected as 16 1-bit, 4 4-bit, 2 8-bit or 1 16-bit value. The 16 neuron sums are multiplexed through a user-defined piece-wise continuous threshold function to produce a 16-bit output. Internal feedback allows for multi-layer networks. Multiple chips can be used to build large networks.

A far more elaborate approach is to put many small processors on a chip. Two architectures dominate such designs: single instruction with multiple data (SIMD) and systolic arrays. For SIMD design, each processor executes the same instruction in parallel but on different data. In systolic arrays, a processor does one step of a calculation before passing its result on to the next processor in a pipelined manner. A systolic array system can be built with Siemens MA-16. The MA-16 provides for fast matrix-matrix operations of 4x4 matrices with 16-bit elements. The multiplier outputs and accumulator have 48-bit precision. Weights are stored off-chip and neuron transfer function are generated off-chip via lookup tables. Multiple chips can be cascaded.

RBF networks provide fast learning and straight-forward implementation. The comparison of input vectors to stored training vectors can be done quickly if non-Euclidean distances, such as the Manhattan block norm (sum of element differences), are calculated with no multiplication. One of the commercial available products is Nestor NI1000 chip. The Nestor NI1000, developed jointly by Intel and Nestor, contains 1024 prototypes of 256 5-bit elements. The chip has two on-chip leaning algorithms and other algorithms microcoded.

1.2.3. Analog VLSI implementation

Analog neural networks can exploit physical properties to perform network operations and thereby obtain high speed and densities. A common output line, for example, can sum current outputs from synapses to the neuron inputs. However, analog design can be very difficult because of the need to compensate for variations in manufacturing, in temperature, etc. Creating an analog synapse involves the complication of analog weight storage and the need for a multiplier being linear over a wide range.

The first analog commercial chip was the Intel 80170NW ETANN [6] (Electrically trainable Analog Neural Network) that contains 64 neurons and 10280 weights. The non-volatile weights are stored as charge on floating gates and a Gilbert multiplier provides 4-quadrant multiplication. A flexible design, including internal feedback and division of the weights into 64x80 banks including 16 biases, allows for multiple configurations including 3-layers of 64 neurons/layer, and 2-layers with 128 inputs and 64 neurons. No on-chip training was provided so a chip-in the loop mode with a PC is necessary.

1.2.4. Hybrid VLSI implementation

Hybrid design attempts to combine the best of analog and digital techniques. Typically, the external inputs/outputs are digital to facilitate integration into digital systems, while internally some or all of the processing is done in analog. Table 1.4 shows some of the hybrid neural network chips available in the market. The AT&T ANNA Artificial Neural Network ALU, for example, is externally digital but uses capacitor charge, periodically refreshed by DAC's, to store weights. Similarly, the Nellcore CLNN-32 chip has 5-bit weights loaded digitally but the processing of the network with Boltzmann style annealing is done in analog.

Table 1.4.Hybrid Neural Network Chips

Name	Architecture	Learn	Precision	Neurons	Synapses	Speed
AT&T ANNA	Feed forward Multilayer	no	3b x 16b	16-25	4096	2.1GCPS
Mesa Research Neuroclassifier	Feed forward Multilayer	no	6b x 5b	6	426	21GCPS
Bellcore CLNN-32	FCR	Boltzmann	6b x 5b	32	992	100MCPS
Ricoh RN-200	Feed forward Multilayer	BP	no information	16	256	3.0GCPS

1.3. Why analog processors

Analog implementations of neural networks have many advantages such as small size and high speed. Synapses, which are the most common elements in a neural network, can be represented at the circuit level by multipliers. For instance, parallel digital multipliers of 8x8 input word lengths have transistor counts on the order of at least several thousand [7]. Analog multipliers of comparable precision use less than 20 transistors. The speed of an analog multiplier is limited only by its bandwidth and can go up to the GHz range. When one looks at neurons, a similar picture can be seen. Again, an adder and the nonlinearity can be realized by less than 20 transistors, whereas the same operations require thousands of transistors in the digital domain [8]-[10].

However, analog neural network implementations have been rather *ad hoc* in that very few, if any, have explored the constraints that circuit non-idealities—like nonlinear synapses [11],[12], neurons that deviate from ideal functions, or errors and limitations in storing weights [13] —bring about. It has been shown in [11] and [12] that multiplier nonlinearity can be a very severe problem even for nonlinearity factors of less than 10 per cent for many applications. Limited precision in storing weights has also proven to be a crucial problem in analog neural network design. The work in this area has been mostly limited to predicting these effects either through simulation or through theoretical analysis and developing some methods to overcome these problems partially. In [12], the effects of some non-idealities have been studied through circuit simulation with SPICE and the importance of circuit level simulation in analog neural network design has been demonstrated.

1.4. Outline of the Thesis

In section two; first, the history of circuit simulator programs are presented. Second, the tools used by the circuit simulator programs are explained. The methods for improving the speed and convergence properties, such as parallel processing, partitioning, are investigated. Then a partitioning method for the simulation of analog neural networks is presented and simulation results are provided.

In section three, the building blocks of the analog neural networks are presented. The building blocks are synapses, activation functions -sigmoid generator for our case-, and current to voltage (IV) converters to convert the summed current outputs of the synapses into voltages.

In section four; first, different approaches of hardware implementations of neural networks are presented. Second, the methods for training the analog neural networks are introduced and the method used in this thesis explained in detail. As the last part of that section, simulations and their results of applying the training algorithm is given.

In Appendix A, the software details of circuit simulator program is given. In Appendix B, the methods used for modeling the circuitry used as synapses and circuitry used as sigmoid generator are explained. In Appendix C, the sample input and output files of the program are given.

2.   CIRCUIT SIMULATOR

Research on the simulation of circuits started in 1950's with relay computers for simulation of electrical circuits [14]. With the improvement of digital computers, most of the researchers attempted to develop circuit simulation programs. Attempts had not been very successful before the early 1970's. In 1969-1970, Rohper initiated a graduate student project, with approximately ten students, into the investigation of circuit analysis using digital computers. All aspects of the problem were studied: network formulation, linearization, integration techniques, sparse-matrix techniques, Gaussian elimination and LU decomposition, pivoting techniques, etc. The result of the project formed the CANCER program. After very extensive use by undergraduate and graduate students in class-assignments the CANCER program evolved into Nagel's SPICE1 program. Nagel, in the doctoral studies, studied thoroughly all of the prior ten years with this type of circuit simulators. He studied thoroughly the aspects of proper choice of all of the components and techniques needed in circuit simulator. In early 1975, SPICE2 emerged, which has become a worldwide CAD tool.

The circuit simulator SPICE2 and the ones evolved at the same years shared common features: a modified nodal analysis to be able to take care of voltage sources, floating sources, and include a first- or second-order backward integration technique, advantage taken for memory considerations and computer run time speed of sparsity of the matrix of the linearized values of the circuit, pivoting to maintain sparsity, Newton-Raphson linearization, and LU solution of the equations.

Today, circuit simulation is one of the most critical and time consuming computational tasks to be performed in integrated circuit design. VLSI circuit design requires extensive, accurate simulation. The attempts to improve the speed mostly decrease the accuracy or attempts to improve the accuracy increase the simulation time. Circuit partitioning techniques are being investigated to improve the speed and/or allow the use of parallel processors [15]-[19].

In this section; first, the general flowchart of SPICE like circuit simulator and the tasks done by simulator is given. Second, the approaches for circuit partitioning are investigated. Then, the circuit partitioning approach applied to analog neural network circuits in this thesis is given. Finally the simulation results and comparison with SPICE2G6 is given.

2.1. Circuit Simulators in General

Circuit simulation includes different types of simulation, and different types of circuits. Table 2.1 shows the types of networks and types of simulation applied to those networks.

Table 2.1. Circuit Simulation Types

Type of network	Problem description
1. Linear resistive (no capacitors or inductors) and Linear dynamic (contains at least one capacitor or inductor)	1. DC analysis 2. AC analysis 3. Transient analysis 4. Noise analysis 5. Tolerance analysis 6. Determination of pole-zero locations of transfer function 7. Generation of symbolic network functions
2. Nonlinear resistive (no capacitor or inductors)	1. Operating-point analysis 2. Driving-point characteristic determination 3. Transfer characteristic determination 4. Find the output waveform due to input time function
3. Nonlinear Dynamics (contains at least one capacitor or inductor)	1. Initial condition, bias, or equilibrium state analysis 2. Transient analysis 3. Steady-state analysis 4. Nonlinear distortion analysis

Most of the circuit simulation problems are solved in two steps. The first step consists of formulating the equilibrium equations in an appropriate form, making use of the Kirchhoff laws and the element characteristics. The second step consists of solving these equations by suitable analytical or numerical techniques. The tasks performed during the simulation of a circuit by SPICE program are

- READIN and SETUP, which read the input circuit description and setup the data structures required in the analysis phase
- model evaluation and matrix load for MOSFET transistor
- matrix load for linear elements
- matrix solution
- data output routines

In this thesis, only nonlinear resistive circuits with MOSFETS are simulated, and DC analysis is performed.

2.1.1. Nonlinear DC Analysis

Most nonlinear dc analysis programs are incorporated into more general transient analysis programs, like SPICE. Equation formulation approaches are divided between nodal and state-variable analysis. In either approach, the nonlinear dc analysis problem reduces to one of solving simultaneous nonlinear algebraic equations. Iterative methods are used for the solution of these equations. SPICE uses nodal analysis, particularly modified nodal analysis and uses Newton-Raphson algorithm for the solution of nonlinear equations simultaneously.

2.1.2. Modified Nodal Analysis

For both linear and nonlinear (or piecewise-linear) circuit problems, nodal equations may be generated by little more than inspection. The relative simplicity of this approach contrasts sharply with implementations required by the state-variable approach. As an example, consider the case of a linear circuit. The nodal equations are of the form

$$Yv = i \tag{2.1}$$

where Y is the nodal admittance matrix, v is the vector of node voltages to be found, and I is a vector representing independent source currents. The term y_{ii} in Y represents the sum of the admittances of all branches connected to node i; y_{ij} is the negative of the sum of the admittances of all branches connecting node i and node j: and i_k is the sum of all source currents entering node k. If a resistor of value R connects nodes 5 and 7, $1/R$ is added to y_{55} and y_{77} and subtracted from y_{57} and y_{75} while if a current source of strength I is directed from node 2 to node 3, I is subtracted from i_2 and added to I_3.

Voltage sources are usually handled in one of two ways. The first one requires that every voltage source appear in series with a resistor so that the source may be transformed into a Norton equivalent current source. The second approach is a generalization of the first but does not depend upon a series resistor. The nodal equations are first assembled including all elements other than voltages sources by assuming the currents of the voltage sources as unknown currents. Which means that an unknown current appears in the equations for each voltage source. The additional equations comes from the voltage relation between the nodes where the voltage sources are connected. Assume that a voltage source with strength V is connect between the nodes 3 and 6. Then $V_6\text{-}V_3\text{=}V$ is the relation between node voltage 3 and 6. The equation 2.1 becomes

$$\begin{bmatrix} Y & C \\ Z & 0 \end{bmatrix}\begin{bmatrix} v \\ i_v \end{bmatrix}=\begin{bmatrix} i \\ V \end{bmatrix} \tag{2.2}$$

where i_v is the unknown currents of the voltage sources, V is the voltage values of the voltage sources, and Z is the matrix which has 1 and -1 at the positions corresponding to the positive and negative polarity nodes of the voltage sources respectively, and C is the matrix which has *1 and -1 a*t the positions corresponding to the positive and negative polarity nodes of the voltage sources respectively.

2.1.3. Functional Iteration

The solution method used by most of the presently available nonlinear dc analysis programs is based on the Newton-Raphson iteration technique [20]. It is one of a broad class of techniques known collectively as functional iteration or fixed point iteration methods.

Given a set of nonlinear equations of the form

$$g(v) = 0 \tag{2.3}$$

The solution technique is to start from some initial set of values $v^{(0)}$ and to generate a sequence of iterates ... $v^{(v-1)}$, $v^{(v)}$, $v^{(v+1)}$, ... which convergence to the solution $\bar{v}$.

Newton-Raphson iteration is most easily introduced by considering the case of a single nonlinear equation.

$$g(v) = 0 \tag{2.4}$$

The function $g(v)$ can be expanded about some point v_0 in a Taylor series to obtain

$$g(v) = g(v_0) + (v-v_0)g'(v_0) + ... = 0 \tag{2.5}$$

where the prime denotes differentiation with respect to v.

If only first-order terms are retained, a rearrangement of equation 2.5 yields

$$v = v_0 - \frac{g(v_0)}{g'(v_0)} \tag{2.6}$$

The form of equation 2.6 suggests that a sequence of iterates might be generated by the following

$$v^{(v+1)} = v^{(v)} - \frac{g(v^{(v)})}{g'(v^{(v)})} \qquad (2.7)$$

Equation 2.7 is the Newton-Raphson iteration function for the scalar case. Note that at the solution $\boxdot$, $g(\boxdot)=0$ as would be expected. The geometrical interpretation of equation 2.7 is illustrated in Figure 2.1 for the simple case of a current source driving an ideal diode. The line tangent to the nonlinearity at the point $(v^{(v)}, g(v^{(v)}))$ has the slope $g'(v^{(v)})$. Its intercept with the voltage axis defines the next voltage iterate in the sequence as shown in the Figure 2.1.

The generalization of the Newton-Raphson procedure to a system of n equations is given by

$$v^{(v+1)} = v^{(v)} - J^{-1}(v^{(v)})g(v^{(v)}) \qquad (2.8)$$

where the Jacobian $J(v)$ of the function $g(v)$ is given by

$$J(v) = \begin{bmatrix} \frac{\partial g_1}{\partial v_1} & \frac{\partial g_1}{\partial v_2} & \Lambda & \frac{\partial g_1}{\partial v_n} \\ \frac{\partial g_2}{\partial v_1} & \frac{\partial g_2}{\partial v_2} & \Lambda & \frac{\partial g_2}{\partial v_n} \\ M & & & M \\ \frac{\partial g_n}{\partial v_1} & \frac{\partial g_n}{\partial v_2} & \Lambda & \frac{\partial g_n}{\partial v_n} \end{bmatrix} \qquad (2.9)$$

A physical interpretation of the elements of $J(v)$ is given below.

A direct application of equation 2.8 necessitates computing the inverse of the $n \times n$ Jacobian matrix. The operation count for inverting a matrix and multiplying the result by a vector is $n^3 + n^2$.

