SIMULATIONS OF ARTIFICIAL NEURAL NETWORK WITH MEMRISTIVE DEVICES

by

Thanh Thi Thanh Tran

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

Boise State University

December 2012

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Thanh Thi Thanh Tran

Thesis Title: Simulations of Artificial Neural Network with Memristive Devices

Date of Final Oral Examination: December 2012

The following individuals read and discussed the thesis submitted by student Thanh Thi Thanh Tran, and they evaluated her presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Elisa H. Barney Smith, Ph.D.	Chair, Supervisory Committee
Kristy A. Campbell, Ph.D.	Member, Supervisory Committee
Vishal Saxena, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Elisa H. Barney Smith, Ph.D., Chair, Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

Dedicated to my parents and Owen.

ACKNOWLEDGMENTS

I would like to thank Dr. Barney Smith who always encourages me to keep pushing my limit. This thesis would not have been possible without her devoting support and mentoring. I wish to thank Dr. Saxena for guiding me through some complex circuit designs. I also wish to thank Dr. Campbell for introducing me to the memristive devices and helping me to understand important concepts to be able to program the device.

I am thankful for having such tremendous support from family and my fiance Owen Ravella in this journey. I also would like to thank my friend Adrian Rothenbuhler for his great help in debugging and providing insights to my research. I also want to thank all the members of the Signal Processing Lab. They helped me overcome stress and frustrations, and kept me motivated to be able to attain my goal.

ABSTRACT

The memristor has been hypothesized to exist as the missing fourth basic circuit element since 1971 [1]. A memristive device is a new type of electrical device that behaves like a resistor, but can change and remember its internal resistance. This behavior makes memristive devices ideal for use as network weights, which will need to be adjusted as the network tries to acquire correct outputs through a learning process. Recent development of physical memristive-like devices has led to an interest in developing artificial neural networks with memristors.

In this thesis, a circuit for a single node network is designed to be re-configured into linearly separable problems: AND, NAND, OR, and NOR. This was done with fixed weight resistors, programming the memristive devices to pre-specified values, and finally learning of the resistances through the Madaline Rule II procedure. A network with multiple layers is able to solve difficult problems or recognize more complex patterns. To illustrate this, the XOR problem has been used as a benchmark for the multilayer neural network circuit. The circuit was designed and learning of the weight values was successfully shown.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ANN – Artificial Neural Network

MRII – Madaline Rule II

BP – Back Propagation

FPGA – Field-Programmable Gate Array

TLU – Threshold logic unit

LIST OF SYMBOLS

η	learning rate
x_i	i^{th} input feature vector
y_j	j^{th} output of a hidden node
z_k	k^{th} output vector
w_{ji}	weights of the hidden layers
w_{kj}	weights of the output layers
t_k	k^{th} target output
J	cost function
G	synapse gain
R_M	memristor resistance

CHAPTER 1

INTRODUCTION

Artificial neural networks (ANNs) themselves are nothing new; they are widely used in pattern recognition, but they are also mostly implemented in software. Hardware implementations of artificial neural network have been attempted with FPGAs [2–4]. Once a network is trained, the weight of each synapse stays fixed and can't be changed on the FPGA without recompiling. Using memristive devices or memristors [5], artificial neural networks can finally be fully implemented in hardware so that they can be retrained by changing the device resistances. Hardware-only ANNs are expected to learn much faster than software ANNs and consume less power. The goal of this thesis is to lay a foundation in building a hardware-only neural network by describing the process of converting the theoretical artificial neural network into a circuit neural network using memristors as synapse weights. The problem is memristors with today's technology cannot be programmed to an exact weight value, therefore, this thesis will also investigate how programming inaccuracy affects the network performance.

There are several important factors that contribute to building an ANN such as the number of hidden layers, the number of inputs, activation functions, training algorithms, and learning parameters... The objective of this thesis is to simulate the ANN with combinations of these factors not only to choose the best learning method but also to pick out more suitable components for the circuit design.

At first, a single node hardware-only ANN using look-up weight resistance will be attempted to configure some linearly separable operators such as AND, NAND, OR, and NOR. If this single node circuit design can realize these basic operators successfully, the fixed weight resistances will be replaced with memristor models and the network will be implemented with a learning algorithm to automatically find the correct resistance weights without a look-up table. This single node will then be duplicated to make a two layer neural network to realize XOR operator. XOR is a simple problem with only two inputs, but it is difficult for an ANN to solve because its outcomes are non-linearly separable. Because of this reason, XOR has been a famous benchmark for a lot of neural network performance, and therefore will be naturally used as a test for the hardware neural network designed in this thesis as well.

Although the scope of this thesis only covers the network simulations, it will attempt to build the first prototype of a single node ANN using discrete circuit components only. The design will be tested with the four basic threshold logic gates AND, NAND, OR, and NOR. The vision is to expand these ANNs into multiple layers with thousands of nodes to mimic how human brains function after gaining some knowledge of how a hardware ANN actually works. The remaining thesis has been organized as follow:

Chapter 2 provides a brief background about theoretical ANNs, threshold logic unit problems, memristors, and some computer simulation tools used in this thesis.

Chapter 3 evaluates two network training algorithms: back propagation and Madaline Rule II with different learning parameters. In the end, it will determine which one is better suited for the ANN.

Chapter 4 describes the circuit realization of the network. Most of the conversion between the theoretical ANNs and the circuit ANNs will be in this chapter, but it also covers the details on how to program a memristor.

Chapter 5 presents the network simulations of single node ANNs, then multilayer ANNs. Both start with fixed weight resistors, then replace fixed resistors with memristors programmed to target resistance values and finally a learning algorithm is implemented for the network to automatically learn the resistance based on input-output patterns.

Chapter 6 talks about some problems encountered, concludes the thesis, and discusses future work for this research.

CHAPTER 2

BACKGROUND INFORMATION

2.1 Artificial Neural Network

An artificial neural network (ANN) is a computational network that attempts to mimic how humans and animals process information through their nervous system cells. This system is formed by trillions of nerve cells exchanging electrical pulses across synapses. A neuron in a neural network is based on one of these nerve cells. In an ANN it is called a node. The strength between these synapses is implemented by the weights of the network. A neural network is constructed of one input layer, hidden layer(s), and one output layer. Figure 2.1 shows an example of such a multilayer neural network. The input feature vectors of the network are x_i, the hidden node outputs are y_j, and the network outputs are z_k. w_{ji} denotes the input to hidden layer weights at the hidden unit j, and w_{kj} denotes the hidden to output layer weights at the output unit k.

At each node, the weighted summation of the inputs is calculated by

$$net_j = w_{j0} + \sum_{i=1}^{c} w_{ji} x_i, \tag{2.1}$$

where w_{j0} is the weight of the bias input, which always stays constant (usually 1). The summation is passed through a nonlinear operator called the activation function $f(.)$ to yield an output. This output is then transmitted to other neurons and the process is repeated

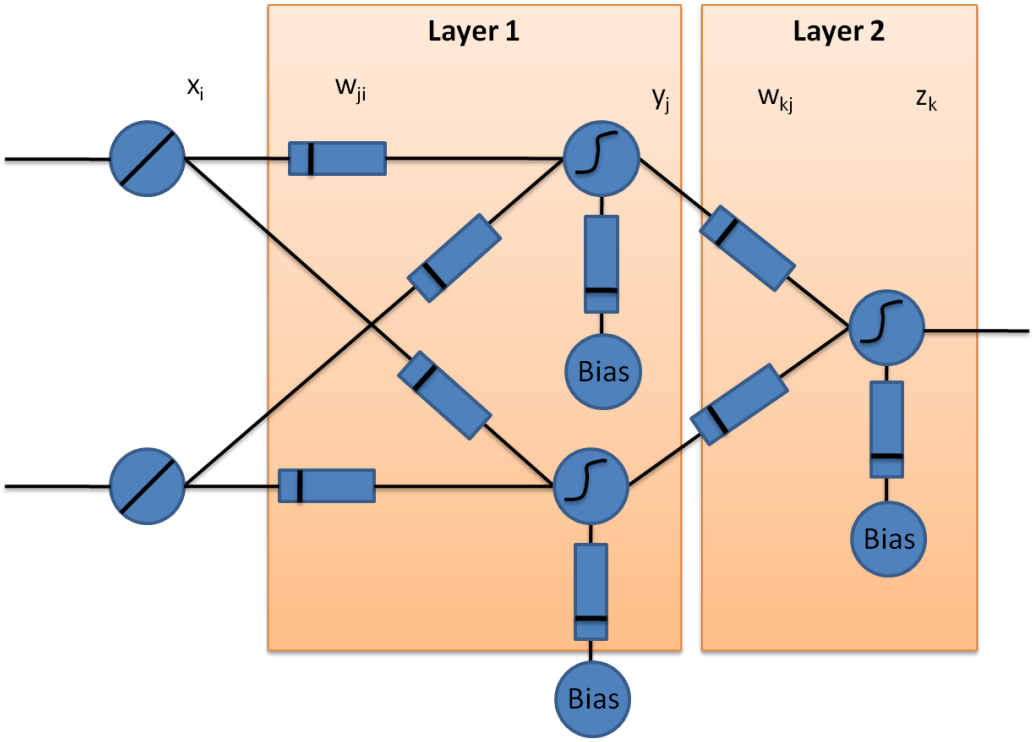


Figure 2.1: An example of a two layer neural network

until it reaches the output layer. In other words, the input of a cell arrives in the form of signals with different weight factors. These signals are built up and finally, the cell will discharge through the output to other cells. A perceptron is a single layer ANN. An Adaptive Linear Neuron or Adaline has the structure of a perceptron but with a hard-limitting activation function. If there is more than one Adaline layer in the ANN, the network is referred to as a Madaline, which stands for multiple Adaline. The output of each layer then acts as a new input feeding into the next layer. With Madalines, the output layer can have as many adaptive units as desired, which greatly increases the power of the network to realize arbitrary input-output relationships presented to it.

The term "feed-forward" ANN comes from the network structure where the output of one layer's neuron feeds forward into the next layer of neurons and never the other way.

The process is repeated at sequential layers until it reaches the output layer. The neurons are fully connected in that all the nodes from one layer are connected to all the nodes from the next layer. Using this feed forward mechanism, the neural network is able to take in the inputs and respond with some outputs. However, the feed forward structure alone is not enough because the output values will just be random numbers. To complete an artificial neural network, a learning algorithm is needed so that the network can learn to adjust its weights to produce desired outputs for the given inputs.

2.2 Network Learning Algorithms

Two ANN learning algorithms will be discussed in this thesis. They are back propagation and Madaline Rule II.

2.2.1 Back Propagation

The most commonly learning algorithm for ANNs is Back Propagation (BP) [6]. BP uses gradient descent to determine a weight combination that minimizes the cost function, which measures the error between the output $\mathbf{z_k}$ of the network and its target value $\mathbf{t_k}$ presented to the network during a training procedure, described by the equation

$$J = \frac{1}{2}(\mathbf{t_k} - \mathbf{z_k})^2. \tag{2.2}$$

To find a local minimum of a function using gradient descent, one takes a small step in the direction of the steepest descent until the local minimum is reached. Figure 2.2 shows an example of 1-D gradient descent. The ordinate is the value of the cost function, which is a function of one independent process variable drawn on the abscissa. After an initialization, a step down-hill is taken in the direction that has the steepest slope. In Figure 2.2, the

algorithm is initialized on the right, so it will take a step to the left until it gets to the local minimum. The size of the step is the learning rate of the algorithm and it gets smaller and smaller as the cost approaches the minimum to avoid overshoot.

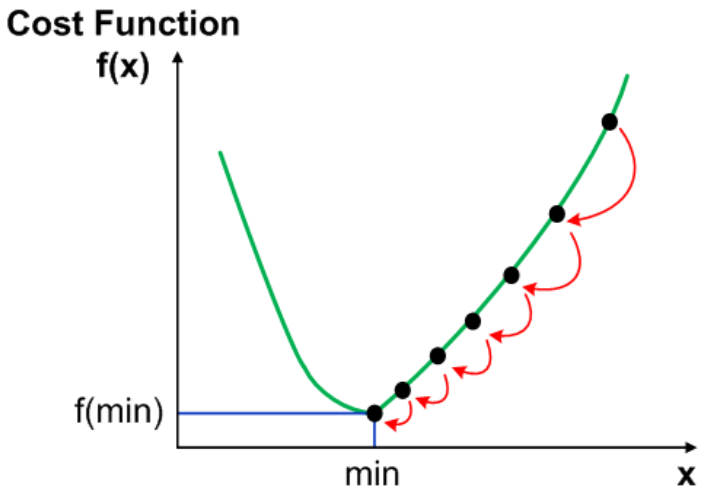


Figure 2.2: Gradient descent method to find the cost function minimum

BP requires a non-linear activation function and the most commonly used is the sigmoid function

$$y = \frac{1}{1+e^{-\lambda x}}, \tag{2.3}$$

where the parameter λ determines the steepness of the sigmoid function. BP will calculate the partial derivative of the cost function with respect to the weights to determine each weight's contribution to the error. Then the weights in each layer are adjusted simultaneously based on their contribution to the errors by using

$$\Delta w = -\eta \frac{\delta J}{\delta w}, \tag{2.4}$$

where η is the learning rate, which indicates the step size of the weight change. J is the cost function taken from Equation 2.2. The weight adjustments at the output layer are then

calculated by first finding the sensitivity δ of unit k using

$$\delta_k = -\frac{\delta J}{\delta net_k} = (t_k - z_k) f'(net_k), \tag{2.5}$$

$$f(net_k) = z_k = f(\sum_{j=1}^{c} w_{kj} y_j), \tag{2.6}$$

and

$$f'(net_k) = \lambda f(net_k)[1 - f(net_k)] = \lambda z_k (1 - z_k). \tag{2.7}$$

Taken together, they give the weight update results of the output layer:

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j. \tag{2.8}$$

Similarly, the same calculation is repeated within the hidden layer to find the sensitivity of unit j as follows:

$$\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj} \delta_k, \tag{2.9}$$

$$f(net_j) = y_j = f(\sum_{k=1}^{c} w_{ji} x_i), \tag{2.10}$$

and

$$f'(net_j) = \lambda f(net_j)[1 - f(net_j)] = \lambda y_j (1 - y_j). \tag{2.11}$$

The weight update results for the hidden layer are then calculated by

$$\Delta w_{ji} = \eta x_i \delta_j = \eta \left(\sum_{k=1}^{c} \delta_{kj} \right) f'(net_j) x_i = \eta \left[\sum_{k=1}^{c} w_{kj} (t_k - z_k) \right] f'(net_j) x_i. \tag{2.12}$$

The disadvantage of back propagation is that it takes a lot of iterations to train a pattern, in addition to finding the appropriate learning rate η_{opt} so that the cost function will converge.

If $\eta < \eta_{opt}$, convergence is assured, but the training process can be needlessly slow. If $\eta_{opt} < \eta < 2\eta_{opt}$, the system will oscillate but nevertheless converge, and the training process is also needlessly slow. If $\eta > 2\eta_{opt}$, the system will diverge [7].

2.2.2 Madaline Rule II

An alternative training algorithm is Madaline Rule II (MRII), which was originally developed in 1988 by Winter and Widrow [8]. MRII expects the activation function to be a hard-limiting function. It uses the principle of minimal disturbance to train the network. The flow chart in Figure 2.3 describes how MRII works.

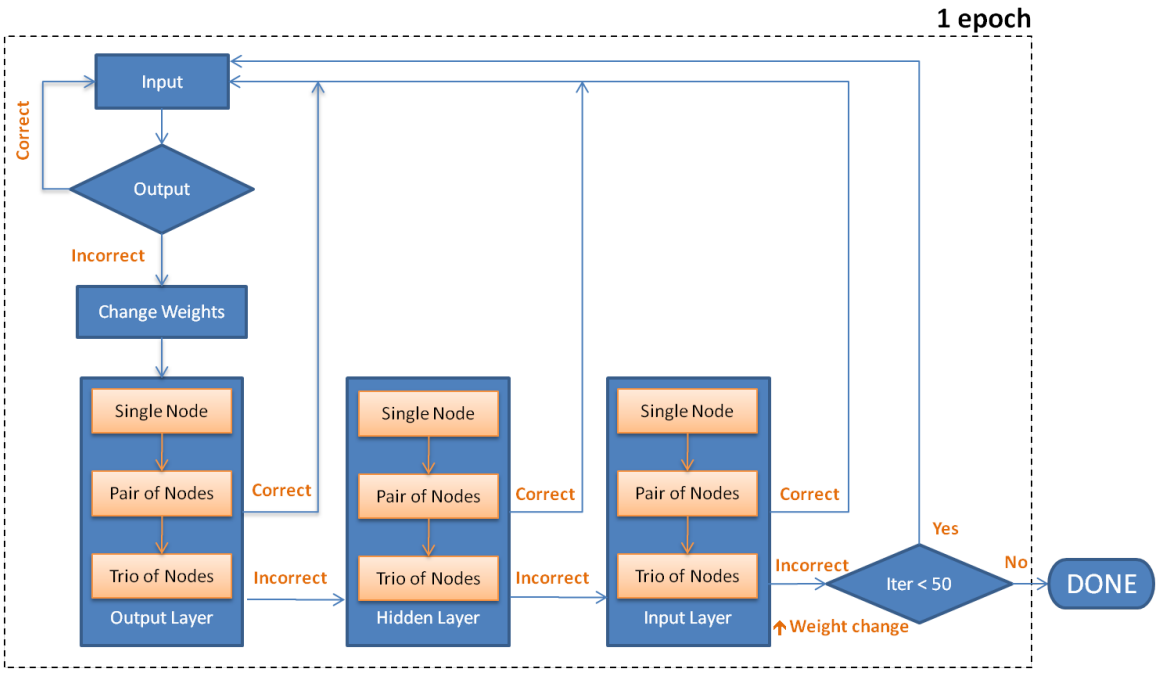


Figure 2.3: Madaline Rule II algorithm

When an input is fed into the network and it responds correctly to the training values, no weight adaptation is done, otherwise, it randomly changes the weights at the input of the node that has the least confidence in its output. After the weights are changed, the network

is tested to see if the change reduced the classification error rate. If this is the case, the change is kept, otherwise, the weight is reset to the original, then another set of weights is changed. If the error is not reduced, this search continues to weights at nodes further from their switching threshold. If there is still an error in classification, a pair of nodes is changed at the same time, then a triplet of nodes... If changing weights in the output layer does not reduce the classification error, the MRII algorithm will move to the hidden layer, working its way deeper back into the network until it reaches the input layer. The amount by which the weights are changed can be a function of the amount of residual errors in the network or a random amount. If after a certain number of iterations the weight change does not increase the overall network performance, the weight change will be increased by a factor called the growth rate. The maximum number of iterations used in this thesis for MRII is 50. After each training epoch of 50 iterations, the network training will be declared either a success or a failure.