An alternative procedure for obtaining new iterates is to solve the linear system of equations

$$J(v^{(v)})(v^{(v)} - v^{(v+1)}) = g(v^{(v)}) \qquad (2.10)$$

A second alternative procedure often is used with nodal analysis is to employ the system of equations

$$J(v^{(v)})v^{(v+1)} = J(v^{(v)})v^{(v)} - g(v^{(v)}) \qquad (2.11)$$

The right-hand side of equation 2.11 is found to have particularly simple interpretation. Gaussian elimination applied to either equation 2.10 or 2.11 reduces the long operation count to $(n^3/3)+n^2-(n/3)$. An even greater advantage is obtainable using sparse matrix methods. The locations of the nonzero elements of the Jacobian Matrix is fixed and remain unchanged from iteration to iteration. The additional time required on the first iteration to record the nonzero structure and determine the optimal variable elimination order is small compared to the total computational time required as the number of iterations becomes large.
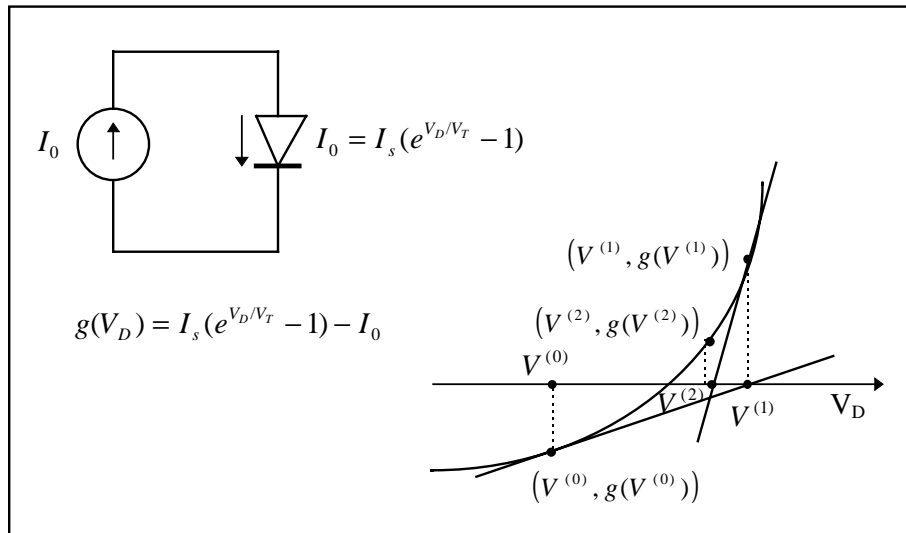
Figure 2.1. Newton-Raphson iteration

2.1.4. Newton-Raphson method applied to Nonlinear DC Analysis

A physical interpretation of the Jacobian matrix and the Newton-Raphson method can be made using the diode circuit of Figure 2.2 (a). The exponential nonlinearity of the diode is linearized about some trial solution voltage V_0. This is equivalent to a Taylor series expansion as indicated previously where only first-order terms are retained. The expansion is of the form

$$I_D = I_D\big|_{V=V_0} + (V - V_0)\frac{\partial I_D}{\partial V}\bigg|_{V=V_0} \tag{2.12}$$

$$I_D = I_s(e^{V_0/V_T} - 1) + (V - V_0)\frac{I_s}{V_T}e^{V_0/V_T} \tag{2.13}$$

$$I_D = I_{D0} + (V - V_0)g_{D0} \tag{2.14}$$

where I_{D0} is recognized as the current through the diode corresponding to the voltage V_0, and the g_{D0} is recognized as the dynamic conductance corresponding to the voltage V_o. Since the diode characteristics as described by equation 2.14 has now been linearized, the diode may be modeled in terms of a Norton equivalent current source I_{DN0} in parallel with the conductance g_{D0}. As can be seen from Figure 2.2 (b). I_{DN0} given by

$$I_{DN0} = I_{D0} - g_{D0}V_0 \tag{2.15}$$

Hence equation 2.13 may be written in terms of I_{DN0} as

$$I_D = g_{D0}V_0 + I_{DN0} \tag{2.16}$$
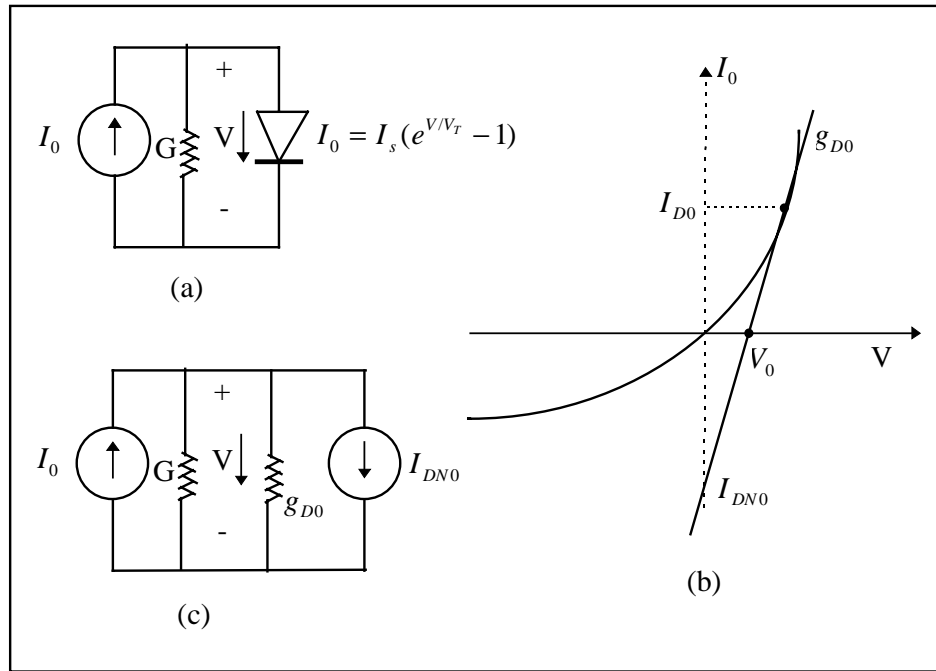
Figure 2.2. Nonlinear dc analysis. (a) Diode circuit. (b) Linearized diode approximation. (c) Linearized circuit model.

The nodel equation for the complete linearized circuit of Figure 2.2 (c). is

$$(G + g_{D0})V = I - I_{DN0} \tag{2.17}$$

In terms of iterates values

$$(G + g_D^{(v)})V^{(v+1)} = I - I_{DN}^{(v)} \tag{2.18}$$

If equation 2.8 is compared with the equation 2.18 the physical interpretation of the Jacobian and the right-hand side of equation 2.8 become more apparent. The Jacobian consists of the nodal conductance matrix of the linear elements of the circuit together with the linearized conductances associated with each nonlinear circuit element. The vector on the right-hand side of equation 2.8 consists of independent source currents and the Norton equivalent source currents associated with each nonlinear element. Thus at each iteration in a nodal analysis, the linearized conductances and Norton equivalent source currents must be recomputed and the linearized conductance matrix equations reassembled.

2.2. Partitioning

Circuit simulation with SPICE or direct method simulators is too costly and time consuming to adequately support the design of VLSI circuits. As the size of the circuits become larger the simulation times increases readily. For the simulation of large circuits, the simulation times are measured by hours, or days instead of seconds. Extensive research has been done in this area to improve the speed of the simulation. Optimized versions of SPICE provides up to 10 times of speed improvement. However, the improvements with direct simulation methods cannot be achieved beyond a limit.

Parallel processing is one of the important tools for direct simulation. Some of the tasks done by the SPICE can be easily parallelized. For an ordinary VLSI circuit, about 40% of the simulation time is used for MOSFET model evaluation and matrix load for MOSFET transistors, and 40% of the simulation time is used for the solution of matrix equation. MOSFET model evaluation and matrix load can be easily parallezible, by assigning different MOSFET's to different processors, however, the matrix solution task is not easy to parallelize. There are different attempts to parallelize the matrix solution. In this thesis, we are interested in circuit partitioning rather than the use of parallel processors. Therefore, the partitioning approaches used for parallelizing the simulation task, or to make use of the circuit latency is the main issue for this work. Table 2.2. gives some of the programs and their properties which make use of the latencies in circuit simulation.

Table 2.2. Programs making use of the latencies in circuit simulations.

Program	Properties
SPLICE	Relaxation methods are used to exploit time-domain latency
RELAX	Relaxation methods are used to exploit time-domain latency
SAMSON	Partitions the circuit into smaller subcircuits and each subcircuit is processed with its own independent time step
SLATE	Relaxation methods are used to exploit time-domain latency
SUPPLE	Partitions the circuit into blocks for parallel processing
CONCISE	Relaxation methods and parallel processing is used

2.2.1. Node tearing for circuit partitioning

The idea of node tearing is to divide a network into a set of subnetworks [15]. The nets, or nodes, that are common between them are tearing nodes as seen in Figure 2.3. Each subnetwork can then be analyzed separately and the solution is obtained in terms of tearing nodes. This is equivalent to a particular ordering of the nodal equations such that the resulting conductance matrix has bordered block diagonal structure. Each block represents a subcircuit .
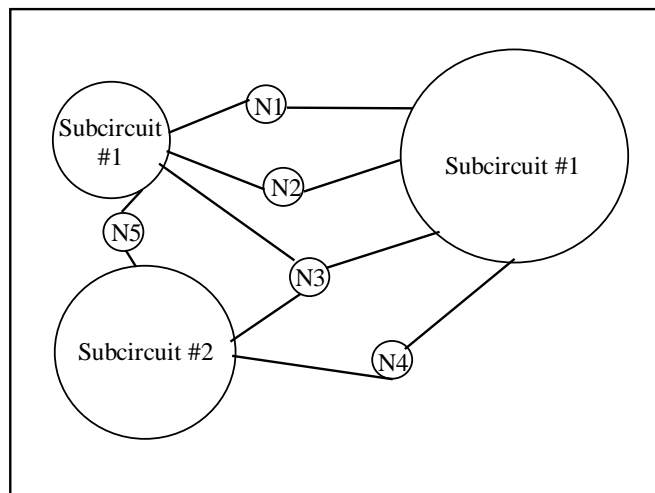
Figure 2.3. Node tearing results in a set of nodes that connect together a set of a subcircuits.

This structure can be efficiently utilized if every subcircuit has its own copy of the interconnect matrix. The entries in each subcircuit's interconnect matrix represent only the devices contained in the subcircuit. The conductance matrix of the subcircuit can be analyzed to obtain a conductance matrix that represents the Norton equivalent of the subcircuit at the tearing nodes. These matrices can be obtained to form the interconnect matrix which is then solved to determine the voltages of the tearing nodes.

The outline of matrix solution method when using node tearing and bordered block diagonal matrix is

- Load all devices in each subcircuit to that subcircuit matrix

- Solve each subcircuit matrix to obtain the conductance matrix that represents the Norton equivalent of the subcircuit at the interconnect nodes (LU and forward substitute each subcircuit to the first interconnect node)

- Add the Norton equivalent's of the subcircuits together to obtain the interconnect matrix

- Solve the interconnect matrix to obtain the tearing node voltages.

- Backward substitute each subcircuit to obtain subcircuit variables.

2.2.2. Other partitioning techniques

Different approaches are used for circuit partitioning. Some of them relies on the circuit definition given by the user. The input netlist is given in subcircuits which may also consists of further subcircuit blocks. In one of waveform relaxation techniques [19] the circuit is hierarchically partitioned using the given subcircuit definitions. Waveform relaxation techniques mostly requires unidirectional signal flow and weak feedback between partitioned blocks which is also required in the partitioning technique proposed in this thesis.

2.3. Partitioning Applied to Analog Neural Networks

Analog neural networks are specially designed to be used in real time applications. The speed of analog neural networks is a very important issue during the design. However, the feedforward structure and the regularity of the neural network allows the determination of the overall speed of the network easily. If the building blocks of the neural network are designed carefully to give a fast response, the overall structure is assured to fulfill the desired response time. This means that, for most applications, the DC transfer characteristics of the neural networks will be the important issue for the designer. Therefore, DC simulation will be sufficient for most cases. A simulator designed specifically for this purpose can be applicable to analog neural networks.

It is previously mentioned that, the most commonly used tool for circuit simulation is SPICE. However, the size of a neural network circuit for an ordinary example is very large, which makes SPICE inappropriate for the simulation of analog neural networks. The simulation time of SPICE for neural network circuits increases almost quadratically with the circuit size. Besides, when the circuit size is increased beyond a limit, SPICE starts to have difficulties in simulating the network. The solution for this problem can be solved by using partitioning techniques.

For DC analysis, if the layers are completely decoupled, by which we mean that the outputs of the neurons are not loaded by the inputs of the synapses of the next layer, the circuit can be partitioned into decoupled blocks which can be simulated separately starting from the input layer. Most of the neural network implementations uses the CMOS technology. Considering this technology, the assumption of being completely decoupled holds and allows us to partition the network.

The decoupled blocks consist of neurons and the synapses which are connected to input of these neurons. The output of the analog neural network is found starting from the first layer. The neurons in the first layer are simulated and the outputs are found. Then the output values of the first layer are then applied to the next layer as independent voltage sources being input to the synapses and the neurons in that layer are also simulated. This way, the input is propagated to the output, just like in traditional neural network simulators.

The partitions are simulated using the circuit simulation program developed in the thesis. The circuit simulator program is explained in Appendix A

2.3.1. Partitioning: Software Details

The structure of multilayer feedforward neural networks is a regular and repetitive structure. The structure consists of neurons and synapses as building blocks. The program accepts the circuit definitions of the neurons and synapses. It does not require the netlist of the whole neural network circuits, and it does not use the whole circuit to create the

partitions. The program uses the same input format with SPICE. The circuit definitions of neurons and synapses are entered as subcircuit definitions of SPICE program. The name of the subcircuit for a neuron is chosen to be NEURON and the name of the subcircuit for SYNAPSE is taken to be MULTIPLIER. A file including the definition of synapse and neuron is defined by the user. An example input file for this purpose and the created netlist file is given in Appendix C.
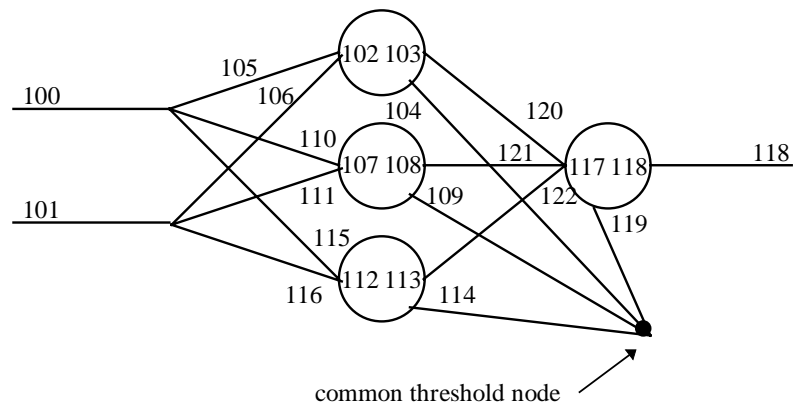
Figure 2.4. Node numbering during automatic netlist creation

Figure 2.4 shows the block diagram and assigned node numbers for an MLP structure of 2x3x1. This structure is used for XOR case.

Node numbering flowchart:

1. Number the input nodes sequentially starting from node number 100. (For this case 100 and 101)
2. Start from first hidden layer
3. Number the neuron nodes: input, output (For the first neuron 102, 103)
4. Number the threshold synapse weight input (104)
5. Number the synapse weights (105)
6. If next neuron exists goto 3
7. Goto to the next layer if the output layer is not reached

The netlist creation procedure is not necessary for the simulation of neural network. It is only used to check the accuracy of the partitioning algorithm. The synapse, IV converter,

and activation function generator circuits used in the following example is given in section 3. The network is simulated with partitioning, without partitioning and with SPICE. The node voltages at the interconnection nodes of the circuit given in Figure 2.4 simulated with these methods are given in Table 2.3 and Table 2.4. Two sets of node voltages are found with the SPICE parameter *RELTOL* set to *0.001* and *0.0001* respectively.

Table 2.3. Node voltages for *RELTOL=0.001*

Node Number	SPICE	Without Partitioning	With Partitioning
103	-0.7103	-0.7083	-0.7093
108	1.0807	1.0793	1.0793
113	0.7912	0.7898	0.7898
118	-0.8270	-0.8261	-0.8263

The differences between the simulation results is due the SPICE parameter *RELTOL*. This parameter is used to make the convergence decision for the MOSFET equations during the simulations. Using the default value for *RELTOL,* which is *0.001*, produces three digits of accuracy for the node voltages. Since the required accuracy for the node voltages is *4* digits, the default value of *RELTOL* is not adequate.

Table 2.4. Node voltages for *RELTOL=0.0001*

Node Number	SPICE	Without Partitioning	With Partitioning
103	-0.7095	-0.7097	0.7099
108	1.0800	1.0802	1.0802
113	0.7910	0.7910	0.7910
118	-0.8260	-0.8260	-0.8260

When the value of *RELTOL* is set to *0.0001* the node voltages come out to be perfectly matching. Therefore, the circuit simulator and partitioning algorithm developed in this thesis is proven to work correctly and in a reliable manner.

2.4. Simulations

Different sized analog neural network structures are created and simulated with SPICE, without partitioning, and with partitioning. Table 2.5 shows the size, simulation times, and the memory requirements for different structures.

Table 2.5. Simulation results for different MLP structures

Structure	synapses +opamps	Fets	Nodes	SPICE2G6		Without partitioning		With partitioning	
				Memory (k)	Time (sec)	Memory (k)	Time (sec)	Memory (k)	Time (k)
2x1	3	49	148	672	3.6	448	0.8	560	0.8
2x2x1	9	147	426	908	13.3	596	2.7	812	1.7
2x3x1	13	213	611	980	19.6	672	4.3	956	2.4
2x4x1	17	279	796	1040	26.3	748	6.6	1084	3.0
2x5x1	21	345	981	1100	31.9	820	8.8	1208	3.7
2x5x2	27	445	1258	1212	49.3	928	14.5	1400	5.1
2x5x3	33	545	1535	*	*	1028	20.6	1564	6.3
2x5x4	39	645	1812	*	*	1128	26.6	1696	7.0
2x5x5	45	745	2089	*	*	1264	32.3	1912	8.8
4x8x7	103	1721	4766	*	*	2136	128.7	3112	19.4

 * SPICE2G6 did not converge up to the predefined number of iterations of SPICE.
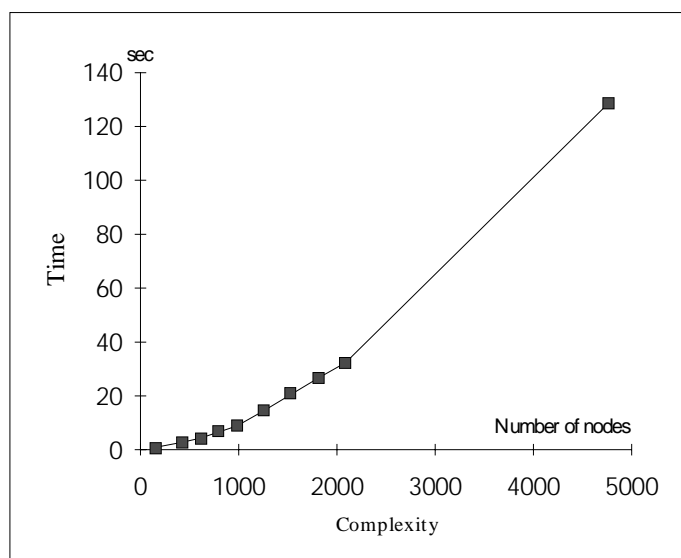
Figure 2.5. Simulation time for different MLP structures (without partitioning)
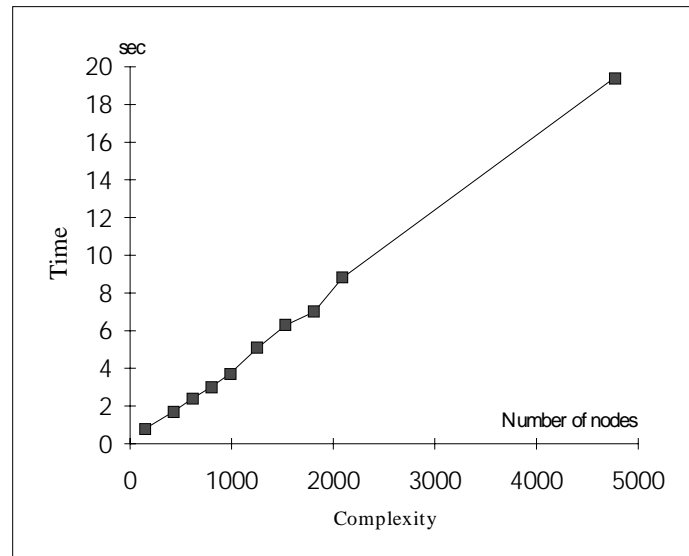
Figure 2.6 Simulation time for different MLP structures (with partitioning)

When the analog neural network circuit was partitioned, the simulation time decreased remarkably. As seen in Figure 2.5 and Figure 2.6, simulation time increases almost linearly for the partitioned case and almost quadratically for the non-partitioned case. Remarkable decreases in simulation time shows the effectiveness of partitioning. We can now simulate large neural network circuits in smaller CPU times and without any convergence problems. In spite of the fact that, partitioning increases the memory requirements, the improvement in the simulation times is worth using that extra memory.

3. BUILDING BLOCKS OF NEURAL NETWORK CIRCUITS

In this section, the realizations and constraints on the realization of synapse and activation functions are presented. Circuits for synapses and activation functions used in simulations throughout the thesis are presented as well. The circuits given in this section are designed using 1 micron ES2 technology. However, various realizations of neural network circuits are beyond the scope of this thesis and therefore they are not treated to the full extent.

3.1. Synapse

The design of multipliers used as synapse in neural network circuits imposes many often conflicting constraints on the designer. Among these are small size, high speed, linearity, and four quadrant multiplication.

Figure 3.1 shows the circuit used as a synapse. This circuit is a modified version of the well-known Gilbert multiplier [21],[22]. The inputs are in the form of voltage differences and are denoted by the couples x_1-x_2, and y_1-y_2. The output of the original Gilbert multiplier is a current difference which is proportional to the multiplication of the voltage differences (x_2 - x_1) and (y_1 - y_2)

$$I_{diff} = K(x_2 - x_1)(y_2 - y_1) \tag{3.1}$$

where K is the proportionality constant. The current difference is then converted to a single ended current (Z) through current mirrors. This improves the linearity of the multiplier; therefore, modeling procedure becomes easier which is one of the most important tasks during the training phase. Beside the improvement in linearity, the current mirror plays a role as buffer which will allow easy interfacing with the following circuitry. The following circuitry sums the current outputs of the synapses. Buffering at this point prevents the loading effects among the synapses which cannot be modeled easily. The characteristics of

the synapse circuit in Figure 3.2 show that the multiplier works almost linearly within the range of (*-1,1*) for the input voltage difference (x_2 - x_1) and (*-1,1*) for the input voltage difference (y_1 - y_2).
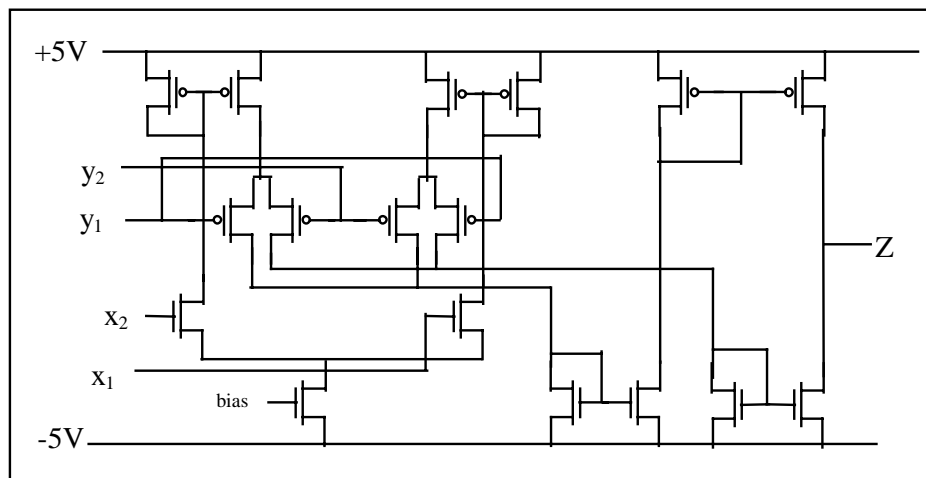
Figure 3.1 Four-quad. Gilbert multiplier
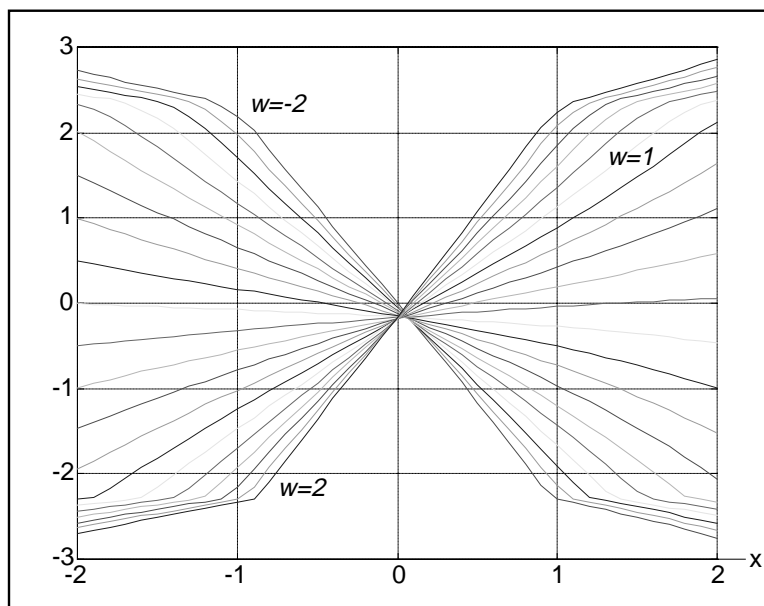
Figure 3.2 Characteristics of synapse circuitry

3.2. OPAMP

The outputs of the synapses can easily be summed. The summation is done by connecting all current outputs together. The summed current then must be converted to a voltage by a current to voltage (IV) converter. Figure 3.3 shows an OPAMP that can be used as IV converter.
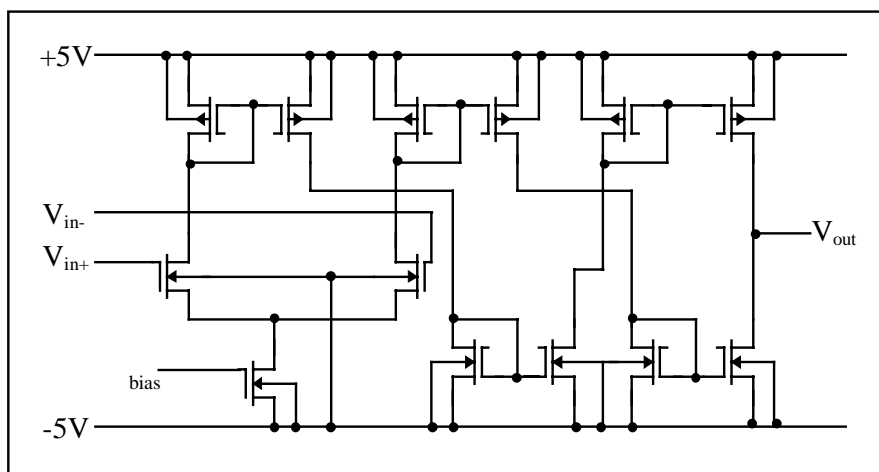
Figure 3.3 OPAMP
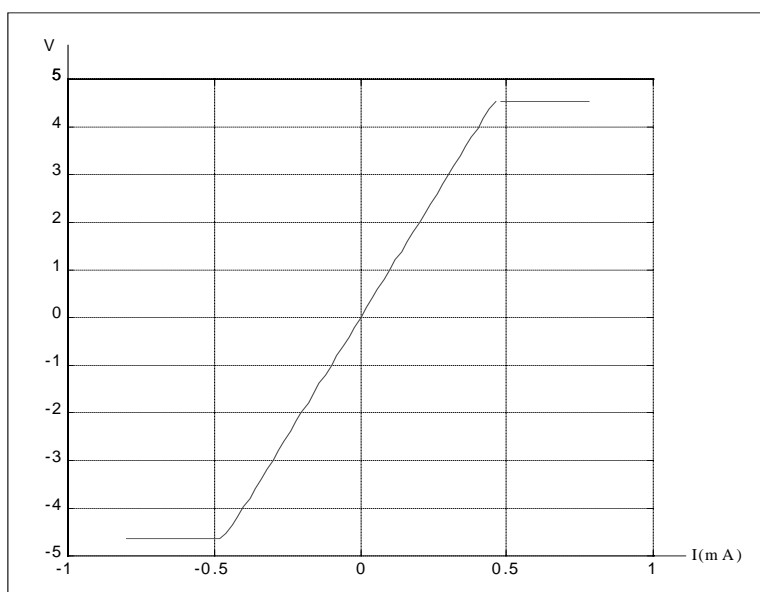
Figure 3.4 Characteristics of OPAMP (IV Converter)

This OPAMP is a two-stage OPAMP where the first stage is a differential amplifier whose differential current output is mirrored into the next stage and converted to a single ended output through circuitry very similar to the synapse circuit above. The Figure 3.4 shows the characteristics of OPAMP (IV Converter) circuitry.

3.3. Activation function: Sigmoid generator

The activation function does not have to be sigmoid; however, it has to be a soft threshold, it has to be continuously differentiable and monotone increasing. Sigmoid function is one of the functions used in neural networks.