2.3 Threshold Logic Unit Problems

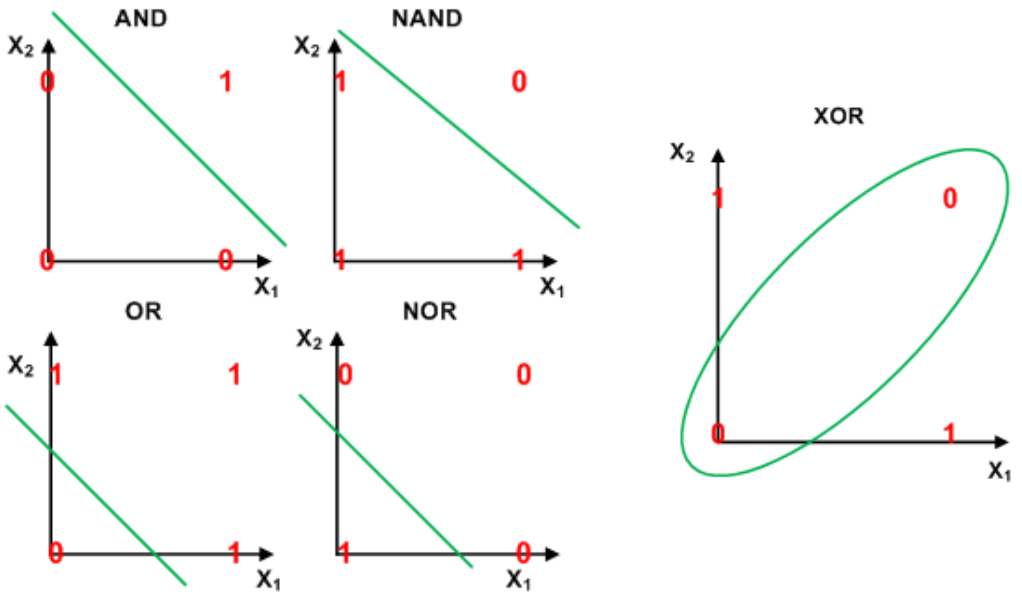
In this thesis, threshold logic unit problems will be used as a test bench to measure the performance of the ANN built with memistors. There are 16 possible binary logic operations that can be created with two binary inputs X_1 and X_2. These are shown in Table 2.1. Except for XOR and XNOR, the 14 remaining logic operations are linearly separable. Figure 2.4 shows an example of the difference between linearly separable and non-linearly separable problems. The AND, NAND, OR, and NOR operators are linearly separable because the input set can be separated by a single straight line decision boundary to group like outputs. In contrast, XOR is not linearly separable because the set cannot be separated by a single straight line. The 16 logic operators are separated into these two categories

because the linearly separable problems can be realized using just a single layer network with a hard-limiting activation function, while non-linearly separable problems require at least a two layer network.

Table 2.1: 16 possible binary logic operations

INPUTS	X_1	0	0	1	1
	X_2	0	1	0	1
OUTPUTS	FALSE	0	0	0	0
	A AND B	0	0	0	1
	A DOESN'T IMPLY B	0	0	1	0
	TRUE A	0	0	1	1
	A IS NOT IMPLIED BY B	0	1	0	0
	TRUE B	0	1	0	1
	A XOR B	0	1	1	0
	A OR B	0	1	1	1
	A NOR B	1	0	0	0
	A XNOR B	1	0	0	1
	NOT B	1	0	1	0
	A IS IMPLIED BY B	1	0	1	1
	NOT A	1	1	0	0
	A IMPLIES B	1	1	0	1
	A NAND B	1	1	1	0
	TRUE	1	1	1	1

There are multiple possible weight values that will correctly realize most problems. Figure 2.4(a) shows some examples of linearly separable problems: the AND, NAND, OR, and NOR operators. These operators have high tolerance in decision boundaries, meaning that there are many ways to draw a straight line to separate the inputs based on the output result. Therefore, it is suspected that the weights for these problems will have a high tolerance as well. In theory, these weight values can be infinite because the output logic is thresholded at 0. However, the purpose of the thesis is to build a practical circuit for an ANN, and the hardware can only take in a limited gain from the weights. Therefore, for this discussion, these weights are limited to the values between [-10,10]. This range



(a) Linearly separable problems: AND, NAND, OR, NOR operators

(b) Non-linearly separable problem: XOR operator

Figure 2.4: The linearly separable vs. non-linearly separable problems and their decision boundaries

can illustrate the four logic operations used in this thesis, and also is logistic for designing hardware ANN circuits.

2.3.1 Linearly Separable TLUs

To find a sample set of weights that works for linearly separable problems, an exhaustive search was performed to find the correct weights in the weight space. Because these problems are linearly separable, a single node with two inputs, one bias input, one output and three weights w_0, w_1, w_2 as seen in Figure 2.5 can be used. The output of the feed-forward network was calculated for each weight combination. For every operation, a set of weights is said to be qualified for that logic operation if and only if the network produces correct logic outputs for all four input pairs (-1,-1), (-1,1), (1,-1), and (1,1). The result is shown in Figure 2.6. The qualified regions for the four logic operations AND,

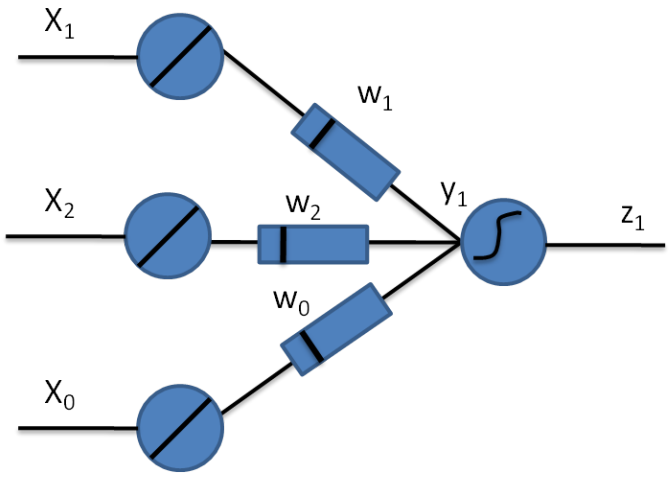


Figure 2.5: Adaline Structure

OR, NAND, and NOR are highlighted. All the weight combinations within each region can classify the corresponding logic gate correctly. From the weight space in Figure 2.6, a working set of weights was manually picked and is displayed in Table 2.2.

Table 2.2: A sample set of weights for linearly separable problems extracted from the weight space in Figure 2.6

	w_1	w_2	w_0
NAND	-0.5	-0.7	0.5
NOR	-0.5	-0.7	-0.5
AND	0.3	0.6	-0.5
OR	0.3	0.6	0.5

A weight for the bias node further from 0 provides better tolerance. Another interesting observation is that one can change the logic from an AND to an OR, or a NAND to a NOR just by switching the sign of the bias weight. Last but not least, this insight explains why most of the threshold logic ANNs use bias inputs. Looking at the plane $w_0 = 0$ in Figure 2.6, the number of weight combinations that do not require a bias weight are distributed discontiuously on a diagonal line. Therefore, it is possible to have a set of weights that do

not require a bias node for these ANNs, however, the probability of finding them will drop dramatically without using the bias node.

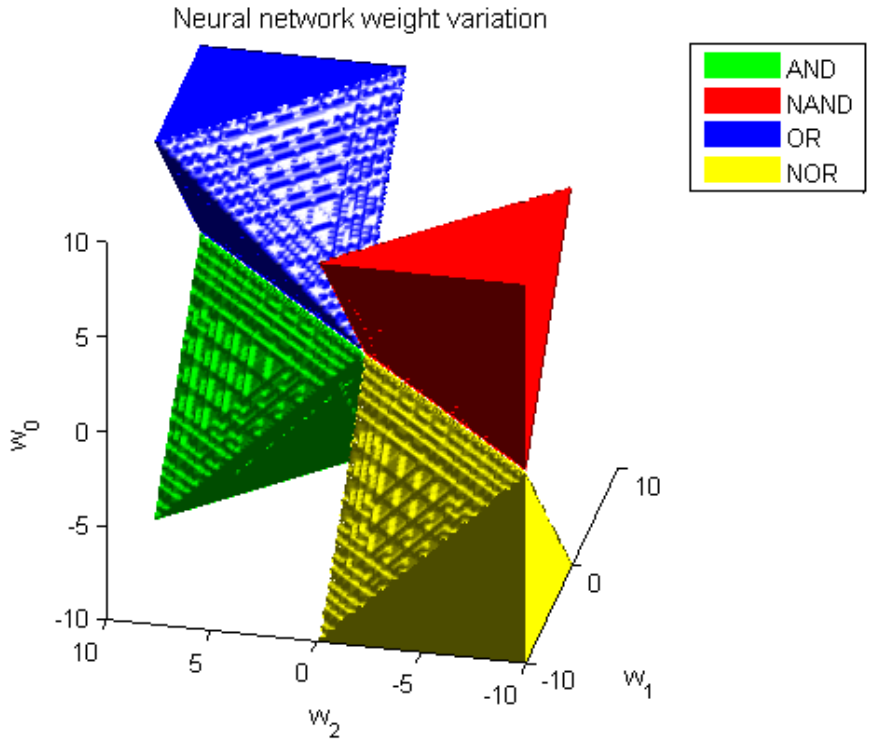


Figure 2.6: Weight variation for the AND, NAND, OR, and NOR operators

2.3.2 Non-Linearly Separable TLUs

XOR and XNOR behave differently than the other operations because they are not linearly separable. They require a second layer with a total of 9 weights, 6 more than the previous one layer ANN structure, Figure 2.1. Because of its simplicity and extremely non-linear decision boundary, XOR is famous for being a benchmark for a lot of neural network prototypes. Table 2.3 shows a sample weight combination that works for the XOR operator.

Table 2.3: A sample set of XOR weight values

	w_1	w_2	w_0
hidden weights	-0.6485	-0.4646	0.6592
	-1.9410	-1.5920	-1.2104
output weights	0.3732	-0.4063	-0.2250

2.4 Memristors

The memristor was hypothesized to exist by Chua in 1971 [1]. Since then, several other groups have also made memristive-like devices, those that simply show variable resistance and hysteresis in the device IV curve, but have not shown explicitly to exhibit the properties of memristance as proposed by Chua [1] including Dr. Campbell's group at Boise State University [9], Strukov et al. from Hewlett-Packard [10]. There has been a debate about whether these devices are called memristors and in 2011, Chua explicitly suggested that all the memristive devices that have a common fingerprint consisting of a dense continuum of pinched hysteresis loops whose area decreases with the frequency are called memristor [5].

A memristor is a passive device that behaves like a resistor, but whose history influences its resistance state. Unlike potentiometers, a memristor is an integrated circuit device and is programmable. The resistance of the memristors used in this thesis can be changed through a writing or erasing procedure, which is only effective if the applied voltages exceed a certain threshold, otherwise the memristor will act as a linear resistor. Applying a positive voltage above the threshold causes the memristor to write, which means that the device decreases its intrinsic resistance. In contrast, applying a negative voltage causes the memristor to erase, which means that the intrinsic resistance increases. This behavior makes memristors ideal for use as ANN weights, which will need to be adjusted as the network tries to acquire correct outputs through its learning process. Often, a short pulse is used to write or erase a memristor so that the it is not put under a constant stress.

Since discovery, important applications of memristors include ultradense, semi-non-volatile memories and learning networks that require a synapse-like function [10]. Recently, Adhikari et al. has proposed a neural network hardware architecture using memristor bridge synapse to solve the problem of non-volatile weight storage to recognize images of cars from behind [11].

2.5 Computer Simulation Tools

In preparation for building the actual hardware, the neural network was first simulated on the computer. The two software tools used in this thesis are Cadence and MATLAB/Simulink. The ANN discrete circuit is built in Cadence, and Simulink controls the learning algorithm of the network. Section 2.5.1 will describe the process of interfacing Cadence with Simulink. Section 2.5.2 will talk about the Cadence memristor model.

2.5.1 Cadence Simulink Cosimulation

The Cadence Virtuoso AMS Designer Simulator provides a bidirectional link between MATLAB Simulink and Cadence, enabling cosimulation and early verification of ANN board design that combines both analog and digital components. The interface is implemented by either the Cadence coupler block placed in the Simulink model or the AMS coupler block placed in the Virtuoso schematic to represent the Simulink model. These two coupler modules give the user a lot of flexibility to configure input and output parameters. There are three cosimulating modes: Cadence initiates the simulation and waits for Simulink to run, Simulink initiates the simulation and waits for Cadence to run, or both can be running at the same time.

Assuming that the standalone simulations work correctly in MATLAB/Simulink and Cadence SpectreRF, these steps are required to set up the cosimulation:

1. Insert and configure the SpectreRF in the Simulink schematic.

2. Setup the netlist to adopt the SpectreRF and MATLAB cosimulation.

3. Run the cosimulation.

2.5.2 Cadence Memristor Model

The memristor model representing the weights used in this thesis was provided by Dr. Vishal Saxena based on "A memristor device model" by C. Yakopcic et al. [12]. The memristor model code written in VerilogA is included in Listing 2.1. The "state input" pin "*xd*" is used to sneak into the memristor state. The state input value is inversely proportional to the memristor resistance. Figure 2.7 shows the device resistance varies from 120Ω to 170KΩ as the initial value of the state input pin varies from 0.001 to 1. Although, the model for this memristor can operate at the high resistances region (MΩ), only the low resistance region (KΩ) is purposely chosen in this thesis to avoid other discrete circuit components having to work with a large dynamic range.

Figure 2.8 shows a basic experimental set up to characterize the memristor model. A series of positive and negative pulses with an amplitude of 200 mV and 1 μs pulse width were applied to the model. Figure 2.9 shows how the memristor state responds to the writing and erasing processes. Applying a positive pulse causes the meristor to write, which decreases the memristor resistance, meaning that the "state" will be increasing. In contrast, applying a negative pulse causes the memristor to erase, which increases the memristor resistance, meaning that the "state" will be decreasing. This memristive model can be

updated in the future to match with the actual memristors fabricated and packaged by Dr. Campbell's group at Boise State University.