A sigmoid generator introduced in [23] is used after the OPAMP to generate the activation function for the neuron. This generator is depicted in Figure 3.5 and the characteristics are plotted in Figure 3.6.
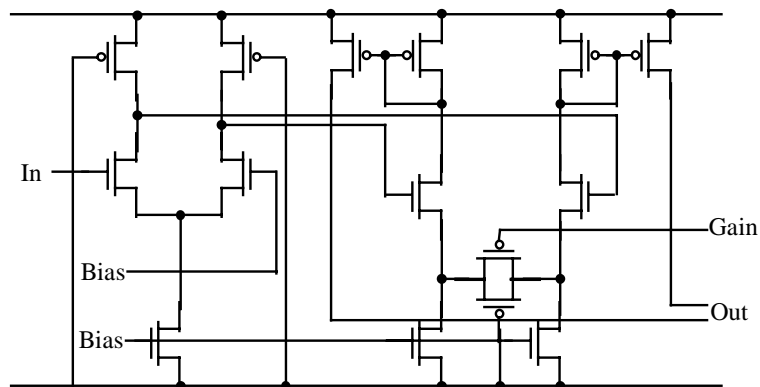
Figure 3.5. Sigmoid generator
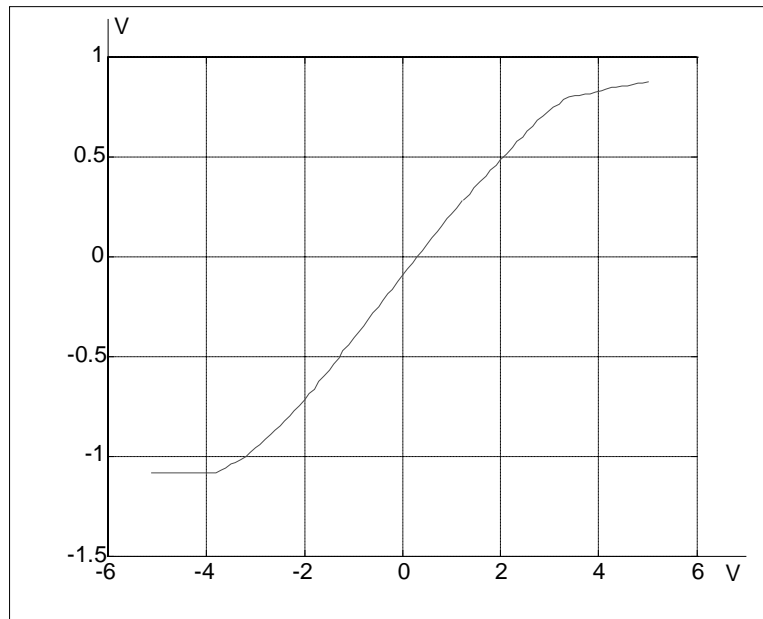
Figure 3.6 Characteristics of sigmoid generator

3.4. Interconnection

During this thesis the layout is not generated for any of these circuits. Only the subcircuit definitions of the synapse, OPAMP and Sigmoid generator are used. The interconnection of those blocks would be made via metal lines in layout. The interconnections in simulations are implemented in Figure 3.7 show the interconnections of the synapse, OPAMP, and sigmoid generator blocks for the XOR example case. The XOR example uses a 2x3x1 neural network structure.
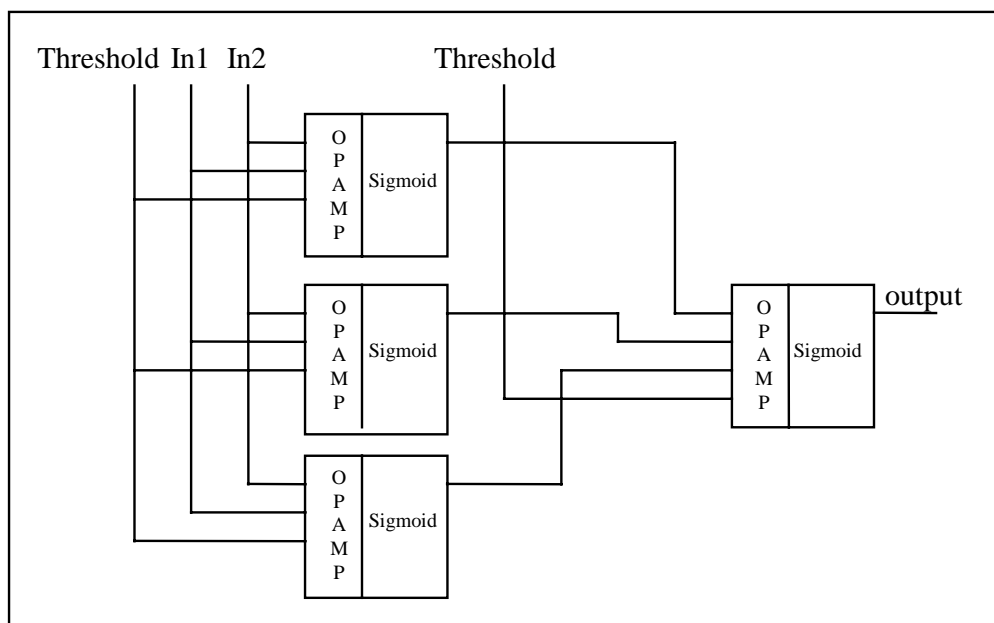
Figure 3.7. Example interconnection for XOR example (2x3x1)

4. TRAINING

In this chapter, training approaches of analog neural network implementations are investigated. Some of the techniques used in literature will be presented. Then the approach used in this thesis will be explained in detail. Training results, limitations, and problems encountered during simulations are also presented.

4.1. Training Approaches

There are three different types of analog neural network implementations, which are distinguished by the way in which the weights are modified [1].

- Non-learning network. In this method, the weights are hardwired through the implementation of the fixed gain multiplier. In this case, the weights to be hardwired must be calculated before the operation of the neural network. The calculations are done on a computer which uses the model of the actual neural network. The performance of this method depends heavily on the matching between the model and the real circuit. Perfect matching between the simulated model and the implementation model cannot be achieved. Therefore, matching problem is a major limitation for this type of network

- Neural-networks in analog hardware implementation with externally adjustable weight construction. For this realization, the weights are again computed on a host computer and downloaded to the chip. Then, the weights are fine tuned with the so called "chip-in-the-loop" method. In this method, the chip is used for forward pass, host computer is used for feedback (weight adaptation) pass. By this way, the matching of the model and analog hardware is considerably increased with respect to the situation without "chip-in-the-loop" training.

- Neural network with on-chip learning. In this, both the feedforward structure and all circuitry required for "on-chip-learning" are realized on the chip. In this case, both

feed-forward pass and feed-back pass are done on the chip. Advantage of this approach among others are the high speed of the chip which is caused by the parallel operation and absence of interface with a host computer.

The disadvantages of on chip learning are the high demand on the weight adaptation hardware, most of the chip area is used for the weight adaptation circuitry which is only utilized during weight adaptation and the resolution requirements of weights during training. In [24], [25] it is shown that the resolution for the weights during training must be about 14 to 16 bits, which cannot be realized by analog circuits in a small chip area.

Taking into account all three implementations, the minimal size hardware implementation seems to be non-learning implementation. The moderate performance approach is the second one, and if a hardware with 14 bit resolution can be implemented on a small chip area, the third approach will be superior.

Using the first implementation approach, if the model of the hardware analog neural network perfectly matches the real neural network, this implementation will be the superior one. An almost perfectly matching model can be the Spice model of the analog network. As the first part of this thesis, a simulator is developed for analog neural networks. This simulator can easily be used for training as well. When we train the neural network with this simulator, this will be equivalent to either the second approach or third approach. In both cases, the important thing is using an algorithm to train the network. Although the SPICE model almost perfectly matches the real circuitry, it is not possible analytically to find the derivative of the output error with respect to the weight. The derivative cannot be calculated in a direct manner. There is one major approach, which given in section 4.2, in the case where the derivatives of the activation function are not available

4.2. Perturbation Algorithms

Recently, the Madaline Rule III was suggested as an alternative to back-propagation algorithm for analog implementation [28]. This rule can be regarded as implementing gradient evaluation using "node perturbation" according to

$$\Delta w_{ij} = -\eta \frac{\Delta E}{\Delta sum_i} x_j \qquad\qquad (4.1)$$

where η is a constant, ⬚ is the change in the total mean square error in the

output corresponding the perturbation to the sum_j (⬚), $sum_i = \sum_j w_{ij} x_j$, and $x_j = f\left(sum_j\right)$ with f being the nonlinear activation function. Therefore, in addition to the actual hardware needed for the normal operation of the network, the implementation of the Madaline Rule III learning rule for N-neuron network in analog VLSI requires

- an addressing of module and wires routed to select and deliver the perturbation to each neuron;
- either one or N multiplication hardware to compute the term $\left(\Delta E / \Delta sum_i\right)x_j$ in addition to the multiplication by the learning rate (if one multiplier used then additional multiplexing hardware is required);
- an addressing module and wires routed to select and read x_j terms required.

If greater flexibility is required in the sense for off-chip access to the gradient values then the states of the neurons (x_j) would be needed to be made available which will require a multiplexing scheme or N chip pads.

An alternative method to perturbation is "weight perturbation," where the gradient is approximated to a finite difference [26], [27]. "Weight perturbation" algorithm is a cheaper solution with respect to the complexity.

The gradient with respect to the weight can simply be evaluated by approximation

$$\frac{\partial E}{\partial w_{ij}} = \frac{\Delta E}{\Delta_{pert} w_{ij}} + high\ order\ \text{terms} \tag{4.2}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{E(w_{ij} + \Delta_{pert} w_{ij}) - E(w_{ij})}{\Delta_{pert} w_{ij}} + high\ order\ \text{terms} \tag{4.3}$$

where $\Delta_{pert} w_{ij}$ is the perturbation. If the perturbation is small enough, update rule becomes

$$\Delta w_{ij} = \eta\ \frac{E(w_{ij} + \Delta_{pert} w_{ij}) - E(w_{ij})}{\Delta_{pert} w_{ij}} \tag{4.4}$$

where $E(\ )$ is the total mean square error produced at the output of the network for a given pair of input and training patterns and a give value of weights.

For this method, η and $\Delta_{pert} w_{ij}$ are both constants, and the analog implementation version of this update rule can simply be written as

$$\Delta w_{ij} = \frac{\eta}{\Delta_{pert} w_{ij}} \Delta E(w_{ij}, \Delta_{pert} w_{ij}) \tag{4.5}$$

The weight update hardware involves the evaluation of the error with perturbated and unperturbated weight and then the multiplication by a constant.

This technique is ideal for VLSI implementation for the following reasons:

- As the gradient $\delta E/\delta w_{ij}$ is approximated to $(E_{pert} - E) / \Delta_{pert} w_{ij}$, no back-propagation pass is needed and only forward path is required. This means, in terms of analog VLSI implementations that no bi-directional circuits and hardware for the back-propagation

are needed. The hardware used for the operation of the network is used for training. Only simple circuits to implement the weight update are required. This simplifies considerably the implementation.

- Compared with node perturbation, this technique does not require the two neurons addressing modules, routing, and extra multiplication.

4.3. Modification in Backpropagation

The backpropagation algorithm is given in introduction. The backpropagation algorithm is applicable to the cases where the synapses are linear and the outputs of the synapses are described by *wx*. However, analog implementation of neural networks does not yield such a easy relation between the output and inputs of the synapses. Although it is possible to implement synapses with transfer characteristics very close to linear multiplication, the gain of this circuit would not be 1 for most cases. Because of the differences between ideal and analog synapses, the backpropagation algorithm should be modified to allow the use of approximate models of analog synapses during training.

Modifications on backpropagation algorithm do not alter equation 1.2. However equation 1.3 should be modified to incorporate the non-linearity of synapses. Which incorporated into equation 1.3 as the derivative of the synapse function with respect to the input of the synapse.

$$\delta_m = f'(s_m(X)) \sum_j \delta_j \frac{\partial g(w_{jm}, x_j)}{\partial x_j} \qquad (4.6)$$

Equation 1.4 should also be modified to be incorporated into equation 1.4 as the derivative of the synapse function with respect to the weight of that synapse.

$$w_{mi,new} = w_{mi,old} + 2\mu\delta_m \frac{\partial g(w_{mi}, x_i)}{\partial w_{mi}} \qquad (4.7)$$

These two modifications will be sufficient to incorporate the nonlinearity of synapses into backpropagation algorithm.

The nonlinearity of the synapses affects the convergence of the learning algorithm. Depending on the type of the nonlinearity and the application, the convergence time becomes larger or smaller.

4.4. Madaline Rule III implemented with Circuit Simulator

Perturbation algorithm for training analog neural networks is presented in previous sections. The superior properties of weight perturbation over Madaline Rule III are explained in terms of "chip-in-the-loop" training. However, in our case the extra circuitry needed for perturbation does not have any meaning, because training is implemented on software. The weight perturbation algorithm, on the other hand, increases the simulation time remarkably for our case. Because the neural network circuit should be simulated after every weight perturbation.
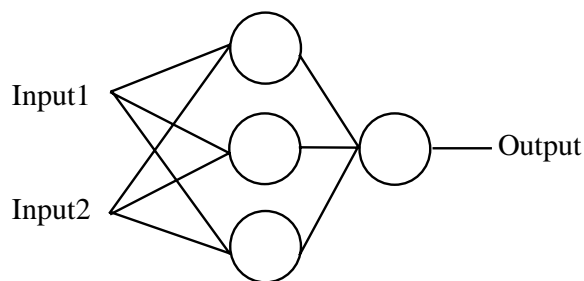
Figure 4.1 Neural network structure for XOR example

The neural network structure for simple XOR example is shown in Figure 4.1. The inputs and corresponding output values for this example is given in Table 4.1. This is a 2x3x1 structure and has 4 neurons and 13 weights including threshold values. This means that weight perturbation algorithm for this simple case will require about three times more simulation time.

Table 4.1 Inputs and Output for XOR example

Input 1	Input 2	Output
-0.8	-0.8	-0.8
-0.8	0.8	0.8
0.8	-0.8	0.8
0.8	0.8	-0.8

Another perturbation technique is also tried during the simulations. This technique perturbs all the weights at the same time. This technique eventually gave the same effect of the Madaline Rule III, because when all the weights of the synapse connected to a neuron are perturbed at the same time, the input of the neuron is also perturbed proportional to the number of synapse times the weight perturbation and the gain of the multiplier. This method has some advantages when the neurons are working around saturation, but the mathematical background of this perturbation technique is not investigated in this thesis. This perturbation method could be a part of further research subject in this area.

In this thesis, Madaline Rule III is used for perturbation algorithm. However, the simulation time is very critical in this application. Because of this, Madaline Rule III is applied with two different approaches. First one updates the weights just after the perturbation, whereas in the second one, the weights are updated after perturbating all neurons. The latter one requires almost half simulation time as the first one.

The two methods and their result will be represented with the simple XOR example, whose structure, inputs and outputs are given above.
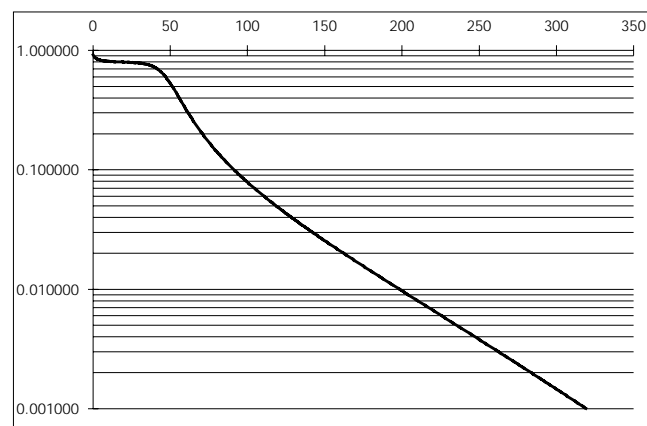
Figure 4.2 Epochs versus error during backpropagation

The starting point of the simulations is the modified backpropagation algorithm with the models of the synapse and neuron. Figure 4.2 shows the decrease in the error during the backpropagation. Table 4.2 shows the weights after backpropagation. Table 4.3 shows the results of backpropagation algorithm where the error has dropped below 0.1 per cent.

Table 4.2 Weighs after backpropagation

	Weight1	Weight2	Weight3	Threshold
Neuron1	2.309854	2.596771		-2.333146
Neuron2	-2.736185	-2.512728		-1.385036
Neuron3	-0.449041	0.588115		-1.322927
Neuron4	-3.044850	-3.264449	0.920185	-1.225071

Table 4.3 show the simulation results of the circuit simulator, which is equivalent to SPICE simulation results. The error becomes 10 per cent. This shows that the small mismatches between the models and real circuitry creates large errors. At this point, to eliminate the effects of the mismatches between the models and real circuitry, Madaline Rule III is applied for this example.

Table 4.3 Outputs of neural network with models

Input 1	Input 2	Output
-0.8	-0.8	-0.7987
-0.8	0.8	0.7995
0.8	-0.8	0.7990
0.8	0.8	0.7989

Table 4.4 Output of neural network simulated by SPICE

Input 1	Input 2	Output
-0.8	-0.8	-0.8342
-0.8	0.8	0.8793
0.8	-0.8	0.8793
0.8	0.8	-0.7663

Before giving the simulation results, let us give the flowcharts of the Madaline Rule III implementations.

- Update weights after every perturbation
    1. Apply first pattern
    2. Find the output with circuit level simulation
    3. Perturb the first neuron in the first layer
    4. Simulate the neuron to find the output
    5. Simulate the network starting from next layer
    6. Find the error difference and update the weights
    7. Simulate the neuron to find the output
    8. Simulate the network starting from next layer
    9. If not the last neuron, perturbates he next neuron, goto 4
    10. If not last layer, goto next layer, perturbate the first neuron, goto 4
    11. If not the last pattern, apply next pattern and find the output
    12. End of this epoch, if the error goal is not reached, goto 1

- Update weights after pertubating all neurons. This is almost the same with the previous method. The difference is at the 6th step. The error difference is calculated but the weights are not updated. Then the previous outputs are restored. If we do not count the simulation after applying the pattern, in this method the simulation time is decreased by almost 50 per cent.
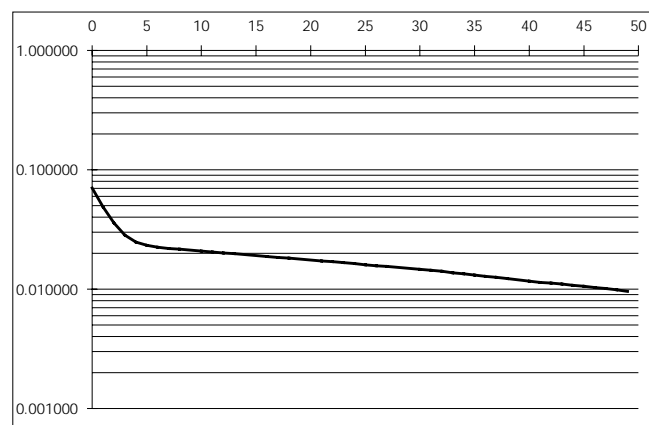
Figure 4.3 Epochs versus error during Madaline Rule III, method 1

Figure 4.3 shows the decrease in error during Madaline Rule III with method 1, whereas Figure 4.4 shows the decrease in error during Madaline Rule III with method 2. The comparison of two figures shows that, the error goal is reached with almost same number of epochs. The second one requires somehow 10 per cent more epochs. However, the decrease in simulation time will be more then this amount. Therefore, the use of the second method will be more advantageous.
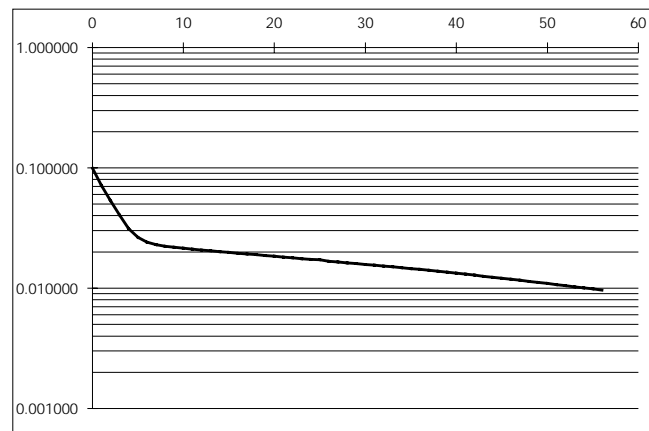
Figure 4.4 Epochs versus error during Madaline Rule III, method 1

Table 4.5 and Table 4.6 show the outputs after the implementation of Madaline Rule III. with both methods. Both results show that Madaline Rule III eliminates the effects of mismatches between the models and real circuitry. In both case, the error dropped below 0.1 per cent.

Table 4.5 Outputs after Madaline Rule III with method 1

Input 1	Input 2	Output
-0.8	-0.8	-0.8194
-0.8	0.8	0.8016
0.8	-0.8	0.8018
0.8	0.8	-0.8022

Table 4.6 Outputs after Madaline Rule III with method 1

Input 1	Input 2	Output

-0.8	-0.8	-0.8187
-0.8	0.8	0.8011
0.8	-0.8	0.8006
0.8	0.8	-0.8017

Table 4.7 and Table 4.8 show the weights after Madaline Rule III with both methods. The weights are changed in the same direction in both methods and the error goal is reach in both cases. Therefore, we can conclude that the application of Madaline Rule III solves the problems encountered during the training phase of the analog neural networks.

Table 4.7 Weighs after Madaline Rule III with method 1

	Weight1	Weight2	Weight3	Threshold
Neuron1	2.297416	2.516154		-2.319543
Neuron2	-2.613019	-2.530443		-1.369658
Neuron3	-0.464094	0.596265		-1.317426
Neuron4	-2.929680	-2.990174	1.002994	-1.289026

Table 4.8 Weighs after Madaline Rule III with method 2

	Weight1	Weight2	Weight3	Threshold
Neuron1	2.307288	2.506321		-2.338838
Neuron2	-2.634151	-2.508823		-1.393296
Neuron3	-0.475559	0.603014		-1.316461
Neuron4	-2.921889	2.979390	1.002994	-1.306904

4.5. Applications and Results

Madaline Rule III is tested on larger examples. Two different kinds of neural network applications are used to test the algorithm. First one is a function approximation application. The sine function is approximated with a neural network structure of 1x10,. the second application is a pattern recognition application. The numerals from 0 to 9 are recognized by a neural network whose structure is 20x5x10.

4.5.1.   Function Approximation: Sine Function

As a function approximation application, sine wave approximation is selected. A multilayer perceptron neural network with 1x10x1 structure is used in this example. The neural network is trained with modified backpropagation algorithm until the rms error drops below *1* per cent. Figure 4.5 show the rms error in backpropagation algorithm. After training the neural network with modified backpropagation algorithm, the network is simulated with circuit simulator. Figure 4.6 show the sine wave approximation after modified backpropagation together with the original sine wave. This example shows that small mismatches in models of the synapse and neurons resulted in large errors.
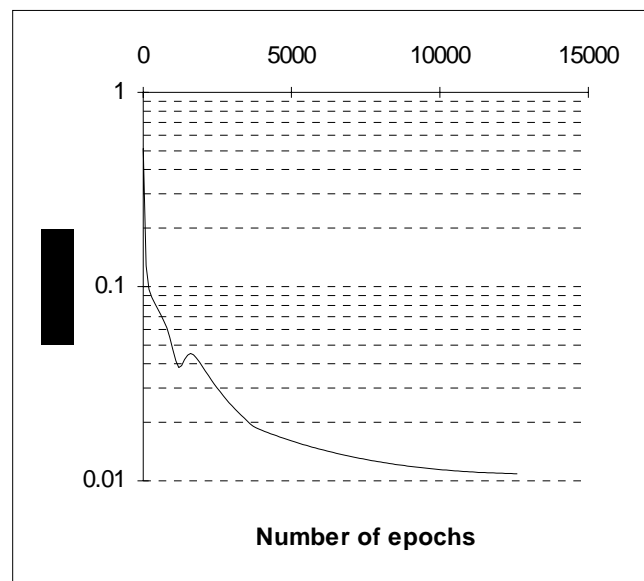
Figure 4.5 Error versus number of epochs in modified backpropagation
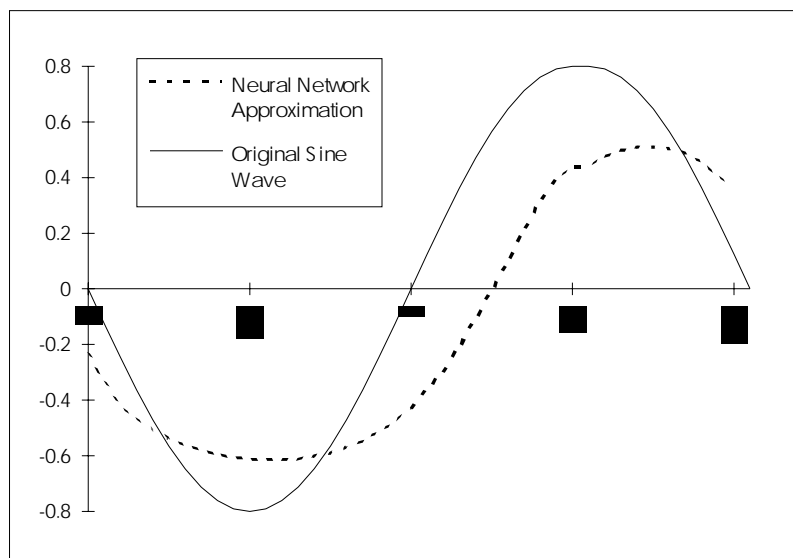
Figure 4.6 Neural network approximation of sine wave after modified backpropagation
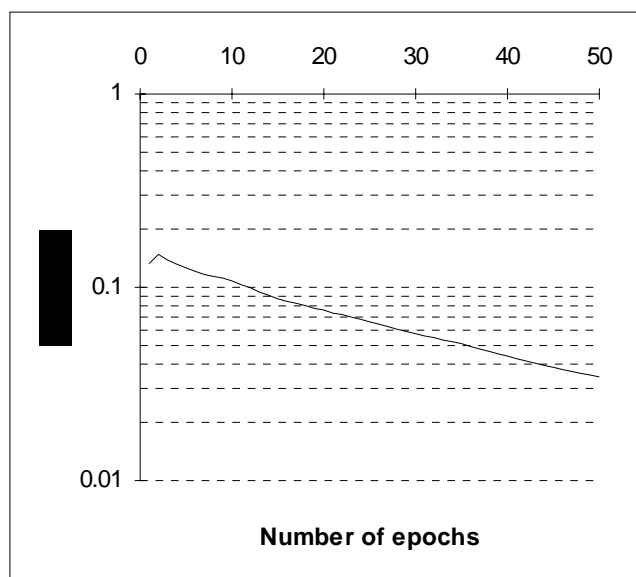
Figure 4.7 Error versus number of epochs during Madaline Rule III
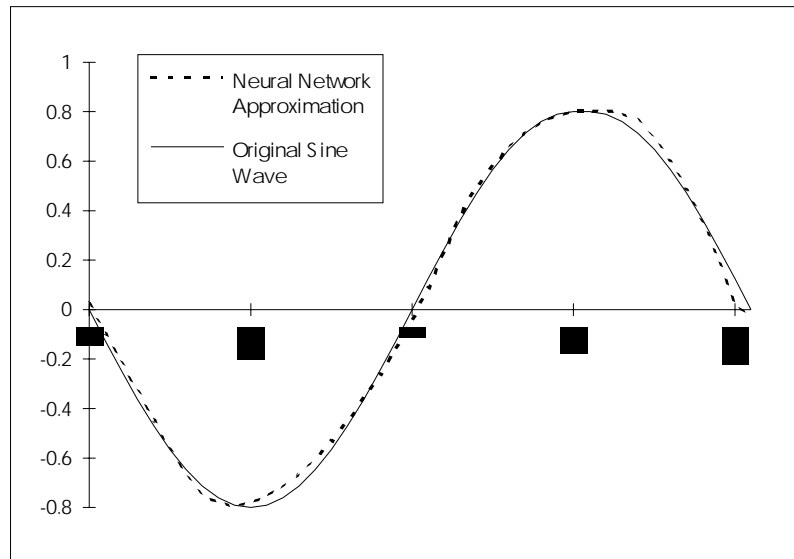
Figure 4.8 Neural network approximation of sine wave after Madaline Rule III

To eliminate the effects of model mismatches, Madaline Rule III is used. The neural network is trained for 50 epochs with Madaline Rule III. Figure 4.7 shows the rms error during Madaline Rule III and Figure 4.8 show the sine wave approximation after Madaline Rule III together with the original sine wave.

The results shows that the perturbation algorithm (Madaline Rule III) is successful in eliminating the problems created because of the model mismatches. Another advantage of the method used is that there is no need for extra hardware like in the case of "on-chip-training" or "chip-in-the-loop" training. One can claim at this point that the weight perturbation algorithm used in "chip-in-the-loop" training does not requires extra hardware, this is not totally correct, because there is need for extra hardware to be able to communicate with the host computer.