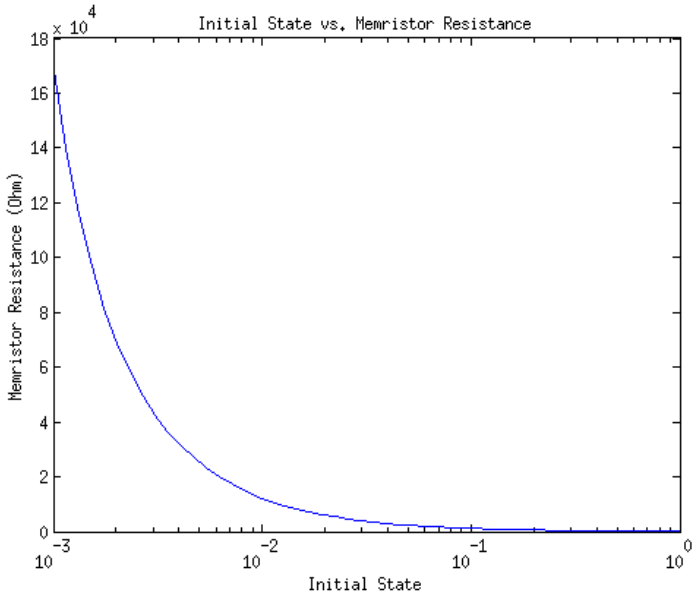


Figure 2.7: Device resistance based on initial state

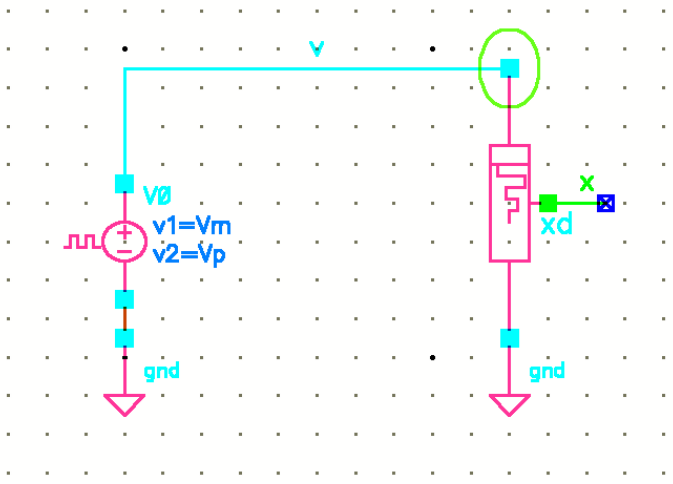
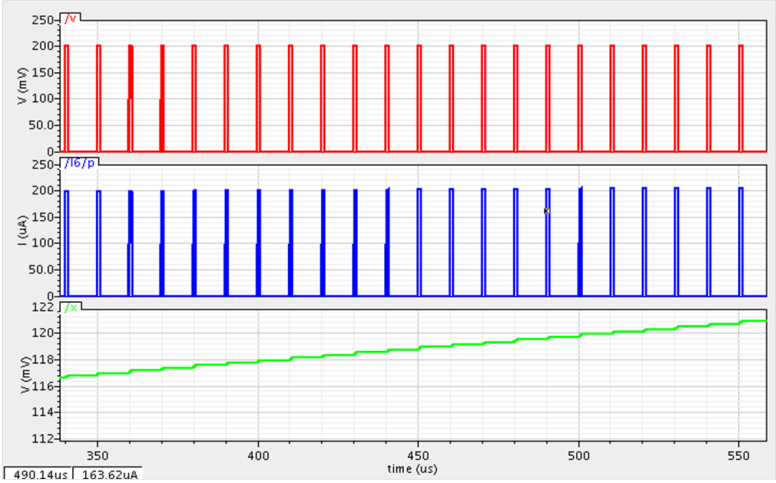
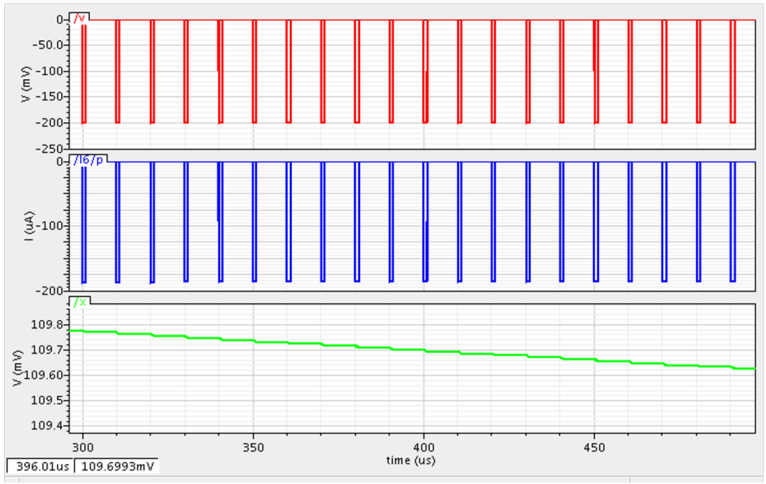


Figure 2.8: Basic experiment to characterize the memristor behavior



- V_{in}= Positive pulses of 200mV 1µm width
- Red – input voltage, Blue – current through the device
- Green – movement of the device state 'x' (x_0=0.1)

(a) writing process, decreasing the device resistance



- V_{in}= Negative pulses -200mV of 1µm width
- Red – input voltage, Blue – current through the device
- Green – movement of the device state 'x' (x_0=0.1)

(b) Erasing process, increasing the device resistance

Figure 2.9: Movement of the memristor state in response to the writing and erasing process

```
1  //////////////////////////////////////////////
2  // VerilogA model for BSU Memristor
3  // Version: 1.0
4  // vishalsaxena@boisestate.edu
5  //
6  // Boise State University
7  // ECE Dept. March 2012
8  //////////////////////////////////////////////
9
10 `include ``constants.vams"
11 `include ``disciplines.vams"
12
13 module Memristor_BSU_v10(p,n,xin,xd);
14 inout p, n;
15 inout xin;
16 output xd;
17
18 electrical p, n, xin, xd;
19
20 // Device parameters
21 parameter real Vp = 0.16;
22 parameter real Vn = 0.15;
23 parameter real Ap = 4000;
24 parameter real An = 4000;
```

```
25 parameter real xp = 0.3;
26 parameter real xn = 0.5;
27 parameter real alpha_p = 1;
28 parameter real alpha_n = 5;
29 parameter real a1 = 0.17;
30 parameter real a2 = 0.17;
31 parameter real b = 0.05;
32 parameter real t0 = 2e-9;
33 parameter real dt = 0.1e-9;
34
35 // State variable and parameters
36 real x0;
37 real x, g, f;
38 real assert;
39 real wp, wn;
40 branch (p, n)  memr;
41
42 // Initial conditions
43 analog begin
44         @(initial_step) begin
45                 f=1; g=0;
46                 x0=V(xin); assert=0;
47         end
48
49         // Using timer to make it work with Simulink Cosim
```

```
50          // Simulink inputs arrive late into spectre (use t0)
51          @(timer(t0)) begin
52                  x0 = V(xin);
53                  assert=1;
54          end
55
56          // Restart the integrator at time = t0+dt
57          @(timer(t0+dt)) begin
58                  assert=0;
59          end
60
61          // find g(V(t))
62          if (V(memr) > Vp)
63                  g = Ap*(exp(V(memr))-exp(Vp));
64                  else if (V(p,n) < -Vn)
65                  g = -An*(exp(-V(memr))-exp(Vn));
66                  else g = 0;
67
68          // find f(x)
69          if (V(memr)>0)
70                  if (x>xp) begin
71                          wp = (xp-x)/(1-xp) + 1;
72                          f = exp(-alpha_p*(x-xp))*wp;
73                  end
74                  else f = 1;
```

```
75          else
76                  if (x<=1-xn) begin
77                          wn = x/(1-xn);
78                          f = exp(alpha_n*(x+xn-1))*wn;
79                  end
80                  else f = 1;
81
82          // State update (integrator)
83          x = idt(g*f, x0, assert);
84
85          // Assign outputs
86          I(p,n) <+ a1*x*sinh(b*V(memr));
87          V(xd) <+ x; // Normalized state
88  end
89  endmodule
```

Listing 2.1: VerilogA Memristor Model Code

CHAPTER 3

EVALUATION OF NETWORK TRAINING ALGORITHM

In simulation, every memristor model can be programmed correctly to a specific resistance value within a certain tolerance. However, in reality there is a process variation that causes the memristors to respond inconsistently to the same programming pulses. Progress is being made toward developing a more precise way to program the memristors by the neuromorphic group at Boise State University. In the mean time, experiments were run to evaluate the effect of imprecision in weight changes of memristors in training an XOR problem. Although, the network will be built first to recognize linearly separable problems, the classic XOR operator is chosen in this evaluation to get results that are applicable to a wider range of problems. If the network can train an XOR operator, it is more likely to be able to train other linearly separable problems also.

A small amount of Gaussian noise with variance σ was added to the basic training rule

$$w_{kj}^{(n+1)} = w_{kj}^{(n)} + \eta \delta_k y_j + N_\sigma, \tag{3.1}$$

$$w_{ji}^{(n+1)} = w_{ji}^{(n)} + \eta \delta_j x_i + N_\sigma. \tag{3.2}$$

Two training algorithms were evaluated: back propagation and Madaline Rule II. Each algorithm will be run with different learning parameters to determine which one is better suited to be implemented in the hardware ANN. Figure 3.1 shows an overview of the

experiments in this section. Again, BP will be evaluated with three different steepness of the sigmoid functions and MRII with varying the growth rate and number of epochs.

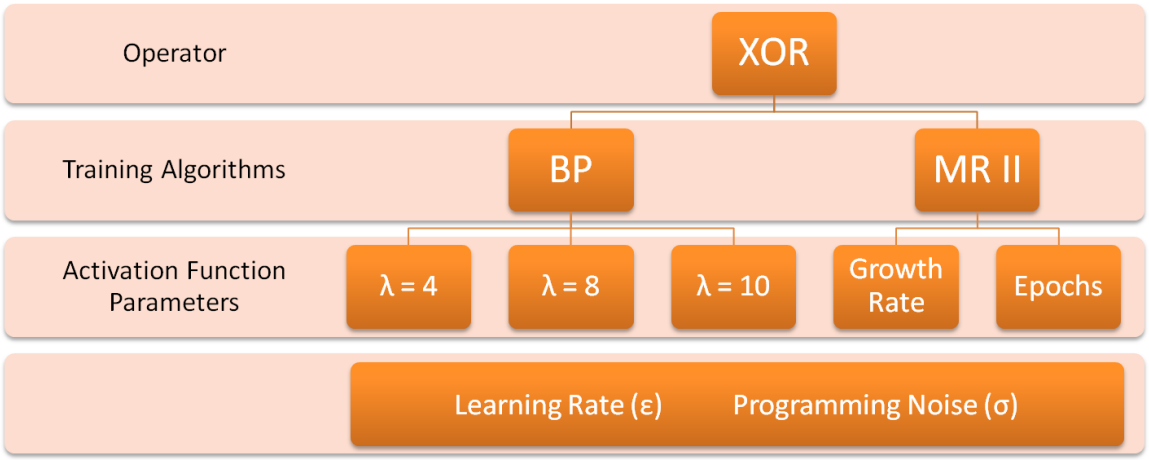


Figure 3.1: Experiments to evaluate how programming noise affects the network performance with different learning parameters

3.1 Back Propagation

With back propagation, besides the learning parameter η, the activation function parameter λ is probably the most important factor that determines the network learning ability. Three different sigmoidal activation functions were used based on different values of λ in Equation 2.3. The larger λ makes the curve steeper. For each value of λ a preliminary set of experiments were run to choose a suitable learning rate η. A larger λ requires a smaller η. In this thesis, BP will be explored with different learning rate and weight noise and $\lambda = \{4, 8, 10\}$. Each experiment was run 100 times and the learning success rate was recorded in Figure 3.2. The maximum number of iterations allowed for BP in these experiments is 10,000.

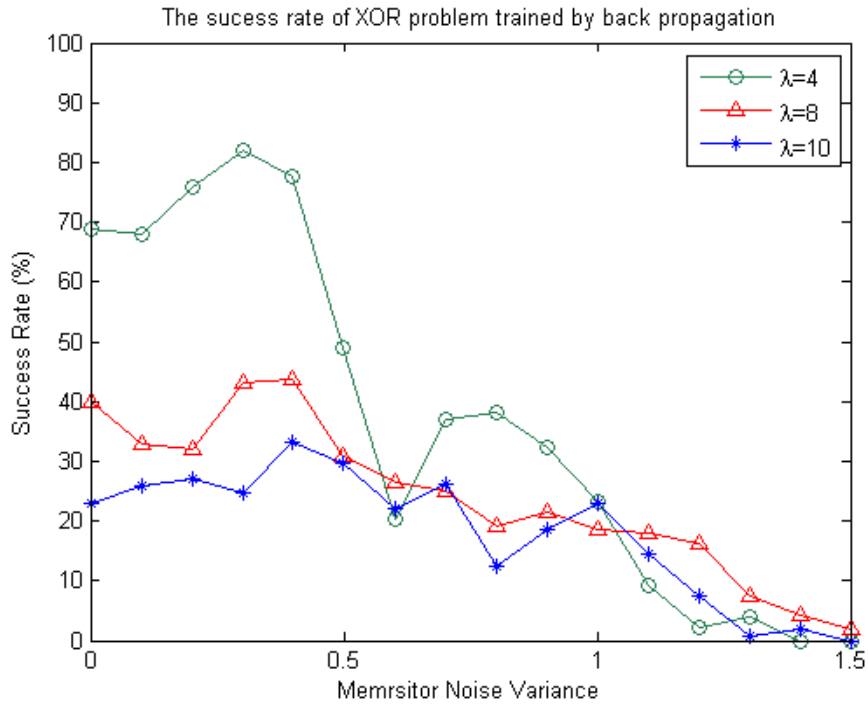


Figure 3.2: Training performance for XOR ANN with back propagation training versus weight change size

Training an XOR sometimes does not always converge to a correct solution after a lot of BP iterations. It occasionally gets stuck in a local minimum. To fix this problem, a momentum factor can be added to the algorithm to help BP get out whenever it is stuck in a local minimum. However, it is beyond the scope of this thesis. When moderate amounts of additive Gaussian noise were included in the weight changes ($\sigma < 0.4$), the percentage of instances when the network properly trained increases as shown in Figure 3.2. This is because, just like momentum, a small amount of noise is able to kick BP out of a local minimum. The average number of iterations needed also decreases as seen in Figure 3.3. Therefore, a moderate imprecision in memristive weights would actually help the network train better. This is not a surprise, the effect of imprecision in the weights has been explored and it was determined that it is not detrimental to the ANN operation [13]. However, as the

amount of noise increases past 0.5, the training performance declines dramatically. When the noise variance reaches 1, the network becomes unstable and fails to solve the given problem. In addition, as λ increases leading to a steeper sigmoid function, the training is less effective and eventually fails when the sigmoid becomes a step function. This is understandable because BP calculates the weight changes based on the derivative of the activation function. Thus, to explore the step activation function, the Madaline Rule II will be employed.

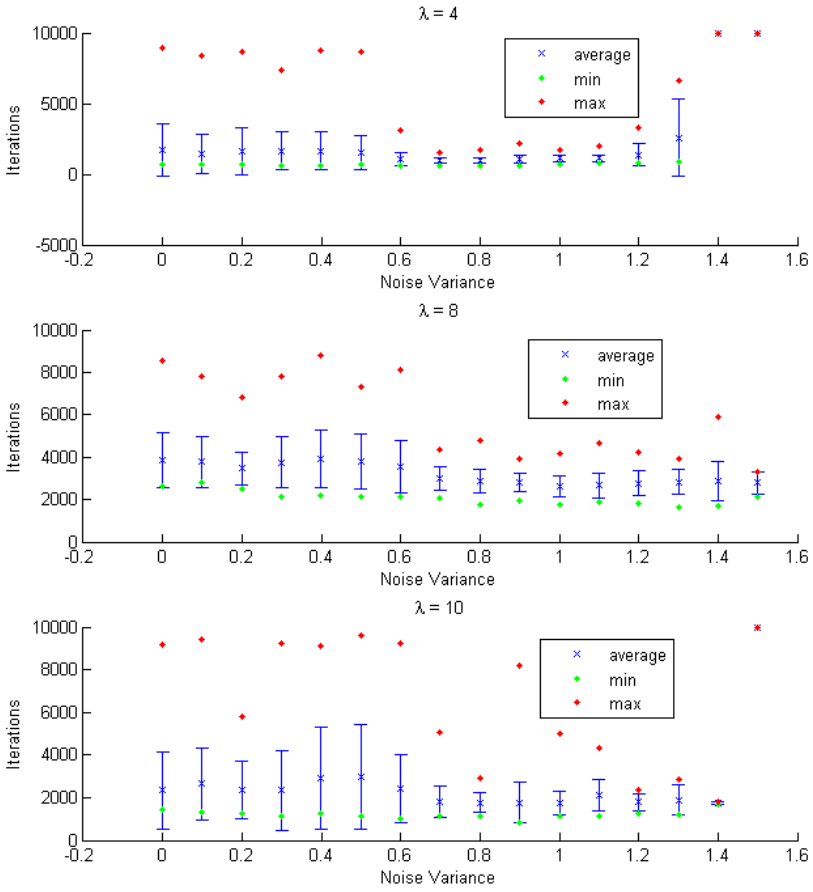


Figure 3.3: Number of iterations versus weight change size for BP. Error bars indicate variance of results about the mean.

3.2 Madaline Rule II

Since MRII randomly perturbs network weights rather than calculating the weight changes based on the error gradient, the effect of weight setting noise is not relevant, but analysis of the size of the weight change is relevant. For each experiment the network was trained 100 times with the weight randomly initialized and using different starting seeds in the random number generator. The standard deviation of the random perturbation was gradually increased in Figure 3.4. The success rates for training within the maximum of 50 iterations allowed increased as the perturbation size was increased, but reached a maximum at 25%.