4.5.2.   Pattern Recognition: Numeral Recognition

In this example a multilayer perceptron neural network with 20x5x10 structure is used. The neural network is trained to recognize the numbers shown in Figure 4.9. The numbers are applied to the neural network as a vector with dimension of 20. Table 4.9 shows the desired

values for the numbers. The desired values are such that, the desired values corresponding to the number is 1.0, and the desired values at other position are all -1.0.
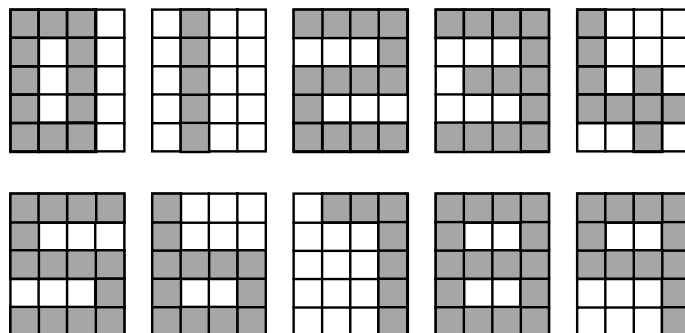
Figure 4.9 The numbers

The neural network is trained with the modified backpropagation algorithm like in the previous example. Then the network is simulated with the circuit simulator and the results are cheched. The rms error at this point is found find out to be around 20 percent. After the modified backpropagation algorithm, the network is trained with Madaline Rule III and the error dropped to about 10 percent. Table 4.10 shows the outputs after Madaline Rule III. The network can successfully recognize the original images.

Table 4.9 The desired values for the numbers

1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	1.0

Table 4.10 The outputs after Madaline Rule III

1.08	-0.87	-0.87	-0.87	-0.88	-0.88	-0.87	-0.88	-0.87	-0.87
-0.87	1.00	-0.88	-0.87	-0.87	-0.87	-0.88	-0.87	-0.87	-0.87
-0.87	-0.88	1.00	-0.88	-0.87	-0.87	-0.88	-0.88	-0.87	-0.88
-0.88	-0.88	-0.88	1.00	-0.87	-0.88	-0.88	-0.87	-0.87	-0.87
-0.88	-0.88	-0.87	-0.87	1.00	-0.87	-0.88	-0.87	-0.87	-0.88

-0.88	-0.87	-0.87	-0.88	-0.87	1.00	-0.88	-0.87	-0.88	-0.87
-0.87	-0.87	-0.88	-0.87	-0.88	-0.88	1.01	-0.87	-0.87	-0.88
-0.88	-0.87	-0.87	-0.87	-0.87	-0.87	-0.87	1.08	-0.88	-0.88
-0.87	-0.87	-0.87	-0.87	-0.87	-0.88	-0.87	-0.87	1.00	-0.88
-0.87	-0.87	-0.88	-0.87	-0.87	-0.87	-0.88	-0.87	-0.88	1.00

Some noise is added to the original pattern to create test patterns. Figure 4.10 shows the noisy images used as test patterns. Table 4.11 shows the outputs for noisy patterns. Seven out of ten images are recognized for these test patterns.
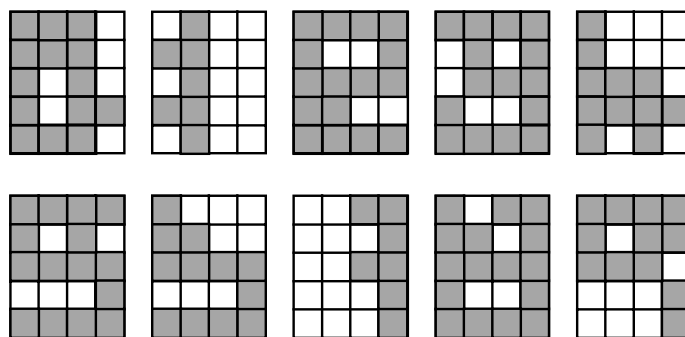
Figure 4.10 The Noisy Numbers

Table 4.11 Outputs of the neural network for noisy patterns

0.77	-0.87	-0.87	-0.87	-0.84	-0.04	-0.87	-0.87	-0.87	-0.87
-0.88	1.07	-0.88	-0.88	-0.88	-0.87	-0.88	-0.88	-0.87	-0.87
-0.87	-0.87	0.08	-0.88	-0.87	-0.87	-0.87	-0.88	0.15	-0.87
-0.88	-0.88	-0.88	0.92	-0.87	-0.88	-0.88	-0.87	-0.88	-0.87
-0.87	-0.87	-0.87	-0.87	-0.84	-0.87	0.22	-0.87	-0.87	-0.88
-0.88	-0.87	-0.87	-0.87	-0.87	0.75	-0.87	-0.87	-0.87	-0.87
-0.87	-0.88	-0.88	-0.88	-0.82	-0.73	1.08	-0.87	-0.87	-0.73
-0.87	-0.88	-0.88	-0.87	-0.88	-0.87	-0.87	-0.14	-0.87	-0.79
-0.87	-0.87	-0.87	-0.87	-0.87	-0.82	-0.87	-0.87	1.08	-0.84
-0.87	-0.87	-0.87	-0.87	-0.88	-0.87	-0.87	-0.88	-0.88	0.67

This example is presented here to show the effectiveness of the circuit simulator besides to show the effectiveness of Madaline Rule III. Because the neural network structure used for this example is a considerably large circuit. It has 150 synapses and 15 neurons, which is

equal to about 3000 MOSFETs. It is not easy to simulate such a circuit with SPICE or other simulators. With the partitioning technique used, large neural network circuits are simulated without much problems in smaller times compared with the other simulators.

5. CONCLUSION

A circuit level simulator for analog neural networks is developed. The simulator is designed for the solution of nonlinear resistive circuits especially for annealing neural network circuits implemented with mosfets. This simulator uses the modified nodal analysis, Newton-Raphson algorithm, and LU decomposition for the solution of node voltages. Modified nodal equations and Newton Raphson algorithm is explained in section 2.

The simulator uses circuit partitioning techniques to decrease the simulation time and to improve the convergence of the simulation. Analog neural networks are partitioned into decoupled blocks which consist of a neuron and all the synapses connected to that neuron. The partitions are simulated separately and the outputs of the neurons in a layer are fed to the inputs of the synapses at the succeeding layer. In this manner the output of an analog neural network is found. Simulation times for different analog neural network structures and their comparisons are given in section 2. Partitioning in simulation of analog neural network circuits is performed for the first time. The results show that the simulator developed in this thesis has superior performance over all conventional circuit simulators in both convergence and speed criteria.

Building blocks of an analog neural network is created for the simulations performed in this thesis. In section 3, multiplier circuit used as synapse, current to voltage converter circuit used for addition of synapses' output, a sigmoid generator circuit used as activation function of a neuron, and their characteristics are given.

Implementation methods of analog neural networks and their training approaches are investigated. Three types of analog implementations are given in section 4. All of these approaches have some problems. In this thesis, a new approach is proposed. This approach implements "chip-in-the-loop" training on the circuit simulator and uses perturbation algorithms. Among the perturbation algorithms, Madaline Rule III is implemented. The implementation of circuit level simulation based training algorithms does not exist in the literature. The simulator is used for the forward pass, which is performed with the

hardware in conventional methods, and the control shell of the developed software performs the feed-back pass as explained in section 4. Three examples are examined: XOR, sine wave approximation as an example of function approximation and, numeral recognition as an example of patterns recognition.

The XOR example is used because of its popularity and simplicity. With this example the partitioning technique, automatic netlist generation, modification of backpropagation algorithm and implementation of Madaline Rule III are presented.

The bacpropagation algorithm is modified to allow the integration of models of the building blocks of the neural networks and its modification is given in section 4. After training the neural network with modified backpropagation, the output of the neural network is tested with circuit level simulator and the results are found to be different from the expected results. Then, the network is trained with Madaline Rule III and the results are again tested with circuit level simulator. The results are found to be perfectly matching. The simulations show that, the proposed training approach solves some of the problems of hardware implementations of the analog neural networks.

As further work, the circuit simulator and training algorithm should be tested with different synapse and neuron circuits. After the simulations, the circuits should be implemented and the resultant neural network hardware should be tested to verify the performance of the circuit level simulator and trainer.

6. CIRCUIT SIMULATOR

The general tasks performed by the circuit simulator program are given below.

1. Read the input file and store all the device and model parameters in linked list structures.
2. Renumber the nodes and expand subcircuit definitions into the circuit netlist.
3. Process the model parameters and add internal nodes of the devices whenever they exist in device models.
4. Create modified nodal equations matrix by reserving all branches on the matrix.
5. Reorder the equations to eliminate singularity problems which may occur during LU decomposition.
6. Reorder the node numbers to minimize number of fills which are the nonzero coefficients created during LU decomposition [10].
7. Solve modified nodal equations iteratively until the solution converges

6.1. Read In Circuit Definitions

The simulator accepts the circuit definition in the form of SPICE netlist definition file. The simulator is designed to be fully compatible with SPICE in terms of netlist definitions to allow the use of available circuit layout extractor programs. The simulator program does not include all device types. The devices which can be simulated with the circuit simulator program are

- Independent voltage sources
- Independent current sources
- Voltage controlled voltage sources
- Resistors
- MOSFETs.

The circuit netlist file is read in with the *readincircuit* routine which is declared as:

```
void readincircuit(fileprocess *file,struct devicetype *devices,int sub);
```

where sub informs the subroutine that it is reading in a subcircuit definition, and *file if the object which is responsible for reading the netlist file line by line.

Before starting to explain the detail of the circuit simulator program, it is necessary to explain the data structures used in this program.

The main types of structures used to store the input netlist in link lists are.

1. Device type

This structure forms the head of the link list structure. The heads of all devices are stored in this structure. Besides, the user defined node numbers and arrays used to store the matrix elements for LU decomposition are stored in this structure.

```
struct resistortype* resistor;       /* Link list for resistors    */
struct voltagetype*  voltage;        /* Link list for voltage sour.*/
struct currenttype*  current;        /* Link list for current sour.*/
struct voltageCvoltagetype* voltageCvoltage; /* Link list for V    */
                             /* controled coltage source   */
struct mosfettype*   mosfet;         /* Link list for mosfets      */
struct subtype*      sub;            /* Link list for subcircuits  */
struct mosfetmodeltype* mosfetmodel;/* Link list for mosfet models*/
struct subcircuittype* subcircuit;  /* Link list for subcircuit   */
                                     /* definition                 */
int    numofnodes;                   /* Number of total nodes      */
int    NTTBR;                        /* Number of brunches in cir. */
int    *NODEVS;                      /* Number of voltage defined  */
                                     /* elements connected to nodes*/
int    nodes[SizeofMatrix];          /* User defined nodes         */
real   *C;                           /* Node voltages              */
real   *diagonal;                    /* Diagonal elemets           */
real   *utriangle;                   /* Upper triangular elements  */
real   *ltriangle;                   /* Lower triangular elements  */
int    *ucols;                       /* Upper triagular columns    */
int    *urowst;                      /* Upper trangular row starts */
int    *lrows;                       /* Lower triagular columns    */
int    *lcolst;                      /* Lower trangular row starts */
int    *nodereorder;                 /* Reordered node numbers     */
int    *rowreorder;                  /* Reordered row numbers      */
```

2. Resistor type

This structure stores the resistors in the circuit

```
char name[MAXNAMELENGTH + 1];          /* Name of the resistor      */
int  node1,node2;                      /* User defined nodes        */
int  n1,n2;                            /* Remapped nodes            */
real *n1n1,*n1n2,*n2n1,*n2n2;          /* Matrix location for       */
                                       /* matrix emtries            */
real value;                            /* Resistance value          */
struct resistortype* next;             /* Link to the next resistor */
```

3. Voltage type

This structure stores the independent voltage sources in the circuit

```
char name[MAXNAMELENGTH + 1];          /* Name of the resistor      */
int  node1,node2;                      /* User defined nodes        */
int  n1,n2,n3;                         /* Remapped nodes            */
real *n1n3, *n2n3, *n3n1, *n3n2;       /* Matrix location for       */
                                       /* matrix emtries            */

real value;                            /* Voltage value             */
struct voltagetype* next;              /* Link to the next source   */
```

4. VoltageCvoltage type

This structure stores the voltage dependent voltage sources in the circuit and includes the

following entries in addition to the voltage type

```
int  node3,node4;                      /* Controlling nodes         */
int  nc1,nc2;                          /* Remapped nodes            */
real *n3nc1, *n3nc2;                   /* Matrix location for       */
                                       /* matrix emtries            */
```

5. Current type

```
char       name[MAXNAMELENGTH + 1]; /* Name of the resistor      */
int node1,node2;                       /* User defined nodes        */
int n1,n2;                             /* Remapped nodes            */
real *n1n3, *n2n3, *n3n1, *n3n2;       /* Matrix location for       */
                                       /* matrix entries            */
real       value;                      /* Current value             */
struct     currenttype* next;          /* Lint to the next source   */
```

6. LX type

This structure stores the mos transistor's device specific parameters.

```
real  L, W;                            /* Lenght & Width            */
real  AD, AS;                          /* Drain & source area       */
real  VDS, VGS, VBS;                   /* Initial junction voltages */
real  devmode;                         /* Mode of operation         */
real  VON;                             /* Von voltage               */
```

```
real  VDSAT;                          /* VD_sat voltage              */
```

7. Mosfet type

This structures store the mos transistors in the circuit

```
char name[MAXNAMELENGTH + 1];          /* Name of the transistor    */
int  node1,node2,node3,node4;          /* User defined nodes        */
int  nd,ng,ns,nb,nD,nS;                /* Remapped nodes            */
real *nodes10, *nodes14, *nodes16, *nodes17, *nodes18, *nodes19;
real *nodes20, *nodes21, *nodes22, *nodes23, *nodes24, *nodes25;
real *nodes27, *nodes29, *nodes30, *nodes31, *nodes32;
                              /* Matrix location for matrix entries  */
struct mosfetmodeltype* model;         /* Link to the model         */
struct LXtype LXO;                     /* Operating point parameters */
struct locvtype LOCV;                  /* Device specific parameters */
char modelname[MAXNAMELENGTH + 1];     /* Model name                */
struct mosfettype* next;               /* Link to the next transistor*/
```

8. LX type

This structure stores the operating point values for different mosfet parameters

```
real VBS, VBD, VGS, VDS;               /* Junction voltage          */
real CD;                               /* Drain current             */
real CBS;                              /* Bulk to source current     */
real CBD;                              /* Bulk to drain current      */
real GM;
real GDS;
real GMBS;
real GBD;                              /* Bulk to drain conduct.     */
real GBS;                              /* Bulk to source conduct.    */
```

$$GM = \frac{\partial I_D}{\partial V_{GS}}, \quad GDS = \frac{\partial I_D}{\partial V_{DS}}, \quad GMBS = \frac{\partial I_D}{\partial V_{BS}}$$

9. Sub type

This structure stores the subcircuits in the circuit

```
char    name[MAXNAMELENGTH + 1];       /* Name of the subcircuit     */
char    type[MAXNAMELENGTH + 1];       /* Name of the defining sub.  */
int     nodes[10];                     /* User defined nodes         */
int     remapednodes[10];              /* Remapped nodes             */
int     numofnodes;                    /* Number of nodes            */
struct subcircuittype *subcircuit;     /* Link to the sub.definition */
struct subtype *next;                  /* Link to the next subcircuit*/
```