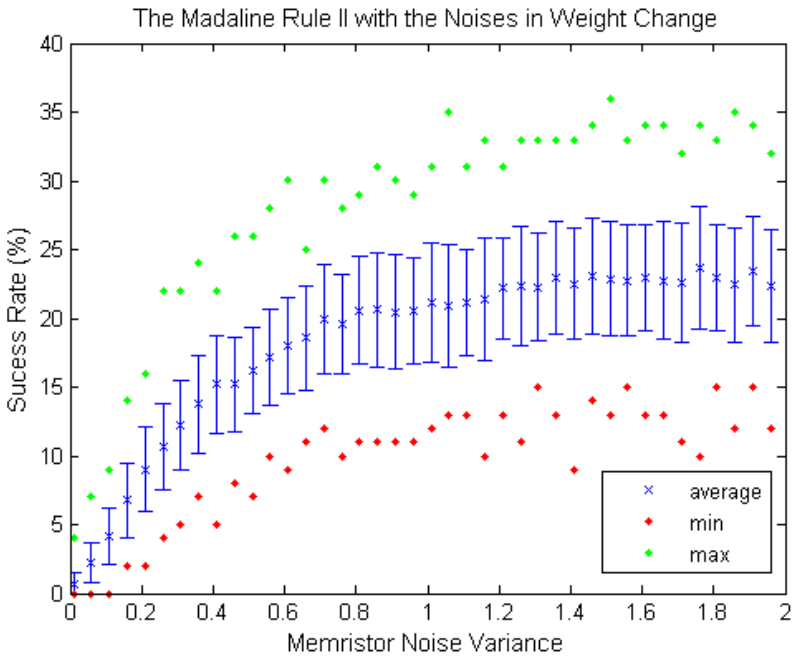


Figure 3.4: Training performance for XOR ANN with Madaline Rule II training versus weight change size

If the first weight change causes an overall network improvement, but leads to setting weights that cannot be changed to the weight values that are eventually needed, MRII will fail. The XOR network in this thesis only has three nodes that the MRII algorithm

can almost exhaustively search for at which node the weights should be changed. If the training is not proceeding well initially, the network is likely to fail within 50 iterations. A solution for this is to make the weight change larger so the network will jump to a significantly different decision boundary. To try this, the size of the change in weights was set to increase by a growth rate if all the possibilities were tried and no increase in network performance was noted. Once the error rate decreases, the weight change was returned to the original weight change amount. Figure 3.5 shows the success rate versus the base weight change standard deviation and weight growth rate. As the growth rate increases, the training is accomplished more regularly, but not so much compared to the base algorithm after the grow rate is larger than 3.

Because MRII has a low learning success rate to begin with, a revised training procedure was developed in such a way that if the network does not train after 50 iterations, the weights are reset and the MRII process is rerun. Each time the weights are reset, a new epoch is launched. This process is repeated up to a certain number of epochs. Figure 3.6 shows that the modified version of MRII was able to achieve the success training rate of 100% within 20 epochs.

3.3 Conclusion

Training with MRII is much faster than BP even with having to reset the network (50 MRII iterations x 20 epochs vs. 10,000 BP iterations). Plus, MRII depends on the randomness of the weight change, so it is not sensitive to noise, so it is well suited for using memristors as weights. In addition MRII is designed for networks that use the hard-limiting activation function, which is already available for implementing in circuit ANNs as a comparator. BP might still be considered in the future for different types of problems when a chip is

developed where a non-linear activation function can be realized, but the rest of this thesis will be focusing on implementing the network using the MRII learning algorithm.

Figure 3.5: MRII with different variable perturbation growth rate and learning rate has a very low learning success rate (maximum at 25%).

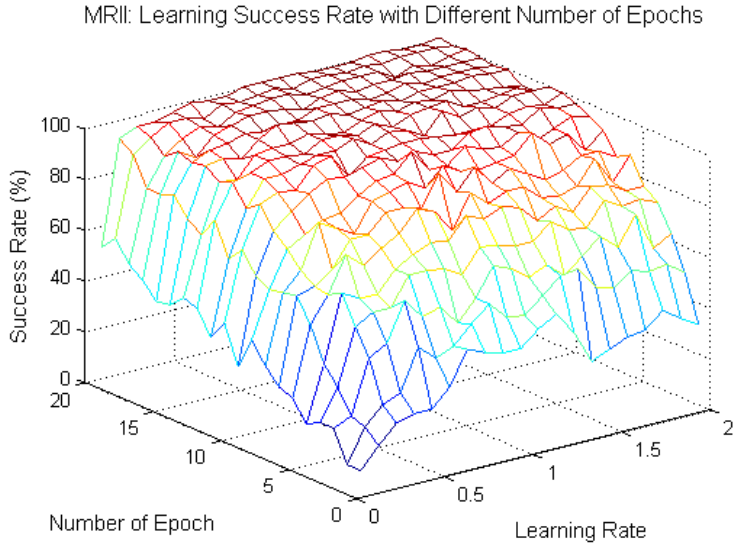


Figure 3.6: The success rate was increased to 100% with the new method of resetting the network and retraining for multiple epochs.

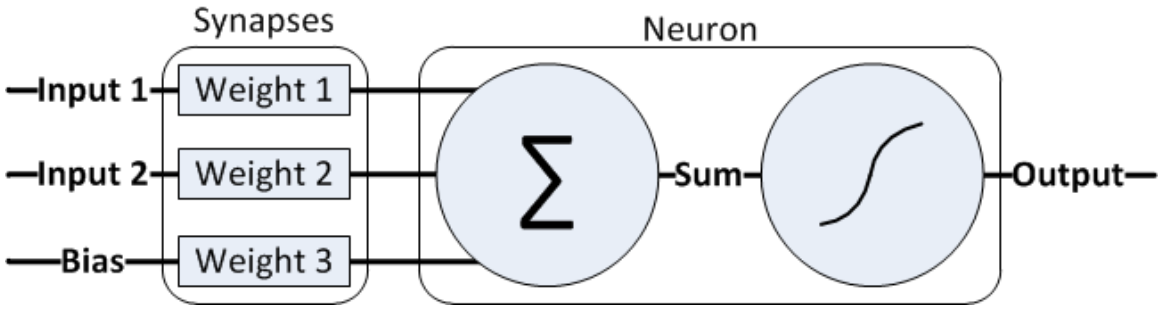
CHAPTER 4

CIRCUIT REALIZATION OF NETWORKS

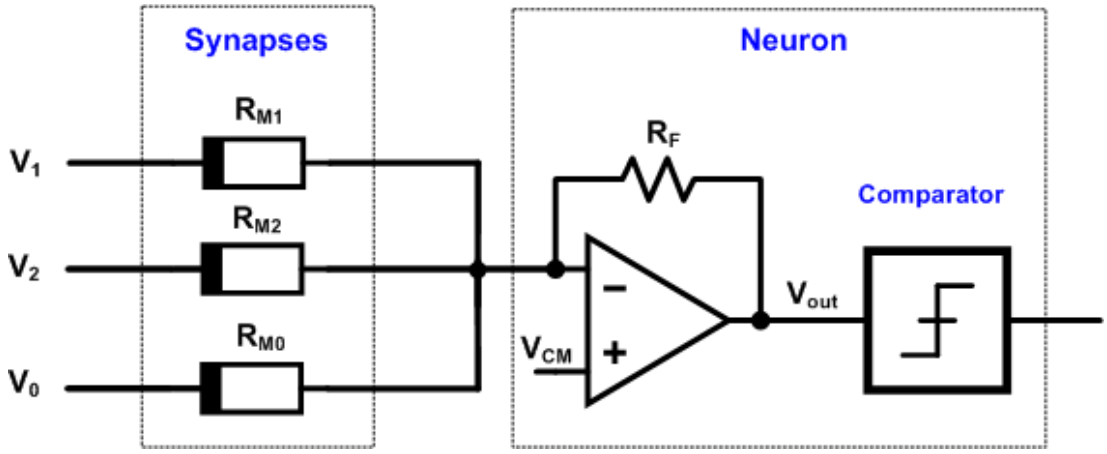
The goal of this thesis is to explore and develop a method to build a hardware-only ANN with memristors through simulations in Cadence and MATLAB. The circuit that realizes a single layer network will be first introduced in Section 4.1, then later it will be expanded into multiple layers to be tested with an XOR problem. The weights are the memristor conductance, which are by default always positive, however, for ANNs we desire the weights to sometimes be negative. Section 4.1 will show the circuit design to accommodate this problem. Section 4.2 will describe the process of converting theoretical ANNs into circuit ANNs. Section 4.3 will describe the method of programming a single memristor. Lastly, Section 4.4 will describe the complete structure of one synapse.

4.1 The ANN Circuit

Figure 4.1 shows the traditional view of a neuron and its equivalent circuit structure. Three memristors R_{M1}, R_{M2}, and R_{M0} are used as the two input weights and the bias weights respectively. The weighted summation is replaced by an op-amp to sum up all the input currents. The summation current will be passed through a comparator acting as an activation function. The weights in this case are the conductance of the memristors, clearly there is a problem that they do not allow for negative weights. A circuit has been developed to overcome this problem and is shown in Figure 4.2. R_0 and R_1 have the same



(a) Traditional view



(b) Circuit view

Figure 4.1: A neuron with two input synapses and one bias synapse

resistance value so that they invert the positive input voltage coming out of the op-amp. The output current is the summation of the two currents going through R_M and R_N as follow

$$I = \frac{V_{IN}}{R_M} + \frac{-V_{IN}}{R_N} \tag{4.1}$$

The negative weight problem is overcome by adjusting the resistance of the memristor so the total output current is negative. This current is then amplified through the summation op-amp to provide negative output voltage. An appropriate value will be chosen later for R_N so that the gain of this circuit is within the desired weight limit. The operating range of the memristors chosen in this thesis is between 10 KΩ and 100 KΩ. Adjusting the device

resistance R_M within this range will swing the output voltage from a negative to a positive voltage, and therefore will solve the negative weight problem.

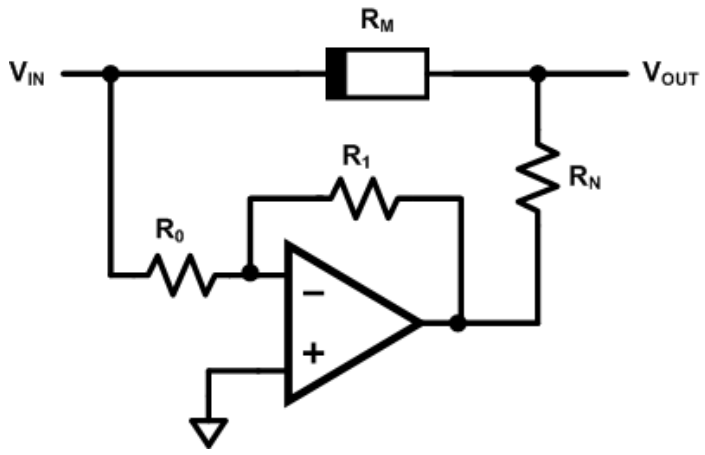


Figure 4.2: One synapse with negative weight

4.2 Converting Weights into Resistance Values

In the process of building the actual hardware for the network, the theoretical weight values for the network in Section 2.3 need to be converted to resistance values. The calculations to achieve this were developed at Boise State University jointly by Adrian Rothenbuhler and myself. Figure 4.3 shows the equivalent circuit of the theoretical synapses and their summation in Figure 4.1. Within the synapse, the voltage at the negative terminal V_N is calculated by

$$V_N = V_{CM} - R_2 * \frac{V_{IN} - V_{CM}}{R_1} = 2V_{CM} - V_{IN}. \tag{4.2}$$

The current going to the summation op-amp is calculated by

$$I_S = I_1 + I_2 = \frac{V_{CM} - V_{IN}}{R_M} + \frac{V_N - V_{CM}}{R_N}. \tag{4.3}$$

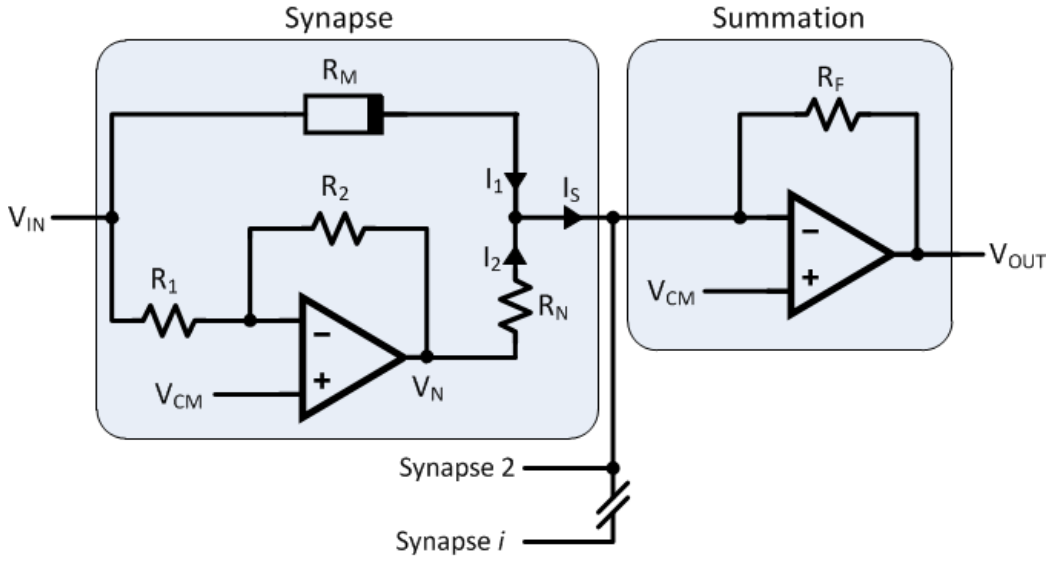


Figure 4.3: Synapse supporting negative weights feeding into the summation circuit of the neuron

Using the summation current calculated in Equation 4.3, the output voltage of the circuit can be calculated using

$$V_{OUT} = V_{CM} - R_F * \sum_{i=1} I_S. \tag{4.4}$$

The gain G of the circuit is calculated by

$$G = \frac{V_{OUT} - V_{CM}}{V_{IN} - V_{CM}} = R_F(\frac{1}{R_N} - \frac{1}{R_M}). \tag{4.5}$$

Knowing G, the memristor resistance can be calculated using

$$R_M = \frac{R_F R_N}{R_F - G * R_N}. \tag{4.6}$$

In order to convert the weight into actual resistance for R_M, R_N and R_F have to be calculated first by reversing Equations 4.5 and 4.6 as

$$R_N = R_{M_{High}} - \frac{R_{M_{High}} G_{High}}{R_{M_{High}} G_{High} - R_{M_{Low}} H_{Low}} (R_{M_{High}} - R_{M_{Low}}), \tag{4.7}$$

$$R_F = \frac{R_N (R_{M_{High}} G_{High} - R_{M_{Low}} G_{Low})}{R_{M_{High}} - R_{M_{Low}}}, \tag{4.8}$$

where G_{High} and G_{Low} are the highest and lowest weight of the ANN, while $R_{M_{High}}$ and $R_{M_{Low}}$ are the highest and lowest optimal range of the memristors. The two resistors R_N and R_F set the gain of the circuit to be within the desired weight limit while adjusting the resistance of the memristor R_M between 10 KΩ and 100 KΩ. Following Equations 4.7 and 4.8, R_N and R_F were calculated to be 33.33 KΩ and 500 KΩ, respectively. Table 4.1 shows a summary of the conversion of the weights and logic levels between the theoretical ANNs and the circuit ANNs.

Table 4.1: Conversion between theoretical ANNs and circuit ANNs

	Theoretical ANNs	**Circuit ANNs**
Weights	-10	10 KΩ
	10	100 KΩ
Logic level	-1	-100 mV
	1	100 mV

4.3 Programming a Memristor

Programming a memristor is challenging because the device material has an inconsistent response to the programming pulse. Therefore, a method of programming the device has been developed based on the difference between the original and the target resistance.

4.3.1 Programming Circuit

Recall from Section 2.4, there are two modes in programming a memristor: writing and erasing. Writing will decrease the device resistance and erasing will increase the device

resistance. For the memristor model used in this thesis, a 1 μs pulse with an amplitude higher than the threshold voltage is used to program the memristor so that the device is not under a constant stress. Figure 4.4 shows the schematic of the programming circuit [14].