10. Mosfetmodel type

This structure stores the mosfet model definitions for the circuit

```
char        name[MAXNAMELENGTH + 1];
int         type;
real        LEVEL;  /*  1 Level                                   */
real        VTO;    /*  2 Zero-bias threshold voltage             */
real        KP;     /*  3 Transconductance parameter              */
real        GAMMA;  /*  4 Body-effect parameter                   */
real        PHI;    /*  5 Surface inversion potential             */
real        LAMBDA; /*  6 Channel-length modulation               */
real        RD;     /*  7 Drain ohmic resistance                  */
real        RS;     /*  8 Source ohmic resistance                 */
real        CBD;    /*  9                                         */
real        CBS;    /* 10                                         */
real        IS;     /* 11 Bulk-junction saturation current        */
real        PB;     /* 12 Bulk-junction potential                 */
real        CGSO;   /* 13 Gate-source overlap capacitance per meter */
real        CGDO;   /* 14 Gate-drain overlap capacitance per meter  */
real        CGBO;   /* 15 Gate-bulk overlap capacitance per meter   */
real        RSH;    /* 16 Source and drain sheet resistance       */
real        CJ;     /* 17 Zero-bias bulk capacitance per sqr-meter */
real        MJ;     /* 18 Bulk-junction grading coefficient       */
real        CJSW;   /* 19 Zero-bias perimeter cap. per sqr meter  */
real        MJSW;   /* 20 Perimeter capacitance grading coefficient */
real        JS;     /* 21 Bulk-junction saturation current per m² */
real        TOX;    /* 22 Thin oxide thickness                    */
real        NSUB;   /* 23 Substrate doping                        */
real        NSS;    /* 24 Surface state density                   */
real        NFS;    /* 25 Surface-fast state density              */
real        TPG;    /* 26 Type of gate material                   */
real        XJ;     /* 27 Metallurgical junction depth            */
real        LD;     /* 28 Literal diffusion                       */
real        UO;     /* 29 Surface mobility                        */
real        UCRIT;  /* 30 Critical electric field for mobility    */
real        UEXP;   /* 31 Exponential coefficient for mobility    */
real        UTRA;   /* 32 Transverse field coefficient            */
real        VMAX;   /* 33 Maximum drift velocity of carriers      */
real        NEFF;   /* 34 Total channel charge coefficient        */
real        XQC;    /* 35 Coefficient of channel charge share     */
real        KF;     /* 36 Flicker-noise coefficient               */
real        AF;     /* 37 Flicker-noise exponet                   */
real        FC;     /* 38 Bulk-junction forward bias coefficient  */
real        DELTA;  /* 39 Width effect on threshold voltage       */
real        THETA;  /* 40 Mobility modulation                     */
real        ETA;    /* 41 Static feedback on threshold voltage    */
real        KAPPA;  /* 42                                         */
real        PRM43;
real        PRM44;
real        PRM45;
```

11. Subcircuit type

This structure stores the subcircuit definitions in the circuit

```
struct devicetype devices;         /* Link to the devices in sub.*/
char   name[MAXNAMELENGTH + 1];    /* Name of the subcircuit    */
int    nodes[10];                  /* Node Numbers              */
int    numofnodes;                 /* Number of nodes           */
struct subcircuittype *next;       /* Link to the next subcircuit*/
```

6.2.   Node Remapping and Subcircuit Expansion

A circuit netlist can contain any node numbers. However, while creating the modified nodal equations the node numbers must be ordered sequentially starting from 1 (or if we include ground node, starting from 0). A circuit netlist can also contain multiple subcircuits which are defined separately. Therefore, the subcircuit definition should be expanded to get a flat circuit hierarchy. Node remapping and subcircuit expansion is done by the subroutine *errchk*, which is declared as

```
void errchk(struct devicetype *devices)
```
where devices contains the devices in the circuit.

6.3.   Mosfet Model Parameter Processing

While entering the model parameters, user should not enter all model parameters. Therefore, the program should use the default parameters when some parameters are not entered, or some parameters should be calculated from other given parameters. Mosfet model for SPICE also contains internal nodes, so these internal nodes should be added to netlist when they are necessary. This task is done by the subroutine *reversenodes* which is declared as

```
void reversenodes(struct devicetype *devices);
```

In addition to this, the mosfet models corresponding to each device in the circuit should be found by searching the model by its name, which is entered by the user for each mosfet. This task is done by the subroutine *modelnames* which is declared as

```
void modelnames(struct devicetype *devices);
```

6.4. Modified Nodal Equations

Up to this point, the user defined netlist is modified and a flat, sequential node numbered netlist is acquired. At this point modified nodal equations should be created. For a circuit with *N* nodes excluding ground node and including currents of voltage defined elements, there are *N* equations to be solved simultaneously. This *N* equation can be represented as $A_{NxN} \, B_N = C_N$, where *A* represents the modified nodal equation matrix, *B* represents the node voltages, and *C* represents the current values of the current sources and the current values of the nonlinear devices around operating point.

The matrix *A* for a circuit with *N>20* is very sparse. The sparsity of the matrix *A* goes above 97% when A becomes larger. This means that it is not necessary to store *NxN* matrix in the memory. Instead of storing an *NxN* matrix, all the brunches of the circuit are stored in a link lists. For every row of the matrix, a link list is used to store the non-zero matrix elements in that row. The elements of the link list also include the column number of their own. The program reserve all elements on the matrix for the available branches in the circuit by using the *reserve* subroutine which is defined as

```
reserve (struct devicetype *devices);
```

6.5.   Reorder Equations & Node number

After reserving the matrix locations for modified nodal equations, it is necessary to guarantee that the diagonal elements are not zero for LU decomposition. The voltage equation for voltage defined elements contain a zero diagonal element, because the voltage equation is given as

$$V_{n2} - V_{n1} = V$$

where V_{n1}, V_{n2}, and V is the node voltage of where the voltage defined element is connected and the voltage value of the voltage define element respectively. This equation is store at the row n_3 of the matrix. Therefore the diagonal element n_3 x n_3 is zero. In order to prevent a diagonal element from being zero the equations are reorder by the subroutine *reorder* which is declared as

```
reorder(struct devicetype *devices);
```

During LU decomposition, fills are created. Fills are the matrix location which are zero before LU decomposition and become non-zero after LU decomposition. Reordering the node numbers decreases the number of fills remarkably which eventually decrease the solution time of the modified nodal equation. This task is done by subroutine *solvesparse* which is declared as

```
solvesparse(struct devidetype *devices);
```

6.6.   Solution of Modified Nodal Equations

Modified nodal equations are solved iteratively. As the starting point, all node voltages are initialized to zero and mosfets are initialized such that the gate-to-source junction is just at the on-off transition potential namely at threshold voltage, bulk-to-source and drain-to-source are at zero potential.

The matrix is loaded with the linearized device parameters and using LU decomposition, forward substitution and backward substitution, the equations are solved and new node voltages are found. Then, the nonlinear devices are checked for convergence. If all devices are converged the solution is found, otherwise the iterations are continued from the matrix load point until convergence. Solution of equations is done by the routine *dcop*, which is declared as

```
void dcop(struct devicetype *devices)
```

6.7. *An Example to Explain the Solution Steps*

This example is used to explain node remapping, subcircuit expansion and creation of matrix of equation.

```
Example Ciruiut
Vdd 3 0 10
Vgg 2 0 2
X1 3 2 9 SUB
*
.subckt SUB 15 12 11
M1 11 12 0 0 NFET
X1 11 15 res
.ends sub
*
.subckt res 1 2
R1 1 2 100k
.ends
*
.MODEL NFET NMOS LEVEL=2
*
.end
```

The devices and all the node number of the devices after node remapping and subcircuit expansion are given below.

```
Name: R1        User defined nodes: 1, 2 Remapped nodes: 5, 1
Name: Vdd       User defined nodes: 3, 0 Remapped nodes: 1, 0, 2
Name: Vgg       User defined nodes: 2, 0 Remapped nodes: 3, 0, 4
Name: M1        User defined nodes: 11, 12, 0, 0 Modelname:NFET
                Remapped nodes: 5, 3, 0, 0, 6, 7
```

The first device processed is the voltage source Vdd. The nodes of the voltage source Vdd are assigned as 1,0,2. The ground node is left as it is. The current node numbers to be assigned becomes 3 at this point. The second device process is the voltage source Vgg. The nodes of the voltage source Vgg are assigned as 3,0,4. The current node numbers to be assigned becomes 5 at this point.

The node numbers of the subcircuit X1 is assigned as 1,3,5 since the node 3 was remapped to 1, the node 2 was remapped as 3 and the node 9 is remapped as 5.

This means that the nodes 15,12,11 will be replaced with the nodes 1,3,5.

So the mosfet node numbers becomes 5,3,0,0.

The nodes of X1 in the subcircuit SUB are remapped to 5,1. This means that the nodes of RES 1,2 will be replaced with 5,1. So the remapped node numbers of R1 becomes 5,1.

After the node remapping and subcircuit expansion is completed, the internal nodes of the MOSFET are assigned. The internal nodes of the MOSFET becomes 6,7.

	1	2	3	4	5	6	7
1	•	•			•		
2	•						
3				•			
4			•				
5	•				•	•	
6			•		•	•	•
7			•			•	•

The resistor has entries at the position 11, 55, 15, 51. The voltage source Vdd has entries at 21,12. The voltage source Vgg has entries at 34,43. The other entries stem from the MOSFET.

The diagonal entries of 22,33,44 are all zero because of the voltage sources. If a voltage source is connected to a node to which a conductance is connected, it creates one diagonal with zero entry. Otherwise, the voltage source creates 2 diagonals with 0 entry. This is the

case with the voltage source Vgg, because the node to which Vgg is connected has no conductance (Gate of the Mosfet).

The rows are reordered and the matrix is made symmetrical. Then the node numbers are reordered in order to minimize the number of fill, which will be created during the LU factorization.

	1	2	3	4	5	6	7
1	•	•			•		
2	•	•			•		
3			•			•	•
4				•			
5	•	•			•	•	
6			•		•	•	•
7			•			•	•

7. MODELING

7.1. Synapse Modeling

The multipliers used as synapse are assumed to be linear. However, real multipliers are not perfectly linear. One reason for the nonlinearity is the saturation of the output due to the current driving capability of the output stage of the multiplier. Another reason for the nonlinearity is that, the approximations done during the derivation of the output relation of the multiplier does not hold when the inputs become larger. Therefore, synapses cannot be modeled only using linear multiplication. Taking into account the saturation of the output, the synapses are modeled with the following function:

$$g(w,x) = A + Bw_i + Cx_j + Dw_ix_j + Ew_i^2x_j + Fw_ix_j^2 + Gw_i^2x_j^2 + Hw_i^3x_j^2 + Kw_i^2x_j^3 + Lw_i^3x_j^3$$

The data for the multiplier is taken in a small neuron structure, with two synapses connected to a neuron in order to take into account the effects of the current summing circuitry. The inputs of the synapse are denoted by w and x, and the output of the synapse which is the output current is denoted as z.

A set of data is acquired from the simulation of the synapse for a the set of input values (w_{min}, w_{max}) x (x_{min}, x_{max}).

In order to find the values of the coefficients of the function $g(w,x)$ we should minimize the square error with respect to all coefficients. The square error is given as:

$$SquareError = ES = \sum_i \sum_j \left(g(w_i, x_j) - z_{ij} \right)^2$$

To be able to minimize the error with respect to the coefficients $A, B, C, D, E, F, G, H, K, L$ the partial derivative of the square error with respect to all coeeficients should be zero.

$$\frac{\partial ES}{\partial A} = 0, \ \frac{\partial ES}{\partial B} = 0, \ \frac{\partial ES}{\partial C} = 0, \ \frac{\partial ES}{\partial D} = 0, \ \frac{\partial ES}{\partial \mathrm{E}} = 0,$$

$$\frac{\partial ES}{\partial F} = 0, \ \frac{\partial \mathrm{ES}}{\partial G} = 0, \ \frac{\partial ES}{\partial H} = 0, \ \frac{\partial ES}{\partial K} = 0, \ \frac{\partial ES}{\partial L} = 0. \ \text{where}$$

$$\frac{\partial ES}{\partial A} = \sum_i \sum_j 2(A + Bw_i + Cx_j + Dw_i x_j + Ew_i^2 x_j + Fw_i x_j^2 + Gw_i^2 x_j^2 + Hw_i^3 x_j^2 + Kw_i^2 x_j^3 + Lw_i^3 x_j^3 - z_{ij})$$

$$\frac{\partial ES}{\partial B} = \sum_i \sum_j 2w_i(A + Bw_i + Cx_j + Dw_i x_j + Ew_i^2 x_j + Fw_i x_j^2 + Gw_i^2 x_j^2 + Hw_i^3 x_j^2 + Kw_i^2 x_j^3 + Lw_i^3 x_j^3 - z_{ij})$$

$$\frac{\partial ES}{\partial C} = \sum_i \sum_j 2x_j(A + Bw_i + Cx_j + Dw_i x_j + Ew_i^2 x_j + Fw_i x_j^2 + Gw_i^2 x_j^2 + Hw_i^3 x_j^2 + Kw_i^2 x_j^3 + Lw_i^3 x_j^3 - z_{ij})$$

$$\frac{\partial ES}{\partial D} = \sum_i \sum_j 2w_i x_j(A + Bw_i + Cx_j + Dw_i x_j + Ew_i^2 x_j + Fw_i x_j^2 + Gw_i^2 x_j^2 + Hw_i^3 x_j^2 + Kw_i^2 x_j^3 + Lw_i^3 x_j^3 - z_{ij})$$

$$\frac{\partial ES}{\partial E} = \sum_i \sum_j 2w_i^2 x_j(A + Bw_i + Cx_j + Dw_i x_j + Ew_i^2 x_j + Fw_i x_j^2 + Gw_i^2 x_j^2 + Hw_i^3 x_j^2 + Kw_i^2 x_j^3 + Lw_i^3 x_j^3 - z_{ij})$$

$$\frac{\partial ES}{\partial F} = \sum_i \sum_j 2w_i x_j^2(A + Bw_i + Cx_j + Dw_i x_j + Ew_i^2 x_j + Fw_i x_j^2 + Gw_i^2 x_j^2 + Hw_i^3 x_j^2 + Kw_i^2 x_j^3 + Lw_i^3 x_j^3 - z_{ij})$$

$$\frac{\partial ES}{\partial G} = \sum_i \sum_j 2w_i^2 x_j^2(A + Bw_i + Cx_j + Dw_i x_j + Ew_i^2 x_j + Fw_i x_j^2 + Gw_i^2 x_j^2 + Hw_i^3 x_j^2 + Kw_i^2 x_j^3 + Lw_i^3 x_j^3 - z_{ij})$$

$$\frac{\partial ES}{\partial H} = \sum_i \sum_j 2w_i^3 x_j^2(A + Bw_i + Cx_j + Dw_i x_j + Ew_i^2 x_j + Fw_i x_j^2 + Gw_i^2 x_j^2 + Hw_i^3 x_j^2 + Kw_i^2 x_j^3 + Lw_i^3 x_j^3 - z_{ij})$$

$$\frac{\partial ES}{\partial K} = \sum_i \sum_j 2w_i^2 x_j^3(A + Bw_i + Cx_j + Dw_i x_j + Ew_i^2 x_j + Fw_i x_j^2 + Gw_i^2 x_j^2 + Hw_i^3 x_j^2 + Kw_i^2 x_j^3 + Lw_i^3 x_j^3 - z_{ij})$$

$$\frac{\partial ES}{\partial L} = \sum_i \sum_j 2w_i^3 x_j^3(A + Bw_i + Cx_j + Dw_i x_j + Ew_i^2 x_j + Fw_i x_j^2 + Gw_i^2 x_j^2 + Hw_i^3 x_j^2 + Kw_i^2 x_j^3 + Lw_i^3 x_j^3 - z_{ij})$$

Using the data acquired from simulations and above conditions, 10 equations are created with 10 unknown coefficients. These equations is then solved using Gaussian elimination and coefficients are found.