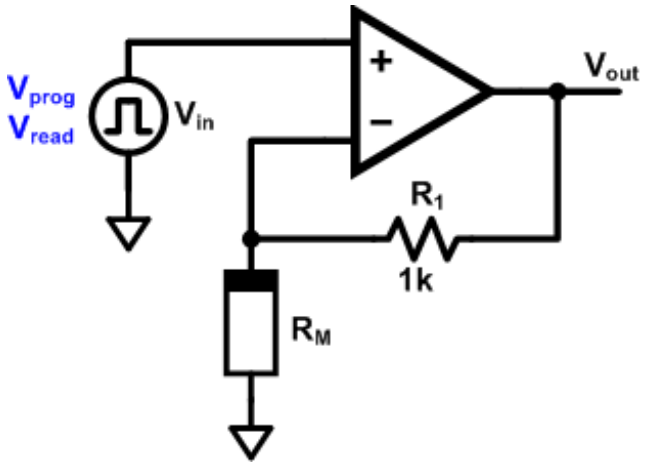


Figure 4.4: Programming circuit

To read the current value of the memristor, the same circuit will be used, however, with an amplitude pulse that is much smaller than the threshold voltage. For the memristor model used in this thesis, a (100 mV, 1 μs) pulse is chosen to be sufficiently small in order to avoid disturbing the current state of the device while reading it. The memristor resistance is then determined by measuring the output voltage of the circuit and applying Ohm's law

$$R_M = \frac{V_{prog}}{V_{out} - V_{prog}} R_1. \tag{4.9}$$

4.3.2 Programming a Single Memristor

The programming circuit in Figure 4.4 was built in Cadence first and then it was connected to Simulink using the Cadence coupler block. A MATLAB script was written to provide the voltage inputs to automatically drive the memristor to a desired resistance. Pseudocode for the algorithm is in Listing 4.1. The script will keep generating programming pulses with

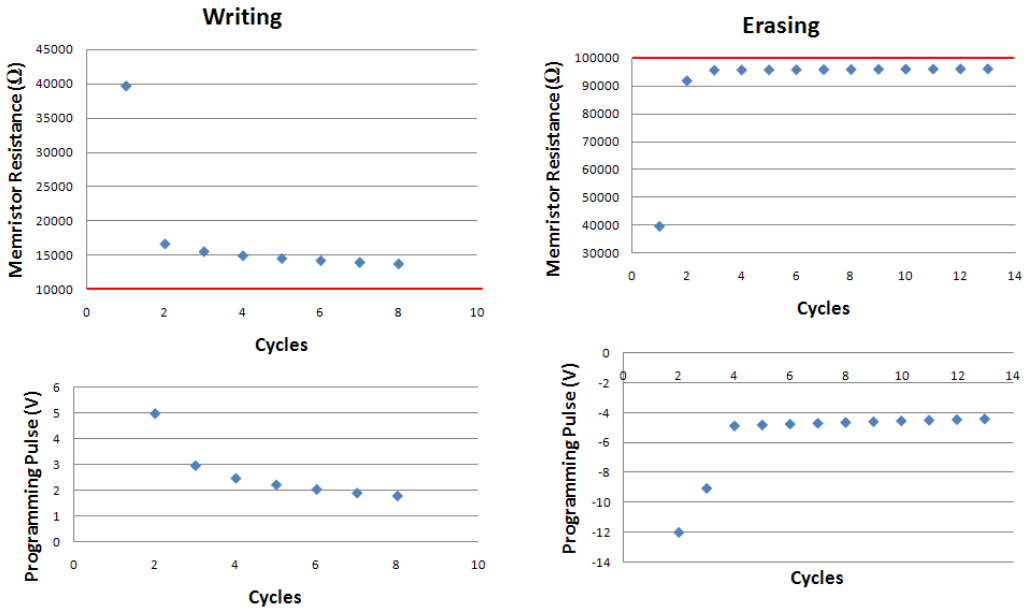
amplitudes proportional to the difference between the current resistance R_i of the memristor and the desired value R_d. If the difference is large, it will send a bigger amplitude pulse and vice-versa. This amplitude gets smaller and smaller as the memristor approaches the desired resistance. The erasing process is more sensitive than the writing process, therefore, the gain in the erasing process is a factor of 1/3 smaller than the gain in the writing process. The programming pulse is limited below V_{max} so that it will not destroy the device nor cause clipping in the output voltage of this programming circuit.

```
1  while (abs(R_i - R_d) > tolerance)
2      error = R_in - R_d;
3      if error > 0
4          gain = 0.3*V_gain;
5      else
6          gain = V_gain;
7      end
8      Vpulse = sign(error) * min(abs(errA)*gain, Vmax);
9      new_Ri = sim;
10     Ri = new_Ri;
11 end
```

Listing 4.1: Programming a memristor algorithm

After each programming state, the MATLAB script will generate a reading pulse to read the current resistance of the device. In short, the programming circuit will keep sending pulses with an adapted amplitude to program, then read the memristor until its resistance is within a certain tolerance of the desired value. This behaves like a state machine in hardware. The operating range of the memristor in this thesis is between 10 KΩ and 100

KΩ. The device was initialized at 40 KΩ, the desired value for the writing mode was 10 KΩ, and for the erasing mode was 100K Ω. Figure 4.5 shows the programming pulses and the memristor current resistance after each programming cycles. The target resistance is highlighted in red. The script was able to drive the device to the desired resistance within a tolerance of 4 KΩ within an average of 10 programming cycles.



(a) Programming from 40 KΩ to 10 KΩ (b) Programming from 40 KΩ to 100 KΩ

Figure 4.5: Programming a single memristor

4.4 Feed Forward Circuit

Figure 4.6 shows the structure of one neuron including the feed forward mechanism. The op-amp is used as a synapse summation. This circuit is added to the programming circuit as shown in Figure 4.7. The negative weight circuit is omitted in this schematic. The switch (θ, R_b) will be controlling to which circuit the memristor is connected. Depending on which

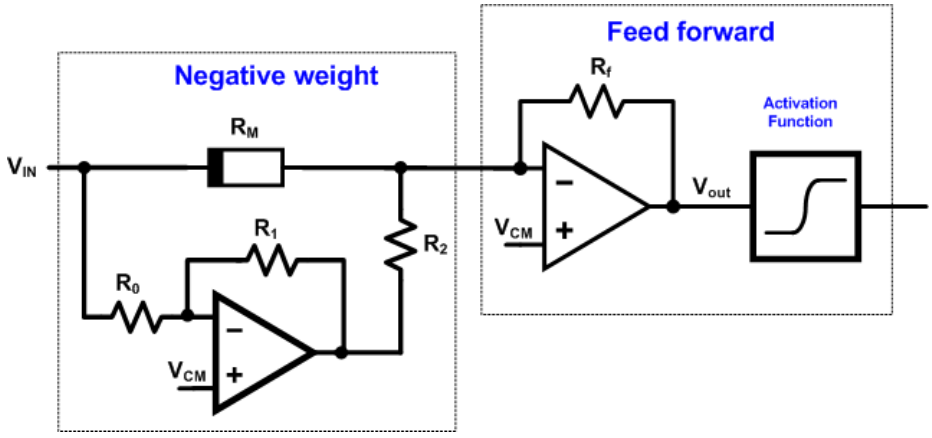


Figure 4.6: Feed forward circuit

process the memristor is currently in, it can either be connected to the programming circuit or to the feed forward circuit.

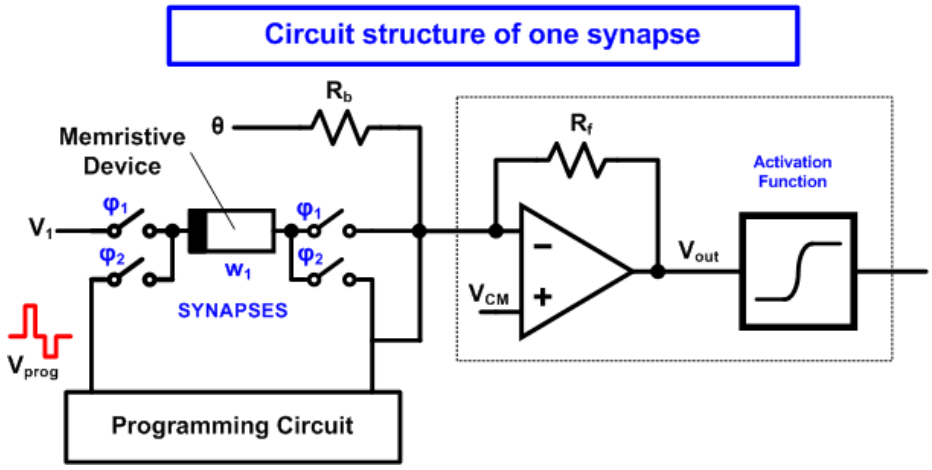


Figure 4.7: A complete circuit of one synapse

The next step is to connect this circuit to the controllers in Simulink. Figure 4.8 shows the finished model of one neuron that can operate in both programming mode and feed forward mode. If the memristor resistance needs to be read or adjusted, the "control programming line" block will turn on the programming/reading pulse generator. If just

the feed forward operation is required, then the “control feed forward line” will turn on the feed forward pulse generator.

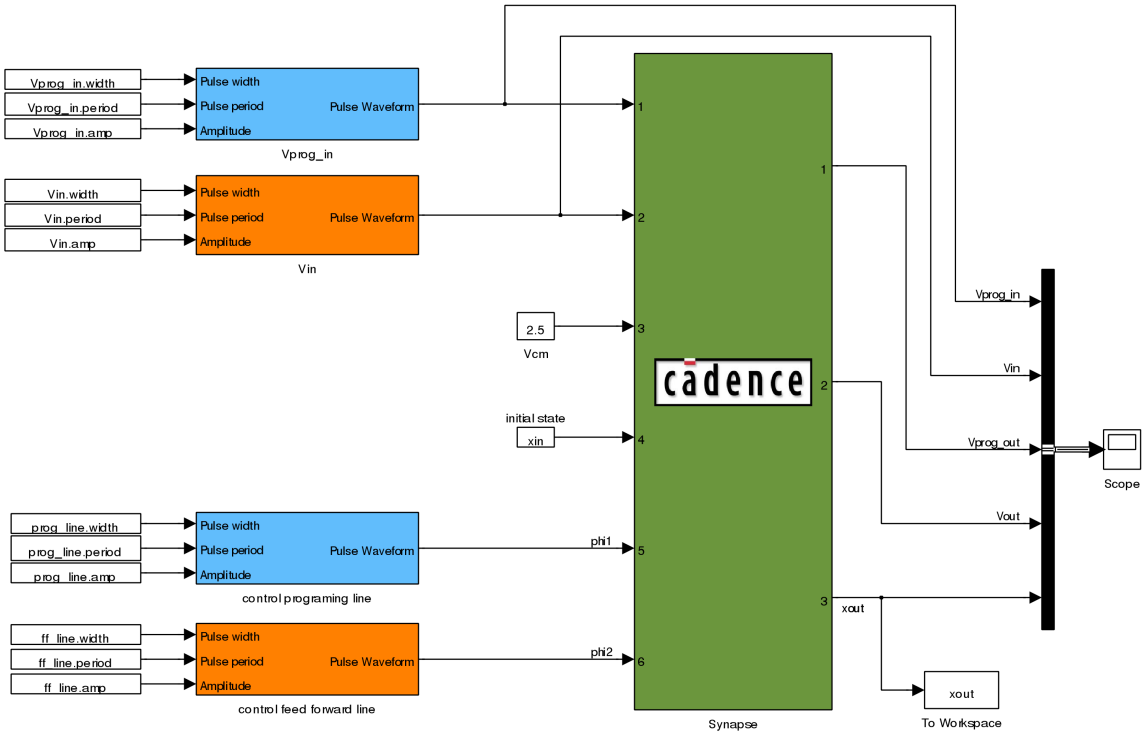


Figure 4.8: The Cadence memristor model controlled by Simulink to operate at the feed forward mode or programming mode

CHAPTER 5

NETWORK SIMULATION

At first, one single node ANN is built to simulate the linearly separable operators, then this structure will be expanded into a two layer ANN to simulate non-linealy separable operators. The procedure of implementing the memristors as weights into these ANNs are divided into following simulations

- Adaline with fixed resistances set from look-up weights for linearly separable operators
- Adaline with memristors programmed with weights from the look-up table
- Adaline with the MRII training algorithm to learn the weights
- Madaline with fixed resistances set from look up weights for XOR
- Madaline with memristors programmed with weights from the look-up table
- Madaline with the MRII training algorithm to learn the weights.

5.1 Single Node (Adaline) Circuits

The first simulation is to construct an Adaline structure to realize four basic threshold logic gates: AND, NAND, OR, and NOR. The weights needed for these binary logic operators are listed in Table 2.2. These values are converted into actual resistor values shown in

Table 5.1 based on the equations in Section 4.2. In this first experiment, these weights are manually calculated without any training method. The purpose of doing this is to verify that the circuits that were designed can be used to realize the four listed logic operators by simply re-programming the memristor resistance to a different target value.

5.1.1 Cadence Simulation - Fixed Resistance

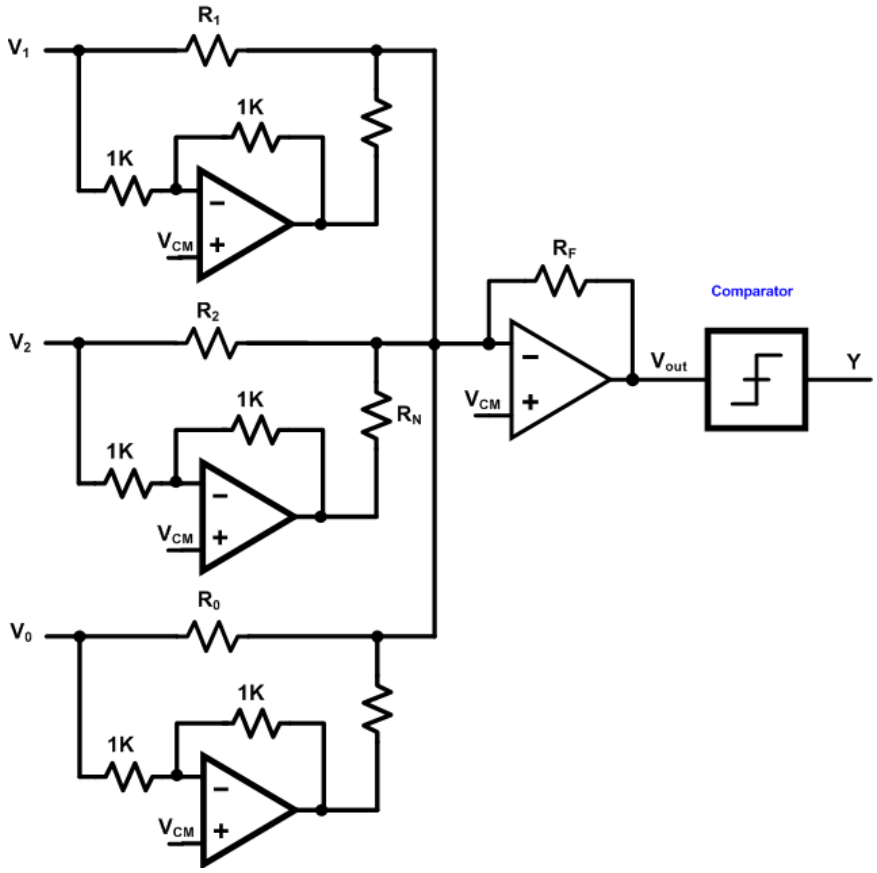


Figure 5.1: Schematic of an Adaline with fixed weight resistors in Cadence

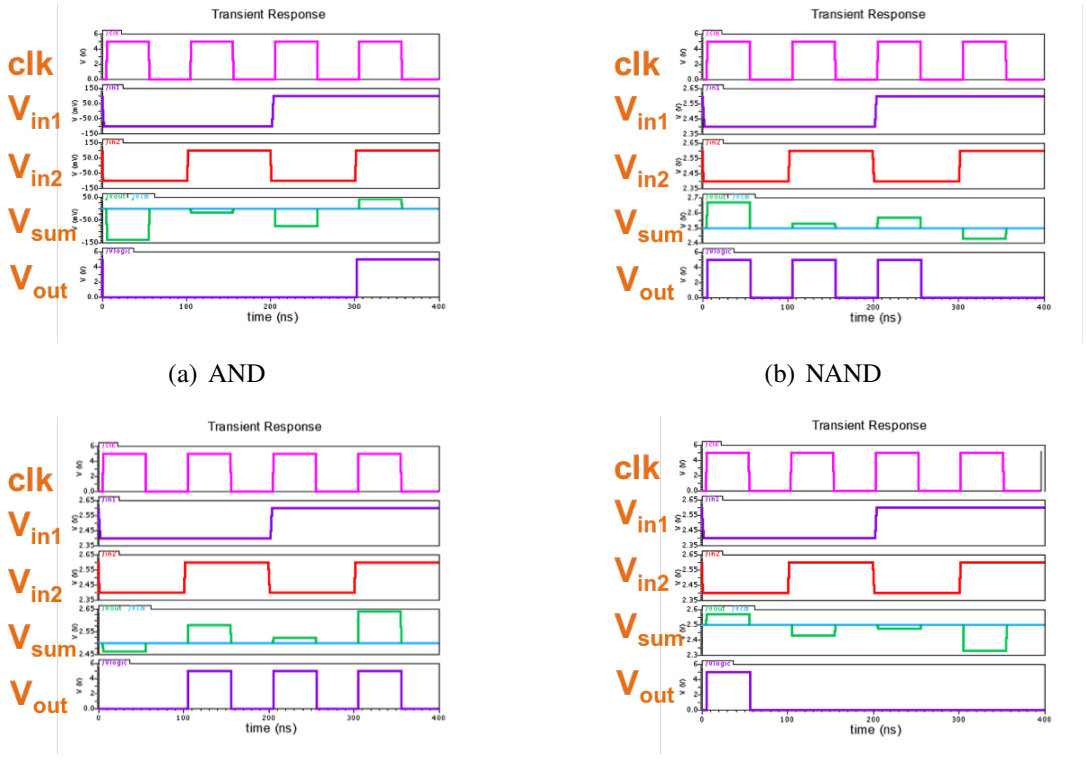
The single node network was first built using fixed weight resistors specified in Table 5.1. Figure 5.1 shows the circuit structure for such a network. The two binary inputs are V_1 and V_2, while V_0 is the bias input, which will always be a high input. The circuit

is controlled by a switch to make sure that the circuit is not on while the inputs are switching between their high and low states. The negative resistor R_N used in this circuitry is calculated to be 1.98 KΩ and the feedback resistor R_F is 2.02 KΩ. This is to ensure that the gain between output and input voltages is limited to the range between -1 and 1, which qualifies for the weights listed in Table 2.2. The Adaline circuitry is centered around a virtual ground V_{CM} = 2.5 V. Anything below this threshold voltage will be considered as a negative input, and anything above this threshold voltage will be considered as a positive input. For an Adaline, the neuron output will be passed through a comparator, which acts as a hard-limiting activation function, to determine the logic gate result.