7.2. *Sigmoid Generator Modeling*

A normalized sigmoid generator function is given as follows:

$$f(x) = \frac{1}{1+e^{-x}}$$

When the sigmoid generator is implemented in hardware, the above simple function cannot be implemented. The minimum value of the function can be different from zero, and the range of the sigmoid generator can be different from one. Besides, the hardware implementation of the sigmoid generator can have offsets. Therefor all of the above mentioned effects should be taken into account and a general function should be used for sigmoid generator model. The function used as the model is

$$f(x) = A + \frac{B}{1+e^{(Cx+D)}}$$

The data for sigmoid generator is again acquired by simulation. A set of inputs is given and the outputs are taken. The input is denoted by x and output is denoted by y.

In order to find the values of the coefficients of the function f(x) we should minimize the square error with respect to all coefficients. The square error is given as:

$$SquareError = ES = \sum_i \left(f(x_i) - y_i \right)^2$$

However, this function is not easy to be minimized with respect to all coefficients *A, B, C,* and *D*. The values of the coefficients *A* and *B* can be found easily by looking at the output values. The coefficient *A* should be equal to the minimum value of the output values and the value of *B* should be the range of the output values which is the difference of the maximum output and minimum output value. After finding the values of *A* and *B* the function is modified so that we get a function which is linear with respect to input *x*.

$$g(x) = Cx - D = \ln\left(\frac{B}{f(x) - A}\right)$$

Since we know the values of A and B we can find the output values for $z=g(x)$. Now we have to find the coefficients C and D, which is done by minimizing the square error with respect to C and D. The square error is given by

$$SquareError = ES = \sum_i \left(g(x_i) - z_i\right)^2 = \sum_i \left(Cx_i + D - z_i\right)^2$$

Therefore,

$$\frac{\partial ES}{C} = \sum_i 2x_i(Cx_i + D - z_i) = 0$$

$$\frac{\partial ES}{D} = \sum_i 2(Cx_i + D - z_i) = 0$$

Solving the above two equations, we will find the values of C and D.

The derivative of this general sigmoid function in terms of itself is:

$$f'(x) = C\left(\frac{(f(x) - A)(f(x) - A)}{B} - (f(x) - A)\right)$$

8. EXAMPLE INPUT AND OUTPUTS

8.1. Example Input Netlist File

```
Neuron + Synapse Circuits
* Definition of a neuron
.subckt neuron 1 4 2
Xop 1 4 3 opamp
Rf  1 3 10k
Xsig 3 2 sigmoid
.ends
*definition of an OPAMP
.subckt multiplier 1 114 112 111 106 108 105
M0    1 100 100    1 pfet L=1.0U W=3.0U
M1    1 100 101    1 pfet L=1.0U W=3.0U
M2    1 102 103    1 pfet L=1.0U W=3.0U
M3    1 102 102    1 pfet L=1.0U W=3.0U
M4    1 104 104    1 pfet L=1.0U W=12.0U
M5    1 104 105    1 pfet L=1.0U W=27.0U
M6  101 106 107    1 pfet L=1.0U W=2.0U
M7  101 108 109    1 pfet L=1.0U W=2.0U
M8  103 108 107    1 pfet L=1.0U W=2.0U
M9  103 106 109    1 pfet L=1.0U W=2.0U
M10 110 111 100 114 nfet L=1.0U W=1.5U
M11 102 112 110 114 nfet L=1.0U W=1.5U
M12 110 113 114 114 nfet L=1.0U W=3.0U
M13 107 107 114 114 nfet L=1.0U W=1.5U
M14 104 107 114 114 nfet L=1.0U W=3.0U
M15 109 109 114 114 nfet L=1.0U W=1.5U
M16 105 109 114 114 nfet L=1.0U W=11.0U
VBIAS 113 0 -7.8
.ends
*definition of a sigmoid generator
.subckt sigmoid 112 106
M0    1 100 101    1 PFET L=1.0U W=3.0U
M1    1 100 102    1 PFET L=1.0U W=3.0U
M2  104 103   1    1 PFET L=1.0U W=9.0U
M3  103 103   1    1 PFET L=1.0U W=3.0U
M4  105 105   1    1 PFET L=1.0U W=3.0U
M5  106 105   1    1 PFET L=2U W=19.250U
M6  107 108 109    1 PFET L=2.0U W=3.0U
M7  107 100 109    1 PFET L=2.0U W=3.0U
M8  102 110 111 100 NFET L=1.0U W=1.5U
M9  101 112 111 100 NFET L=1.0U W=1.5U
M10 103 102 109 100 NFET L=2.0U W=4U
M11 105 101 107 100 NFET L=2.0U W=4U
M12 111 113 100 100 NFET L=1.0U W=9.0U
M13 109 113 100 100 NFET L=1.0U W=6.0U
M14 107 113 100 100 NFET L=1.0U W=6.0U
M15 104 104 100 100 NFET L=1.0U W=1.5U
M16 106 104 100 100 NFET L=2.0U W=3U
VIN-  110 0 -1
VBIAS 113 0 -3
VDD   1   0  5
```

```
VSS    100 0 -5
VGAIN 108 0 -5
.ends

.subckt opamp 103 105 102
M0 100 100   1 1 pfet L=2.0U W=30.0U
M1 101 100   1 1 pfet L=4.0U W=6.0U
M2 102 100   1 1 pfet L=2.0U W=100.0U
M3 104 103 101 1 pfet L=5.0U W=15.0U
M4 106 105 101 1 pfet L=5.0U W=15.0U
M5 104 104   2 2 nfet L=10.0U W=3.0U
M6 106 104   2 2 nfet L=10.0U W=3.0U
M7 102 106   2 2 nfet L=7.0U W=150.0U
IR 100 0 200U
VDD 1 0 5
VSS 0 2 5
.ends
.MODEL NFET NMOS LEVEL=2 TOX=2.0E-08 VTO=0.82
+LD=0.125U NSUB=2.5E+16 GAMMA=0.76
+UO=690.0 UEXP=0.35 UCRIT=35.0K VMAX=70.8K DELTA=0.0
+RSH=55.0 NEFF=30.0 LAMBDA=0.0 NFS=3.1E+11 NSS=0.0
+XJ=0.25U CJ=350.0U MJ=0.43 CJSW=450.0P
+MJSW=0.43 CGDO=310.0P PB=0.675 CGSO=310.0P JS=2.0U
 *
.MODEL PFET PMOS LEVEL=2 TOX=2.0E-08 VTO=-1.4
+LD=0.047U NSUB=2.5E+16 GAMMA=0.78
+UO=231.0 UEXP=0.35 UCRIT=71.0K VMAX=320.0K DELTA=0.10
+RSH=75.0 NEFF=0.88 LAMBDA=3.8E-02 NFS=1.0E+10 NSS=0.0
+XJ=0.45U CJ=540.0U MJ=0.510 CJSW=760.0P
+MJSW=0.510 CGDO=300.0P PB=0.7 CGSO=300.0P JS=10.0U
.end
```

8.2. Automatically Created Netlist for XOR Example

```
X100 1 2 100 0 105 0 102 multiplier
V100 105 0 -0.1200
X101 1 2 101 0 106 0 102 multiplier
V101 106 0 0.7600
X102 1 2 20 0 104 0 102 multiplier
V102 104 0 1.0000
X103 102 0 103 neuron
X104 1 2 100 0 110 0 107 multiplier
V104 110 0 0.8600
X105 1 2 101 0 111 0 107 multiplier
V105 111 0 -0.9800
X106 1 2 20 0 109 0 107 multiplier
V106 109 0 -0.7800
X107 107 0 108 neuron
X108 1 2 100 0 115 0 112 multiplier
V108 115 0 -0.2000
X109 1 2 101 0 116 0 112 multiplier
V109 116 0 -0.6200
X110 1 2 20 0 114 0 112 multiplier
V110 114 0 0.7400
X111 112 0 113 neuron
X112 1 2 103 0 120 0 117 multiplier
V112 120 0 0.6200
X113 1 2 108 0 121 0 117 multiplier
V113 121 0 -0.6000
X114 1 2 113 0 122 0 117 multiplier
V114 122 0 0.5800
X115 1 2 20 0 119 0 117 multiplier
V115 119 0 -0.0200
X116 117 0 118 neuron
Vdd 1 0 12
Vss 0 2 12
```

9.   REFERENCES

1.   Annema, A-J., *Feed-forward Neural Networks, Vector Decomposition Analysis, Modelling and Analog Implementation*, Boston: Kluwer Academic Publishers, 1995.

2.   C. S. Lindsey and Th. Lindblad, "Review of hardware neural networks: A user's perspective," *Proc. of the 3rd Workshop on Neural Networks: From Biology to High Energy Pyhsics*, Isola d'Elba, Italy, Sept. 26-30, 1994.

3.   Hertz, Jack., Krogh, A., Palmer, R. G., *Introduction to the theory of Neural Computation*, New York: Addison Vesley, 1991.

4.   O. Rossetto et. al., "Analog VLSI synaptic matrices as building blocks for neural networks," *IEEE Micro. Mag.*, pp 56-63, Dec 1989.

5.   R. P. L.ppmann, "An introduction to computing with neural nets," *IEEE ASSP Magazine*, pp 4-22, April 1987

6.   Intel 80170NX ETANN Data Sheets, Feb. 1991

7.   H. Binici, G. Dündar, and S. Balkýr, "A new multiplier architecture based on radix-2 conversion scheme," *Proceedings of ECCTD '95*, pp 439-442, Istanbul, 1995.

8.   P. Treleaven, M. Pacheco, and M. Vellasco, "VLSI architectures for neural networks," *IEEE Micro. Mag.*, pp 8-27, Dec 1989.

9.   O. Rossetto et. al., "Analog VLSI synaptic matrices as building blocks for neural networks," *IEEE Micro. Mag.*, pp 56-63, Dec 1989.

10.   G. Dündar and K. Rose, "Analog neural network circuits for ASIC fabrication," *Proc. of the 5th. IEEE ASIC Conference*, Rochester, 1992, pp 419-422.

11.   G. Dündar, F-C. Hsu, and K. Rose, "Effects of nonlinear synapses on the performance of multilayer neural networks," *Neural Computation*, in press.

12.   A. Þimsek, M. Civelek, and G. Dundar, "Study of the effects of nonidealities in multilayer neural networks with circuit level simulation," *Proc. of 8th Melecon*, pp. 613-616, Bari, Italy,May 1996.

13.   G. Dundar and K. Rose, "The effects of quantization on multilayer neural networks," *IEEE Transactions on Neural Networks*, Vol.6, No. 6, pp. 1446-1451, Nov. 1995.

14.   D. O. Pederson, "A Historical review of Circuit Simulation", *IEEE Transactions on Circuits and Systems*, Vol. cas-31. No. 1, January 1984

15. P. F. Cox, R. G. Burch, P. Yang, D. E. Hocevar, "New Implicit Integration Method for Efficient Latency Exploitation in Circuit Simulation", *IEEE Transactions on Computer-Aided Design* Vol. 8. N0. 10. October 1989

16. G. D. Hachtel, A. L. Sagiovanni-Vincentelli, "A survey of Third-Generation Simulation Techniques", *Proceedings of the IEEE*, Vol. 69, No. 10. October 1981

17. P. F. Cox, R. G. Burch, D. E. Hocevar, P. Yang, B. D. Epler, "Direct Circuit Simulation Algorithms for Parallel Processing", *IEEE Transactions on CAD,* Vol. 10. No. 6, June 1991

18. L. Pederson, S. Mattison, "The Design and implementation of a Concorent Circuit Simulation program for Microcomputers", *IEEE Transaction on CAD*, Vol. 12, No. 7, July 1993

19. P. Saviz, O. Wing, "Circuit Simulation by Hierarchical Waveform Relaxation", *IEEE Transaction on CAD*, Vol. 12, No. 6, June 1993

20. W. J. McCalla, D. O. Pederson, "Elements of Computer- Aided Circuit Analysis," ?????????

21. B. Gilbert, "A precise four quadrant multiplier with subnanosecond response," *IEEE Journal of Solid State Circuits*, Vol 3., pp 365-373, 1968.

22. F. J. Kubetal, "Programmable analog matrix multipliers," *IEEE Journal of Solid-State Circuits*, Vol 25, pp 207-214, 1990

23. T. Shima et.al., "Neuro Chips with On-chip Back-Propagation and/or Hebbian Learning," *IEEE Journal of Solid-State Circuits*, Vol. 27, pp. 1868-1976, Dec. 1992.

24. R.C. Frye, E. A. Rietman and C. C. Wong, "Back-Propagation Learning and Nonidealities in Analog Neural Nework Hardware," *IEEE Trans. Neural Networks*, Vol. 2, pp. 110-117, 1991

25. P. W. Hollis, J.S. Harper and J. J. Paulos, "The Effects of Presicion Constraints in a Bacpropagation Learning Network," *Neural Computation*, Vol. 2, pp. 363-373, 1990

26. M. Jabri, Barry Flower, "Weight Perturbation: An Optimal Architecture and Leraning for Analog VLSI Feedforward and Recurrent Multilayer Networks", *IEEE Trans. on Neural Networks*, Vol.3. No.1. January 1992

27. P. W. Hollis. J. J. Paulos, "A Neural Netwrok Learning Algorith Tailored for VLSI Implementation", *IEEE Trans. on Neural Networks*, Vol.5 No.5 September 1994

28. B. Widrow and M. A. Lehr, "30 years of adaptive neural networks: Perceptron, Madaline, and Backpropagartion", *Proceedings of the IEEE*, Vol. 78, No. 9, September 1990.

29. L. W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," *MEMO ERL-M520, Univ. Calif. Berkeley, Electronic Research Lab,* May 1975.

30. Chua, L.O., Lin, P-M., *Computer aided analysis of electronic circuits: algorithms and computational techniques*, Prentice Hall, Inc., 1975, New Jersey.