Table 5.1: Resistance values converted from Table 2.2 theoretical weights

	R_1	R_2	R_0
NAND	1.33 KΩ	1.17 KΩ	3.88 KΩ
NOR	1.33 KΩ	1.17 KΩ	1.33 KΩ
AND	2.81 KΩ	4.81 KΩ	1.33 KΩ
OR	2.81 KΩ	4.81 KΩ	3.88 KΩ

The simulation result for an AND gate is shown in Figure 5.2(a). The first window shows the clock signal of the circuit, the second and third windows show the two binary input signals. The amplitude of these two inputs is set within a range of 100mV so that it will not disturb the current state of the memristor. The fourth window shows the output voltage superimposed on the 2.5 V virtual ground. The logic output of the comparator is displayed in the fifth window. It is confirmed that this circuit with the set of resistances in Table 5.1 works correctly for an AND gate. Figure 5.2(b) - 5.2(d) show similar results for the NAND, OR, and NOR gates using the same Adaline ANN structure and the other resistances listed in Table 5.1. This concludes that the conversion from the theoretical ANNs to the circuit ANNs works correctly.



(a) AND (b) NAND

(c) OR (d) NOR

Figure 5.2: The result of the Adaline simulation with fixed weight resistors

5.1.2 Adaline Prototype

After successfully simulating the Adaline circuit, the first Adaline ANN prototype in hardware was constructed using three potentiometers instead of the actual memristor devices. Four LF-411 op-amps were used, three of them were for the negative circuit of the network inputs, while the last op-amp was for the node summation. Figure 5.3 shows the entire circuit on a breadboard. The two voltages from Table 4.1 indicating logic levels were fed into the network to verify that this circuit design in hardware and not just Cadence simulation. It is confirmed by varying the resistance of the three potentiometers that this synapse circuitry works for all four operators: AND, NAND, OR, and NOR.

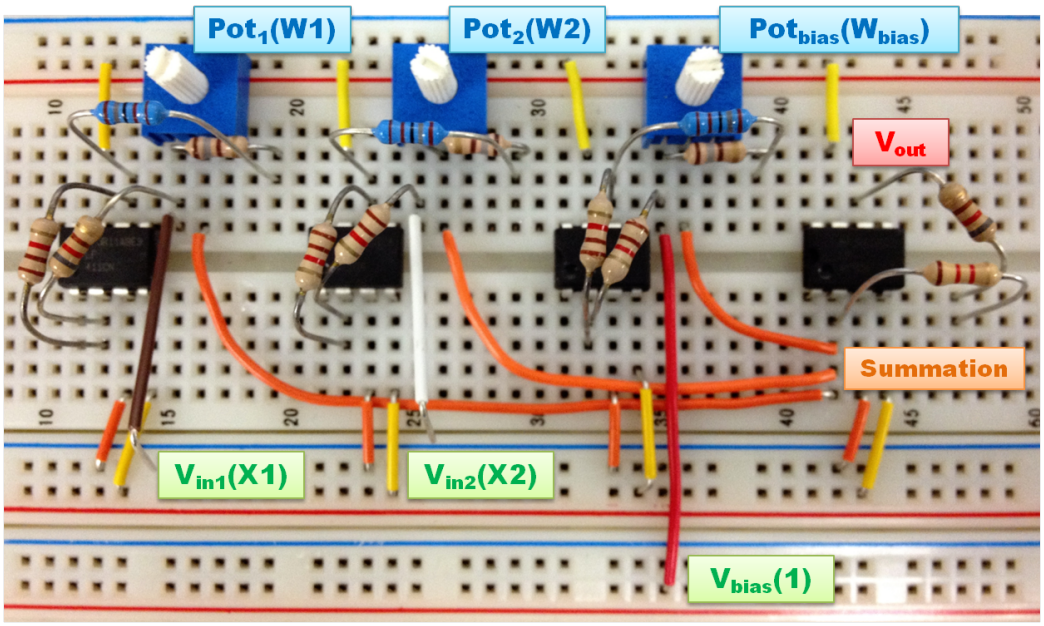


Figure 5.3: 1st prototype of an Adaline on a breadboard

5.1.3 Cadence Simulation - Memristors

After successfully simulating the four basic logic operations with fixed resistors, this experiment is then repeated using the same Adaline structure but replacing these fixed resistors with memristor models. These devices are initialized randomly between [10 KΩ, 100 KΩ] and will be programmed individually to reach the resistance values of these fixed resistors in Table 5.1. The schematic of an Adaline without the negative weight circuit components is shown in Figure 5.4.

Using the algorithm in Listing 4.1, each memristor in the network was individually programmed in an iterative process just like in Figure 4.5 until it reached the target resistance within a tolerance of 100 Ω. After all the memristors were programmed to match all the target resistances, a set of feed forward voltages was applied to the network to confirm the logic outputs for all four AND, NAND, OR, and NOR operations. The results are similar to Figure 5.2 showing that up to this point of the simulation, the Adaline circuit

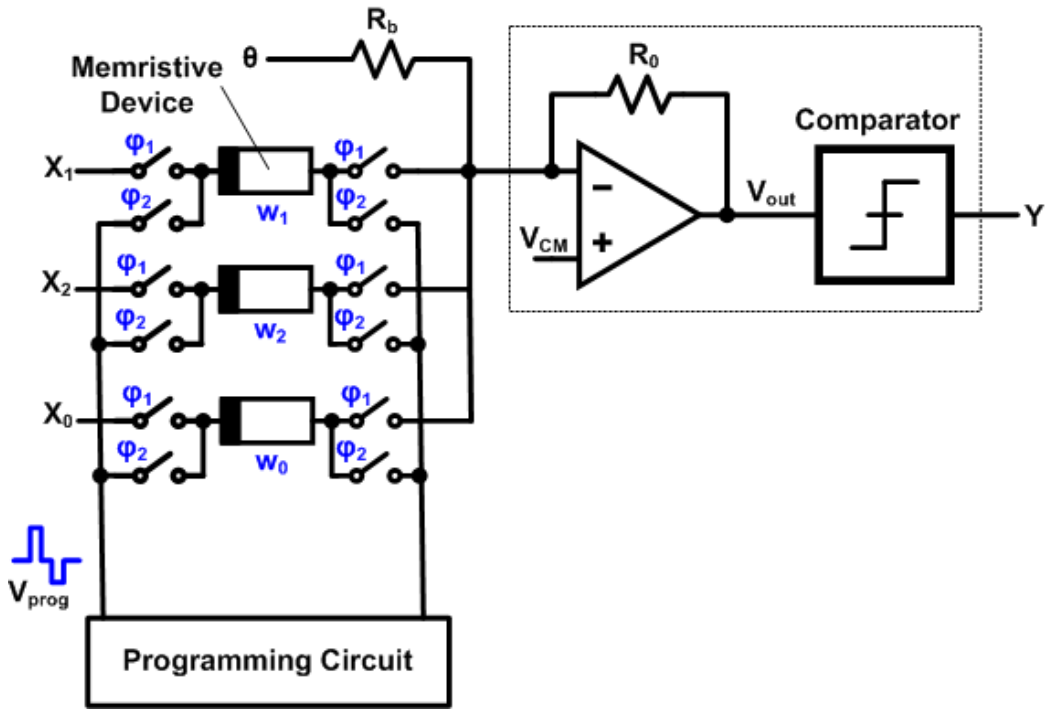


Figure 5.4: Adaline with Memristors

built with memristors is working correctly with look-up resistance values for all four basic logic operators, and the memristor programming procedure functions correctly also.

5.1.4 Adaline with Madaline Rule II Training Algorithm

In each of the previous sections, the weight or resistance values were determined and set in the device. The goal is to implement a learning process where the ANN will determine appropriate weight or resistance values based on some input-output combinations. The discussion in Section 3 led to the decision to use MRII as the training algorithm. MRII calculates feed-forward values for all four input pairs, then notes the number of incorrect outputs of the hard-limiting activation function. The minimum disturbance principle is used to select the neuron that has the lowest absolute value comming out of the hardlimiter. The weights for this neuron are then adjusted to reverse the hardlimiter outputs. The weight

change is accepted if the number of incorrect outputs is reduced, otherwise, the weight stays the same. Since this Adaline network for the four basic logic operations only has one neuron, this neuron will always be chosen if weight adjustment is needed regardless of the minimum disturbance principle result. The choice of which memristor to adjust will vary.

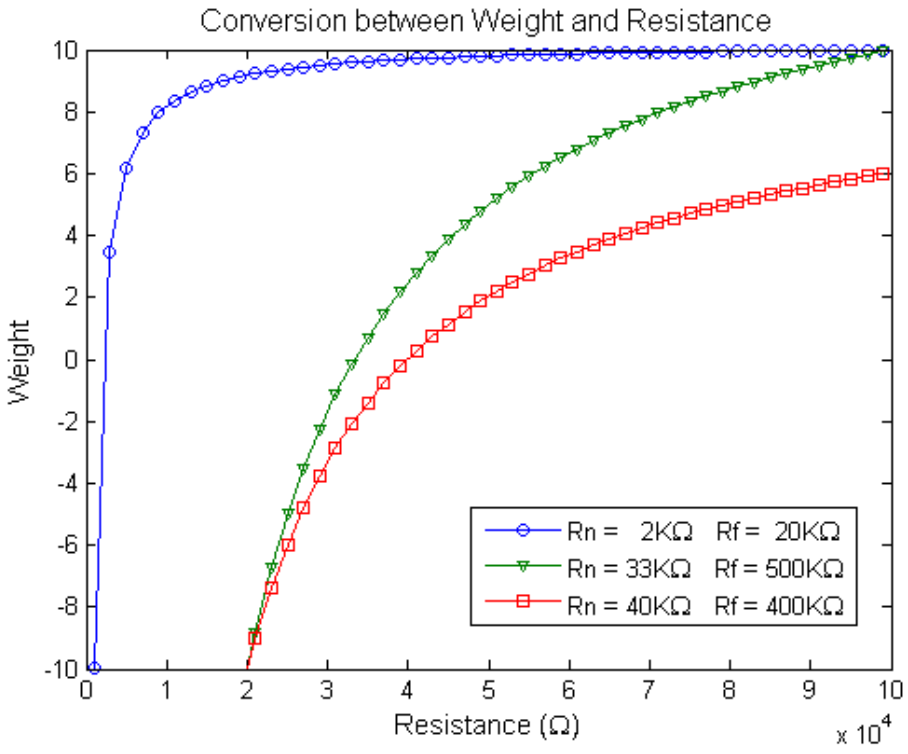


Figure 5.5: Conversion between theoretical weights and resistance values

The network was trained with only one epoch, a learning rate of 0.5 and a growth rate of 3. The maximum number of iterations was set to be 30. The memristor resistance range used in this thesis is between 10 KΩ and 100 KΩ. MRII updates weights by a random amount of Δw, which decides ahead of time, see Section 2.2.2. At this stage, we do not have good information about the relationship between the change in weight Δw vs. the change in resistance ΔR, and the relationship between ΔR vs. the change in programming pulse ΔP

sent to the memristor. Therefore, the MRII algorithm is implemented by programming the memristor to the resistance that is converted from Δw within a tolerance of 4 KΩ.

The conversion between the weight and resistance values is displayed in Figure 5.5. This curve can be manipulated by adjusting the resistors R_F and R_N of the synapse circuit in Figure 4.3. The steeper the curve is, the faster it is to train the memristor to reach its desired resistance, but the less tolerance it has for programming noise. For example, with R_N = 2 KΩ, R_F = 20 KΩ, one can change the ANN weights from -10 to 0 by adjusting the device resistance from 1 KΩ to 5 KΩ. This means that it does not take a lot of cycles to program the memristor, however, if the memristor is not programmed to the correct resistance, it will make a big difference in the weight update and cause the network to fail. In contrast, the conversion curve is less steep with R_N = 33 KΩ, R_F = 500 KΩ, therefore, adjusting the weight results in less weight change. This allows a higher resistance tolerance in programming the devices, but it also takes more iterations to reach a desired resistance. Even though this conversion still favors the positive weight, it now has a better range [20 KΩ, 35KΩ] to program the negative weight. Ideally, the best conversion would be a synapse circuit that has a linear conversion between the weights and the resistance, but this will increase the running time of the simulation. In addition, it is difficult in practice to program the current memristors to have a resistance less than 10 KΩ. Therefore, to compensate for this problem and to maintain an appropriate simulation run time, the middle curve is chosen. Hence, 33 KΩ and 500 KΩ resistors were used for R_N and R_F, respectively.

The simulation was repeated 100 times for each threshold logic unit, during this period, if the resistance ever grows outside of the range of [20 KΩ, 90 KΩ], the network is reset to start over again. Each training is said to be successful if it produces correct outputs for all four input pairs within 30 iterations.

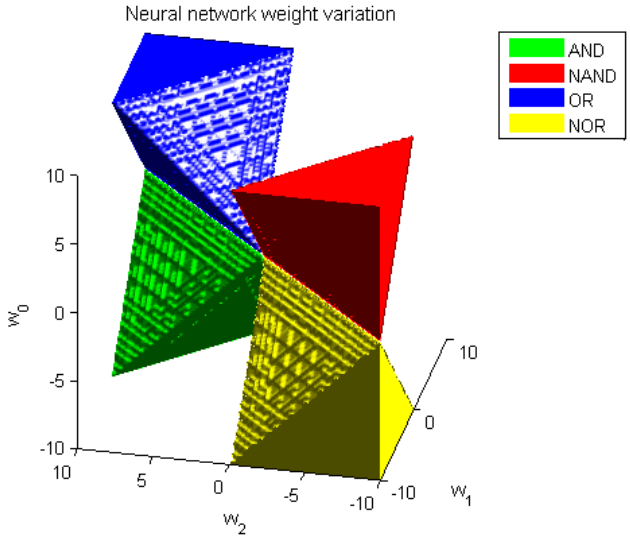
5.1.5 Adaline Simulation Results

The training was repeated 100 times for each logic operator and the success rate was recorded in Table 5.2. The learning success rate within 30 iterations of training is listed in Table 5.2. Results from individual resistance are listed in Appendix A. As predicted, the OR gate has the highest training success rate, and the NOR gate has the lowest success rate. This is due to the non-linear conversion between the weight and the resistance in Figure 5.5; the positive weights always have a higher tolerance than the negative weights. The weight variation in Section 2.3.1 and its equivalent resistance variation using this conversion is plotted in Figure 5.6. The OR gate has all its weights in the positive region, therefore it also has the highest tolerance for the resistance weights. In contrast, the NOR gate has all its weight in the negative region, therefore it has the lowest tolerance for the resistance weights, making it the most difficult case to program the weights. Recall from Chapter 3 that training an XOR with MRII was able to achieve 100% learning success rate after 20 epochs. Therefore, even though the NOR operator has only 60% learning success rate, this can be improved by increasing the number of epochs used in training.

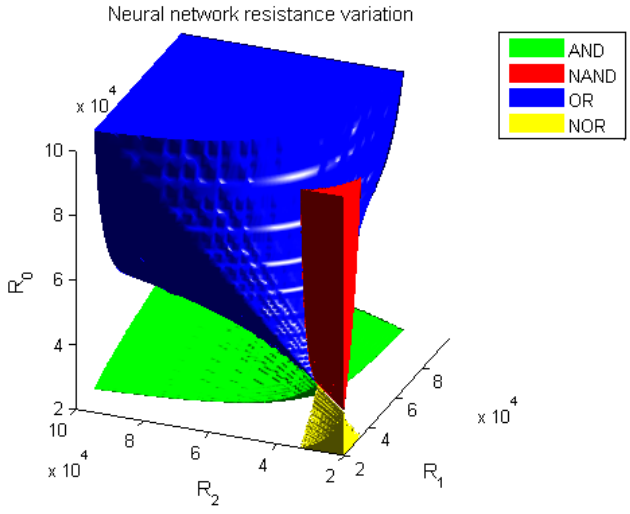
A complete cycle of training takes about two minutes to run, but the cycle stops as soon as a weight solution is found, therefore each simulation lasts roughly two hours. Looking at the average number of iterations in Table 5.2, the OR gate only takes 4.84 iterations while the NOR gate takes 6.68 iterations to finish training because NOR has the lowest tolerance for its resistance weights as seen in Figure 5.6. The NOR gate also has the highest standard deviation in the number of iterations because of the high sensitivity of the memristor resistance below 25 KΩ.

Table 5.2: Success rate for the four basic logic gates trained using MRII with maximum of 30 iterations

	OR	**AND**	**NAND**	**NOR**
Success Rate (%)	100	97	81	60
Avg # Iterations	4.84	5.15	5.84	6.65
Var # Iterations	8.22	18.03	25.36	44.54



(a) Weight space



(b) Resistance space

Figure 5.6: Inaccuracy variations for the AND, NAND, OR, and NOR operators

5.2 Multiple Node (Madaline) Circuit

The second experiment is to build a two layer ANN for XOR by expanding the Adaline structure from Section 5.1. The output of the first layer is fed into the input of the second layer. This input should have the same logic level as the first layer, which is -100mV for a low and 100mV for a high. However, the comparator, which is used as a hardlimiter activation function, has 0V as a low and 5V as a high. Therefore, a voltage divider circuit is added between the layers to shift the logic level back to the circuit ANN logic level, which is from -100mV to 100mV. Similar to the Adaline circuit experiment, the Madaline circuit is first simulated using fixed resistor weights. Then, the fixed resistors will be replaced with memristors and finally the Madaline will be implemented using the MRII training algorithm to produce the correct outputs for the XOR.

5.2.1 Cadence Simulation - Fixed Resistors

The Adaline circuit was duplicated three times to construct the Madaline circuit. A voltage divider was inserted between layers to convert the high voltage logic output of each node to a memristor logic input level. The schematic of the XOR neural network circuit is shown in Figure 5.7. The circuit has a total of nine memristors, of course each device was isolated from the programming circuit and the network circuit by switches just like in Figure 5.4, but for simplicity, these circuit components were not drawn in this schematic.

Similar to the experiment with the Adaline circuit, the memristors are first introduced by fixed resistor weight from Table 5.3. Again, the two inputs are V_1 and V_2, while V_0 is the bias input. The result in Figure 5.8 shows that this network structure was able to realize the XOR problem.

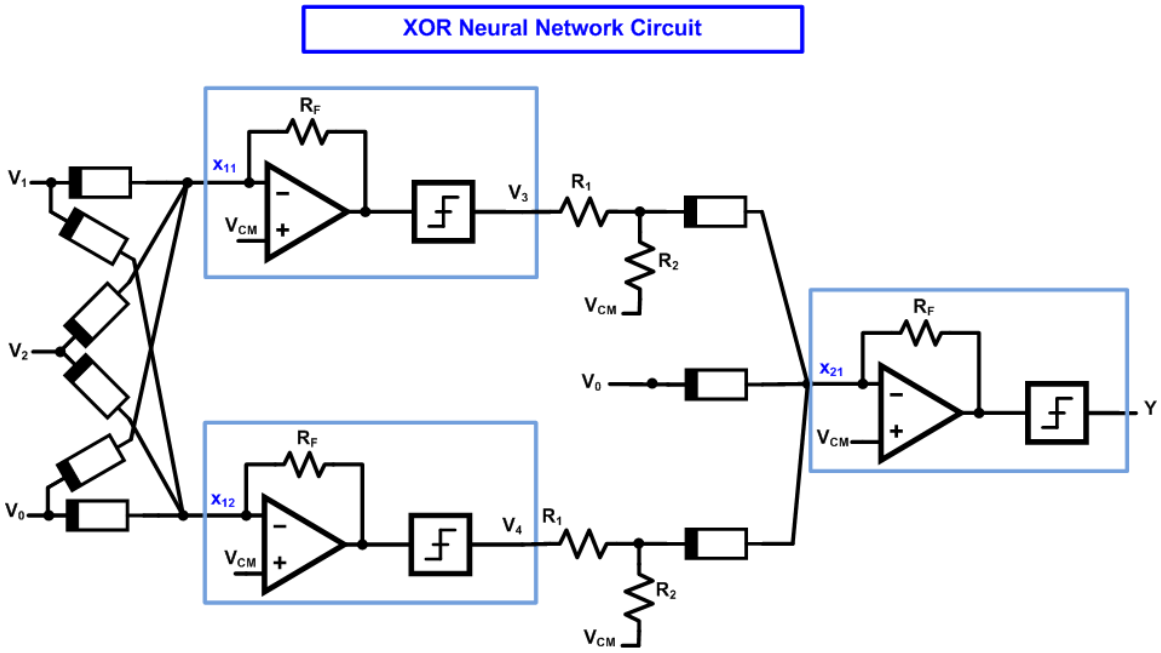


Figure 5.7: XOR neural network circuit

Table 5.3: XOR equivalent resistors of Table 2.3

	R_1 (KΩ)	R_2 (KΩ)	R_0 (KΩ)
hidden layer	31.952	32.332	34.866
	29.514	30.135	30.844
output layer	34.184	32.454	32.776

5.2.2 Cadence Simulation - Memristors Look Up

The next step is to program the memristors to the desired weight listed in Table 5.3 with a tolerance of 100 Ω. The resistor weight values in Table 5.3 are not optimal, so they require a high accuracy. The final programmed resistance is shown in Table 5.4. All the memristors were initialized to values between 10 KΩ and 100 KΩ. Each memristor took an average of 38 programming cycles to get to its desired resistance values. The network was able to realize the XOR problem correctly using the programmed memristors.

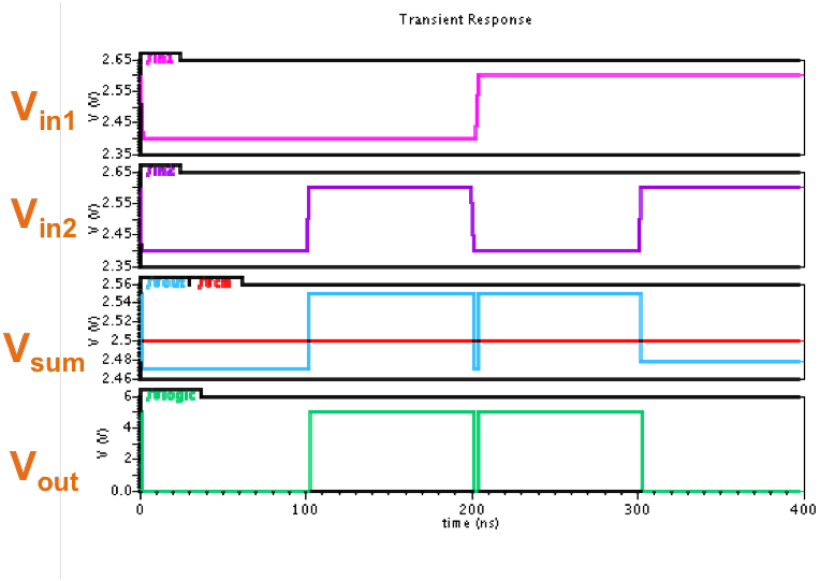


Figure 5.8: The result of the XOR simulation with a multilayer ANN using fixed weight resistors

Table 5.4: Programming memristors to target resistance weights for XOR operator

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
Target	31,952	32,332	34,866	29,514	30,135	30,844	34,184	32,454	32,776
Programmed	31,993	32,360	34,778	29,535	30,187	30,893	34,101	32,489	32,771
# Cycles	40	38	44	48	48	40	26	33	30

5.2.3 Multilayer Network With Madaline Rule II Training Algorithm

The final simulation is to implement MRII so that the network can learn to realize an XOR by just simply providing it the desired inputs and outputs. The network was trained with a learning rate of 0.5 and a growth rate of 3 just as in the Adaline simulation. The maximum number of allowed iterations was increased to from 30 to 50 because XOR has a small zone in the weight space and it is harder for the network to learn. The weight was still limited to values between -10 and 10, and the resistance range was still between 10 KΩ and 100 KΩ. The conversion between the weights and the resistance values was kept the same, therefore R_F was 500 KΩ and R_N was 33 KΩ. Out of 10 simulation runs, the network was able to

realize the XOR four times and the final resistance weights are recorded in Appendix B. Again, this can still reach 100% learning success rate by increasing the number of epochs in MRII.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This thesis has described how to use memristors to build a hardware-only ANN. The conversion between the theoretical weights and the resistance values was analyzed. The circuits were designed and confirmed to work correctly through simulations. These circuits includes: single layer ANN, multilayer ANN, negative weight circuit, and adjusting the interface between the first and second layer of a multilayer ANN. This thesis was able to illustrate the learning ability of the hardware ANN designed with memristiors by solving linearly and non-linearly separable problems. Using Madaline Rule II with the modified epoch parameter, the network was able to achieve 100% learning success rate within 50 iterations. The programming inaccuracy of the memristors was able to help the learning process.

Improvement can still be made towards the conversion between the theoretical weights and the resistance values. This limitation is because the network is being designed with discrete circuit components. This can be handled in a better manner using an on-chip CMOS circuit, and we should not have a problem to achieve a linear conversion between the weights and resistance values. More experiments need to be conducted to characterize the memristive devices to fill in the gap of straight conversion between resistance changes and programming pulses in MRII learning algorithm.

Memristors are fragile and programming them is challenging because of their incon-

sistent response to different pulse widths and amplitudes. A better feedback programming circuit needs to be developed. If the resistance cannot reach a desired value after a certain number of cycles, it is best to just fully erase the memristors and try again. A better way to reduce the number of programming cycles to preserve the device is to design the noise around random pulses sent to the device. Also, the back propagation training algorithm can be used with CMOS technology to produce a non-linear activation function.

Finally, we do not want to just stop at the simulation. Progress is being made at Boise State University in building a hardware ANN with actual memristive devices and the Madaline Rule II implemented on an FPGA. Future work will include fabricating the entire multilayer ANN on chip.

REFERENCES

[1] Leon O. Chua. Memristor - the missing circuit element. *Circuit Theory, IEEE Transactions on*, 18(5):507 – 519, September 1971.

[2] Jihong Liu and Deqin Liang. A survey of FPGA-Based hardware implementation of ANNs. In *Neural Networks and Brain, 2005. ICNN B '05. International Conference on*, volume 2, pages 915 –918, October 2005.

[3] Xiaoping Zhu and Yaowu Chen. Improved FPGA implementation of probabilistic neural network for neural decoding. In *Apperceiving Computing and Intelligence Analysis (ICACIA), 2010 International Conference on*, pages 198 –202, December 2010.

[4] Wang Yonggang, Du Junwei, Zhou Zhonghui, Yang Yang, Zhang Lijun, and Peter Bruyndonckx. FPGA based electronics for PET detector modules with neural network position estimators. *Nuclear Science, IEEE Transactions on*, 58(1):34 –42, February 2011.

[5] Leon O. Chua. The fourth element. *Proceedings of the IEEE*, 100(6):1920 –1927, June 2012.

[6] Wen Jin, Zhao Jia Li, Luo Si Wei, and Han Zhen. The improvements of BP neural network learning algorithm. In *Signal Processing Proceedings, 2000. WCCC-ICSP 2000. 5th International Conference on*, volume 3, pages 1647 –1649 vol.3, 2000.

[7] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley Interscience, second edition, 2001.

[8] Rodney Winter and Bernard Widrow. Madaline Rule II: a training algorithm for neural networks. In *Neural Networks, 1988., IEEE International Conference on*, volume 1, pages 401 – 408, July 1988.

[9] Antonio S. Oblea, Achyut Timilsina, David Moore, and Kristy A. Campbell. Silver chalcogenide based memristor devices. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1 –3, July 2010.

[10] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 453:80 – 83, 2008.

[11] Shyam. P. Adhikari, Changju Yang, Hyongsuk Kim, and Leon O. Chua. Memristor bridge synapse-based neural network and its learning. *Neural Networks and Learning Systems, IEEE Transactions on*, PP(99):1, July 2012.

[12] Chris Yakopcic, Tarek M. Taha, Guru Subramanyam, Robinson E. Pino, and Stanley Rogers. A memristor device model. *Electron Device Letters, IEEE*, 32(10):1436 –1438, October 2011.

[13] Maryhelen Stevenson, Rodney Winter, and Bernard Widrow. Sensitivity of feedforward neural networks to weight errors. *Neural Networks, IEEE Transactions on*, 1(1):71 –80, March 1990.

[14] Kolton Drake and Kristy A. Campbell. Chalcogenide-based memristive device control of a LEGO Mindstorms NXT servo motor. *AIAA Infotech@Aerospace Conference and Exhibit*, March 2011.

APPENDIX A

ADALINE RESULTS OF TRAINING WITH MRII

Table A.1: Resistor values trained with MRII for an OR operator

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
1	46	47	39	4.13	4.36	2.18	2
2	45	56	42	3.89	6.07	3.10	8
3	41	37	41	2.80	1.49	2.80	3
4	39	40	44	2.18	2.50	3.64	5
5	36	36	36	1.11	1.11	1.11	2
6	43	41	42	3.37	2.80	3.10	8
7	42	36	39	3.10	1.11	2.18	9
8	35	43	43	0.71	3.37	3.37	11
9	34	36	36	0.29	1.11	1.11	1
10	39	47	46	2.18	4.36	4.13	2
11	36	37	35	1.11	1.49	0.71	2
12	47	35	46	4.36	0.71	4.13	9
13	53	43	47	5.57	3.37	4.36	9
14	39	39	39	2.18	2.18	2.18	2

Continued on next page

Table A.1 – *Continued from previous page*

Exp	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
15	39	37	37	2.18	1.49	1.49	2
16	39	48	43	2.18	4.58	3.37	6
17	36	36	38	1.11	1.11	1.84	4
18	48	43	38	4.58	3.37	1.84	9
19	40	38	37	2.50	1.84	1.49	4
20	37	40	41	1.49	2.50	2.80	6
21	36	41	42	1.11	2.80	3.10	4
22	40	38	38	2.50	1.84	1.84	4
23	40	36	40	2.50	1.11	2.50	3
24	36	36	34	1.11	1.11	0.29	1
25	40	54	47	2.50	5.74	4.36	7
26	51	38	45	5.20	1.84	3.89	11
27	41	35	41	2.80	0.71	2.80	2
28	35	37	36	0.71	1.49	1.11	4
29	39	44	39	2.18	3.64	2.18	4
30	40	35	40	2.50	0.71	2.50	10
31	47	45	35	4.36	3.89	0.71	5
32	39	51	46	2.18	5.20	4.13	2
33	42	36	39	3.10	1.11	2.18	3
34	40	40	43	2.50	2.50	3.37	5
35	37	44	42	1.49	3.64	3.10	7
36	36	37	36	1.11	1.49	1.11	2

Continued on next page

Table A.1 – *Continued from previous page*

Exp	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
37	35	46	45	0.71	4.13	3.89	5
38	40	42	45	2.50	3.10	3.89	5
39	40	37	43	2.50	1.49	3.37	6
40	39	44	41	2.18	3.64	2.80	8
41	43	44	48	3.37	3.64	4.58	2
42	39	40	40	2.18	2.50	2.50	2
43	41	43	42	2.80	3.37	3.10	3
44	39	36	40	2.18	1.11	2.50	2
45	41	42	37	2.80	3.10	1.49	5
46	40	38	47	2.50	1.84	4.36	14
47	39	46	40	2.18	4.13	2.50	4
48	44	35	46	3.64	0.71	4.13	3
49	41	53	42	2.80	5.57	3.10	7
50	47	38	44	4.36	1.84	3.64	2
51	37	36	38	1.49	1.11	1.84	2
52	35	40	39	0.71	2.50	2.18	4
53	38	35	37	1.84	0.71	1.49	6
54	40	39	45	2.50	2.18	3.89	5
55	38	36	37	1.84	1.11	1.49	3
56	37	35	37	1.49	0.71	1.49	6
57	37	40	39	1.49	2.50	2.18	5
58	35	36	37	0.71	1.11	1.49	2

Continued on next page

Table A.1 – *Continued from previous page*

Exp	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
59	61	54	37	6.80	5.74	1.49	9
60	59	57	35	6.53	6.23	0.71	3
61	50	40	50	5.00	2.50	5.00	4
62	37	51	50	1.49	5.20	5.00	8
63	43	41	41	3.37	2.80	2.80	7
64	41	37	38	2.80	1.49	1.84	2
65	44	47	38	3.64	4.36	1.84	2
66	39	36	37	2.18	1.11	1.49	2
67	38	36	40	1.84	1.11	2.50	3
68	46	37	48	4.13	1.49	4.58	2
69	38	37	39	1.84	1.49	2.18	5
70	39	38	37	2.18	1.84	1.49	3
71	40	38	39	2.50	1.84	2.18	6
72	38	39	35	1.84	2.18	0.71	6
73	48	50	67	4.58	5.00	7.54	12
74	37	36	39	1.49	1.11	2.18	2
75	37	41	40	1.49	2.80	2.50	4
76	37	36	35	1.49	1.11	0.71	3
77	36	46	47	1.11	4.13	4.36	8
78	40	35	40	2.50	0.71	2.50	7
79	44	43	38	3.64	3.37	1.84	6
80	53	60	44	5.57	6.67	3.64	5

Continued on next page

Table A.1 – *Continued from previous page*

Exp	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
81	44	36	46	3.64	1.11	4.13	3
82	36	39	37	1.11	2.18	1.49	2
83	37	43	42	1.49	3.37	3.10	4
84	35	34	35	0.71	0.29	0.71	1
85	41	40	43	2.80	2.50	3.37	11
86	35	43	43	0.71	3.37	3.37	6
87	39	45	40	2.18	3.89	2.50	8
88	45	42	60	3.89	3.10	6.67	9
89	38	41	38	1.84	2.80	1.84	8
90	36	36	36	1.11	1.11	1.11	2
91	52	60	37	5.38	6.67	1.49	9
92	63	62	38	7.06	6.94	1.84	4
93	35	40	41	0.71	2.50	2.80	2
94	38	36	37	1.84	1.11	1.49	5
95	36	35	35	1.11	0.71	0.71	1
96	42	39	38	3.10	2.18	1.84	3
97	39	39	36	2.18	2.18	1.11	3
98	51	47	38	5.20	4.36	1.84	7
99	42	44	39	3.10	3.64	2.18	6
100	45	43	52	3.89	3.37	5.38	2
Mean	40.92	41.22	40.86	2.59	2.64	2.60	4.84
Variance	31.67	38.48	26.44	2.09	2.55	1.77	8.22

Table A.2: Resistor values trained with MRII for an AND operator

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
1	40	38	29	2.50	1.84	-2.24	5
2	39	39	31	2.18	2.18	-1.13	7
3	39	39	32	2.18	2.18	-0.62	8
4	36	37	30	1.11	1.49	-1.67	4
5	35	35	33	0.71	0.71	-0.15	1
6	45	36	28	3.89	1.11	-2.86	4
7	38	40	32	1.84	2.50	-0.62	5
8	34	36	32	0.29	1.11	-0.62	2
9	35	36	32	0.71	1.11	-0.62	2
10	46	48	31	4.13	4.58	-1.13	12
11	42	41	29	3.10	2.80	-2.24	4
12	50	39	27	5.00	2.18	-3.52	5
13	39	36	32	2.18	1.11	-0.62	2
14	39	40	30	2.18	2.50	-1.67	5
15	34	36	32	0.29	1.11	-0.62	1
16	46	36	28	4.13	1.11	-2.86	8
17	39	35	30	2.18	0.71	-1.67	3
18	34	36	31	0.29	1.11	-1.13	2
19	40	34	30	2.50	0.29	-1.67	2

Continued on next page

Table A.2 – *Continued from previous page*

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
20	35	35	31	0.71	0.71	-1.13	2
21	34	34	33	0.29	0.29	-0.15	1
22	49	48	30	4.80	4.58	-1.67	8
23	44	40	29	3.64	2.50	-2.24	7
24	35	35	32	0.71	0.71	-0.62	2
25	39	38	32	2.18	1.84	-0.62	5
26	36	35	33	1.11	0.71	-0.15	2
27	44	50	27	3.64	5.00	-3.52	4
28	35	35	33	0.71	0.71	-0.15	2
29	34	35	33	0.29	0.71	-0.15	1
30	40	46	27	2.50	4.13	-3.52	6
31	38	39	29	1.84	2.18	-2.24	4
32	35	35	33	0.71	0.71	-0.15	2
33	35	36	32	0.71	1.11	-0.62	1
34	38	38	32	1.84	1.84	-0.62	3
35	34	35	32	0.29	0.71	-0.62	1
36	37	37	33	1.49	1.49	-0.15	5
37	38	38	30	1.84	1.84	-1.67	6
38	46	47	32	4.13	4.36	-0.62	13
39	40	44	28	2.50	3.64	-2.86	9
40	35	35	32	0.71	0.71	-0.62	2
41	48	47	30	4.58	4.36	-1.67	2

Continued on next page

Table A.2 – *Continued from previous page*

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
42	40	39	32	2.50	2.18	-0.62	3
43	39	36	29	2.18	1.11	-2.24	6
44	44	54	23	3.64	5.74	-6.74	12
45	36	36	33	1.11	1.11	-0.15	5
46	34	36	32	0.29	1.11	-0.62	3
47	43	58	27	3.37	6.38	-3.52	24
48	43	36	27	3.37	1.11	-3.52	9
49	42	41	31	3.10	2.80	-1.13	3
50	34	34	33	0.29	0.29	-0.15	1
51	37	37	31	1.49	1.49	-1.13	3
52	36	36	33	1.11	1.11	-0.15	2
53	51	39	26	5.20	2.18	-4.23	4
54	38	38	32	1.84	1.84	-0.62	3
55	40	60	26	2.50	6.67	-4.23	14
56	51	65	29	5.20	7.31	-2.24	6
57	41	42	29	2.80	3.10	-2.24	12
58	40	43	28	2.50	3.37	-2.86	6
59	39	39	33	2.18	2.18	-0.15	6
60	39	41	31	2.18	2.80	-1.13	4
61	42	41	32	3.10	2.80	-0.62	5
62	35	36	33	0.71	1.11	-0.15	1
63	35	35	31	0.71	0.71	-1.13	1

Continued on next page

Table A.2 – *Continued from previous page*

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
64	43	47	29	3.37	4.36	-2.24	6
65	44	37	28	3.64	1.49	-2.86	6
66	40	40	32	2.50	2.50	-0.62	5
67	44	40	30	3.64	2.50	-1.67	6
68	35	35	32	0.71	0.71	-0.62	1
69	37	35	30	1.49	0.71	-1.67	3
70	34	34	33	0.29	0.29	-0.15	1
71	41	44	29	2.80	3.64	-2.24	6
72	37	37	32	1.49	1.49	-0.62	5
73	37	38	29	1.49	1.84	-2.24	11
74	43	43	32	3.37	3.37	-0.62	3
75	35	35	31	0.71	0.71	-1.13	1
76	46	45	23	4.13	3.89	-6.74	23
77	38	39	30	1.84	2.18	-1.67	14
78	43	47	31	3.37	4.36	-1.13	7
79	43	39	30	3.37	2.18	-1.67	5
80	38	37	31	1.84	1.49	-1.13	3
81	43	39	31	3.37	2.18	-1.13	7
82	35	36	31	0.71	1.11	-1.13	7
83	54	54	31	5.74	5.74	-1.13	4
84	38	40	28	1.84	2.50	-2.86	4
85	43	43	32	3.37	3.37	-0.62	4

Continued on next page

Table A.2 – *Continued from previous page*

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
86	39	39	30	2.18	2.18	-1.67	4
87	36	36	32	1.11	1.11	-0.62	5
88	51	45	30	5.20	3.89	-1.67	12
89	49	37	26	4.80	1.49	-4.23	4
90	39	39	33	2.18	2.18	-0.15	4
91	35	34	32	0.71	0.29	-0.62	1
92	38	38	29	1.84	1.84	-2.24	9
93	35	35	32	0.71	0.71	-0.62	2
94	34	34	33	0.29	0.29	-0.15	1
95	50	62	19	5.00	6.94	-11.32	14
96	38	39	31	1.84	2.18	-1.13	4
97	37	40	30	1.49	2.50	-1.67	5
Mean	39.69	39.82	30.36	2.23	2.20	-1.60	5.15
Variance	23.24	39.06	6.09	2.05	2.58	2.70	18.03

Table A.3: Resistor values trained with MRII for a NAND operator

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
1	27	27	37	-3.52	-3.52	1.49	6
2	33	32	34	-0.15	-0.62	0.29	7
3	33	32	35	-0.15	-0.62	0.71	2
4	32	32	35	-0.62	-0.62	0.71	2
5	25	30	54	-5.00	-1.67	5.74	12
6	32	32	35	-0.62	-0.62	0.71	1
7	32	32	37	-0.62	-0.62	1.49	4
8	29	26	50	-2.24	-4.23	5.00	6
9	32	32	35	-0.62	-0.62	0.71	1
10	32	32	34	-0.62	-0.62	0.29	1
11	29	31	39	-2.24	-1.13	2.18	2
12	25	21	68	-5.00	-8.81	7.65	16
13	31	31	34	-1.13	-1.13	0.29	3
14	33	33	35	-0.15	-0.15	0.71	3
15	27	30	43	-3.52	-1.67	3.37	4
16	27	27	54	-3.52	-3.52	5.74	10
17	33	32	35	-0.15	-0.62	0.71	1
18	32	33	35	-0.62	-0.15	0.71	1
19	26	23	42	-4.23	-6.74	3.10	4
20	32	33	35	-0.62	-0.15	0.71	2

Continued on next page

Table A.3 – *Continued from previous page*

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
21	30	28	50	-1.67	-2.86	5.00	15
22	29	33	39	-2.24	-0.15	2.18	3
23	26	31	43	-4.23	-1.13	3.37	10
24	27	24	44	-3.52	-5.83	3.64	4
25	30	30	35	-1.67	-1.67	0.71	9
26	27	24	48	-3.52	-5.83	4.58	5
27	31	32	37	-1.13	-0.62	1.49	6
28	30	29	45	-1.67	-2.24	3.89	4
29	33	33	35	-0.15	-0.15	0.71	1
30	28	31	45	-2.86	-1.13	3.89	2
31	32	33	35	-0.62	-0.15	0.71	1
32	32	33	36	-0.62	-0.15	1.11	5
33	32	33	36	-0.62	-0.15	1.11	3
34	28	28	39	-2.86	-2.86	2.18	4
35	26	32	46	-4.23	-0.62	4.13	6
36	32	26	45	-0.62	-4.23	3.89	11
37	27	27	36	-3.52	-3.52	1.11	25
38	31	31	35	-1.13	-1.13	0.71	2
39	30	30	41	-1.67	-1.67	2.80	7
40	26	26	44	-4.23	-4.23	3.64	6
41	32	31	37	-0.62	-1.13	1.49	3
42	33	31	36	-0.15	-1.13	1.11	3

Continued on next page

Table A.3 – *Continued from previous page*

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
43	25	23	51	-5.00	-6.74	5.20	6
44	24	29	57	-5.83	-2.24	6.23	16
45	30	31	37	-1.67	-1.13	1.49	5
46	29	30	38	-2.24	-1.67	1.84	4
47	33	33	34	-0.15	-0.15	0.29	1
48	33	32	35	-0.15	-0.62	0.71	1
49	33	33	35	-0.15	-0.15	0.71	1
50	32	32	35	-0.62	-0.62	0.71	1
51	32	28	40	-0.62	-2.86	2.50	6
52	29	30	36	-2.24	-1.67	1.11	3
53	28	29	40	-2.86	-2.24	2.50	13
54	30	32	38	-1.67	-0.62	1.84	2
55	31	28	42	-1.13	-2.86	3.10	3
56	28	27	38	-2.86	-3.52	1.84	11
57	31	29	37	-1.13	-2.24	1.49	12
58	25	27	38	-5.00	-3.52	1.84	18
59	32	32	35	-0.62	-0.62	0.71	1
60	26	32	44	-4.23	-0.62	3.64	16
61	31	28	49	-1.13	-2.86	4.80	7
62	29	29	37	-2.24	-2.24	1.49	5
63	28	24	43	-2.86	-5.83	3.37	10
64	32	31	38	-0.62	-1.13	1.84	3

Continued on next page

Table A.3 – *Continued from previous page*

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
65	29	28	39	-2.24	-2.86	2.18	12
66	32	31	36	-0.62	-1.13	1.11	1
67	32	32	35	-0.62	-0.62	0.71	1
68	28	32	41	-2.86	-0.62	2.80	7
69	31	31	38	-1.13	-1.13	1.84	15
70	32	32	36	-0.62	-0.62	1.11	4
71	26	24	43	-4.23	-5.83	3.37	15
72	28	30	44	-2.86	-1.67	3.64	3
73	32	28	39	-0.62	-2.86	2.18	5
74	30	27	40	-1.67	-3.52	2.50	11
75	25	26	36	-5.00	-4.23	1.11	4
76	33	25	52	-0.15	-5.00	5.38	11
77	33	33	34	-0.15	-0.15	0.29	1
78	30	28	38	-1.67	-2.86	1.84	8
79	33	33	35	-0.15	-0.15	0.71	1
80	29	31	38	-2.24	-1.13	1.84	3
81	31	31	37	-1.13	-1.13	1.49	3
Mean	29.86	29.73	39.89	-1.88	-2.00	2.21	5.84
Variance	6.77	8.63	39.53	2.42	3.56	2.74	25.36

Table A.4: Resistor values trained with MRII for a NOR operator

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
1	33	32	32	-0.15	-0.62	-0.62	1
2	31	27	29	-1.13	-3.52	-2.24	6
3	30	32	31	-1.67	-0.62	-1.13	4
4	25	26	31	-5.00	-4.23	-1.13	9
5	33	32	33	-0.15	-0.62	-0.15	1
6	27	26	24	-3.52	-4.23	-5.83	5
7	22	25	24	-7.73	-5.00	-5.83	19
8	22	32	21	-7.73	-0.62	-8.81	13
9	34	33	33	0.29	-0.15	-0.15	1
10	31	31	33	-1.13	-1.13	-0.15	3
11	29	22	24	-2.24	-7.73	-5.83	10
12	32	30	31	-0.62	-1.67	-1.13	2
13	25	27	21	-5.00	-3.52	-8.81	10
14	32	31	31	-0.62	-1.13	-1.13	1
15	32	32	32	-0.62	-0.62	-0.62	1
16	31	28	29	-1.13	-2.86	-2.24	3
17	32	33	32	-0.62	-0.15	-0.62	1
18	25	23	30	-5.00	-6.74	-1.67	26
19	26	30	25	-4.23	-1.67	-5.00	24
20	31	28	28	-1.13	-2.86	-2.86	6

Continued on next page

Table A.4 – *Continued from previous page*

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
21	29	29	29	-2.24	-2.24	-2.24	4
22	32	32	32	-0.62	-0.62	-0.62	1
23	24	27	24	-5.83	-3.52	-5.83	7
24	34	34	34	0.29	0.29	0.29	1
25	28	28	31	-2.86	-2.86	-1.13	3
26	26	24	28	-4.23	-5.83	-2.86	4
27	30	27	27	-1.67	-3.52	-3.52	2
28	22	27	25	-7.73	-3.52	-5.00	9
29	29	28	31	-2.24	-2.86	-1.13	11
30	32	33	31	-0.62	-0.15	-1.13	1
31	21	25	24	-8.81	-5.00	-5.83	13
32	32	33	32	-0.62	-0.15	-0.62	1
33	27	24	25	-3.52	-5.83	-5.00	5
34	22	28	20	-7.73	-2.86	-10.00	18
35	32	31	33	-0.62	-1.13	-0.15	1
36	28	30	27	-2.86	-1.67	-3.52	6
37	28	30	32	-2.86	-1.67	-0.62	4
38	33	32	31	-0.15	-0.62	-1.13	1
39	30	33	31	-1.67	-0.15	-1.13	4
40	31	32	31	-1.13	-0.62	-1.13	1
41	27	26	30	-3.52	-4.23	-1.67	6
42	29	28	25	-2.24	-2.86	-5.00	20

Continued on next page

Table A.4 – *Continued from previous page*

Experiment	R_1 **(KΩ)**	R_2 **(KΩ)**	R_0 **(KΩ)**	w_1	w_2	w_0	**Iterations**
43	29	23	21	-2.24	-6.74	-8.81	4
44	32	32	33	-0.62	-0.62	-0.15	1
45	31	31	33	-1.13	-1.13	-0.15	3
46	32	28	29	-0.62	-2.86	-2.24	4
47	25	24	24	-5.00	-5.83	-5.83	18
48	27	29	25	-3.52	-2.24	-5.00	12
49	30	28	28	-1.67	-2.86	-2.86	22
50	28	27	26	-2.86	-3.52	-4.23	10
51	33	32	32	-0.15	-0.62	-0.62	2
52	31	31	31	-1.13	-1.13	-1.13	2
53	32	32	32	-0.62	-0.62	-0.62	1
54	31	29	28	-1.13	-2.24	-2.86	19
55	33	32	32	-0.15	-0.62	-0.62	1
56	30	26	25	-1.67	-4.23	-5.00	14
57	22	28	22	-7.73	-2.86	-7.73	4
58	21	21	29	-8.81	-8.81	-2.24	5
59	26	27	28	-4.23	-3.52	-2.86	7
60	32	32	33	-0.62	-0.62	-0.15	1
Mean	28.90	28.88	28.63	-2.61	-2.54	-2.80	6.65
Variance	13.28	10.34	14.34	6.26	4.45	6.91	44.54

APPENDIX B

MADALINE RESULTS OF TRAINING WITH MRII

Table B.1: Resistor values trained with MRII for an XOR operator

Exp	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
1	33,332	34,658	33,012	35,133	32,577	32,889	35,468	35,467	34,981
2	33,030	34,084	32,856	35,122	32,968	31,748	35,090	33,960	35,300
3	31,952	32,332	34,866	29,514	30,135	30,844	34,184	32,454	32,776
4	32,276	35,412	32,210	42,546	29,980	30,803	35,081	38,509	38,483